ABSTRACT

XIAO, XUSHENG. Cooperative Testing and Analysis via Informed Decision Making. (Under the direction of Tao Xie and Laurie Williams.)

Software is pervasive in all aspects of our life, and thus it is critical to ensure high quality of software. Software quality includes both functional quality and non-functional quality. Functional quality refers to functional correctness of software, and non-functional quality supports the delivery of functional requirements, such as security. Failing to ensure high quality of software can result in serious consequences, such as causing software to behave incorrectly and compromising users' security and privacy. To improve software quality, software testing/analysis tools can be used to automate certain activities in software development and maintenance, reducing manual efforts of quality assurance.

Although tool automation is important in software testing and analysis for reducing manual efforts, tools face various problems when dealing with complex software, and such problems would be still difficult for the tools to tackle in the foreseeable future. For example, my ICSE 2011 work shows that even a state-of-the-art test-generation tool may achieve not more than 65% code coverage due to problems in dealing with method calls to external libraries. For some of these problems, tool users (e.g., developers and testers) may help the tools tackle these problems, such as providing mock objects to simulate the behaviors of the external libraries. With the provided mock objects, the test-generation tools can be reapplied to generate more test inputs, achieving new coverage and carrying out software testing more effectively. In such *cooperation*, the advances on test-generation tools free the users from labor-intensive and tedious tasks (e.g., manually producing test inputs for coverage) and the tools try their best in automating the tasks, while the users focus their efforts in addressing only the problems faced by the tools (e.g., providing mock objects). Note that the efforts in addressing the problems are typically less than the efforts in asking the users to finish all the remaining work for the tasks (e.g., generating test inputs for all the not-covered code). For example, with the provided mock objects, the tools can be reapplied to explore the not-covered code and generate more test inputs for covering the newly explored code. In this way, the users do not need to generate test inputs for all the newly explored code. If more problems are encountered in exploring the newly explored code, the users can provide their help again.

To support such new types of cooperation enabled by the advances on software testing /analysis tools, there is a strong need to provide a user-tool interface[1] that communicates suf-

[1] These user-tool interfaces are the places where the users interact with the tools. Unlike user interfaces in Human-Computer Interface (HCI) research, we do not focus on the representation of information and the ways for the users to manipulate the information in these interfaces, but focus on the contents being displayed or manipulated.

ficient and precise information to the users (e.g., reporting the problems faced by the tools), allowing the users to help the tools address the problems. However, many software testing/-analysis tools still do not communicate sufficient or precise information to the users, and little research has been done on such interfaces for these new types of cooperation. For example, most test-generation tools often report only the achieved coverage, but not the problems faced by the tools; for a few tools that report the problems, many reported problems are false warnings. Such situations pose difficulties for the users to help the tools address the problems.

To maximize the value of software testing and analysis for improving software quality, this dissertation proposes a framework of *cooperative testing and analysis*, which provides interfaces that enable the users to make informed decisions in cooperation with software testing/analysis tools. With the advances on software testing/analysis tools, many new types of cooperation emerge for various software engineering (SE) tasks. In these types of cooperation, the tools free the users from tedious and mechanic subtasks, and the users' subtasks have been shifted to the subtasks that are dependent on the users' domain knowledge and expectations. To better support such new types of cooperation, there is a strong need to provide the interfaces for communicating sufficient and precise information to the users.

In this dissertation, our framework focuses on providing interfaces to support two types of cooperation with software testing/analysis tools for SE tasks:

- **Problem-Diagnosis Interface**: For certain SE tasks where automated tools drive the tasks towards a goal of testing and analysis, users can help the tools address the encountered problems since the complicated characteristics of the code under test are beyond the capability of the tools. For such SE tasks, our framework provides *problem-diagnosis interfaces* to support a type of cooperation, called *problem-diagnosis cooperation*, as follows. Users first set up and apply a tool to conduct initial testing and analysis on a program. The provided interface then shows the feedback to the users, including the effectiveness of the tool and the problems faced by the tool; these problems are precisely identified and the benefits of solving these programs are accurately estimated using analyses of the program and the tool. By looking at the problems, the users provide guidance to the tool based on the feedback, helping the tool address the problems. Such process forms a feedback loop and can be repeated until the effectiveness is satisfied or the users run out of patience.

- **Behavior-Diagnosis Interface**: For certain SE tasks where users inspect software behaviors to determine whether the behaviors are expected for achieving a goal of testing and analysis, tools reveal and explain software behaviors to help the users make informed decisions, because behavior expectation lies in the mind of the users and the tools need the users to make decisions. For such SE tasks, our framework provides *behavior-diagnosis*

interfaces to support a type of cooperation, called *behavior-diagnosis cooperation*, as follows. Users first set up and apply a tool to conduct initial testing and analysis on a program. The provided interface then shows a list of behaviors related to the goal for the users to inspect, and the users make decisions on whether these behaviors are expected. Furthermore, the interface also provides explanations for the behaviors. Based on whether the behaviors are in the form of either concrete instances or abstract models, these explanations can be in the form of (1) *extension*: a larger scope of information to collect more concrete instances or extend the models of the behaviors, (2) *instantiation*: instantiations of the behaviors' abstract models, or (3) *abstraction*: inferred models from the concrete instances of the behaviors. Such explanations help the users make informed decisions on the behaviors to be inspected.

The key difference between problem-diagnosis cooperation and behavior-diagnosis cooperation is that problem-diagnosis cooperation typically engages the users to diagnose some intermediate results used for producing the final results, while behavior-diagnosis cooperation typically engages the users to diagnose the final results directly. For example, after a test-generation tool generates test inputs for a program, problem-diagnosis cooperation engages the users to diagnose the reported problems and address the problem for helping the tool in achieving higher coverage. If the tool generates an input that causes the program to throw uncaught exceptions, behavior-diagnosis cooperation engages the users to diagnose the exceptions and confirm whether such exceptions indicate real faults. Also, the information of how the exception-throwing states are reached in the failing executions is shown to the users (i.e., a larger scope of information is used to explain the exceptions), helping the users make informed decisions in diagnosing exceptions.

The basic rationale of cooperative testing and analysis is that tools and users typically have their respective strengths and weaknesses. For example, the tools are good at automating mechanic and repetitive tasks that have well-defined goals, such as generating a random number or searching an array of numbers for a specific number. Although the users are not good at such tasks, the users are good at tasks that are dependent on domain knowledge and user expectations, such as addressing problems faced by the tools (e.g., providing mock objects based on their domain knowledge about the external libraries) and confirming whether a software behavior (e.g., sending text SMS) is expected for an application (e.g., a navigation mobile application). Thus, creating interfaces that enable the users and the tools to do subtasks that they are good at provides opportunities to improve the effectiveness of various SE tasks. For an SE task on which our framework is applied, we assume user subtasks and tool subtasks are already identified. Optimizing the allocation of user subtasks and tool subtasks for an SE task is out of the scope of this dissertation.

We propose a number of approaches under cooperative testing and analysis, where each approach provides the interface to support cooperation between tools and their users for a specific SE task, improving either functional or non-functional quality of software.

First, we propose approaches to better support cooperation for automated test generation, ensuring functional correctness of software. The goal of automated test generation is generating test inputs for achieving high code coverage. However, even the state-of-the-art test-generation tools have difficulties in achieving high coverage for complex software. To improve the effectiveness of the test-generation tools, our approaches provide a problem-diagnosis interface that precisely reports the problems faced by the test-generation tools, and shows the ordering of the problems based on their estimated benefits. Such interface enables the problem-diagnosis cooperation for test-generation tools. In this cooperation, the tools automatically generate test inputs for achieving coverage of a program and report problems faced by the tools for not-covered code. Based on the users' domain knowledge of the program under test, the users provide their guidance to address the problems. To identify major types of problems that prevent test-generation tools from achieving high coverage for complex software, we first conduct empirical studies on popular open-source projects. Based on the study results, we then propose an approach to precisely identify these problems. To allow prioritization of the problems, we further propose an economic-analysis approach to accurately estimate the benefit of solving a problem, i.e., how much coverage improvement can be obtained by solving the problem.

Second, we propose approaches that provide behavior-diagnosis interfaces to better support cooperation in security analysis, with the focus on mobile privacy control and security policy extraction from requirements documents. The goal of mobile privacy control is controlling mobile applications' accesses to the users' privacy-sensitive information, i.e., the users making decisions on whether to allow or deny permissions that protect certain privacy-sensitive information. However, existing mobile platforms show only what privacy-sensitive information the mobile applications request to use (i.e., what permissions are requested by the mobile applications), but do not explain how such privacy-sensitive information is used by the mobile applications, causing the users to make uninformed decisions on controlling their privacy. To improve mobile privacy control, besides showing what permissions are requested by the mobile applications, our approach provides a behavior-diagnosis interface that shows information flows of the requested permissions (i.e., showing what types of privacy-sensitive information flow to what output channels). Such a larger scope of information explains how the users' privacy-sensitive information is used by the mobile applications, enabling the behavior-diagnosis cooperation for mobile privacy control. In this cooperation, the tools identify permissions requested by the mobile applications for the users to inspect and explain the permissions by showing information flows. Based on the users' domain knowledge about the mobile applications' functionality and security requirements, the users inspect the information flows to determine whether the infor-

mation flows are expected for the mobile applications, and make decisions on granting accesses for the permissions.

The goal of security policy extraction is controlling resource accesses for a software system. In practice, security policies are commonly written in Natural Language (NL) and are supposed to be written in security requirements. However, often ACPs are buried in NL documents such as requirement documents, and these NL software documents could be large in size, making it tedious and error-prone to manually inspect these NL documents for extracting security policies. To improve security policy extraction, our approach provides a behavior-diagnosis interface that shows the security policies automatically extracted from the sentences in requirements documents, and the users make decisions on whether the security policies are expected based on their domain knowledge about the software system's functionality and security requirements. Also, scenario-based functional requirements (such as use cases) contain sequences of action steps that describe actors (principals) who access different resources for achieving some functionalities and help developers determine what system functionality to implement. Validating these action steps against the extracted policies explains what action steps violating the policies. Thus, our behavior-diagnosis interface further shows the requirements documents where the security policies are extracted as policy-witness scenarios, and the inconsistencies between action steps and the extracted policies as policy-violation scenarios. Such policy-witness and policy-violation scenarios use the instantiations of the extracted security policies as explanations for the security policies, enabling the behavior-diagnosis cooperation for security policy extraction. In this cooperation, the tools extract security policies for the users to inspect and explain the security policies by showing the policy-witness and policy-violation scenarios. Based on the users' domain knowledge about the software system's functionality and security requirements, the users inspect the policy-witness and policy-violation scenarios (instantiations of the security policies) to determine whether these scenarios are expected for the software system, and make decisions on the extracted security policies.

Finally, we propose an approach that provides a behavior-diagnosis interface to better support cooperation in performance analysis, with the focus on identifying workload-dependent performance bottlenecks (WDPBs). The goal of performance analysis is detecting performance faults that unexpectedly compromise the performance of a program. In such task, it is difficult for tools to know the oracle for performance faults, i.e., how slow is slow enough to consider a method as a performance fault. For such task, it is better for users to make decisions on whether the costs of certain methods are expected based on the users' domain knowledge about the functionality and performance requirements of the program under analysis. To identify WDPBs, many existing performance analysis approaches report *what* are the expensive methods for the users to inspect without explaining *how* these methods become expensive, posing challenges for the users to identify WDPBs. To improve performance analysis, besides showing the costs

of the executed methods, our approach provides a behavior-diagnosis interface that shows the complexity models of methods. Our approach infers complexity models of workload-dependent loops from multiple executions of the program on multiple workloads. Such complexity models inferred from the concrete executions explain how the costs of certain methods grow in larger workloads, enabling the behavior-diagnosis cooperation for performance analysis. In this cooperation, the tools compute the costs of the executed methods for the users to inspect and explain the cost growth of the methods by showing the inferred complexity models. Based on the users' domain knowledge about the functionality and performance requirements of the program under analysis, the users inspect the complexity models of the methods to determine whether the cost growths represented by the models are expected for the methods, and make decisions on whether the methods are performance faults.

Our empirical results show that the approaches developed under our framework provide interfaces to effectively support cooperation for the respective SE tasks. In particular, our results demonstrate the effectiveness of our approaches in identifying problems faced by test-generation tools and estimating the benefits of solving the problems on real-world open source projects, enabling the problem-diagnosis cooperation for test-generation tools and thereby improving software correctness. Our results also show that our security-analysis approaches effectively compute information flows for open-source mobile applications and extract security policies from both open source and proprietary functional requirements documents. Such results show that our security-analysis approaches enable the behavior-diagnosis cooperation for mobile privacy control and security policy extraction, and thereby improve software security and privacy. Finally, our results show that our performance-analysis approach effectively infers complexity models from executions of real-world open source projects and helps detect new performance faults not detected by existing related approaches, enabling the behavior-diagnosis cooperation for performance analysis and thereby improving software performance. Note that no human subjects experiments appear in this work because our framework focuses on providing interfaces to support cooperation with software testing/analysis tools in various SE tasks, and our evaluations aim to assess the quality of the information presented by the interfaces.

Cooperative Testing and Analysis via Informed Decision Making

by
Xusheng Xiao

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Emerson Murphy-Hill	Douglas Reeves
Tao Xie Co-chair of Advisory Committee	Laurie Williams Co-chair of Advisory Committee

DEDICATION

To my wife, my parents, and my grand parents.

BIOGRAPHY

Xusheng Xiao was born in Shantou, Guangdong, China. He received his B.E. degree from Chongqing University in 2007. In 2009, he started graduate studies at North Carolina State University in Raleigh, completing an M.S. degree in 2011. In 2013 and 2014, he was a visiting student at the University of Illinois at Urbana-Champaign. His research in software engineering focuses on improving cooperation between software testing/analysis tools and their users. He has been awarded the ICSE SRC Best Project Representing an Innovative Use of Microsoft Technology at ACM SRC Grand Final 2012. His work on mobile security is integrated into TouchDevelop developed by Microsoft Research and is granted a U.S. patent. Before joining the Ph.D. program, Xusheng was a Consultant/Developer who specialized in Agile Software Development in ThoughtWorks. During his PhD program, his internship experiences include IBM T.J. Watson Research Center (Summer 2010), Microsoft Research Redmond (Summers 2011 and 2013), Microsoft Research Asia (Fall 2011), and NEC Laboratories America (Summer 2012). He is a student member of ACM and ACM SIGSOFT.

ACKNOWLEDGMENTS

I would like to start by giving my deepest thanks to my advisor, Tao Xie, who supports me with freedom and guidance throughout my Ph.D. studies. During the past five years, with the largest possible freedom offered by Tao, I was able to pursue my own research interests, collaborate with other groups, and do internships in industry research labs, etc. Tao has also provided me valuable technical and professional advice, which improves my research skills, technical writing, and effective presentation, and will continue to benefit me in my future career. His dedicated mentorship made my five-year Ph.D. studies a fruitful and fulfilling experience. I want to thank my co-advisor Laurie Williams for her support of my dissertation research. I also would like to thank Emerson Murphy-Hill and Douglas Reeves for serving on my dissertation committee and providing valuable feedback on my dissertation research. Portions of this dissertation were previously published at ICSE 11 [205] (Chapter 4), ASE 13 [200] (Chapter 4), ASE 12 [203] (Chapter 6), FSE 12 [201] (Chapter 7), ISSTA 13 [199] (Chapter 8). The material included in this dissertation has been updated and extended.

I thank my internship mentors and collaborators in Microsoft Research and IBM Research, who have immensely helped me with the work described in this dissertation. Nikolai Tillmann and Jonathan de Halleux (Microsoft Research, USA) provided support for the Pex tool and offered their insightful feedback for my research on automated test generation. Nikolai Tillmann, Jonathan de Halleux, Manuel Fahndrich, and Michal Moskal (Microsoft Research, USA) helped me in integrating my approach into the TouchDevelop platform and provided the TouchDevelop programs for my evaluation. Amit Paradkar (IBM Research, USA) provided support for the natural language processing (NLP) parser for my research on extraction of ACP rules from use cases, and Suresh Thummalapenta (Microsoft, USA; formerly IBM Research, India) provided his help for my evaluations. Shi Han and Dongmei Zhang (Microsoft Research, China) provided support for the performance profiling tool and guidance for my research on performance analysis. I thank Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta (NEC Laboratories America) for their guidance and help during my 2012 summer internship in NEC Laboratories America, and thank Mark Marron and Sumit Gulwani (Microsoft Research, USA) for their mentoring and help during my 2013 summer internship in Microsoft Research. Thanks also go to other students I met during my internships for their technical discussions. Working with such great researchers and students has helped expand my research to a broader scope.

I am very much thankful to all my peers at the Automated Software Engineering group, Tao Xie's research group at NC State University/University of Illinois at Urbana-Champaign, for their constant support, insightful discussions and useful feedback on my research: Suresh Thummalapenta, Kunal Taneja, Madhuri Marri, JeeHyun Hwang, Rahul Pandita, Justin Gorham,

Kiran Shakya, Mithun Acharya, Linghao Zhang, Yuan Yao, Yu Xiao, Sihan Li, Wei Yang, and Blake Bassett. My research was supported by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, CNS-1318419, an NCSU CACC grant, Army Research Office (ARO) grants W911NF-08-1-0443 and W911NF-08-1-0105 managed by NCSU SOSI, and a Microsoft Research Software Engineering Innovation Foundation Award. Gratitude to the Department of Computer Science at NC State University for Teaching Assistantship opportunities and financial support during my Ph.D studies.

I would like to thank Tao Xie, Zhendong Su, Sumit Gulwani, and Nikolai Tillmann, who wrote recommendation letters for me and helped me throughout my job search process. I also would like to thank Mark Marron, who invited me to present my research and visit Microsoft Research for a week during my job search. I thank Milos Gligoric, William Enck, and Mohsen Vakilian for their feedback on my research statement and presentation. I also want to thank the people from different schools who helped my job search.

Last but not least, I am very grateful to my family. My dear wife Ling Chen, my parents Weixiong Xiao and Liling Zhang, my sister Xuyan Xiao, and my grandparents Wei Zhang and Lijuan Yang provide me their continuous support, encouragement, and love. Without their support, this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

Software is pervasive in all aspects of our life, and thus it is critical to ensure high quality of software. Software quality includes both functional quality and non-functional quality. Functional quality refers to functional correctness of software, and non-functional quality supports the delivery of functional requirements, such as security. Failing to ensure high quality of software would result in serious consequences. Software with poor functional quality has many functional faults, causing the software to behave incorrectly. Recently, a functional fault causes Knight Capital's computers to execute a series of automatic orders that were supposed to be spread out over a period of days, costing Knight Capital about 440 million dollars [23]. A report by National Institute of Standards and Technology found that software faults cost the U.S. economy about $60 billion each year [151]. Software with poor non-functional quality may compromise users' security and privacy. For example, posting your kid's photo using applications with geo-tagging may expose your kid's exact location to strangers [19].

To improve software quality, software testing/analysis tools can be used to automate certain activities in software development and maintenance, reducing manual efforts of quality assurance. For example, structural test-generation tools can be used to produce test inputs, automatically achieving high structural coverage [30, 125, 42, 103]. Static-analysis tools can be used to prove pre-defined properties, automatically verifying the correctness of software [37, 83].

Although tool automation is important in software testing and analysis for reducing manual efforts, tools face various problems when dealing with complex software; such problems would

be still difficult for the tools to tackle in the foreseeable future. For example, my ICSE 2011 work shows that even a state-of-the-art test-generation tool may achieve not more than 65% code coverage due to problems in dealing with method calls to external libraries [205], and static-analysis tools cannot prove certain properties in complex software [66]. For some of these problems, tool users (e.g., developers and testers) may help the tools tackle these problems, such as providing mock objects to simulate the behaviors of the external libraries. With the provided mock objects, test-generation tools can be reapplied to generate more test inputs, achieving new coverage and carrying out software testing more effectively. In such *cooperation*, the advances on test-generation tools free the users from labor-intensive and tedious tasks (e.g., manually producing test inputs for coverage) and the tools try their best in automating the tasks, while the users focus their efforts in addressing only the problems faced by the tools (e.g., providing mock objects). Note that the efforts in addressing the problems are typically less than the efforts in asking the users to finish all the remaining work for the tasks (e.g., generating test inputs for all the not-covered code). For example, with the provided mock objects, the tools can be reapplied to explore the not-covered code and generate more test inputs for covering the newly explored code. In this way, the users do not need to generate test inputs for all the newly explored code. If more problems are encountered in exploring the newly explored code, the users can provide their help again.

To support such new types of cooperation enabled by the advances on software testing/analysis tools, there is a strong need to provide a user-tool interface that communicates sufficient and precise information to the users (e.g., reporting the problems faced by the tools), allowing the users to help the tools address the problems. These user-tool interfaces are the places where the users interact with the tools. Unlike user interfaces in Human-Computer Interface (HCI) research [109], we do not focus on the representation of information and the ways for the users to manipulate the information in these interfaces, but focus on the contents being displayed or manipulated.

However, many software testing/analysis tools still do not communicate sufficient or precise information to the users, and little research has been done on such interfaces for these new types of cooperation. For example, most test-generation tools often report only the achieved coverage, but not the problems faced by the tools; for a few tools that report the problems, many of the reported problems are false warnings. Such situations pose difficulties for the users to help the tools address the problems.

To maximize the value of software testing and analysis for improving software quality, this dissertation proposes a framework of *cooperative testing and analysis*, which provides *interfaces* that enable the users to make informed decisions in cooperation with software testing/analysis tools. With the advances on software testing/analysis tools, many new types of cooperation emerge for various software engineering (SE) tasks. In these types of cooperation, the tools free

the users from tedious and mechanic subtasks, and the users' subtasks have been shifted to the subtasks that are dependent on the users' domain knowledge and expectations. To better support such new types of cooperation, there is a strong need to provide the interfaces for communicating sufficient and precise information to the users.

1.2 Tasks and Challenges

Under cooperative testing and analysis, we propose a number of approaches that provide interfaces to support cooperation for three important SE tasks: automated test generation, security analysis, and performance analysis. These tasks are major SE tasks that improve both functional quality and non-functional quality of software, with the focus on assuring three major attributes in software quality: functional correctness, security, and performance. The tasks and challenges are described below.

- Software testing is one of the most widely-used approaches for improving functional correctness of software, but it is typically a labor-intensive and costly process [146, 30]. To reduce manual efforts in conducting testing, automated test generation is employed to automatically produce high-covering tests. Although tools of automated test generation can effectively handle certain programs, they face problems when dealing with complex programs in practice. Based on recent studies [204, 87, 116], the top two major types of problems that prevent these tools from achieving high code coverage are (1) the object-creation problem (OCP), where the tools fail to generate sequences of method calls to construct desired object states for covering certain branches; (2) the external-method-call problem (EMCP), where the tools cannot deal with method calls to external libraries, such as native system libraries or pre-compiled third-party libraries. For these types of problems, the tool users can provide guidance to the tools, helping the tools address these problems. As an example of providing guidance to the tools, the developers can write factory methods that encode sequences of method calls to produce desired object states to deal with OCPs [184]. To deal with EMCPs, the tool users can instruct the tools to instrument and explore the external libraries or write mock objects [188, 181] to simulate the behaviors of the external libraries. However, most test-generation tools report only the achieved coverage, but do not explain the problems faced by the tools. Even for a few tools that report the problems (such as Pex [184], a state-of-the-art test-generation tool), the reported problems are often false warnings. Therefore, the users have to reason about the problems that cause certain branches not to be covered. Such reasoning is time-consuming, tedious, and error-prone, preventing the users from providing their guidance effectively.

- The increasing popularity of smartphones has made them a target for mobile threats of malware and potentially unwanted applications [76]. To address such security issues, the predominant smartphone platforms (Apple, Google, and Windows Phone) rely on either employees to manually validate applications, or end users to make decisions on granting permissions to access privacy-sensitive information. The manual validation process is costly and delays publishing of applications. It is also incomplete, since it cannot examine every execution path to detect violations of privacy policies [91]. Access-control granting by users provides information about *what* private information these applications may access, rather than *how* these applications use private information, causing the users to make uninformed decisions on how to control their privacy. These privacy control mechanisms lead to a situation where the users simply install applications without questioning the requested permissions, even if the applications may silently leak private information [80, 189, 72].

- Access control is one of the most fundamental and widely used privacy and security mechanisms. Access control is often governed by an Access Control Policy (ACP) [165] that includes a set of rules specifying which principals (such as users or processes) have access to which resources. It is important to ensure correct specification of ACPs and correct enforcement of ACP specifications; otherwise, there could be serious consequences such as allowing an unauthorized user to access protected resources. In practice, ACPs are commonly written in Natural Language (NL) and are supposed to be written in security requirements, a type of non-functional requirements. However, often ACPs are buried in NL documents such as requirement documents, and these NL software documents could be large in size, often consisting of hundreds or even thousands of sentences (iTrust, an open source healthcare project consists of 37 use cases [113] with 448 use-case sentences), where a portion of the sentences describing ACPs (117 sentences in iTrust) are buried among other sentences [201]. Also, in software development, there exists an inherent gap between ACPs specified using domain concepts used in requirements and the actual system implementation developed using programming concepts. This gap poses problems for ensuring the consistency between requirements and manually-constructed policies based on system implementations. Manually inspecting large requirements documents to extract ACPs is labor-intensive and tedious. For example, a proprietary IBM enterprise application (that we used in our evaluations) includes 659 use cases with 8,817 sentences. Thus, it is very tedious and error-prone to manually inspect these NL documents for identifying and extracting ACPs for policy modeling and specification.

- Performance problems commonly exist in real-world applications running in systems of various sizes from smartphones to servers [118, 89], and performance analysis helps devel-

opers identify performance faults that cause severe scalability reductions and even financial losses [31, 96, 32]. As a type of widespread performance problems, workload-dependent performance bottlenecks (WDPBs) in responsive actions, which are expected to return instantly, cause software hangs (i.e., unresponsiveness of software applications) [172]. Traditional performance testing that mainly relies on black-box random testing or manual input design often misses WDPBs since WDPBs may not surface on small or even relatively large workloads [143]. In addition, many existing performance analysis approaches report *what* are the expensive methods for users to inspect without explaining *how* these methods become expensive, posing challenges for the users to identify WDPBs.

1.3 Summary

To address these preceding problems and thereby maximize the value of software testing and analysis for improving software quality, in this dissertation, we propose a framework of *cooperative testing and analysis*, which provides interfaces that enable users to make informed decisions in cooperation with software testing/analysis tools. Our framework focuses on providing interfaces to support two types of cooperation with software testing/analysis tools for various SE tasks:

- **Problem-Diagnosis Interface**: For certain SE tasks where automated tools drive the tasks towards a goal of testing and analysis, the users can help the tools address the encountered problems since the complicated characteristics of the code under test are beyond the capability of the tools. For such SE tasks, our framework provides *problem-diagnosis interfaces* to support a type of cooperation, called *problem-diagnosis cooperation*, as follows. Users first set up and apply a tool to conduct initial testing and analysis on a program. The provided interface then shows the feedback to the users, including the effectiveness of the tool and the problems faced by the tool; these problems are precisely identified and the benefits of solving these programs are accurately estimated using analyses of the program and the tool. By looking at the problems, the users provide guidance to the tool based on the feedback, helping the tool address the problems. Such process forms a feedback loop and can be repeated until the effectiveness is satisfied or the users run out of patience.

- **Behavior-Diagnosis Interface**: For certain SE tasks where users inspect software behaviors to determine whether the behaviors are expected for achieving a goal of testing and analysis, tools reveal and explain software behaviors to help the users make informed decisions, because behavior expectation lies in the mind of the users, and the tools need the users to make decisions. For such SE tasks, our framework provides *behavior-diagnosis*

interfaces to support a type of cooperation, called *behavior-diagnosis cooperation*, as follows. Users first set up and apply a tool to conduct initial testing and analysis on a program. The provided interface then shows a list of behaviors related to the goal for the users to inspect, and the users make decisions on whether these behaviors are expected. Furthermore, the interface also provides explanations for the behaviors. Based on whether the behaviors are in the form of either concrete instances or abstract models, these explanations can be in the form of (1) *extension*: a larger scope of information to collect more concrete instances or extend the models of the behaviors, (2) *instantiation*: instantiations of the behaviors' abstract models, or (3) *abstraction*: inferred models from the concrete instances of the behaviors. Such explanations help the users make informed decisions on the behaviors to be inspected.

The key difference between problem-diagnosis cooperation and behavior-diagnosis cooperation is that problem-diagnosis cooperation typically engages the users to diagnose some intermediate results used for producing the final results, while behavior-diagnosis cooperation typically engages the users to diagnose the final results directly. For example, after a test-generation tool generates test inputs for a program, problem-diagnosis cooperation engages the users to diagnose the reported problems and address the problem for helping the tool in achieving higher coverage. If the tool generates an input that causes the program to throw uncaught exceptions, behavior-diagnosis cooperation engages the users to diagnose the exceptions and confirm whether such exceptions indicate real faults. Also, the information of how the exception-throwing states are reached in the failing executions is shown to the users (i.e., a larger scope of information is used to explain the exceptions), helping the users make informed decisions in diagnosing exceptions.

The basic rationale of cooperative testing and analysis is that tools and users typically have their respective strengths and weaknesses. For example, the tools are good at automating mechanic and repetitive tasks that have well-defined goals, such as generating a random number or searching an array of numbers for a specific number. Although the users are not good at such tasks, the users are good at tasks that are dependent on domain knowledge and user expectations, such as addressing problems faced by the tools (e.g., providing mock objects based on their domain knowledge about the external libraries) and confirming whether a software behavior (e.g., sending text SMS) is expected for an application (e.g., a navigation mobile application). Thus, creating interfaces that enable the users and the tools to do subtasks that they are good at provides opportunities to improve the effectiveness of various SE tasks. For an SE task on which our framework is applied, we assume user subtasks and tool subtasks are already identified. Optimizing the allocation of user subtasks and tool subtasks for an SE task [62] is out of the scope of this dissertation.

We propose a number of approaches under cooperative testing and analysis, where each approach provides the interface to support cooperation between users and tools for a specific SE task, improving either functional or non-functional quality. In summary, this dissertation makes the following major contributions:

- **Problem-Diagnosis Cooperation for Identifying Problems Faced by Structural Test Generation.** To understand what types of problems prevent test-generation tools from achieving high structural coverage of complex object-oriented programs, we conduct empirical studies on open-source projects [204, 200]. Our studies on open-source projects leverage Pex [184], a state-of-the-art test-generation tool based on Dynamic Symbolic Execution (DSE) [167, 92, 125], to generate test inputs for the open-source projects, and we manually study the not-covered branches to identify the problems faced by Pex. We observe that most of the not-covered branches are due to two major types of problems: OCPs and EMCPs.

  The goal of automated test generation is generating test inputs for achieving high code coverage. Based on the study results, test-generation tools have difficulties in achieving high code coverage due to two major types of problems: OCPs and EMCPs. To improve the effectiveness of the test-generation tools, we propose an approach, Covana [204, 198], that precisely identifies the causes of the observed symptoms primarily by determining whether not-covered branches have data dependencies on problem candidates. Based on Covana, we provide a problem-diagnosis interface that shows the problems faced by structural test-generation tools, enabling the problem-diagnosis cooperation for test-generation tools. In this cooperation, tools automatically generate test inputs for achieving coverage of a program and report problems faced the tools for not-covered code. Based on the users' domain knowledge of the program under test, the users provide factory methods for addressing OCPs, and provide mock objects for addressing EMCPs.

  Our evaluations on two open-source projects (xUnit and QuickGraph) show that Covana effectively identifies 43 EMCPs out of 1610 EMCP candidates with only 1 false positive and 2 false negatives, and 155 OCPs out of 451 OCP candidates with 20 false positives and 30 false negatives.

- **Problem-Diagnosis Cooperation for Prioritizing Problems Faced by Structural Test Generation.** When developers apply test-generation tools on complex software in practice, often a long list of causes to not-covered code (i.e., symptoms) can be presented to the developers. Given limited time, the developers may not be able to address all the causes. Thus, it is desirable to enable the developers to maximize their testing goals within the given time of addressing the causes. For example, if the goal is to achieve high overall

structural coverage, the developers could first address problems that bring high coverage improvement.

To enable such economical cooperation of developers and tools, we propose an economic-analysis framework, called EcoCov, that accurately estimates the benefit of addressing a cause and the cost of eliminating a symptom. Based on EcoCov, our approach provides the problem-diagnosis interface to show the ordering of the causes based on their estimated benefits, extending the problem-diagnosis interface of precisely reporting problems faced by structural test-generation tools.

We name the main idea of our EcoCov framework as *angelic diagnosis*[1] because it uses an angelically nondeterministic operator to generate desired values to bypass the problems. In this framework, we concretize addressing a cause as solving a problem, either an OCP or an EMCP, and eliminating a symptom as covering a not-covered branch. We evaluate EcoCov on three open-source C# projects and the evaluation results show that EcoCov achieves averagely 91.1% precision and 96.6% recall in estimating the problem benefit.

- **Behavior-Diagnosis Cooperation for Mobile Privacy Control.** Existing privacy-control approaches employed by mobile platforms show limited success, partly because these approaches report only *what* permissions are used by mobile applications, rather than *how* these permissions are used by the applications, making the users make uninformed decisions on controlling their privacy.

  The goal of mobile privacy control is controlling mobile applications' accesses to the users' privacy-sensitive information, i.e., the users making decisions on whether to allow or deny permissions that protect accesses to certain privacy-sensitive information. To improve mobile privacy control, besides showing what privacy-sensitive information the mobile applications request to use (i.e., what permissions are requested by the mobile applications), our approach provides a behavior-diagnosis interface [203] that shows information flows of the requested permissions (i.e., showing what types of privacy-sensitive information flowing to what output channels). Such a larger scope of information, i.e., information flows, explains how the users' privacy-sensitive information is used by the mobile applications, enabling the behavior-diagnosis cooperation for mobile privacy control. In this cooperation, the tools identify the permissions requested by the mobile applications for the users to inspect and explain the permissions by showing information flows. Based on the users' domain knowledge about the mobile applications' functionality and security requirements, the users inspect the information flows to determine whether the information

[1]The meaning of angelic in this dissertation is related to but different from that in angelic debugging [49] and angelic programming [39]. Here, an angelic value makes a problem disappear while in angelic debugging and programming, an angelic value corrects an execution.

flows are expected for the mobile applications, and make decisions on granting accesses for the permissions.

We built our approach into TouchDevelop [186], and evaluated our approach by studying 546 applications published by 194 users. Since our approach requires the users to make decisions for each permission in a given mobile application, it is very important to reduce the decisions that the users have to make. The results show that among the 546 applications, our approach reduces the need to make access granting decisions to only 10.1% (54) of all applications.

- **Behavior-Diagnosis Cooperation for Security Policy Extraction from Natural-Language Software Documents.** Ensuring the correctness and consistency of ACPs is crucial to prevent security vulnerabilities. However, in practice, ACPs are commonly written in Natural Language (NL) and buried in large documents such as requirements documents, not amenable for automated techniques to check for correctness and consistency. It is tedious to manually extract ACPs from these NL documents and validate NL functional requirements such as use cases against ACPs for detecting inconsistencies.

  The goal of security policy extraction is controlling resource accesses for a software system. To improve security policy extraction, we propose an approach, called Text2Policy [201], to automatically extract ACPs from NL software documents and resource-access information from NL scenario-based functional requirements. Based on Text2Policy, our approach provides a behavior-diagnosis interface that shows the security policies automatically extracted from the sentences in requirements documents, and the users make decisions on whether the security policies are expected based on their domain knowledge about the software system's functionality and security requirements. Also, scenario-based functional requirements (such as use cases) contain sequences of action steps that describe actors (principals) access different resources for achieving some functionalities and help developers determine what system functionality to implement. Validating these action steps against the extracted policies explains what action steps violating the policies. Thus, our behavior-diagnosis interface further shows the requirements documents where the security policies are extracted as policy-witness scenarios, and the inconsistencies between action steps and the extracted policies as policy-violation scenarios. Such policy-witness and policy-violation scenarios use the instantiations of the extracted security policies as explanations of the security policies, enabling the behavior-diagnosis cooperation for security policy extraction. In this cooperation, the tools extract security policies for the users to inspect and explain the security policies by showing the policy-witness and policy-violation scenarios. Based on the users' domain knowledge about the software system's functionality and security requirements, the users inspect the policy-witness and policy-

violation scenarios (instantiations of the security policies) to determine whether these scenarios are expected for the software system, and make decisions to include or reject the security policies.

We conducted three evaluations on the collected ACP sentences from publicly available sources along with use cases from both open source and proprietary projects. Our results show that Text2Policy effectively identifies ACP sentences with the precision of 88.7% and the recall of 89.4%, extracts ACP rules with the accuracy of 86.3%, and extracts action steps with the accuracy of 81.9%. These action steps can be used to validate against the ACP rules directly, identifying the inconsistencies as policy-violation scenarios.

- **Behavior-Diagnosis Cooperation for Identifying Workload-Dependent Performance Bottlenecks.** To identify WDPBs that cause software hangs, traditional performance testing that mainly relies on black-box random testing or manual input design often misses WDPBs since WDPBs may not surface on small or even relatively large workloads [118, 143, 89]. In addition, many performance analysis approaches report *what* are the expensive methods for users to inspect without explaining *how* these methods become expensive.

  The goal of performance analysis is detecting performance faults that unexpectedly compromise the performance of a program, and our focus is to detect WDPBs. We propose an approach, ΔInfer [199], which infers *complexity models* of workload-dependent loops from executions on different workloads. For example, a complexity model of some loop can be linear, $a \cdot n + b$, in terms of the input size n. Based on ΔInfer, our approach provides a behavior-diagnosis interface that shows the complexity models of methods besides the costs of the executed methods. Our approach infers complexity models of workload-dependent loops from multiple executions of the software on multiple workloads. Such complexity models inferred from the concrete executions explain how the costs of certain methods grow in larger workloads, enabling the behavior-diagnosis cooperation for performance analysis. In this cooperation, the tools compute the costs of the executed methods for the users to inspect and explain the cost growth of the methods by showing the inferred complexity models. Based on the users' domain knowledge about the functionality and performance requirements of the program under analysis, the users inspect the complexity models of the methods to determine whether the cost growth represented by the models are expected for the methods, and make decisions on whether the methods are performance faults.

  Our evaluations on two open-source projects (7Zip [3] and Notepad++ [6]) show that ΔInfer infers complexity models with a high prediction accuracy and effectively identifies

10 performance faults (8 of them are new faults) without the need of executing the projects on large workloads.

1.4 Scope

The approaches presented in this dissertation focus on automated test generation, security analysis, and performance analysis. Our approaches for improving automated test generation focus on structural test generation that generates test inputs for a program unit (such as a class) written in modern object-oriented languages (such as C# and Java). The goal of structural test generation focuses on achieving high structural coverage [30]. Our approaches for security analysis focus on computing information flows from mobile applications and extracting ACP policies from requirements documents [113] written in NL. Our analysis for mobile applications focuses on TouchDevelop [186] applications, but can be extended to other mobile applications in Android and iOS with additional techniques described in Chapter 9. Our approach of extracting ACP policies focuses on requirements documents, such as scenario-based requirements (e.g., use cases [113]) that specify sequences of action steps to describe user interactions with the software system under development. Our approach for performance analysis focuses on identifying WDPBs in GUI applications that employ a single UI thread to update GUI. UI updates are allowed in only the UI thread, and thus operations in the UI thread are expected to return instantly [15]. Such GUI applications are widely seen in daily applications, such as Windows Form applications [12] and Java AWT [2]/Swing [11].

The approaches presented in this dissertation focus on sequential programs but not concurrent programs. Although a majority of our approaches focus on open-source projects available on the Internet, our approaches are general and can be applied on proprietary software artifacts as well, as shown in the evaluation results of our Text2Policy approach [201]. Our approaches focus on analyzing both structured software artifacts, such as source code and traces, and unstructured software artifacts, such as NL requirements documents. Finally, our approaches focus on both testing functional correctness or program robustness and analyzing other quality attributes such as performance and security. However, there exist various other quality attributes such as energy efficiency and maintainability. Chapter 9 discusses our new approaches that target at quality attributes beyond functional correctness, performance, and security.

1.5 Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background information of cooperative testing and analysis and three major SE tasks: automated test generation, security analysis, and performance analysis, and surveys related work. Chapter 3 presents

the key concepts of cooperative testing and analysis. The subsequent chapters present a number of approaches developed under our framework for three major SE tasks: automated test generation, security analysis, and performance analysis. In particular, Chapter 4 presents the studies and the approach for identifying problems faced by test-generation tools. Chapter 5 describes the approach that enables economical cooperation with test-generation tools via estimating the benefits of solving problems faced by the test-generation tools. Chapter 6 introduces the approach that computes information flows to enable behavior-diagnosis cooperation for mobile privacy control. Chapter 7 describes the approach that extracts ACP rules and action steps from NL requirements documents to enable behavior-diagnosis cooperation for security policy extraction. Chapter 8 describes the approach that infers complexity models for methods and loops to enable behavior-diagnosis cooperation for performance analysis. Chapter 8 presents suggestions for future work. Finally, Chapter 10 concludes with a summary of the contributions and lessons learned.

CHAPTER 2

Background and Related Work

This chapter presents background information and discusses how our research relates to other research in cooperative testing and analysis, automated test generation, security analysis, and performance analysis. Section 2.1 describes existing research on cooperative testing and analysis and how our research is different from the existing research. Section 2.2 presents the background of structural test generation and discusses other research that relates to our research on improving structural test generation. Section 2.3 presents the background of mobile privacy control and access control policies, and discusses their related work. Section 2.4 describes existing research on performance analysis and how our research is different from the existing research.

2.1 Cooperative Testing and Analysis

Our framework of cooperative testing and analysis provides interfaces to support cooperation between software testing/analysis tools and their users. Indeed, providing interfaces to support such cooperation has been long investigated in the research community. However, unlike existing research that focuses on how to optimize the ways of representing and manipulating information [109, 191], our research focuses on what contents are displayed or manipulated. Also, our framework assumes that the subtasks in an SE task have been assigned to users and tools before our framework is applied on the SE task. Thus, our research is different from the research that focuses on allocation of tasks for users and tools [62, 193]. There exists other research that studies cooperation between humans for SE tasks [136]. While such research also

focuses on cooperation for SE tasks, our research focuses on the cooperation between users and tools for SE tasks, rather than the cooperation between users.

Our framework focuses on providing interfaces to support two types of cooperation: *problem-diagnosis cooperation* and *behavior-diagnosis cooperation*. We next present related work on these two types of cooperation.

2.1.1 Problem-Diagnosis Cooperation

Our framework of cooperative testing and analysis supports problem-diagnosis cooperation that users cooperate with software testing/analysis tools by addressing the problems faced by the tools. There exists some related work on leveraging user help to improve testing and analysis goals.

St-Amour et al. [177] propose an approach to report optimizations that the compiler could perform if the compiler has additional information, enabling users to provide further help for the compiler optimization in the source program. Dillig et al. [67] propose an approach to compute small and relevant queries that capture the facts that an analysis is missing when automated static analysis fails to verify a program. These queries are presented to users who decide whether the answers to queries are yes or no, and the answers are then used to either verify the program or prove the existence of a real faults. These approaches report additional information that causes the tools to perform poorly, similar to our problem-diagnosis cooperation. However, our problem-diagnosis cooperation further emphasizes the feedback loop between users and tools, where the process of problem fixing and problem identification is repeated until the effectiveness of the tools is satisfied or the users lose patience. Moreover, our EcoCov estimates the benefits and costs of addressing problems faced by test-generation tools, allowing users to focus on more important problems.

In this dissertation, we propose approaches that provide problem-diagnosis interfaces to support the problem-diagnosis cooperation for test-generation tools. We present the background information of automated test generation and related work in Section 2.2.

2.1.2 Behavior-Diagnosis Cooperation

Our framework of cooperative testing and analysis supports behavior-diagnosis cooperation that software testing/analysis tools cooperate with users by revealing and explaining software behaviors for the users to inspect.

Most of the related work of behavior diagnosis focuses on diagnosing software behaviors based on patterns or models to detect faults [65], performance anomalies [29] or vulnerabilities [192], while our behavior-diagnosis cooperation focuses on using tools to explain behaviors for improving users' understanding of the behaviors. There also exists research in explaining

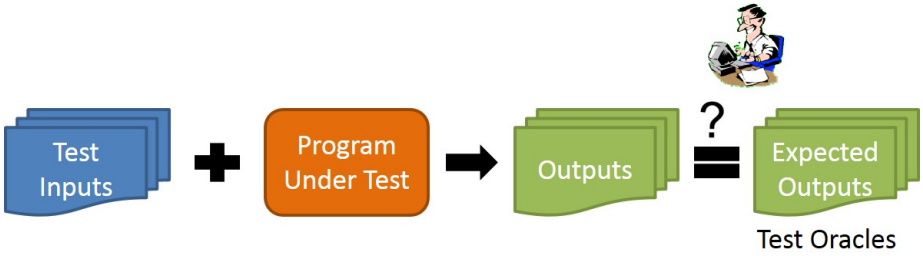


Figure 2.1: Overview of Developer Testing

software faults using visualization and record/replay techniques [128, 126, 127]. Unlike their explanations based on visualization and record/replay techniques, our research focuses on explaining software behaviors using extensions, instantiations, and abstractions based on whether the behaviors are represented as instances or models. There exists research that provides extensions for the models of software behaviors (such as using data flows to extend the permissions in iOS [71]), infers abstractions from the instances of software behaviors (such as inferring complexity models from executions [95, 213, 56]), and provides instances of the models of software behaviors (such as testing security policies [138, 111]). Such research typically asks the users to inspect the analysis results, while our behavior-diagnosis cooperation uses such analysis to explain the behaviors that the users are inspecting.

In this dissertation, we propose approaches that provide behavior-diagnosis interfaces to support the behavior-diagnosis cooperation for security analysis and performance analysis. We present the background information of security analysis and performance analysis and related work in Sections 2.3 and 2.4.

2.2 Automated Test Generation

In this section, we first introduce the background of software testing, and then describe the background of automated test generation.

2.2.1 Developer Testing

Software testing is one of the most widely used approaches for improving software quality in practice. A common practice of software testing is that an independent group of testers test the software under development after the functionality is developed, before the software is shipped to the customers. Besides testing done by testers, developer testing, where developers test their own code as they work on the code, has been widely recognized as a valuable practice of improving software quality [202]. Developer testing is usually in the form of *unit* testing, where developers

```
1  public class Queue {
2      private ArrayList data = new ArrayList();
3      public void enqueue(Object o) {
4          data.add(o);
5      }
6      public boolean empty() {
7          if(data.size() == 0) {
8              return true;
9          }
10         return false;
11     }
12     ...
13 }
14
15 @Test
16 public void testEnqueue() {
17     Queue s = new Queue();
18     s.enqueue(1);
19     Assert.assertEquals(1, s.size());
20 }
```

Figure 2.2: Example JUnit test case for the `Queue`

write and run unit tests for individual units of code (e.g., a class or a method) to ensure that the code behaves as expected. These unit tests are typically independent to each other, although recent work shows that some tests may have dependencies to global variables [216]. These unit tests help the developers to gain high confidence of the program unit under test and reduce the cost of fixing faults by detecting the faults early in the software development life cycle. Recent focus on test-driven development (TDD) [41], where tests are written before the program unit, further signifies the importance of developer testing. The popularity and benefits of developer testing have been well witnessed in industry.

Figure 2.1 shows the overview of developer testing. Typically, developer testing includes four major tasks: (1) generate test inputs, (2) create expected outputs (also referred to as test oracles [38]), (3) run test inputs, and (4) verify actual outputs. As shown in Figure 2.1, test inputs are executed on a program under test to produce outputs. These generated outputs are compared with expected outputs to check whether tests pass or fail. Among these four tasks, there exist many open source testing frameworks such as JUnit [90] (Java), NUnit [148] (.NET) and xUnit [208] (.NET), and CppUnit [77] (C++) for automating the last two tasks of running test inputs and verifying actual outputs. Figure 2.2 shows an example JUnit test case `testEnqueue` that tests the `enqueue` method of the `Queue` class. In this test case, Lines 17 and 18 represent test inputs, whereas Line 19 represents the expected output. The JUnit

testing framework helps automatically execute `testEnqueue` and verify that the return value of `s.size()` is the same as the expected value `1` (shown in Line 19).

Although these frameworks assist in reducing effort for the third and fourth tasks, developers still need to perform the first two tasks manually. Manual developer testing is known to be tedious and labor intensive. In addition, it is difficult for manual testing to exercise comprehensive behaviors of the program unit under test, and thus may not expose hidden faults of the program unit. For example, developers may miss certain test inputs (such as corner or special test inputs) that can expose faults in the program unit because of program complexity and limited resources allocated for the task.

In this dissertation, we focus on the problem of generating test inputs for achieving high structural coverage [30], i.e., structural test generation. Although achieving full code coverage does not guarantee that the code is fault-free, covering a part of code is necessary for exposing the faults in this part of the code. Without expected outputs, the generated test inputs can help detect robustness-related faults such as `null dereference` and `division by zero`. Although it is infeasible for developers to create expected outputs for the large number of generated test inputs, the developers can use specifications [140] to improve the effectiveness of generating test inputs and checking program behaviors. If specifications are not available (which may be difficult for the developers to write), code coverage criteria [224] such as statement coverage and block coverage can also be used to select a subset of generated test inputs for the developers to manually create the expected outputs. Next we present the background of structural test generation.

2.2.2 Structural Test Generation

Structural test-generation approaches aim to produce test inputs to achieve high structural coverage of the program under test, such as statement or branch coverage [30]. A major reason for achieving high structural coverage is that generating test inputs for covering a line of code is necessary for exposing the faults in this line of code. Next we describe two state-of-the-art structural test-generation approaches.

Random Test Generation. Random test generation is one of the most popular testing approaches. It is simple in concept, easy to implement, and effective in generating useful inputs, even if the specifications are incomplete and the source code is not available for analysis. Typically, random test generation treats the program under test as a black box and randomly chooses inputs from the input domain of the program. A straightforward technique for random test generation is to generate inputs by randomly choosing a value from the input domain according to the uniform distribution. There are also other advanced strategies for sampling inputs, e.g., adaptive random testing [53] and feedback-based techniques [152, 215, 217] for

object-oriented programs. Since random test-generation techniques do not analyze the program, random techniques have relatively low costs and generate test inputs efficiently. However, a common limitation of random techniques is that they often fail to generate inputs for some corner cases.

DSE-based Test Generation. Symbolic execution [125], a case of abstract interpretation [58], analyzes programs by tracking symbolic values instead of actual values. Symbolic execution associates symbolic values with program variables (such as program inputs and local variables), and statically simulates program execution to track these symbolic values. Along the simulated execution, symbolic execution collects constraints on symbolic values obtained from predicates in branch statements to form a *Path Condition*, which is a symbolic expression that can be used to reason about all program inputs that take the same path through the program.

Dynamic Symbolic Execution (DSE) [167, 92, 184] explores the feasible paths of the program under test and generates test inputs to exercise the paths. DSE initially executes the program under test symbolically with arbitrary or default inputs and collects path conditions as the constraints on inputs from the executed branch statements. DSE then systematically negates part of these constraints to form new path conditions, and leverages a constraint solver to solve these path conditions for obtaining new test inputs. These new test inputs steer the future explorations towards different paths of the program, iteratively collecting new constraints and achieving new structural coverage, such as statement and branch coverage [30]. With the advances of research on constraint solvers, e.g., Z3 [61] and CVC3 [40], DSE-based tools, such as Pex [184] and SAGE [93], show promising results in producing high-covering test inputs during unit or security/system testing [47, 48, 167, 184]. Pex has found serious faults in an already-well tested component of the .NET runtime, and SAGE has been continually running on more than 100 machines to detect security vulnerabilities since 2008.

2.2.3 Empirical Studies of Problems Faced by Structural Test Generation

This section presents the related work on empirical studies of structural test generation.

Studies on Test Generation. Lakhotia et al. [131] conduct an empirical study on applying test-generation tools CUTE [167] (a symbolic-execution-based tool) and AUSTIN [130] (a search-based tool) to achieve branch coverage of C programs. Fraser et al. [87] present a study of applying a search-based tool EvoSuite [86] on a set of open-source applications. They identify that dependencies on the environment inhibit high coverage achieved by test-generation tools. Kim et al. [123] propose a distributed concolic algorithm [124], and present an empirical study to show that their technique achieves several orders-of-magnitude increase in speed of test generation compared to concolic testing. All these studies focus on the coverage or scalability of testing the whole applications, while our work provides in-depth studies to identify major types

of problems faced by test-generation tools and presents an approach to precisely identify the problems.

2.2.4 Problem Identification for Structural Test Generation

This section presents related work (to our approach Covana) on identifying problems faced by software testing and analysis tools.

Explaining Failures of Program Analysis. Dincklage and Diwan [190] propose an analysis language and build a system to produce reasons when program analyses fail to produce desirable results. The objective of their approach is to express arbitrary data flow analyses using their analysis language and compute reasons for the failures. Zhang et al [219] propose an approach to infer documentation for explaining why tests fail and indicating the changes that can cause the tests to pass. Although Covana is remotely related to these approaches in terms of helping explain causes of residual structural coverage in the form of problems, Covana focuses on a quite different problem and includes significantly different techniques needed for addressing unique challenges in identifying problems that prevent test-generation tools from achieving high structural coverage.

Coverage Analysis. Pavlopoulou and Young [156] developed a residual coverage monitoring tool (for Java), which provides rich feedback from actual use of deployed software. Since their approach aims to reduce the performance overhead for gathering structural coverage from deployed software, their approach does not provide a way to analyze the coverage, while our approach analyzes the residual structural coverage gathered from DSE to filter out irrelevant problem candidates.

Problem Identification for Symbolic Execution. Anand et al. [33] propose type-dependence analysis, which performs a context- and field-sensitive interprocedural static analysis to identify the parts of the program under test that may be unsuitable for symbolic execution, such as third-party libraries. Their approach identifies external-method calls that are problematic in symbolic execution by carrying out static analysis to determine whether an external-method call receives symbolic values as arguments. To identify EMCPs, Covana considers not only data dependencies of arguments of external-method calls on program inputs, but also data dependencies of partially-covered branch statements on external-method calls for their return values.

2.2.5 Economical Analysis

This section presents related work (to our approach EcoCov) on economical analysis on solving problems faced by structural test generation.

Angelic Analysis. Angelic nondeterminism was first proposed by Floyd [84] to concisely express backtracking algorithms back in 1967. Recently, angelic nondeterminism has been successfully used in programming and debugging. Barman et al. [39] propose a programming methodology based on the refinement of angelic programs. They use the choose operator to represent the code that has not been implemented yet, and thus allow the execution of incomplete programs. Since the incomplete program is executable, programmers can run the program to better figure out how to make the program behave as expected. Chandra et al. [49] propose angelic debugging where they replace the value of a suspicious faulty expression with an angelic value that rescues the failing tests. If such angelic value exists and changing the expression may not break passing tests, then the expression is presented to developers as a repair candidate.

EcoCov is related to angelic programming and debugging in that we all replace the value of a code element with an angelic value. To compute angelic values, we all use constraint solvers to solve the collected constraints. However, the main difference is that the angelic values in angelic programming and debugging are values that correct the program executions (e.g., rescue the failing tests), while the angelic values in our diagnosis are values that make the problems encountered by the test-generation tools disappear. Hence, the constraints that we collect are different.

Economical Analysis. Horwitz [107] proposes an approach that aims to provide developers with the cost-benefit analysis of executing a not-covered component. Her approach computes the value for the must-execute-set metric by employing a control-dependence graph to compute the number of the not-covered parts (of the program under test) that will be executed if a predicate takes a value v. Unfortunately, such static-analysis approach ignores the capabilities of test-generation tools in generating various values for the not-covered code part that is dependent on B, producing imprecise estimation of the benefits. Moreover, such approach does not consider implicit branches introduced in object-oriented programs. For example, a member function call may throw an exception if the receiver object is a `null` object. Such extra control flows introduced by exceptions change the computation of the must-execute-set significantly.

Economic models are used to assess software engineering methodologies and predict their costs and benefits. Freimut et al. [88] propose a model to assess the cost effectiveness of inspection processes, and Do et al. [69] propose a model to assess the costs and benefits of regression testing processes. Besides economic models for software engineering methodologies, economic models are used for testing and analysis tools. Kumar et al. [129] propose an approach that models the cost of a fault, the cost of a tool, and the realized, potential value of a static analysis tool, and performs analysis on the model to evaluate the economic value of the tool. These proposed models assess the benefits and costs of methodologies or static analysis tools, while EcoCov focuses on the problems faced by test-generation tools.

2.3 Security Analysis

Security is one of the most important quality attributes of software. In this dissertation, our research focuses on mobile privacy control and extracting access control policies.

In recent years, the rapid growth of smartphones and mobile applications spur the occurrences of mobile threats of malware and potentially unwanted application (PUA) on the Android platform. Over the year of 2012, Android's mobile threat share has risen from 66.7% in 2011 to 79.0% in 2012 [76]. Thus, addressing the issues of mobile threats is very critical. Access control is one of the most fundamental and widely used privacy and security mechanisms. Access control is often governed by an Access Control Policy (ACP) [165] that includes a set of rules specifying which principals (such as users or processes) have access to which resources. Failing to correctly specify ACPs can result in serious consequences such as allowing an unauthorized user to access protected resources, and thus it is critical to ensure the correct specification of ACPs. Next we describe related work on both mobile privacy control and access control policies.

2.3.1 Mobile Privacy Control

This section presents the related work on mobile privacy control and information flow analysis.

User-Aware Application Capabilities. Mobile-device platforms such as Android and social-network platforms such as Facebook use manifests to show application capabilities and request permissions at install time. Other mobile-device platforms such as iOS and research approaches such as TaintDroid [72] report application capabilities the first time the applications try to access a resource. The capabilities shown in the manifests are either claimed by developers [164] or present only part of the requested application capabilities. Felt et al. [79] propose an approach that uses static analysis to map API calls used by applications to permissions. However, they adapt automated testing methodology to test the applications and identify APIs that require permissions, while our approach annotates the APIs with permissions and uses static checking.

Information Flow Analysis. Xie and Aiken [206] present an approach that statically computes summaries of blocks and procedures of PHP and detects security vulnerabilities at the block level, intraprocedural level, and interprocedural level. Their approach does not handle reference-type flows caused by references of containers and objects, and would lose track of flows after built-in procedure calls (e.g., `senses->take camera picture`) that cannot be analyzed by their approach. To address these problems, our approach uses mutable locations to simplify analysis of reference-type flows and tracks untampered- and tampered-classified information for classifying safe and unsafe flows.

The closest work related to our approach is PiOS [71], which studies private information leakage in actual iOS binaries. The PiOS approach statically computes data flow along control

flow paths from sources to sinks to determine whether there exists a user prompt along that path. PiOS emits warnings if such a flow is found without a user prompt. For the purposes of safe-guarding TouchDevelop users, the PiOS approach is insufficient because: (1) the PiOS analysis is not conservative; it misses flows that are too long or use indirect flow, (2) the prompts the PiOS identifies may be unrelated, show nothing of the leaked information, or show tampered information. PiOS also does not use the static information to control user prompting and privacy settings as our approach does.

Language-based information flow [163] allows developers to annotate variables with security attributes. These attributes can be used by compilers to enforce information flow controls. For example, Slam [37] shows that information flow labels can be applied to a simple language with reference types and Jif [145, 144] extends the Java language with statically-checked information flow annotation. Laminar [162] allows developers to specify security regions and provide information flow controls on both language and JVM/OS levels. Although these language-extending approaches are effective in guaranteeing information flow controls, they impose additional burdens on developers when writing applications; such burdens are undesirable for writing scripts on mobile devices in the context of TouchDevelop, especially for beginners.

Dynamic taint analysis [72, 223] has been applied to track information flows on both mobile platforms such as Android and desktop platform such as Windows. These approaches track tainted data during runtime, providing accurate runtime information about leaks. However, to reduce runtime overhead, these approaches usually ignore implicit flows raised by control structures. Moreover, it is impractical to dynamically execute all execution paths of these applications to detect potential information leaks. Such limitations make these approaches inappropriate for computing information flows for all submitted applications.

Access Granting. Mobile-device platforms such as Android and social-network platforms such as Facebook use manifests to request permissions at install time. Once permissions are given by users, the permissions cannot be changed. iOS and Windows User Account Control [142] prompt a dialog to request permissions from users when the applications try to access a resource or make security or privacy-related system-level changes. Instead of presenting only information about the access to resources, our approach presents information flows to describe what applications may do with private user information. Our access granting also provides a way for users to try out applications before using private information, and these settings can be changed at will.

Zhu et al. [223] propose an approach that uses dynamic taint analysis to track user data as it flows through applications. Their approach allows users to choose among logging the action, blocking the system call, or randomize the tainted data. Chen et al. [54] also propose an approach for shadowing data that the user wants to keep private and blocking network transmissions that contain data made available (by the user) to the application for only on-

device use. Our anonymized/real/abort setting is inspired by their approach, but we use static information flow analysis extended with tampered information to classify flows as safe/unsafe flows and provide default access settings, rather than runtime information.

Automated Security Validation of Mobile Applications. Gilbert et al. present a vision of making mobile applications more secure via automated validation [91]. They propose using commodity cloud infrastructure to emulate smartphones and run the submitted applications to dynamically track information flows and actions. Based on the information flow and action tracking, they propose to automatically detect malicious behavior and misuse of sensitive data via further analysis of dependency graphs [82] or natural language processing. Such an approach is akin to automated testing and suffers from the same problems, namely coverage. It is difficult to drive applications automatically into exercising all data and control paths. Thus, in the end, such an approach gives only a partial view of the behavior and does not safe-guard users.

2.3.2 Extraction of Access Control Policies

This section presents related work on extraction of ACPs.

Manual Extraction of ACPs from NL Documents. He and Anton [105] propose a manual approach, called Requirements-based Access Control Analysis and Policy Specification (ReCAPS), to extract ACPs from various NL documents, including requirements documents, design documents and database design, and security and privacy requirements. During the extraction, their approach also clarifies ambiguities in requirements documents and identifies inconsistencies among requirements documents and database design. Their objective is to derive a comprehensive set of ACP rules, similar to our approach. However, Text2Policy adapts NLP techniques and provides new analysis techniques to automate the process of ACP extraction, while their approach is manual.

Template Matching. Etzioni et al. [75] propose an approach to extract lists of named entities found on the web using a set of patterns. Their approach is related to the ACP extraction of our approach, since both use patterns to extract information. However, their patterns are based on the low-level POS tags (such as NP and NPList), while our semantic patterns are based on grammatical functions of phases (such as subject, main verb group, and object). Text2Policy employs semantic patterns that are more general and provide high precision in identifying ACP sentences as shown in our evaluations.

NLP to Analyze API Documents. Pandita et al. [154] propose an approach that analyzes the meta-data of API descriptions, programming keywords, and semantic patterns from POS tags to infer method specifications from API documents. Zhong et al. [220] propose an approach that builds action-resource pairs from API documents via NLP analysis based on ma-

chine learning, and infers automata for resources from action-resource pairs and class/interface hierarchies. Both of these approaches focus on parsing API documents, and use the specific characteristics of API documents to improve the NLP analysis. For example, different parts of API documents can be mapped to different parts of code structures, such as class/method names, return values, and parameter names. However, the contents of requirements documents usually cannot be mapped directly to code structures, thus making their approaches inappropriate on analyzing requirements documents.

NLP to Assist Privacy-Policy Authoring. The SPARCLE Policy Workbench [120, 121, 45, 46] employs the shallow-parsing technique [147] to parse privacy rules and extract the elements of privacy rules based on a pre-defined syntax. These elements are then used to form policies in a structured form, so that policy authors can review it and then produce policies in a machine-readable form, such as EPAL [34] and XACML [9, 150] with a privacy-policy profile. Michael et al. [141] propose an approach to map NL policy statements to an equivalent computational format suitable for further processing by a policy workbench. However, neither of these approaches can identify sentences describing a policy rule. These approaches parse all the input statements for policy extraction by assuming that the input statements are policy statements, while our approach identifies ACP sentences from requirements documents using semantic patterns. Both of these approaches provide simple templates to extract elements for constructing policy rules, while our approach provides more general semantic patterns. Additionally, their approaches cannot infer negative meaning of sentences.

Use-Case Analysis. Sinha et al. [170, 169] adapt NLP techniques to parse and represent use-case contents in use-case models. The extraction of use-case contents to formal models is similar to the action-step extraction in our approach. However, our approach focuses on extracting access requests for validation against the specified and extracted ACPs. Moreover, we provide corresponding analysis techniques to infer human actors from previous sentences.

2.4 Performance Analysis

Performance problems exist widely in released software [118, 89]. As a type of widespread performance problems, software hangs cause unresponsiveness of software applications [194, 172]. A recent study of hang problems [172] shows that 27.04% of the 233 studied hang faults are caused by time-consuming operations in responsive actions[1], such as expensive computations in the UI thread for GUI applications. Among the expensive operations that cause hang problems, some of these operations are constantly expensive (such as server initializations), whereas some of them depend on the input workloads. These problems are referred to as workload-dependent performance bottlenecks (WDPBs). WDPBs are usually caused by workload-dependent loops

[1]Actions that are expected to return instantly.

(referred to as WDPB loops)[2] that contain certain relatively expensive operations, such as temporary-object creation/destruction [207], file I/O, and UI updates. We next present related work on performance analysis, which is related to our ΔInfer approach.

Model Inference using Multi-Profiles. Goldsmith et al. [95] propose an approach that fits performance measurements of clusters of basic blocks to workload sizes. Zaparanuks et al. [213] propose an approach that infers an empirical cost function of an application automatically, and Coppa et al. [56] propose an approach that measures the size of the input given to a generic code fragment. Unlike their model inference based on sorted or random inputs, our approach iteratively refines the inferred models based on the model accuracy from the previous iteration. Moreover, our approach uses context-sensitive analysis to address complex contexts in GUI applications. Westermann et al. [195] propose an approach to infer the prediction models between interdependent, performance-relevant configuration parameters and the performance metric of interest. All these approaches infer a single complexity model for a program, while our approach infers context-sensitive complexity models for locations inside a program, and identifies workload-dependent loops using complexity transitions.

Static Analysis. Chang et al. [50] propose an approach that combines taint analysis with control dependency analysis to detect high-complexity control structures, such as recursive calls and nested loops. Wang et al. [194] propose an approach that statically searches for the intersections of blocking and responsive invocations as the potential hang faults based on patterns around method invocations. Approaches of purely static analysis face challenges in identifying implicit loops or resolving complex contexts in GUI applications, and in handling many runtime features, such as indirect method calls via function pointers.

Performance Analysis. Traditional performance analysis centers around analyzing the performance measurements obtained by profiling program executions, such as call-tree and call graph profiles [31]. There also exist approaches that assist searching [32] and summarizing [174] profiles to find performance problems. These approaches rely on manual efforts to explore traces or search to identify bottlenecks, while our approach infers WDPBs from multiple profiles.

Foo et al. [85] and Jiang et al. [117] propose approaches to learn signatures or baselines from previous runs, and then detect performance problems by comparing current runs against the derived performance signatures or baselines. Grechanik et al. [97] propose an approach that automatically clusters the input space into good and bad performance test cases, and drill down to the most significant methods to identify performance bottlenecks. Zhang et al. [214] propose a symbolic-execution-based approach to generate load tests for exposing performance bottlenecks. Han et al. [102] mines large-scale traces to identify performance faults. All these approaches require WDPBs to surface on the analyzed executions, suffering from the insufficiency issue of identifying WDPBs. Nistor et al. [149] propose an approach to detect performance problems via

[2]A loop whose iteration count depends on the input workload.

identifying loops whose computation has similar memory-access patterns across loop iterations. Their approach relies on loop events for analysis, and thus cannot identify implicit loops without modelling UI libraries calls.

2.5 Summary

This chapter has laid out the background for the research developed in this dissertation and discussed how our research is related to other previous research in automated test generation, security analysis, and performance analysis.

In particular, our research on automated test generation focuses on studying problems faced by test-generation tools and supporting the problem-diagnosis cooperation between developers and test-generation tools. Thus, it is related to previous studies of test-generation tools and the problems faced by these tools. However, our studies focus on identifying problems that cause low coverage, while their studies focus on how high covereage the tools can achieve. Our Covana approach focuses on reporting problems to enable the cooperation between developers and test-generation tools, and thus it is related to other cooperative analysis approaches that bring human helps to improve tool effectiveness. It is also related to the work on analyzing the residual coverage and explaining failures of program-analysis tools. While their approaches focus on explaining the failures of the tools, our Covana approach focuses on using dependency analysis to prune irrelevant problems, improving the precision on identifying OCPs and EMCPs faced by the test-generation tools. Our EcoCov approach is related to other economical analysis approaches on program-analysis tools. Unlike their approaches that build models to predict cost, our EcoCov uses angelic values to simulate the effect of solving the problems, producing more accurate estimation.

Our research on security analysis focuses on mobile privacy control and security policy extraction. Our research on mobile privacy control improves privacy control on mobile applications, and thus is related to other work on detecting information leaks. Different from detecting information leaks, our work explains how permissions are used by mobile applications, enabling users to make better decisions on controlling their privacy. Our research on extracting security policies develops an approach that adapts NLP techniques to extract ACP rules from NL requirements documents such as use cases. Thus, our work is related to other research on extraction of ACP rules. However, rather than employing manual approaches or semi-automated approaches, our approach is fully automatic. Moreover, by adapting NLP techniques, our approach does not require the requirements documents to be written in controlled NL.

Our research on performance analysis focuses on identifying WDPBs. Existing work on performance analysis mostly focuses on detecting expensive methods, and thus misses WDPBs that do not surface on small workloads. Our approach addresses such challenge by inferring

complexity models from multiple profiles on multiple workloads to predict performance. Existing approaches on inferring complexity models from multiple profiles are mostly context insensitive, while our approach is context sensitive. Moreover, our approach proposes a novel algorithm to identify workload-dependent loops. Such loops are shown to cause many performance problems in our evaluations.

CHAPTER 3

Cooperative Testing and Analysis

3.1 Introduction

We next describe the framework of *cooperative testing and analysis* that provides interfaces to support cooperation between software testing/analysis tools and their users. With the advances on software testing/analysis tools, many new types of cooperation emerge for various SE tasks. To support such new types of cooperation enabled by the advances on software testing/analysis tools, there is a strong need to provide a user-tool interface that communicates sufficient and precise information to the users (e.g., reporting the problems faced by the tools), allowing the users to provide help to the tools for address the problems. These user-tool interfaces are the places where the users interact with the tools. Unlike user interfaces in HCI research [109], we do not focus on the representation of information or ways for the users to manipulate the information in these interfaces, but focus on the contents being displayed or manipulated.

The basic rationale of cooperative testing and analysis is that tools and users typically have their respective strengths and weaknesses. For example, the tools are good at automating mechanic and repetitive tasks that have well-defined goals, such as generating a random number or searching an array of numbers for a specific number. Although the users are not good at such tasks, the users are good at tasks that are dependent on domain knowledge and user expectations, such as addressing problems faced by the tools (e.g., providing mock objects based on the users' domain knowledge about the external libraries) and confirming whether a software behavior (e.g., sending text SMS) is expected for an application (e.g., a navigation mobile application). Thus, creating interfaces that enable the users and the tools to do subtasks

that they are good at provides opportunities to improve the effectiveness of various SE tasks. For an SE task on which our framework is applied, we assume user subtasks and tool subtasks are already identified. Optimizing the allocation of user subtasks and tool subtasks for an SE task [62] is out of the scope of this dissertation.

Our framework focuses on providing interfaces to support two types of cooperation with software testing/analysis tools for SE tasks: (1) **Problem-Diagnosis Interface**: for certain SE tasks where automated tools drive the tasks towards a goal of testing and analysis, our framework provides interfaces that precisely report the problems faced by the tools and the benefits of solving these programs. Such interfaces enable users to cooperate with the tools by addressing problems faced the tools. (2) **Behavior-Diagnosis Interface**: for certain SE tasks where users inspect software behaviors to determine whether the behaviors are expected for achieving a goal of testing and analysis, our framework provides interfaces that show a list of behaviors related to the goal for the users to inspect, and present additional information to explain the behaviors. Such interfaces enable the tools to cooperate with the users by analyzing and explaining the software behaviors. We next present the details of the two types of cooperation supported by our framework.

3.2 Problem-Diagnosis Cooperation

In certain SE tasks, tools automatically carry out a task to accomplish a goal of testing and analysis. For example, structural test-generation tools automatically generate test inputs that aim to achieve high structural coverage of the program under test. As we explained in Section 1.2, these tools face various problems (such as dealing with external libraries) when dealing with complex object-oriented programs. To improve tool effectiveness, users, such as developers and testers, provide their guidance to help the tools address these problems. For example, the users can replace external libraries with mock objects that simulate the behaviors of the external libraries, and reapply test-generation tools to generate more test inputs. With the provided mock objects, the tools can generate desired values for the method calls to the external libraries, and achieve higher coverage to carry out software testing more effectively.

In such *cooperation*, the advances on test-generation tools free the users from labor-intensive and tedious tasks (e.g., manually producing test inputs for coverage) and the tools try their best in automating the tasks, while the users focus their efforts in addressing only the problems faced by the tools (e.g., providing mock objects). Note that the efforts in addressing the problems are typically less than the efforts in asking the users to finish all the remaining work for the tasks (e.g., generating test inputs for all the not-covered code). For example, with the provided mock objects, the tools can be reapplied to explore the not-covered code and generate more test inputs for covering the newly explored code. In this way, the users do not need to generate

test inputs for all the newly explored code. If more problems are encountered in exploring the newly explored code, the users can provide their help again.

However, such cooperation is ineffective if the tools do not communicate sufficient information to the users. For example, most test-generation tools often report only the achieved coverage, but not the problems faced by the tools; for a few tools that report the problems, such as Pex [184], many reported problems are false warnings [204]. In other words, existing tools report only the observed *symptoms* (i.e., not-covered branches) to the users. With only the coverage information, the users have to reason about the possible *causes* (i.e., the problems that cause the branches not to be covered) to the *symptoms* before they can provide their guidance. Thus, the key insight to improve such cooperation is that we need a user-tool interface to show problems that prevent the tools from accomplishing the goal. Also, the users often have limited time in addressing the problems, and the users would like to focus on the more important problems based on the estimated benefits of solving the problems. The information presented by the interface improves the users' understanding of the problems faced by the tools, and thereby helping the users make informed decisions on giving help. With this insight, our framework includes problem-diagnosis interfaces that allow the users to cooperate with the software testing/analysis tools by addressing the problems faced by the tools.

The cooperation supported by problem-diagnosis interfaces consists of three phases:

1. **Setup Phase**: first set up and apply a tool to conduct initial testing and analysis on a program.

2. **Feedback Phase**: the provided interface then shows the feedback to the users, including the effectiveness of the tool and the problems faced by the tool; these problems are precisely identified and the benefits of solving these programs are accurately estimated using analyses of the program and the tool.

3. **Action Phase**: the users provide guidance to the tool based on the feedback, helping the tool address the problems.

Such process forms a feedback loop that enables the users and the tools to refine and accomplish testing and analysis goals for various SE tasks. Such cooperation improves the users' understanding of the problems faced by the tools, enabling the users to provide guidance to the tools more effectively.

In this dissertation, we propose approaches that provide problem-diagnosis interfaces for better supporting cooperation in test-generation tools, described in detail in Chapters 4 and 5.

3.3 Behavior-Diagnosis Cooperation

In certain SE tasks, users inspect software behaviors to determine whether the behaviors are expected for achieving a goal of testing and analysis. For such SE tasks, behavior expectation lies in the mind of the users, and the tools need the users to make decisions. For example, the users may expect a social navigation mobile application to share their contacts (accessing and then sending out their contacts), while it is difficult for the tools to infer such exception if most navigation applications do not share contacts. However, it is difficult for the users to observe such software behaviors without help of the tools, typically requiring the users to install the applications and dynamically explore the applications. Thus, to better accomplish such type of tasks, the tools can help the users by revealing a list of software behaviors for the users to inspect, and the users confirm whether such behaviors are expected. For example, when a mobile application is installed, the privacy-control approach employed by Android presents a list of requested permissions for a mobile application and asks the users to determine whether the requested permissions are expected for the application.

However, it is shown that presenting permissions in installation time has limited successes since the tools do not communicate sufficient information to explain the permission uses [80, 189, 72]. As we explained in Section 1.2, such approach shows *what* permissions are used by the applications, but do not explain *how* the applications will use the permissions. Such limited information causes the users to make uninformed decisions on these permissions. Thus, the key insight to improve such cooperation is that we need a user-tool interface to explain the software behaviors, improving the users' understanding of the software behaviors and thereby assisting the users in making better decisions. For example, the tools can explain how the applications use the permissions by showing what types of private information protected by the permissions flow to what output channels. Such explanations provide a larger scope of information for the permission uses, improving the users' understanding of the applications' privacy-sensitive behaviors. With this insight, our framework includes behavior-diagnosis interfaces that allow the tools to cooperate with the users by analyzing and explaining the software behaviors.

The cooperation supported by behavior-diagnosis interfaces consists of three phases:

1. **Setup Phase**: users first set up and apply a tool to conduct initial testing and analysis on a program.

2. **Execute Phase**: the provided interface then shows a list of behaviors related to the goal for the users to inspect and make decisions.

3. **Explain Phase**: based on whether the behaviors are in the form of either concrete instances or abstract models, these explanations can be in the form of (1) *extension*: a larger scope of information to collect more concrete instances of the behaviors or extend

the models of the behaviors, (2) *instantiation*: instantiations of the behaviors' abstract models, or (3) *abstraction*: inferred models from the concrete instances of the behaviors. Such explanations help the users make informed decisions on the behaviors.

The explanation phase offers extra information that aims to improve the users' understanding of software's behaviors, enabling the users to make informed decisions in carrying out an SE task. Consider the example of mobile privacy control for controlling the users' privacy-sensitive information. Existing approaches often show what privacy-sensitive information is requested by mobile applications, i.e., the model of such behaviors shows the *read* of the users' privacy-sensitive information. By using information flows to explain how privacy-sensitive information will be used by the mobile applications, we extend the model to show the *use* of the users' privacy-sensitive information. When the behaviors for the users to inspect are in the form of models, instantiating the models in the software under analysis may help the users better understand the behaviors. For example, simply showing invariants inferred from the executions of a program [74] may be difficult for the users to confirm whether the invariants are expected. By applying the invariants on some other programs to detect violations of the invariants, i.e., instantiating the invariants on some other programs, the users can better understand whether such invariants are expected and refine the invariants if needed. Also, when the behaviors for the users to inspect are in the form of concrete instances, such as executions of a program, inferring models from the instances may help the users better understand the behaviors. For example, it is difficult for the users to know whether an expensive method is a performance fault given the cost of the method. By inferring the complexity model from the executions, it is easier for the users to understand the cost growth, and thus help the users make decisions on whether such costs are expected for the method.

The key difference between problem-diagnosis cooperation and behavior-diagnosis cooperation is that problem-diagnosis cooperation typically engages the users to diagnose some intermediate results used for producing the final results, while behavior-diagnosis cooperation typically engages the users to diagnose the final results directly. For example, after a test-generation tool generates test inputs for a program, problem-diagnosis cooperation engages the users to diagnose the reported problems and address the problem for helping the tool in achieving higher coverage. If the tool generates an input that causes the program to throw uncaught exceptions, behavior-diagnosis cooperation engages the users to diagnose the exceptions and confirm whether such exceptions indicate real faults. Also, the information of how the exception-throwing states are reached in the failing executions is shown to the users (i.e., a larger scope of information is used to explain the exceptions), helping the users make informed decisions in diagnosing exceptions.

In this dissertation, we propose approaches that provide behavior-diagnosis interfaces for better supporting cooperation in security analysis and performance analysis, described in detail in Chapters 6, 7, and 8.

3.4 Summary

In this chapter, we presented a framework, called *cooperative testing and analysis*, that provides interfaces to support cooperation between software testing/analysis tools and their users. Our framework focuses on providing interfaces to support two types of cooperation: problem-diagnosis cooperation where the users cooperate with the tools by addressing problems faced by the tools and behavior-diagnosis cooperation where the tools cooperate with the users by analyzing and explaining software behaviors. Both types of cooperation consist of three phases, and these phases are general for various SE tasks. But the approaches that provide interfaces for the cooperation need to be customized based on the artifacts and the goals of the SE tasks under analysis.

CHAPTER 4

Problem-Diagnosis Cooperation for Identifying Problems Faced by Structural Test Generation

4.1 Introduction

Structural test-generation tools automatically producing test inputs that aim to achieve high structural coverage, such as test-generation tools based on DSE [167, 93, 92, 125]. Although these automated test-generation tools can easily achieve high structural coverage for simple programs, these tools face problems in generating test inputs to achieve high structural coverage when they are applied on complex programs in practice.

To better understand how automated test-generation tools perform for complex programs, we carried out a preliminary study of applying Pex [184], a DSE tool, on four popular open-source object-oriented projects, which have high download counts (study details are described in Section 2). The results show that the total block coverage achieved is 49.87%, with the lowest coverage being 15.54%. Among the problems that we empirically observed, many statements or branches are not covered due to two major types of problems: (1) the external-method-call problem (EMCP), where method calls to external libraries[1] throw exceptions to abort test executions, or their return values are used to decide subsequent branches, causing the branches not to be covered; (2) the object-creation problem (OCP), where tools fail to generate sequences of method calls to construct desired object states for non-primitive method arguments or receiver objects to cover certain branches.

[1]External libraries include native system libraries, such as file system and network socket libraries, and third-party pre-compiled libraries, where source code is not available.

Since these automated tools could not be powerful enough to deal with various complicated situations in real-world code bases automatically without human intervention or guidance, we instantiate the framework of cooperative testing and analysis as a concrete approach to improve automated test generation, where tools and developers cooperate to effectively carry out software testing as follows. Automated test-generation tools are first applied to generate test inputs and achieve coverage without human guidance. After the tools reach the pre-defined limits of resource consumption, the tools stop and report the achieved coverage and the problems that prevent them from achieving higher structural coverage back to developers, such as which external-method call causes branches not to be covered or the state of which object is required to cover certain branches. By looking into the reported problems, the developers provide corresponding guidance to help the tools address the problems. For example, to deal with OCPs, developers can specify factory classes [184] that encode desired method sequences for non-primitive object types. To deal with EMCPs, developers can instruct the tools to instrument the external-method calls or provide mock objects [188] to simulate irrelevant environment dependencies. With the provided guidance, the tools are reapplied to generate test inputs for achieving higher structural coverage.

Besides the top two types of problems (OCPs and EMCPs) encountered during our preliminary study, boundary problems caused by loops are also an important type of problems that prevent test-generation tools from achieving high structural coverage. As a special type of branches, loops can cause the number of paths to be explored to grow exponentially. Even worse, the number of paths becomes infinite due to the presence of input-dependent loops (IDLs)[2], causing DSE to run out of resources (e.g., the allocated time or number of explored paths) before achieving satisfactory coverage [198, 200]. For example, a recent study [131] shows that DSE may keep unfolding an IDL without achieving coverage of any new branches. Thus, even if the users provide guidance to address all OCPs and EMCPs, test-generation tools still may not achieve high coverage if loop problems are ignored. The main reason why OCPs and EMCPs account for most of the problems in our preliminary study is that these object-oriented programs typically employ validation at the beginning of methods, and satisfying these validations requires solving the corresponding OCPs and EMCPs first.

In this dissertation, we focus on the top two major types of problems, OCPs and EMCPs. Without solutions to the OCPs and EMCPs, the body of the methods that contain loops cannot be explored by DSE, and thereby we cannot detect loop problems for these methods.

[2]Input-dependent loops are loops whose iteration numbers depend on some unbounded input.

4.2 Preliminary Study for Problems Faced by Structural Test Generation

This section presents the preliminary study that aims to identify problems faced by test-generation tools when dealing with complex object-oriented programs. Our studies focus on test-generation tools based on DSE. With the advances of research on constraint solvers, e.g., Z3 [61] and CVC3 [40], DSE-based tools, such as SAGE [93] and Pex [184], become promising in generating test inputs for unit testing and (security) system testing [184, 48, 47, 167, 93]. Since 2008, SAGE has been continually running on more than 100 machines in a security testing lab, and Pex found serious faults from an already-well-tested component of the .NET runtime. In our studies, we use Pex [184][3] as the DSE tool to generate test inputs.

In this section, we discuss the different types of problems that we empirically observed by applying a state-of-the-art DSE tool, Pex [184], on four open source projects for achieving structural coverage. The analysis of these problems helps motivate our approach. We choose Pex as the DSE tool in our empirical study and later we implement our approach upon it for two reasons: (1) Pex can explore all public methods of any real-world .NET code bases and generate test inputs automatically; (2) Pex has been applied internally in Microsoft to test core components of the .NET runtime infrastructure and found serious faults [184].

We apply Pex on the core libraries of the four open source projects until all the methods have been explored by Pex or Pex runs out of memory and cannot continue to generate test inputs. These four open sources projects are SvnBridge [179], xUnit [208], Math.NET [139], and QuickGraph [158], which are quite popular and have high download counts. After Pex generates test inputs and produces coverage files, we select 10 source files that achieve low coverage in each project, and manually investigate the problems that contribute to the not-covered statements and branches. The details of the subjects and results can be found in our project web[4]

Table 4.1 shows the distribution of the problems that prevent DSE from achieving high structural coverage. Column "Project" lists the name of each project, Column "LOC" shows the number of lines of codes for each project, and Column "Cov %" shows the block coverage achieved by Pex. The other four columns give the number and the percentage of the not-covered branches caused by different types of problems.

The top major type of problems is object-creation problems (64.79%), shown in Column "OCP", since desired object states cannot be generated. In unit testing of object-oriented code, achieving high structural coverage requires desired object states for the receiver or non-primitive arguments of the method under test (MUT). These desired object states help cover various branches. However, automated approaches are often ineffective in generating method-

[3]Pex is a state-of-the-art test-generation tool developed by Microsoft Research.
[4]http://research.csc.ncsu.edu/ase/projects/covana/

Table 4.1: Main problems for not-covered branches in 10 files from core libraries of four open source projects

Project	**LOC**	**Cov %**	**OCP**	**EMCP**	**Boundary**	**Limitation**
SvnBridge	17.1K	56.26	11 (42.31%)	15 (57.69%)	0 (0%)	0 (0%)
xUnit	11.4K	15.54	8 (72.73%)	3 (27.27%)	0 (0%)	0 (0%)
Math.Net	3.5K	62.84	17 (70.83%)	1 (4.17%)	4 (16.67%)	2 (8.33%)
QuickGraph	8.3K	53.21	10 (100%)	0 (0%)	0 (0%)	0 (0%)
Total	40.3K	49.87	46 (64.79%)	19 (26.76%)	4 (5.63%)	2 (2.82%)

call sequences that produce desired object states to achieve high structural coverage [183], facing object-creation problems that prevent DSE from achieving high structural coverage.

The second major type of problems is external-method-call problems (26.76%), shown in Column "EMCP", since external-method calls cause to lose track of computed symbolic values passed as their arguments or throw exceptions to hinder the exploration. In our study, we encountered many external-method calls, 405 in 40 files, but only 4.7% (19 in 405) are causes for DSE not to achieve high structural coverage. If we simply report every encountered external-method call as an EMCP, we can get many irrelevant problems that are not cause for any not-covered statment or branch.

The third main type of problems is boundary problems (5.63%), shown in Column "Boundary", mostly caused by loops in the program under test. Some programs under test have loops whose number of iterations depends on symbolic values, and DSE keeps increasing the number of iterations of the loops during path exploration, preventing DSE from exploring other paths in the remaining parts of the program.

The last main type of problems is limitations of the used constraint solver (2.82%), shown in Column "Limitation", since the used constraint solver cannot compute exact solutions to floating-point arithmetics. The reason why we did not have so many not-covered branches due to this type of problems is that the used constraint solver generates approximate integers for constraints that contain floating-point arithmetics and these approximate integers can cover certain branches.

Besides our preliminary study, other recent studies [87, 116] also identify OCPs and EMCPs as two major types of problems in compromising the coverage achieved by test-generation tools.

4.3 Precise Problem Identification for Structural Test Generation

Based on the study described in the preceding section, structural tools could not be powerful enough to deal with various complicated situations in real-world code bases automatically without human intervention or guidance. To improve the effectiveness of the test-generation tools, we propose to use the problem-diagnosis cooperation for the test-generation tools. In this cooperation, tools automatically generate test inputs for achieving coverage of a program and report problems faced the tools for not-covered code. Based on users' domain knowledge of the program under test, users provide factory methods for addressing OCPs, and provide mock objects for addressing EMCPs.

To achieve this problem-diagnosis cooperation for structural test-generation tools, the tools need to report the encountered problems and narrow down the investigation scope, thus reducing the required efforts from the developers. Given the generated test inputs from tools and the achieved coverage, it is not difficult to identify the problem candidates for the developers to analyze. For example, locating all the external-method calls in the program under test can be easily achieved by static or dynamic program analysis, and reporting the object types of the program inputs and all their fields to the developers is fairly easy as well. However, the number of such problem candidates could be high, and quite some of these problem candidates (referred to as irrelevant problem candidates) are not causes for the tools not to achieve higher structural coverage. For example, the external-method call `Console.WriteLine` prints only the argument string value. Therefore, instrumenting or mocking this method call cannot result in any increase in coverage. Similarly, some branches may require only the specific object state of a field of the program inputs, and thus there is no need to spend efforts in providing sequences of method calls for all the fields of the program inputs.

Since the number of problem candidates could be large, the tools need to prune irrelevant problem candidates for reducing the efforts of developers in terms of investigation scope. Simple pruning techniques, such as pruning external-method calls by method names or belonging libraries, can result in high false positives and false negatives. In our preliminary study, we observed that if branch statements are data dependent on external-method calls for their return values (i.e., return values are used to decide which branches of the statements to take), the branches of these branch statements are very likely not covered by the generated test inputs, since automated test-generation tools normally cannot instrument or analyze the external-method calls. Hence, we can use compute data dependencies of branch statements containing not-covered branches (referred to as partially-covered branch statements) on external-method calls for their return values, and use the computed data dependencies to effectively identify such external-method calls and prune irrelevant ones. Similarly, we can compute data depen-

dencies of partially-covered branch statements on program inputs and their fields, and use the computed data dependencies to help identify which fields of the program inputs require desired object states to cover certain not-covered branches.

To address the need of precisely identifying problems for developers to provide guidance, we propose a novel approach called Covana [204, 198] that precisely identifies the problems that prevent the tools from achieving high structural coverage and prunes the irrelevant problem candidates using the data dependencies of partially-covered branch statements on problem candidates. Covana consists of three main steps: (1) identify problem candidates based on the types of problems, (2) assign symbolic values to elements of the problem candidates (including return values of external-method calls or program inputs as well as their fields) and perform forward symbolic execution [125] using test inputs generated by the tools as program inputs, (3) compute data dependencies of partially-covered branch statements on program candidates, and prune the candidates that none of partially-covered branch statements have data dependencies on. Since EMCPs and OCPs are the two major types of the problems observed in our preliminary study, we provide two specific techniques to instantiate Covana for identifying these two types of problems. Based on Covana, we provide a problem-diagnosis interface that shows the problems faced by structural test-generation tools, enabling the problem-diagnosis cooperation on test-generation tools.

To show the effectiveness of Covana, we use Dynamic Symbolic Execution (DSE) [167, 92, 184] as an illustrative example of automated structural-test-generation approaches. The primary reason why we choose DSE is that DSE is the most recent state-of-the-art in test generation. We concretize Covana as an extensible framework that collects information from DSE to identify different types of problem candidates and perform forward symbolic execution to compute data dependencies of partially-covered branch statements on problem candidates.

4.3.1 Example

We next explain how Covana, instantiated with two specific techniques, identifies EMCPs and OCPs with two illustrative examples.

External-Method-Call Problem (EMCP)

During the execution of the automatically generated test inputs, external-method calls may prevent the generated test inputs from achieving high structural coverage if the return values of external-method calls are used to decide subsequent branches to take or throw exceptions to terminate test executions. As a real example, the return value of `File.Exists` in Figure 4.1 is used in deciding which branch at Line 3 to take. If the generated test inputs do not contain a file name that exists in the test environment, being mostly the case, the statement

```
1  static string GetDefaultConfigFile(string assemblyFile) {
2      string configFilename = assemblyFile + ".config";
3      if (File.Exists(configFilename))
4          return configFilename;
5      return null;
6  }
7  ...
8  public ExecutorWrapper(string assemblyFilename, ...) {
9      ...
10     assemblyFilename = Path.GetFullPath(assemblyFilename);
11     ...
12 }
13 public AssertActualExpectedException(object expected, object actual, ...) {
14     ...
15     this.actual += String.Format("({0})", actual.GetType().FullName);
16     this.expected += String.Format("({0})", expected.GetType().FullName);
17     ...
18 }
```

Figure 4.1: Three simplified methods from xUnit [208].

at Line 4 cannot be covered by the test inputs. The method `Path.GetFullPath` at Line 10 is another example external-method call that prevents test inputs from achieving higher structural coverage. `Path.GetFullPath` throws exceptions when an invalid of an assembly file is given as the argument. Therefore, if none of the generated inputs includes a valid name, the lines after Line 10 remain not-covered. However, not all the external-method calls can cause problems for achieving high structural coverage. For instance, `String.Format` at Line 15 and 16 in Figure 4.1 formats only the string value of the input and does not affect the coverage achieved by the generated test inputs.

Covana first identifies as problem candidates the external-method calls whose arguments have data dependencies on program inputs. In Figure 4.1, Covana identifies `File.Exists` at Line 3, `Path.GetFullPath` at Line 6, and `String.Format` at Lines 15 and 16 as candidates, since they all have data dependencies on the program inputs. By assigning symbolic values to return values of the candidates and applying forward symbolic execution [92, 167], Covana collects symbolic expressions in the predicates of branch statements. From the symbolic expressions collected from branch statements, Covana extracts elements of problem candidates and considers the branch statements that have data dependencies on problem candidates. If one of the branches at Line 3 is not covered (i.e., Line 3 is a partially-covered branch statement), Covana identifies `FileExists` as an EMCP. On the other hand, there are no partially-covered branch statements that are data dependent on the return values of `String.Format` at Lines 15 and 16, causing `String.Format` at Lines 15 and 16 to be pruned. For `Path.GetFullPath`, if all of its executions throw exceptions, the remaining part of the program, starting at Line 11, remains not-covered. Covana detects

```
1  public class FixedSizeStack {
2      private Stack stack;
3      public FixedSizeStack(Stack stack) {
4          this.stack = stack;
5      }
6      public void Push(object item) {
7          if(stack.Count() == 10) {
8              throw new Exception("full");
9          }
10         stack.Push(item);
11     }
12     ...
13 }
14
15 public void TestPush(FixedSizeStack stack, object item){
16     stack.Push(item);
17 }
```

Figure 4.2: FixedSizeStack implemented using Stack

the exceptions that cause to abort the test executions and identifies `Path.GetFullPath` as an EMCP that causes the area starting at Line 11 not to be covered.

Object-Creation Problem (OCP)

Figure 4.2 shows a class `FixedSizeStack` that has a field `stack` of type `Stack`. Invoking the method `Push` to push objects is required for increasing the size. `Stack` has a field `items` that stores the pushed objects, and the method `stack.Count()` returns the number of objects stored in the `Stack.items`[5]. `FixedSizeStack` has an upper bound of the number of objects that can be pushed into the stack. To bound the size, the method `FixedSizeStack.Push` throws an exception when the size of the stack has reached the bound (10 in the example). The method `TestPush` receives a `FixedSizeStack` object and an object to be pushed as its arguments and invokes the method `FixedSizeStack.Push` to push the object to the stack for testing. To cover the true branch at Line 7 of the method `FixedSizeStack.Push`, the generated test inputs need to include method-call sequences to create a full `FixedSizeStack` whose size is 10.

Since the field `FixedSizeStack.stack` can be assigned directly by invoking the constructor of `FixedSizeStack` and passing an object of `Stack` as an argument (i.e., `FixedSizeStack.stack` is *assignable* for the declaring class `FixedSizeStack`), the difficulty of generating an object state of `FixedSizeStack` whose size is 10 lies in generating an object of `Stack` whose size is 10. Let us assume that automated test-generation tools cannot produce the required object state of `Stack`.

[5]Assume that `Stack.items` is implemented using the object type `List<object>`

Covana first assigns symbolic values to program inputs and their fields, i.e., `FixedSizeStack`, `FixedSizeStack.stack`, and `Stack.items`, and performs forward symbolic execution to compute data dependencies. By computing data dependencies of partially-covered branch statements, Covana figures out that the branch statement at Line 7 (with the true branch not-covered) has data dependencies on `Stack.items`. However, reporting the object type of `Stack.items`, being `List<object>`, results in a false warning. Since by providing method-call sequences for `List<object>`, the tools cannot assign it to the field `Stack.items` since `Stack.items` is assignable for `Stack`.

Based on this observation, Covana constructs a field declaration hierarchy from the field that the branch statement has data dependencies on up to the program input and identifies the declaring class whose field is not assignable as the cause of the OCP. In this example, the constructed hierarchy is `FixedSizeStack`, `FixedSizeStack.stack`, and `Stack.items`. Covana analyzes the field declaration hierarchy starting from the program input. By analyzing `FixedSizeStack` and `FixedSizeStack.stack`, Covana knows that `FixedSizeStack.stack` is assignable for `FixedSizeStack`. Then Covana continues to check `FixedSizeStack.stack` and `Stack.items`. Since the field `Stack.items` can be changed only by invoking the method `Stack.Push` (i.e., not assignable for `Stack`), Covana identifies the object type `Stack`[6] as an OCP that causes the true branch at Line 5 not to be covered.

4.3.2 Approach

In this section, we describe how Covana identifies problem candidates of structural test generation and prunes irrelevant problem candidates by computing data dependencies. Covana consists of three main steps: Problem-Candidate Identification, Forward Symbolic Execution, and Data Dependence Analysis. In the following part of the section, we introduces the overview of Covana and describe these three main steps in detail.

Overview of Covana

In this chapter, we concretize Covana as an extensible framework for identifying problems that prevent DSE from achieving high structural coverage. DSE [167, 92] executes the program symbolically, starting with arbitrary inputs. Along the execution path, DSE collects symbolic constraints on program inputs in branch nodes (being runtime instances of branch statements) to form an expression, called the path condition. To obtain a new path that takes a different branch, one of the branch nodes in the path condition is negated to create a new path condition that shares the prefix up till the node being negated with the original path. Then a constraint solver is used to compute test inputs that satisfy the new path condition. These generated test

[6] `Stack` is the object type of the field `FixedSizeStack.stack`.

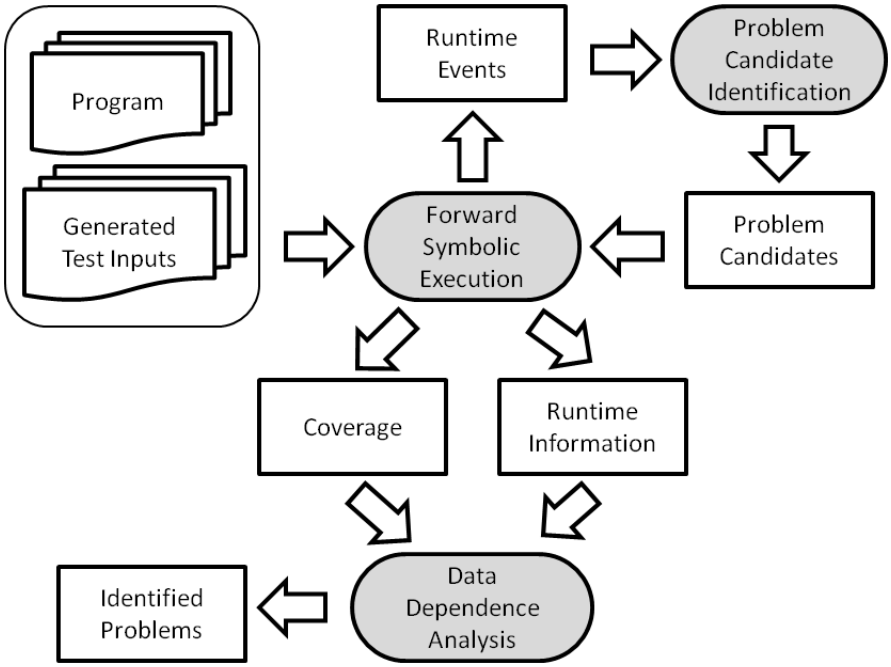


Figure 4.3: Overview of Covana

inputs again are executed on the program to explore different paths of the program. Ideally, all feasible paths can be exercised eventually through such iterations of path variations. However, as we discussed in the introduction, various problems cause DSE not to achieve high structural coverage.

Figure 4.3 shows a high-level overview of Covana. Covana accepts as input a program under test or Parameterized Unit Test (PUT) [187], and generated test inputs from automated test-generation tools (such as a DSE-based tool). Covana then leverages the DSE engine to perform forward symbolic execution on the program or PUT using the test inputs as program inputs (program inputs are assigned with symbolic values). During execution, Covana monitors runtime events triggered by the DSE engine for identifying different types of problem candidates. After identifying problem candidates, Covana assigns symbolic values to elements of these problem candidates, performs forward symbolic execution on these symbolic values, and collects runtime information, such as symbolic expressions and exceptions. Covana then uses the collected structural coverage and runtime information to compute the data dependencies of partially-covered branch statements on problem candidates, and prunes irrelevant problem candidates that none of partially-covered branch statements have data dependencies on. In our current prototype, we instantiate this general approach with two techniques to identify the top two main types of problems: EMCPs and OCPs. We next discuss each step of Covana in detail.

Problem-Candidate Identification

Covana collects necessary information from the DSE engine for identifying different types of problem candidates. The DSE engine executes the program under test or PUT with the generated test inputs symbolically. During execution, Covana monitors different events triggered by the DSE engine and exposes these events as interfaces for specifying different types of problem candidates. There are many kinds of events that can be monitored, such as events of method entry and method exit. We next discuss how these events can be used to identify the problem candidates of EMCPs and OCPs.

Identifying EMCP Candidates

A method exit event is triggered by DSE engine when the execution of a method call is finished. This event comes with detailed method information, including method arguments, method instrumentation information, and so on.

If the method is not instrumented by DSE, the method call is considered as an external-method call, method calls to either system libraries or third-party pre-compiled libraries. If Covana considers all external-method calls as problem candidates of EMCP, then the number of problem candidates can be very large for complex programs. Hence, Covana considers as candidates only the external-method calls whose arguments have data dependencies on program inputs. In this way, the external-method calls that have constant arguments are not considered as problem candidates and are pruned without computing data dependencies. Normally, such external-method calls are method calls that print constant strings or put a thread to sleep for some time, which do not cause DSE not to achieve higher structural coverage and can be safely pruned. Since DSE typically assigns symbolic values to program inputs, to know whether method arguments have data dependencies on program inputs can be achieved easily by checking whether the method arguments contain symbolic expressions of program inputs.

In our preliminary study described in Section 2, we observed that many branches are not covered since the conditions of these branches use the return values of external-method calls (i.e., data dependent on these external-method calls). The reason is that DSE tools and other automated test-generation tools are unlikely to generate different test inputs to cause external-method calls to return desired values since these tools have not instrumented or analyzed external-method calls. Therefore, for the external-method calls whose arguments have data dependencies on program inputs, Covana considers them as EMCP candidates. To illustrate the analysis, we use the example shown in Figure 4.1. During the test execution of the method `GetDefaultConfigFile`, Covana identifies the external-method call `File.Exists` as an EMCP candidate, since its argument `configFilename` has data dependency with `assemblyFile`, which

is the program input. The return value of `File.Exists`, which is used in the branch statement at Line 1, is assigned with a symbolic value for computing data dependencies.

Identifying OCP Candidates

Whenever test inputs are used as the arguments to execute the program under test, the method entry event of the method under test is triggered. In this exposed event, the details of the generated program inputs are collected. Since OCP requires objects of a non-primitive type as program inputs, Covana ignores program inputs whose type is primitive type, such as *int*, *double*, and *boolean*. Covana considers the program inputs of non-primitive types themselves and their fields of non-primitive types as OCP candidates.

Forward Symbolic Execution

Covana performs forward symbolic execution using the test inputs generated by automated test-generation tools as program inputs, and collects runtime information for computing data dependencies. Covana assigns symbolic values to elements of the identified problem candidates (such as return values of external-method calls) and leverages the DSE engine to perform forward symbolic execution for collecting constraints on elements of problem candidates in branch statements. We next discuss how Covana uses this runtime information to compute data dependencies of partially-covered branch statements on problem candidates.

Collecting Symbolic Expressions in Branches. Since elements of problem candidates are assigned with symbolic values, if a branch statement has data dependency on problem candidates, we can find symbolic expressions (on elements of the problem candidates) in the predicates of the branch statement. Such information is later used to compute data dependencies on problem candidates.

Collecting Uncaught Exception. After assigning symbolic values to elements of problem candidates, Covana monitors the program execution. Whenever an uncaught exception is thrown, Covana collects the exception including its stack trace of the exception. As we observed in the preliminary study, if an external-method call throws an exception for the executions of all the generated test inputs, the remaining parts of the program after the call site of the external-method call cannot be covered. Thus, Covana uses the stack trace of an exception thrown at runtime for the analysis of EMCP described in Section 4.3.2.

Data Dependence Analysis

Covana consumes the collected runtime information from the forward symbolic execution to compute data dependencies. For each collected symbolic expression *sym* found in the predicates of a branch statement b, Covana extracts elements of the problem candidates *elem* from

sym. From *elem*, Covana extracts the corresponding problem candidates P and considers b has data dependency on P. Using the collected structural coverage, Covana further computes data dependencies of partially-covered branch statement on problem candidates. With these data dependencies, different analyses further prune irrelevant problem candidates.

EMCP Analysis

Covana first identifies EMCP using the data dependencies of partially-covered branch statements on EMCP candidates. If these exist some partially-covered branch statements that have data dependencies on EMCP candidates for their return values, Covana directly reports such external-method calls as EMCPs. To identify external-method calls that throw exceptions to abort test executions, Covana further analyzes the method calls from the collected stack traces of exceptions thrown during runtime. If these method calls contain any external-method call and the remaining parts of the program after the call site of the external-method call are not covered, Covana identifies the extracted external-method call as an EMCP that causes the remaining parts of the program not to be covered.

Algorithm 1 Object Creation Problem (OCP) Analysis

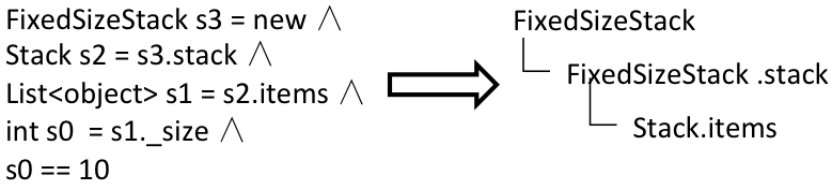
Require: $Fields$ for field declaration hierarchy, B for not-covered branches
Ensure: OCP

```
1: if Length(Fields) == 1 then
2:    OCP = CreateOCP(TypeOf(Fields[0]), B)
3:    return OCP
4: else
5:    Set current = NULL
6:    for i = 1 to Length(Fields) − 1 do
7:       current = Fields[i]
8:       dc = TypeOf(Fields[i − 1])
9:       assg = IsAssignable(current, dc)
10:      if !assg then
11:         OCP = CreateOCP(dc, B)
12:         return OCP
13:      end if
14:   end for
15:   OCP = CreateOCP(TypeOf(current), B)
16:   return OCP
17: end if
```

OCP Analysis

Covana identifies OCPs using the data dependencies of partially-covered branch statements on program inputs and their fields. If a partially-covered branch statement is data dependent on only program inputs, Covana directly reports the program inputs as OCPs. However, if a partially-covered branch statement is data dependent on fields of program inputs, Covana constructs a field declaration hierarchy up to a program input and performs further analysis to identify which field causes tools not to achieve high structural coverage.

To construct a field declaration hierarchy of the field f that a partially-covered branch statement is data dependent on, we can use reflection to obtain the class structure of a program input p and search the fields for finding f (i.e., f is one of the fields of p). If we fail to find f, we continue to search the class structures of the fields of p, similar to graph searching. The search continues until we find f. The fields along the path from p to f are used to construct the field declaration hierarchy. Another way, which Covana adopts, is to use path conditions that lead to not-covered branches and extract fields directly from path conditions, since the symbolic expressions in the path conditions already contain program inputs and their fields. To illustrate the extraction, we use the example shown in Figure 4.2. The figure below shows the path condition that leads to the true branch at Line 5 and the field declaration hierarchy constructed from the path condition.



Algorithm 1 shows our algorithm that identifies OCPs by analyzing the field declaration hierarchy (Fields) and the not-covered branches (B). If the extracted field declaration hierarchy contains only one field (satisfying Line 1 of the algorithm), which should be the program input itself, o ur algorithm reports the program input as an OCP directly.

To identify which field in the field declaration hierarchy causes the OCP, our algorithm analyzes the field declaration hierarchy level by level, starting from Level 2 (i.e., the field of the program input). If a field is assignable for its declaring class (not satisfying Line 10), DSE or other automated test-generation tools can easily create an object of the field's class type and assign the object to the field by invoking the corresponding constructor or public setter method. In this case, it is the object type of the field or a field in the next level(s), not its declaring

class, that causes an OCP. To further decide whether it is the current field or the field in the next level that causes the OCP, our algorithm then continues to check the field in the next level (back to Line 7).

If a field is not assignable for its declaring class (satisfying Line 10), the object state of the field can only be changed by invoking other public state-modifying methods of its declaring class. Hence, Covana reports the type of its declaring class as an OCP. In our example shown in Figure 4.2, the field `Stack.items` cannot be assigned with an object of `List<object>` by invoking any constructor or public setter method of `Stack`. To change the object state of `Stack.items`, DSE needs specific sequences of method calls for `Stack` instead of `List<object>`. As a result, Covana identifies the object type `Stack` as an OCP (Line 11).

4.3.3 Evaluations

In this section, we discuss the two evaluations conducted to show the effectiveness of Covana. In our evaluations, we use two popular .NET applications: xUnit [208] and QuickGraph [158], and answer the following research questions:

- **RQ4.1**: How effective is Covana in identifying the two main types of problems, EMCPs and OCPs?
- **RQ4.2**: How effective is Covana in pruning irrelevant problem candidates of EMCPs and OCPs?

We next provide details of the metrics that we collect in our evaluations. To measure the effectiveness of our approach in identifying EMCPs and OCPs (addressing RQ4.1), we measure the number of problems that Covana finds for the not-covered branches or statements of the subject applications. To measure the effectiveness of our approach in pruning irrelevant problem candidates (addressing RQ4.2), we compare the number of identified problem candidates with the number of identified problem by our approach in applications under test and measure the number of problem candidates pruned by our approach. To address both RQ4.1 and RQ4.2, we measure the false positives, i.e., the number of irrelevant problem candidates that are not pruned by Covana, and the false negatives, i.e., the number of real problems that are identified as irrelevant problem candidates and pruned.

We next provide details on the subject applications and evaluation setup, and the results of the two evaluations.

Subjects and Evaluation Setup

We used two popular .NET applications for evaluating our Covana approach: xUnit [208] and QuickGraph [158]. xUnit is a unit testing framework for .NET program development. xUnit

Table 4.2: Evaluation results showing the effectiveness of Covana in identifying EMCP and OCP

Application Assembly	# File	**Object-Creation Problem (OCP)**				**External-Method-Call Problem (EMCP)**			
		# Identified	# Real	# FP	# FN	# Identified	# Real	# FP	# FN
xUnit	71	68	67	13	12	24	24	0	0
xUnit. Extensions	17	7	5	3	1	2	2	0	0
xUnit.Console	7	2	2	0	0	2	2	0	0
xUnit.Gui	12	3	3	0	0	1	3	0	2
xUnit. Runner.Msbuild	6	15	14	1	0	0	0	0	0
xUnit. Runner.Tdnet	3	5	5	0	0	1	1	0	0
xUnit. Runner.Utility	28	7	12	0	5	9	9	0	0
Quickgraph	3	0	0	0	0	0	0	0	0
Quickgraph. Algorithms	12	7	11	0	4	0	0	0	0
Quickgraph. Algorithms. Graphviz	14	20	20	2	2	4	3	1	0
Quickgraph. Collections	19	6	11	1	6	0	0	0	0
Quickgraph. Concepts	35	5	5	0	0	0	0	0	0
Quickgraph. Exceptions	3	0	0	0	0	0	0	0	0
Quickgraph. Predicates	9	8	8	0	0	0	0	0	0
Quickgraph. Representations	3	2	2	0	0	0	0	0	0
Total	242	155	163	20	30	43	44	1	2

includes 223 classes and interfaces with 11.4 KLOC. QuickGraph is a C# graph library that provides various directed and undirected data structures of graphs. QuickGraph also provides graph algorithms such as depth-first search, topological sort, and shortest path [57]. QuickGraph includes 165 classes and interfaces with 8.3 KLOC.

In our evaluations, we use Pex with the implemented extensions as our DSE test-generation tool. The Pex version used for our evaluation is 0.24.50222.1. We first apply Pex to explore the applications under test and generate test inputs. After test generation and execution, which is automated by Pex, the coverage and the collected runtime information are fed into our stand-alone analysis tool for identifying EMCPs and OCPs.

We next discuss the results of our evaluations in terms of the effectiveness of Covana in identifying EMCPs and OCPs, and in reducing the irrelevant problem candidates.

RQ4.1: Problem Identification

In this section, we address the research question RQ4.1 of how effectively Covana identifies EMCPs and OCPs. To address this question, we measure the number of identified problems, the number of false positives, and the number of false negatives generated by Covana. To measure values for these metrics, we executed the stand-alone analysis tool implemented for our approach with the output information from Pex as inputs, and manually classified the problems reported by our tool as real problems, false positives, and false negatives. To verify EMCP candidates, we either instrument or provide mock objects for the external-method calls identified as EMCP candidates, and reapplied Pex to check whether the not-covered branches can be covered. If so, we classify the EMCP candidates as real problems, or irrelevant problem candidates otherwise. Similarly, to verify OCP candidates, we provide sequences of method calls for the object types of the OCP candidates, and reapplied Pex to check whether the not-covered branches can be covered. If so, we classify the OCP candidates as real problems, or irrelevant problem candidates otherwise.

Table 4.2 shows the results for all the assemblies in both subject applications. Column "# File" lists the number of source files in each application assembly. Columns "Object-Creation Problem (OCP)" and "External-Method-Call Problem (EMCP)" show the statistics of EMCPs and OCPs identified by Covana. Subcolumn "# Real" gives the number of real problems identified by us manually. Subcolumn "Identified" gives the number of problems identified by Covana, and subcolumn "# FP" and "# FN" give the number of false positives and false negatives, respectively. The results show that our approach identifies 43 EMCPs with only 1 false positive and 2 false negatives. In addition, our approach identifies 155 OCPs with 20 false positives and 30 as false negatives. The reason why we have 30 false negatives is that in our prototype analysis tool, we did not implement the logics required to handle `Dictionay` objects (C# version of `HashMap`) and static fields of classes. In our future work, we plan to address these issues by identifying the fields of `Dictionay` objects and static fields of classes as candidates and computing data dependencies of partially-covered branch statements on them.

We next provide examples to describe scenarios where our approach effectively identifies EMCPs and OCPs. We also describe scenarios where our approach produces false positives and false negatives.

Figure 4.4 shows the class `TestClassCommand` of the `Xunit.Sdk` namespace. When we applied Pex to generate test inputs for the method `TestClassCommand.ClassStart`, Pex generated only one test input and achieved low block coverage of 2/27 (7.14%). In the method `TestClassCommand.ClassStart`, the loop at Line 11 requires the field `TestClassCommand.typeUnderTest` to be not null. Since Pex cannot find in the application any public class that implements the interface `ITypeInfo` to create such an object for `TestClassCommand.typeUnderTest`,

```
1  public class TestClassCommand : ITestClassCommand {
2      readonly Dictionary<MethodInfo, object> fixtures = new Dictionary<MethodInfo, object>();
3      Random randomizer = new Random();
4      ITypeInfo typeUnderTest;
5      ...
6      public TestClassCommand(ITypeInfo typeUnderTest) {
7          this.typeUnderTest = typeUnderTest;
8      }
9      public Exception ClassStart() {
10         try {
11             foreach (Type @interface in typeUnderTest.Type.GetInterfaces()) {
12                 ...
13             }
14         }
15         ...
16     }
17     public Exception ClassFinish() {
18         foreach (object fixtureData in fixtures.Values) {
19             ...
20         }
21     }
22 }
```

Figure 4.4: `TestClassCommand` class of xUnit

Pex cannot generate more useful test inputs. Thus, we need to report an OCP of the interface type `ITypeInfo`. By analyzing the data dependencies of the entry branch of the loop at Line 11, our approach extracts the argument object `TestClassCommand` and its field `TestClassCommand.typeUnderTest`. By analyzing `TestClassCommand` and `TestClassCommand.typeUnderTest`, our approach figures out that `TestClassCommand.typeUnderTest` can be assigned by using the public constructor of the class `TestClassCommand`, and correctly reports an OCP of `ITypeInfo`.

Similarly, Pex achieves low coverage block coverage of 6/16 (37.50%) when generating test inputs for the method `TestClassCommand.ClassFinish`. The reason is that the loop at Line 18 requires the field `TestClassCommand.fixtures` to hold at least one item. Since there is no constructor or public setter method to assign an external object to `TestClassCommand.fixtures`, to change the value of `TestClassCommand.fixtures`, other public methods of `TestClassCommand` need to be invoked . Therefore, we need to report the program input `TestClassCommand` as an OCP. However, our approach cannot detect such situation since the object type of the field `fixtures` is `Dictionary` and we did not implement the logics to handle such type. Hence, our approach did not identify the object type of the `fixtures` as an OCP for the not-covered branch at Line 18.

Figure 4.5 shows two methods: (1) the method `ParseCommandLine` of class `Program` in the namespace `Xunit.ConsoleClient` and (2) the constructor of the class `Executor` in the

```
1 static bool ParseCommandLine(string[] args, out string assemblyFile, ...) {
2     assemblyFile = args[0];
3     ...
4     if (!File.Exists(assemblyFile)) {
5         Console.WriteLine("error: assembly file not found: {0}", assemblyFile);
6         return false;
7     }
8     ...
9 }
10
11 public Executor(string fileName) {
12     this.assemblyFilename = Path.GetFullPath(fileName);
13     ...
14 }
```

Figure 4.5: Two methods that have EMCPs in xUnit

namespace `Xunit.Sdk`. For `ParseCommandLine`, Pex achieved low block coverage of 44/154 (28.57%), because it cannot generate test inputs to cause the external-method call `File.Exists` to return true. Since the `out` variable `assemblyFile` is assigned with the value of `args[0]` (and thus has data dependencies on the program input `args[0]`), our approach assigned a symbolic value to the return value of `File.Exist` and found that the branch statement at Line 4 (the false branch not-covered) has data dependency on `File.Exist` for its return value. Thus, our approach correctly reported an EMCP of `File.Exists`. For the constructor of the class `Executor`, Pex achieved low block coverage of 2/5 (40%), because Pex generated a `null` object as the argument for the constructor, which caused the external-method call `Path.GetFullPath` to throw an exception. Our approach collected this exception thrown from `Path.GetFullPath` during runtime. By checking the coverage of the remaining parts of the program after the call site of `Path.GetFullPath`, our approach found that none of them was covered. As a result, our approach reported `Path.GetFullPath` as an EMCP. Although another external method `Console.WriteLine` at Line 5 receives `assemblyFile` as argument and is marked as an EMCP candidate by our approach, this external method did not have any return value, and thus no branch statements have data dependencies on `Console.WriteLine`. As a result, our approach correctly pruned `Console.WriteLine`.

RQ4.2: Irrelevant-Problem-Candidate Pruning

In this section, we address the research question RQ4.2 of how effectively our approach prunes irrelevant problem candidates. To address this question, we compare the number of identified problem candidates with the number of problems reported by our approach, and measure the number of problem candidates pruned by our approach. In addition, we measure the false

Table 4.3: Evaluation results showing the effectiveness of Covana in reducing irrelevant problem candidates

App	Object-Creation Problem (OCP)					External-Method-Call Problem (EMCP)				
	#C	#I	#Pruned	#FP	#FN	#C	#I	#Pruned	#FP	#FN
xUnit	335	107	228 (68.06%)	17	18	1313	39	1274 (97.03%)	0	2
QuickGraph	116	48	68 (58.62%)	3	12	297	4	293 (98.65%)	1	0
Total	451	155	296 (65.63%)	20	30	1610	43	1567(97.33%)	1	2

positives, i.e., the irrelevant problem candidates not pruned by Covana, and the false negatives, i.e., the real problems pruned by Covana. To measure values for these metrics, we executed the stand-alone analysis tool implemented for our approach with the output information from Pex as inputs, and manually classified the problem candidates reduced by our tool as real problems, false positives, and false negatives in the same way as in addressing RQ4.1.

Table 4.3 shows the results of both subject applications. Column "Application" lists the names of the subject applications. Columns "External-Method-Call Problem" and "Object-Creation Problem" show the statistics of EMCPs and OCPs, respectively. Here, the EMCP candidates are all the encountered external-method calls during the test execution, and the OCP candidates are all the non-primitive object types of program inputs and their fields that DSE assigns symbolic values to. In Table 4.3, subcolumn "#C" gives the number of problem candidates, subcolumn "#I" gives the number of problems identified by Covana, and subcolumns "#FP" and "#FN" give the number of false positives and false negatives, respectively. The results show that our approach prunes 97.33% (1567 in 1610) EMCP candidates with only 1 false positive and 2 false negatives and prunes 65.63% (296 in 451) OCP candidates with 20 false positives and 30 false negatives. These results show that our approach effectively reduces the irrelevant problem candidates with low false positives and false negatives.

4.3.4 Discussion

Covana identifies problems faced by tools built for structural test-generation approaches and prunes irrelevant problem candidates to reduce the problem space for investigation. Covana serves as the first step towards problem solving. In fact, identifying problems for developers to investigate is analogical to fault localization before fault fixing. Below, we discuss how Covana can be used to assist other automated test-generation approaches or manual test-generation approaches, and then discuss some issues including those encountered in our evaluations.

Assisting Other Structural Test-Generation Approaches. Given test inputs, no matter whether they are generated by other automated test-generation approaches, such as a random approach, or are generated manually, Covana can be used to identify problems of specific types, such as EMCPs and OCPs. The analysis result of Covana not only can reduce the ef-

```
1  // Lines 1, 2, 4 are external−method calls
2  public static List<RecentlyUsedAssembly> LoadAssemblyList() {
3      ...
4      using (var xunitKey = Registry.CurrentUser.CreateSubKey(XUNIT_KEY_NAME)
5      using (var recentKey = xunitKey.CreateSubKey(RECENT_ASSEMBLIES_KEY_NAME)){
6          for (int index = 0; ; ++index)
7              using (var itemKey = recentKey.OpenSubKey}(index.ToString())) {
8              if (itemKey == null) {
9                  break;
10             }
11             if (itemKey != null) {
12                 ...
13             }
14         }
15     }
16 }
```

Figure 4.6: The method `LoadAssemblyList` in the class `RecentlyUsedAssemblyList` of xUnit

forts of developers in providing guidance to tools, but also can reduce the cost of tools built for other test-generation approaches. The first example is to automatically generate mock objects for only the external-method calls identified as EMCPs by Covana. Since Covana greatly reduces the number of irrelevant problem candidates of EMCP, it becomes possible to generate mock objects for the external-method calls identified as EMCPs. As another example, random approach can assign more probabilities on exploring the object types reported as OCPs by Covana, increasing the chances to achieve higher structural coverage in shorter time. Advanced method-sequence-generation approaches [183] can also be used to address OCPs for increasing coverage.

Static Field. In our evaluations, we observed that a few classes contained static fields that were initialized inside the classes. These static fields were later used by some branches and some of these branches were not covered by DSE. Since DSE did not automatically assign symbolic values to static fields, DSE was not able to collect symbolic constraints on these static fields. In future work, we plan to assign symbolic values to these static fields, so that our approach can collect the symbolic constraints on these static fields for our analysis.

Concrete Arguments for External-Method Calls. Our current Covana implementation identifies the return values of external methods as candidates if the method arguments have data dependencies on program inputs. However, in our evaluations, there were a few external-method calls received concrete values as arguments, and resulted in some not-covered branches. The external-method call `recentKey.OpenSubKey`, shown in Figure 4.6, received a concrete value returned by `index.ToString()`. Since its return value `itemKey` of `recentKey.OpenSubKey` is `null`, the false branch at Line 8 is not covered. In this case, our approach cannot detect the prob-

lem, since our approach does not mark as a candidate the return value of any external-method call that does not receive any symbolic values as an argument. By assigning symbolic values to all external-method calls, our approach can be easily extended to compute data dependencies on every external-method call, no matter whether its arguments have data dependencies on program inputs. However, computing data dependencies on every external-method call may incur many false positives and increase the performance overhead significantly, since the number of external-method calls encountered during the program executions is not trivial. In future work, we plan to conduct experiments to measure the effectiveness and performance overhead when every external-method call is considered as a candidate.

Other Potential Issues. Besides the issues encountered in our evaluations, there are still some potential issues that may affect the effectiveness of our approach: (1) **argument side effect**: some external-method calls may have side effects on the receiver objects or method arguments that have data dependencies on program inputs, causing some subsequent branches not to be covered; (2) **control dependency**: extending our approach to consider control dependency may improve the effectiveness of our approach in some cases; (3) **static analysis**: our approach currently computes dynamic data dependencies based on the executed paths, and may miss some data dependencies on unexecuted paths. Employing static analysis to analyze all the paths is one option to solve the problem. Nevertheless, due to the complexity of programs, static analysis may produce false positives on detected data dependencies, which would compromise the effectiveness of our approach. We plan to conduct experiments to evaluate the effectiveness of incorporating argument side effect, control dependency, and static analysis.

4.4 Summary

Structural test-generation tools face various problems when dealing with complex object-oriented programs in practice. To understand the problems that prevent test-generation tools from achieving high structural coverage, we have conducted an empirical study and identified major types of problems (OCPs and EMCPs) faced by the test-generation tools, with the focus on DSE-based tools. Based on the study results, we have proposed a novel approach, called Covana, which precisely identifies and reports problems that cause structural test-generation tools not to achieve high structural coverage. Covana identifies these problems by computing data dependencies of partially-covered branch statements on problem candidates. We concretize Covana to identify problems faced by DSE and present two techniques to identify EMCPs and OCPs, the top two major types of problems. We also evaluate Covana on two open source projects and the results show that Covana effectively identifies EMCPs and OCPs.

Based on Covana, our approach provides a problem-diagnosis interface that shows problems faced by structural test-generation tools, enabling the problem-diagnosis cooperation on test-

generation tools. In this cooperation, the tools automatically generate test inputs for achieving coverage of a program and report problems faced by the tools for not-covered code. Based on the users' domain knowledge of the program under test, the users provide factory methods for addressing OCPs, and provide mock objects for addressing EMCPs.

CHAPTER 5

Problem-Diagnosis Cooperation for Prioritizing Problems Faced by Structural Test Generation

5.1 Introduction

To help test-generation tools address their problems and achieve better coverage, Covana [205] proposes the problem-diagnosis cooperation for the test-generation tools, where tools and developers cooperate to effectively carry out software testing (described in Chapter 4). When the test-generation tools face difficulties in achieving high coverage, Covana reports not only the observed *symptoms* (e.g., not-covered branches) but also the possible *causes* of the symptoms (e.g., OCPs and EMCPs) back to developers. Developers then can provide guidance to help the test-generation tools address these causes in order to eliminate the observed symptoms. For example, to address a reported OCP, developers can provide a factory method that encodes a method sequence to create a desired object-field state for a non-primitive object type. By using this factory method, the test-generation tools may be able to cover the not-covered branch affected by the OCP.

However, in practice, often a long list of causes can be presented to developers. Given limited time, developers may not be able to address all the causes. Thus, it is desirable to enable developers to maximize their testing goals within the given time of addressing causes. For example, if the goal is to achieve high overall structural coverage, developers could first address causes that bring high coverage improvement. To do so, the developers need to know the coverage benefits brought by addressing each cause.

To enable such economical problem-diagnosis cooperation for test-generation tools, we propose an economic-analysis framework EcoCov [132] that accurately estimates the benefit of addressing a cause. In this framework, we concretize addressing a cause as solving a problem, either an OCP or an EMCP, and eliminating a symptom as covering a not-covered branch. We assume that the benefit of covering each branch is the same (i.e., branches are equally critical). In future work, we plan to conduct further user studies to better measure the benefit of covering a branch. With these assumptions, the benefit of solving a problem (referred to as problem benefit) is the increased branch coverage (the number of newly-covered branch).

To estimate the problem benefit, our main idea is to approximate the solution to the problem with value replacement on the problem-inducing element. A problem-inducing element could be either an object field of an OCP or the return of an EMCP, on which the not-covered branch has data dependencies. Such problem-inducing elements can be automatically identified by cooperative-testing tools like Covana. Given a problem, instead of providing a solution to the problem, EcoCov replaces the problem-inducing element with a choose operator that produces nondeterministic values, and conducts diagnosis by leveraging the test-generation tools to generate a desired value for the element, making the problem disappear. The test-generation tools work on the program with the choose operator (i.e., angelic program) as if there were no problem encountered. In this way, EcoCov simulates the effect of solving a problem and thus gives an estimation on the coverage after solving the problem. The number of newly-covered branches is reported as the problem benefit.

Based on EcoCov, our approach provides the problem-diagnosis interface to show the ordering of the problems based on their estimated benefits, extending the problem-diagnosis interface of precisely reporting problems faced by structural test-generation tools. In this cooperation, the tools automatically generate test inputs for achieving coverage of a program and report problems faced by the tools for not-covered code along with the ordering of the problems based on the benefits of addressing the problems. Based on the users' domain knowledge of the program under test and the benefits of addressing the problems, the users provide factory methods for addressing the selected OCPs, and provide mock objects for addressing the selected EMCPs.

We name the main idea of our EcoCov framework as *angelic diagnosis* since it uses an angelically nondeterministic operator to generate desired values to bypass the problems. The choose operator is angelic because it collaborates with the test-generation tools against the problems; it is nondeterministic because it can generate any suitable value. Essentially, with the choose operator, EcoCov reduces the real problems into constraint-solving problems by neglecting the object encapsulation and environment dependencies. Due to such neglect, EcoCov may generate angelic values that violate related class invariants and environment constraints, and thus produce false positives and false negatives in the estimation. However, in our evaluations, we find few such cases. Even under some of these cases, the estimation provide by

```
1  void Test(Stack stack, string filename, int x, int y) {
2      if (x > 5) { // B1
3          if (!File.Exists(filename)) { // EMCP1 for B2
4              ...
5          }else return;
6      }else {
7          int count = stack.Count;
8          if (count > 15) { // OCP for B3
9              string path = Path.GetFullPath(filename); //EMCP2 for B4 (implicit exceptional branch)
10         }else return;
11     }
12     if (y > 0){ ... } // B5 target branch
13 }
```

Figure 5.1: An example program that contains both OCPs and EMCPs

EcoCov is still acceptable. Compared to the static approach [107], which estimates the coverage by statically computing the must-execute-block set for covering the not-covered branches, our dynamic approach actually runs the test-generation tools on the angelic program to provide a more accurate coverage estimation. The estimation reflects not only the effect of covering the not-covered target branch but also the effect of further explorations by the test-generation tools on the other parts of the program.

We implement EcoCov on Pex [184] from Microsoft Research, a state-of-the-art test-generation tool based on Dynamic Symbolic Execution. Although we choose Pex, our general framework can be easily extended to other symbolic-execution-based test-generation tools. We evaluate EcoCov on three open-source C# projects and the evaluation results show that EcoCov achieves average 91.1% precision and 96.6% recall in estimating the problem benefit.

5.2 Examples

In this section, we use an example to illustrate how we realize angelic diagnosis using dynamic instrumentation and how EcoCov estimates the problem benefits and branch costs.

Figure 5.1 shows a C# example program that contains 1 OCP and 2 EMCPs. To cover the true branch of B3 at Line 8, DSE tools need to generate a `stack` object whose size is larger than 15. Typically, the field `Stack.Count` of `stack` is a private field and can be modified by invoking only `Push` or `Pop` methods. Thus, DSE tools need to generate a sequence of method calls that creates a `stack` object and invokes `Push` 15 times to generate a desired `stack` object. However, since the combination of method calls grows exponentially with the number of method calls in the sequence, the search space is very huge and existing DSE tools cannot easily generate

```
1  void Test(Stack stack, string filename, int x, int y) {
2      if (x > 5) { // B1
3          Moles.Replace("File.Exists", () => PexChoose.Value<bool>("symBool"));
4          if (!File.Exists(filename)) { // EMCP1 for B2
5              ...
6          }else return;
7      }else {
8          if (stack != null){
9              PexInvariant.SetField<System.Int32>(stack, "Count",
                   PexChoose.Value<System.Int32>("symField"));
10         }
11         int count = stack.Count;
12         if (count > 15) { // OCP for B3
13             Moles.Replace("Path.GetFullPath", () => PexChoose.Value<string>("symStr"));
14             string path = Path.GetFullPath(filename); //EMCP2 for B4 (implicit exceptional branch)
15         }else return;
16     }
17     if (y > 0){ ... } // B5 target branch
18 }
```

Figure 5.2: An angelic program transformed from the example in Figure 5.1

such sequence of method calls. Without the desired sequence, DSE tools cannot cover the true branch of B3. Such problems are referred to as OCPs.

The two EMCPs in this example are caused by two external-method calls: `File.Exists` and `Path.GetFullPath`. If DSE tools cannot generate a file name of a file that exists in the system, the true branch of B2 cannot be covered. Similarly, if DSE tools cannot generate a valid name of an existing file, the method `Path.GetFullPath` at Line 9 would throw exceptions (covering the exceptional implicit branch of B4), preventing DSE tools from covering the remaining part of the code after B4.

Dynamic Instrumentation. To realize the angelic diagnosis, we dynamically instrument the program with the angelic CHOOSE operator for each problem. For an EMCP, EcoCov uses Moles [60], a framework that can detour .NET methods to alternative implementations, to replace the external method with an alternative implementation. Such alternative implementation simply contains a choose operator and returns the angelic value generated by the operator. In this way, EcoCov replaces the state of the return variable with an angelic value. Figure 5.2 shows the angelic program transformed from the example in Figure 5.1. The statement at Line 3 is introduced by the dynamic instrumentation for EMCP1. The statement uses the `Replace` method of Moles to redirect the external method `File.Exist` (first parameter) to an alternative implementation represented as a *lambda expression* (second parameter). The alternative implementation returns an angelic value generated from the method `PexChoose.Value` (the choose operator), so that the true branch of B2 may be covered by the angelic value.

For an OCP, EcoCov leverages the C# Reflection libraries to break the object encapsulation constraints and directly set an angelic value generated by the choose operator for the problem-inducing object field. In Figure 5.2, the code snippet Lines 8-10 is introduced by the dynamic instrumentation for the OCP. The code snippet first checks whether the stack is null. If not, it assigns an integer angelic value to the field `Stack.Count`.

Benefit-Cost Estimations. To estimate the problem benefit, EcoCov runs Pex again on the angelic program and uses the number of newly-covered branches as the estimated benefit. For example, in Figure 5.2, assume that we want to estimate the benefit of solving EMCP1, leaving the other two problems not solved. EcoCov first introduces the statement at Line 3 by dynamic instrumentation and then runs Pex on the instrumented program. With this instrumentation, Pex can generate true as the return value for `File.Exists` and covers the true branch of B2. Moreover, after covering the true branch of B2, Pex would continue to explore the remaining part of the program starting from Line 20 and easily cover the both branches of B5. Thus, there are 3 new branches being covered and the estimation on the benefit of solving EMCP1 is 3. Similarly, if we do the instrumentation for only the OCP (leaving EMCP1 and EMCP2 not solved), Pex would generate an integer value larger than 15 for the field `Count` and thus cover the true branch of B3. After covering the true branch of B3, Pex can cover only the exceptional branch of B4 (because we haven't solved the EMCP2) and still cannot reach B5. Thus, the true branch of B3 and the exceptional branch of B4 are the newly-covered branches and the benefit is 2.

5.3 Approach

Figure 5.3 shows the overview of EcoCov. A test-generation tool is applied to generate test inputs for a program under test. By executing the program under test with the generated test inputs, we obtain the achieved coverage. We then apply Covana to analyze the not-covered branches to identify their causing problems (focusing on OCPs and EMCPs). EcoCov accepts as inputs a program under test, the test inputs generated by the test-generation tool, and the problems (OCPs and EMCPs) identified by Covana. EcoCov transforms the program into an angelic program by replacing the problem-inducing elements with choose operators, which produce angelic values to bypass the problem. The transformation is done based on the instrumentation templates that specify the uses of dynamic-instrumentation libraries. EcoCov then apply the test-generation tool on the angelic program and obtain the new coverage. The delta between the new coverage and the original coverage achieved by the test-generation tool is output as the problem benefit.

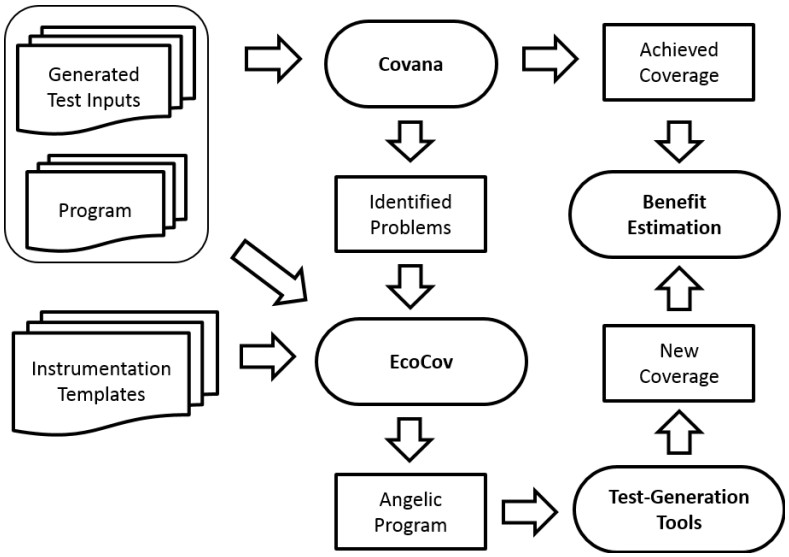


Figure 5.3: Overview of EcoCov

5.3.1 Estimation of Problem Benefit

To estimate the problem benefit, EcoCov replaces the problem-inducing element with the choose operator and leverages the test-generation tools to generate desired values for the operator to bypass the problem. We next describe details on the synthesis of an angelic program, which is the core of the estimation.

Estimation of OCP Benefit

An OCP specifies an object type for which the test-generation tools fail to generate desired states. To bypass the difficulties caused by an OCP, EcoCov synthesizes an angelic problem by replacing relevant object fields with the choose operators to obtain the desired object state. Algorithm 2 shows how to synthesize an angelic program for an OCP.

The algorithm starts by obtaining the desired fields of the OCP (Line 1), which are the fields affecting the not-covered branches. The details of the desired fields and how to obtain them are described later in this section. If no fields but the program input itself is involved in the not-covered branches (e.g., the branch is to perform a null check on the program input), EcoCov synthesizes the code that allocates a not-null symbolic memory for the program input $fields.InputType$ based on the instrumentation template $\mathcal{I}_t.SymObj$ (Lines 3-4). Here, $fields.InputType$ specifies the type of the related program input. If the fields of a program input are involved in the not-covered branches, EcoCov uses the instrumentation template $\mathcal{I}_t.SymField$ to synthesize the code that turns these fields into symbolic values (Lines 6-11). The synthesized

Algorithm 2 Angelic diagnosis for an OCP

Require: $\mathcal{P}$: A program under test
$\mathcal{T}$: Generated test inputs
$\mathcal{OCP}$: An OCP
$\mathcal{I}_t$: Instrumentation templates
Ensure: $\mathcal{P}_a$: An angelic program with choose operators

```
1: fields ← GetDesiredFields(P, T, OCP)
2: txt ← string.Empty
3: if IsProgramInput(fields) then
4:    txt.append(Synthesize(I_t.SymObj, fields.InputType))
5: else
6:    for field ∈ fields do
7:       for ref ∈ field.AP do
8:          txt.append(Synthesize(I_t.FieldCheck, ref))
9:       end for
10:      txt.append(Synthesize(I_t.SymField, field))
11:   end for
12: end if
13: for method ∈ P do
14:   if method.paraHasType(fields.InputType) then
15:      P_a.add(Angelic(method, txt))
16:   else
17:      P_a.add(method)
18:   end if
19: end for
20: return P
```

code also performs null checks on each memory reference to avoid null dereferences. If the program input is an interface, EcoCov synthesizes a class that implements the interface (not shown in Algorithm 2). For each interface method used in the program, EcoCov uses the instrumentation template $\mathcal{I}_t.SymField$ to return a symbolic value. These symbolic values are considered as the angelic values produced by the choose operators. With the synthesized code txt, EcoCov enumerates all the methods in the program $\mathcal{P}$ and inserts txt into a method if any parameter of the method has the type specified in $fields.InputType$. The transformed program is returned as an angelic program.

Identification of Desired Fields. A field of a program input is said to be a desired field of an OCP if (1) it affects a not-covered branch, and (2) the test-generation tools cannot generate an input with a desired value for the field in order to cover the not-covered branch. For example, in Figure 5.1, the field `Count` of the program input `stack` is a desired field because the true branch of B3, which is not covered, depends on `Count`, and the test generation tools cannot

```
1 //Symbolic-object template
2 |input| = choose<|type|>()
3
4 //Object-field-check template
5 if(GetField(|ref|) != null) {
6     |hole|
7 }
8
9 //Symbolic-field template
10 SetField("|fname|", choose<|type|>())
11
12 //Method-redirect template
13 Moles.Replace("|method|", () => choose<|type|>())
```

Figure 5.4: Instrumentation templates used by EcoCov

generate a stack with the field `Count` larger than 15. Given a program, a desired field can be represented as an *access path* (AP) [68], which is a non-empty sequence of memory references to describe how to access the desired field through the program input. For the field `Count`, the AP is `stack.Count`. EcoCov uses such APs to access the desired fields from program inputs when EcoCov synthesizes code to replace the fields with `choose` operators.

To identify the desired fields of OCPs, EcoCov first uses data-dependency analysis to identify the candidate fields of program inputs, on which the not-covered branches have data dependencies. For each candidate field, EcoCov checks if the test-generation tools can generate a value for the field to cover the corresponding not-covered branch. If not, the field is identified as a desired field. Note that EcoCov does not consider control dependencies to identify the desired fields so that some desired fields may be missed. Based on the studies shown in Section 4.2, data dependencies are already precise enough to identify the desired fields.

Instrumentation Templates. Instrumentation templates provide the program templates that specify the uses of dynamic-instrumentation libraries, such as how to access fields from objects and how to turn the fields into symbolic. Research has shown the effectiveness of program templates in synthesizing program invariants [74, 176] and assisting program verification [176, 175]. These templates provide high-level hints from developers, and can be filled with details based on the analysis of programs. Therefore, EcoCov adapts template-based techniques to synthesizes partial code that uses dynamic instrumentation to turn the desired fields of the OCPs into symbolic. EcoCov fills the instrumentation templates with variable names and memory references based on the data-dependence analysis on the desired fields.

As shown in Figure 5.4, EcoCov uses three instrumentation templates for OCPs. (1) *Symbolic-object template* ($\mathcal{I}_t.SymObj$): this template specifies how to dynamically allocate a symbolic memory for a given object type. During synthesis, the hole `|input|` is filled with a variable

name, and the hole `|type|` is filled with the object type of the variable. EcoCov uses this template to allocate a symbolic memory for the program input of the OCP (Lines 4-6 of Algorithm 2). (2) *Object-field-check template* ($\mathcal{I}_t.FieldCheck$): this template specifies how to dynamically access a field of an object instance and how to perform null check of the field. During synthesis, the hole `|Ref|` is filled with a memory reference from an AP, and the hole `|hole|` is filled with any code fragment. EcoCov uses this template to perform null check on each memory reference of an AP (Lines 7-9 of Algorithm 2). (3) *Symbolic-field template* ($\mathcal{I}_t.SymField$): this template specifies how to turn a field into a symbolic value. EcoCov uses this template to turn the desired field into a symbolic value (Line 11 of Algorithm 2). Currently, our templates are customized for .NET language and Pex, but these templates can be easily configured to support other languages and tools, such as Java and Randoop [152].

Estimation of EMCP Benefit

An EMCP specifies an external-method call that throws exceptions to abort test executions, or its return value causes some branches not to be covered. To bypass the difficulties caused by an EMCP, EcoCov uses dynamic instrumentation to redirect the external-method call to an angelic implementation that returns the value from a choose operator.

Method-call Detour. To redirect the external-method call, EcoCov uses the Moles [60] framework to replace the external-method call with the angelic implementation. Everytime the specified external method is called, the method call will be detoured to the corresponding angelic implementation by Moles. The angelic implementations synthesized by EcoCov are context-insensitive mock implementations [188], which uses a choose operator to return a non-deterministic value for each method call. Context-insensitive mock implementations may cause inconsistency problems. For example, invoking `File.Create` with a file name $fname$ to creat a file and then invoking `File.Exists` with $fname$ should return *true*. However, such mock implementation can make `File.Exists` with $fname$ return *false*, causing imprecise estimation for EcoCov. In future work, we plan to investigate parameterized mock objects [135, 181] that use models to describe the dependencies of different methods in the external libraries.

Detour Template. As shown in Figure 5.4, EcoCov uses one instrumentation template for EMCP, *Method-redirect template*, to synthesize the angelic implementation for an external-method call. This template specifies how to redirect an external-method call to the angelic implementation using Moles. During synthesis, the hole `|method|` is filled with the name of the external-method call, and the hole `|type|` is filled its return type. EcoCov starts by obtaining the method name and the return type of the external method specified by the EMCP, and then fills the template with the obtained information, and adds the synthesized code for each method in the program under test $\mathcal{P}$.

Table 5.1: Subjects and their characteristics

Subject	Version	#Classes	#Methods	#KLOC
DSA	0.6	43	504	7.7
BBCode	5.0	27	192	1.9
xUnit	1.9.0	319	1138	12.5
Total	N/A	389	1834	22.1

5.4 Evaluations

We evaluate the effectiveness of EcoCov on three popular open-source projects (DSA, BBCode, and xUnit), answering the following research question: How effective is EcoCov in estimating the benefit of solving a problem (RQ5.1)?

5.4.1 Subjects and Evaluation Setup

Subjects. We use three popular open-source projects: DSA, BBCode, and xUnit in our evaluations. DSA implements basic data structures and algorithms. BBCode is a BBCode-Parser for .NET, which transforms any BBCode into HTML or into an in-memory syntax tree that can be further analyzed. xUnit is a widely used unit testing framework for .NET. Table 5.1 shows various characteristics of the subjects such as their version, the number of classes and methods. We choose these projects because they are very popular open-source projects. In addition, our work on test generation [205, 182] included some of them as subjects and found many OCP and EMCP problems in these projects. Thus, these projects pose sufficient problems for test generation when evaluating EcoCov.

Evaluation Setup. We use Pex [184] as the test-generation tool in our evaluations. We also run Covana along with Pex so that Covana can precisely identify OCPs and EMCPs encountered during test generation and report both the not-covered branches and corresponding problems.

To evaluate the effectiveness of the problem-benefit estimation (RQ5.1), we compare the estimates from EcoCov with the ground truth, i.e., the real benefits, obtained by solving the problem manually. In our evaluations, we use the newly-covered blocks as the benefit of solving a problem. We choose block coverage rather than branch coverage because Pex reports block coverage, and either block coverage or branch coverage is adequate to measure the effectiveness. We first run Pex on our subject projects and get the report from Covana. For an identified problem, we first manually solve this problem by either providing factory methods or mock objects. For an OCP, we create a factory method that encodes a sequence of method calls to modify the object state, which can be used by Pex to generate the desired object states. To simulate environmental dependencies, we create mock objects to replace external-method calls,

Table 5.2: Results on the number of problems being (in)accurately estimated

Project Assembly	**# Problem**	**# Match (%)**	**# FP (%)**	**# FN (%)**
DSA.Algorithms	1	1 (100.0%)	0 (0.0%)	0 (0.0%)
DSA.DataStructures	37	28 (75.7%)	6 (16.2%)	4 (10.8%)
DSA.Utility	3	3 (100.0%)	0 (0.0%)	0 (0.0%)
BBCode	8	5 (62.5%)	3 (27.5%)	0 (0.0%)
xUnit	7	6 (85.7%)	1 (14.3%)	0 (0.0%)
xUnit.Console	4	3 (75.0%)	1 (25.0%)	0 (0.0%)
xUnit.Utility	11	11 (100.0%)	0 (0.0%)	0 (0.0%)
Total	71	59 (83.1%)	11 (15.5%)	4 (5.6%)

enabling Pex to generate desired values for external-method calls [181]. We then run Pex again and mark the set of blocks that are newly covered after solving the problem. This set of blocks is the the ground truth, i.e., the real benefits, that we get.

To attain the results of applying EcoCov for RQ5.1, we repeat the same preceding process of attaining the ground truth, except that we use EcoCov to bypass the problems rather than manually solve the problems. We attain the estimated benefit as the set of newly-covered blocks in this process. Finally, against the ground truth, we calculate the precision and recall of our estimation for each problem. Note that to estimate the benefit for a problem, we select one problem to solve each time, leaving all the other problems unsolved.

5.4.2 RQ5.1: Effectiveness of Estimating Problem Benefit

Our evaluation results for RQ5.1 are summarized in Table 5.2 and Table 5.3. Table 5.2 shows in how many cases EcoCov can accurately estimate the problem benefit. Column "Project Assembly" shows each project's assemblies (i.e., executables or dlls) that contain problems. Column "# Problem" shows the number of problems used in our evaluation in each assembly. Note that we choose only the problems for the branches explored by Pex but not covered by Pex. Solving any of these problems allows Pex to explore other parts of the program and may encounter further problems. We do not use these further problems because we cannot afford to provide manual solutions to so many problems. Moreover, we exclude those problems whose solutions are too complex to solve manually. For example, some factory method requires creation of several more objects to pass the invariant validation. We also exclude object types that cannot be analyzed by Pex, such as `System.Type` from reflection libraries. Column "# Match (%)" shows the number of cases where EcoCov accurately estimates the problem benefit (i.e., produces the same coverage improvement as the ground truth), and the percentage of such cases.

Table 5.3 shows the average precisions and recalls of our estimations on the problem benefit. To calculate the precision and recall, let S_e be the set of **newly-covered** blocks estimated by

Table 5.3: Avg. precision and recall of our estimation

Project Assembly	**# Avg. Prec.**	**# Avg. Rec.**
DSA.Algorithms	100.0%	100.0%
DSA.DataStructures	94.0%	93.5%
DSA.Utility	100.0%	100.0%
BBCode	62.5%	100.0%
xUnit	98.8%	100.0%
xUnit.Console	75.0%	100.0%
xUnit.Utility	100.0%	100.0%
Average	91.1%	96.6%

EcoCov, and S_m be the set of **newly-covered** blocks that are the ground truth, i.e., achieved by the manually-provided solutions.. Then the blocks in the set $S_m \cap S_e$ are the true positives (TP), the blocks in the set $S_e - S_m$ are the false positives (FP), and the blocks in the set $S_m - S_e$ are the false negatives (FN). Columns "# FP (%)" and "# FN (%)" of Table 5.2 show the number of cases where EcoCov has false positives and false negatives of estimating the problem benefits. For each problem, the precision and recall for estimating the problem benefit is calculated as follows,

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

Table 5.2 and Table 5.3 together show that EcoCov is highly accurate in estimating problem benefits. As shown in Table 5.2, in most cases (59/71), EcoCov produces exactly the same coverage improvement as the manually-provided solution does. Such result shows that most of time EcoCov is capable of providing 100% accurate estimations. In addition, Table 5.3 gives information on how far away is the estimated benefit from the ground-truth benefit using precisions and recalls. Overall, EcoCov achieves 91.1% precision and 96.6% recall, showing the high accuracy of the estimation.

Although EcoCov achieves high accuracy in estimating problem benefits, it produces both false positives and false negatives for some cases. In total, there are 12 problems for which EcoCov fails to produce accurate estimations. We manually inspect these problems and find out that the inaccuracy is due to two major factors as follows.

Over-approximation. Over-approximation is the major cause to false positives. There are 8 out of the 12 inaccurate cases that due to the over-approximation nature of our angelic diagnosis. The choose operator used in angelic diagnosis actually replaces the problem-inducing element with a symbolic value. When the test-generation tools perform symbolic execution with this symbolic value, the tools can generate desired concrete values covering not only the target

```
1  public bool Remove(Node n)
2  { //List is not empty
3      if (head == null){ // B1
4          return false;
5      } ...
6      if (head == n) // B2
7        //Node to remove is the head
8        if (head.Next == null){ // B3
9          //Only head in the list
10         head = null;
11       }else{
12         //More than one node in the list
13         head = head.Next;
14       }
15   } ...
16 }
```

Figure 5.5: A simplified example of over-approximation

not-covered branch but also further not-covered branches after the target branch. However, the manually-provided solution may not be as flexible as the symbolic value so that Pex covers fewer branches with the manually-provided solution than with the symbolic value when exploring further code. In our evaluation, we find that such false positives are still acceptable: 4 out of these inaccurate 8 cases still have a precision higher than 80%.

Figure 5.5 shows a simplified example of the over approximation from the `Remove` method of `SinglyLinkedList` in DSA. The branch statement B1 at Line 3 first checks whether the list is empty. If yes, the method returns false without exploring the further part of the code. In this example, Pex is able to invoke only the default constructor of `List` to create an empty list. It cannot generate a non-empty list because the head field of the list cannot be directly modified. Covana reports the head field as the problem-inducing element of an OCP, along with the desired values that the head field is not null. With this report, EcoCov simply assigns a symbolic value to the head field, while developers typically provide a very simple factory method that creates a one-node list because a one-node list is sufficient to cover the false branch of B1 reported by Covana. With this factory method, Pex can move on into the true branch of B2 but cannot cover the false branch of B3 because the factory method can generate only a one-node list. However, since EcoCov turns the field `head` into a symbolic value, Pex can easily generate a head with the Next field not equal to null, and covers the false branch of B3. Thus, the false branch of B3 is a false positive reported by EcoCov.

Constraint Violation. EcoCov neglects the class invariants and environment constraints, which can result in both false positives and false negatives. The other 4 out of the 12 inaccurate cases are all caused by violating the relationship between two fields. Since EcoCov does not track

these relationships, when EcoCov assigns a symbolic value to one of the fields, the relationship may be broken, leading to an invalid object state. When Pex continues to explore the code with this invalid object, EcoCov may or may not produce false positives or false negatives, depending on whether the further executed code uses these two fields.

One example of breaking object relationships is from the `Heap` class in DSA, which has two fields: `array` and `count`. The relationship between them is that the field `count` should be equal to the size of the field `array`. However, in the method `Remove`, to cover a certain branch, EcoCov sets the value of `count` to a value that is not equal to the size of `array`. Later, when Pex continues to explore the further part of the code, Pex explores incorrect executions, and thus results in both false positives and false negatives. In this case, the precision and recall are both low. Hence, in future work, we plan to integrate such class invariants and environment dependencies (if available) into EcoCov to further improve our precision and recall.

5.5 Discussion

Object Dependencies for OCPs. Creating an instance of an object type T may require creating objects of several other object types. For example, the constructor of T may perform validation (such as invariant checking) on the input parameters that will be assigned to certain fields of T. If the validation fails, the constructor throws an exception to prevent creating an invalid object. In this case, our current approach simply creates a dummy object of T without invoking the constructor, which passes the null checking of the object. However, any subsequent field accesses of the dummy object result in exceptions, potentially causing the estimation to have false positives. In future work, we plan to investigate techniques to infer invariants of object types [59], and use the invariants to infer the dependencies among object types, enabling EcoCov to replace multiple fields with the choose operators.

Other Types of Problems. EcoCov focuses on the two major types of problems, OCPs and EMCPs, which prevent test-generation tools from achieving higher coverage [204, 116, 87]. There are three other example types of problems that cause certain branches not to be covered. First, the *infeasible-branch problem* refers to cases where certain branches may be infeasible to cover. For example, a branch may be unreachable (in the dead code), or the path constraints to take the branch under a specific context are not satisfiable. With angelic values that violate class invariants, EcoCov may cover an infeasible branch, causing false positives. In future work, we plan to investigate weakest-precondition computation to identify such branches. Second, the input-dependent-loop problem refers to cases where the iteration count of the loops depends on program inputs. Such loops can cause the program to have enormous or even an infinite number of paths, posing challenges for DSE tools to achieve high coverage, described in Section 4.2. In future work, we plan to investigate techniques to identify input-dependent loops that have side

effects on variables, and compute data dependencies of these variables to identify subsequent not-covered branches that use these variables [198]. Third, the constraint-solving problem refer to cases where constraint solvers cannot easily solve high-order computation or long constraints. In future work, we plan to investigate how to reason about constraint-solving failures and how they affect the coverage of the later branches.

5.6 Summary

Test-generation tools for complex programs in practice often encounter problems such as failing to deal with method calls to external libraries. To improve the effectiveness of test-generation tools, we have proposed cooperative testing, where developers help solve the encountered problems (e.g., providing mock objects). However, in practice, often a long list of problems can be presented to developers, and developers have limited time to solve all the problems. Thus, it is desirable to enable developers to maximize their testing goals within the given time of solving problems, such as achieving as high branch coverage as possible.

To enable such economical problem-diagnosis cooperation for test-generation tools, we have proposed an economic-analysis framework, EcoCov, that estimates the benefit of solving a problem. Our EcoCov framework includes our novel idea of *angelic diagnosis*, which approximates the ideal of solving a problem by instead computing an angelically nondeterministic value whose substitution for the problem-inducing element (e.g., the return of an external-method call) makes the problem disappear. EcoCov conducts diagnosis by running the test-generation tools on the program with the angelic values to bypass the problems, and answer questions for economic analysis. Based on EcoCov, our approach provides the problem-diagnosis interface to show the ordering of the problems based on their estimated benefits, extending the problem-diagnosis interface of precisely reporting problems.

We have shown the effectiveness of EcoCov by applying it to three open-source projects: xUnit, BBCode, and DSA. In our evaluations, EcoCov achieves average 91.1% precision and 96.6% recall in estimating the problem benefit.

CHAPTER 6

Behavior-Diagnosis Cooperation for Mobile Privacy Control

6.1 Introduction

Modern mobile-device platforms like iOS, Android, and Windows Phone provide a central place, called app stores or marketplaces, for finding and downloading third-party applications. A common problem faced by these mobile-device platforms is that the published applications in the marketplace may leak private user information through output channels. Many of these applications access mobile-device resources, such as pictures and GPS that may contain and expose private information, and share them using remote cloud services or web services without notifying users [72].

To mitigate these problems, privacy control mechanisms employed by mobile-device platforms include two major parts: (1) manual app validation by experts: experts employed by an app store manually exercise the functionality provided by an app and observe its behaviors for validation; (2) access-control granting by users: app stores ask for permissions before users can install applications (Android and Windows Phone), or an app requests permissions before it can access users' private information (iOS). The manual validation process is costly and delays publishing of apps. It is also incomplete, since it cannot examine every execution path to detect violations of privacy policies [91]. Access-control granting provides information about *what* private information these applications may access, rather than *how* these applications use private information, causing the users to make uninformed decisions on how to control their privacy. These privacy control mechanism lead to a situation where the users simply install applications

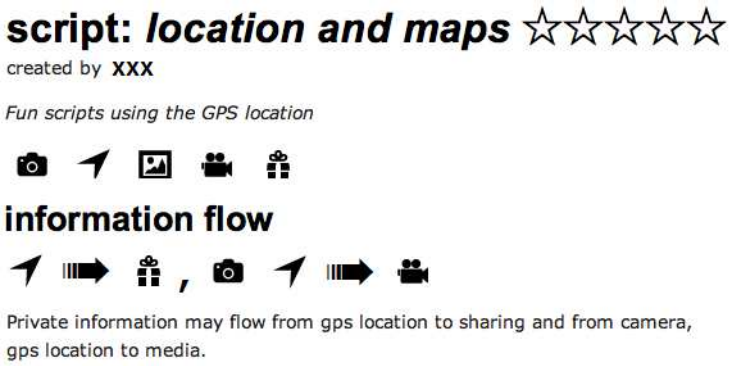


Figure 6.1: Information flow view of a sample script

without questioning the requested permissions, even if the applications may silently leak private information [80, 189, 72].

The goal of mobile privacy control is controlling mobile applications' accesses to the users' privacy-sensitive information, i.e., the users making decisions on whether to allow or deny permissions that protect accesses to certain privacy-sensitive information. To improve mobile privacy control of these mobile platforms, we propose an approach, called user-aware privacy control, that explains the uses of permissions, improving the users' understanding of applications' privacy-sensitive behaviors and reducing efforts for app validation and access-granting. Unlike existing mobile platforms that merely show what privacy-sensitive information the mobile applications request to use (i.e., what permissions requested by the mobile applications), our approach further explains how each permission is used by the application.

To explain the permission uses, our approach automatically computes information flows of private information via static analysis and visualizes the flows, as shown in Figure 6.1. These information flows show what private data types flow to what output channels, explaining how permissions are used by applications. We use the term *Source* to refer to an origin of private information and *Sink* to refer to a point where information may leak from an app. The example in Figure 6.1 shows that the app uses 5 capabilities (Camera, Location, Pictures, Media, and Sharing). Among these, the first 3 are sources, and the last two are sinks. Among the 6 possible flows (3 sources to 2 sinks), our analysis shows that the Location flows to the Sharing sink, and that Camera and Location flow to the Media sink.

Given the computed information flows, our approach employs the mechanism of user-driven access control [159]. When the application is executed for the first time, our approach allows users to choose among *real* information, *anonymized* information, or *abort* execution, as shown in Figure 6.2 (the *abort* option is not yet implemented in TouchDevelop). These settings provide flexible choices for users: (1) using anonymized information (e.g., a fixed picture or a fixed

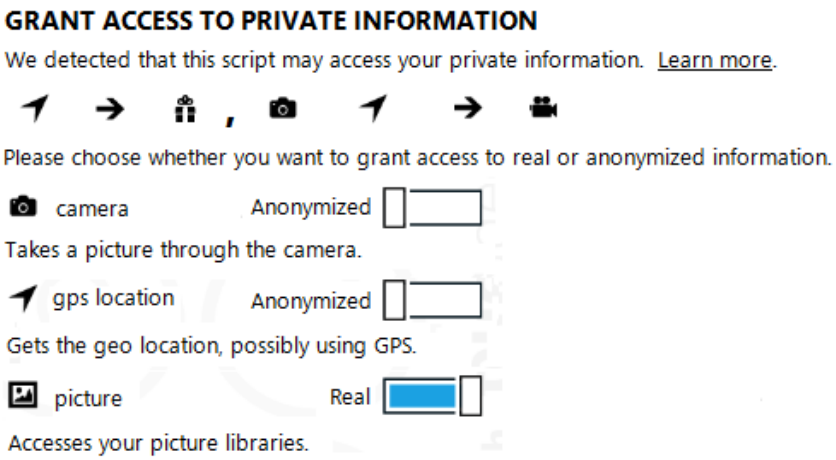


Figure 6.2: Grant access to private information

geolocation), users can experiment with applications before granting access to real information; (2) aborting an execution prevents unintended access to a resource and is helpful for diagnosis.

In addition, our approach allows the users to perform runtime inspection on the information that is to be sent out from the mobile device. Before private information protected by a permission escapes from the mobile device, an explicit dialog is presented to request the user's permissions. In Myer's terminology [35], this dialog corresponds to a declassify step and tampered data has low integrity. For example, in TouchDevelop [186, 18], the sharing of a picture taken directly from the `camera` shows a dialog for the users to review the picture before it leaks from the device. Such information flows do not leak private information without notifying the users and should be safe. We refer to such sinks as *monitored* sinks. Since the users get to review the information at runtime, information flows that flow to *monitored* sinks do not require users' decisions at installation time, reducing users' efforts in granting accesses.

However, information flows whose information are tampered with before flowing to the monitored sinks may escape users' inspection. For example, a malicious app could encode the user's phone number into the color intensity of some pixels inside a picture to be shared. The information flow will reveal that private information from the `camera` and `contact` sources flow to the `share` sink, but a user may be hard pressed to recognize any changed pixels in the picture being posted. Moreover, information flows that flow to *non-monitored* sinks can leak information without notifying users as well.

To address such challenges, our approach further classifies information flows based on a tamper analysis. We define a policy to classify information flows as safe or unsafe: an information flow is safe if only *untampered* private information flows to a *monitored* sink. Our analysis detects unsafe flows by observing whether the information is tampered with before reaching the

sinks, and whether the information flows to a non-monitored sink. Based on the safe/unsafe classification of flows, our policy is to use real information for sources only appearing in safe flows, and anonymized information for all other sources.

Our user-aware privacy control approach strives for a balance between security and user involvement. By employing user-driven access control, our approach ensures that apps gain permissions from the users for private information accessed by apps. To avoid overwhelming the users with access granting—which may annoy the users and cause the users to blindly grant every permission—our approach does not ask the users to grant access to private information accessed by an app but not flowing to sinks. Furthermore, our technique provides default settings that are safe to run a script without further user decisions, thereby reducing risk and user burden.

Based on the computation of information flows, our approach provides a behavior-diagnosis interface that shows information flows of the requested permissions. Such a larger scope of information, i.e., information flows, explains how the users' privacy-sensitive information is used by the mobile applications, enabling the behavior-diagnosis cooperation for mobile privacy control. In this cooperation, the tools identify permissions requested by mobile applications for the users to inspect and explain the permissions by showing information flows. Based on the users' domain knowledge about the mobile applications' functionality and security requirements, the users inspect the information flows to determine whether the information flows are expected for the mobile applications, and make decisions on granting accesses for the permissions.

We built a prototype of our privacy control into TouchDevelop, a novel mobile platform that enables the users to write apps directly using touch screens. In TouchDevelop, apps are written using a scripting language that is expressive enough to create applications or games, utilizing most features of mobile devices [185]. We call apps written in TouchDevelop "scripts". Users can publish their scripts in a "script bazaar", where other users can install and run them on their own devices. TouchDevelop is thus similar to other mobile-device platforms, except that we use no manual validation, only automatic information flow analysis. Our approach works well with the TouchDevelop platform for several reasons: (1) all code is made available through the script bazaar as source; (2) the expressiveness of the language enables apps to be created in fewer lines, allowing efficient static analysis on whole scripts; (3) the language does not allow reflection, `eval`, or native calls to platform APIs, making code analysis easier [108].

6.2 TouchDevelop Language

TouchDevelop allows users to create applications using an imperative and statically typed language [185]. A TouchDevelop script consists of a number of actions (procedures) and global variables. The body of actions consists of: (1) expressions that either update local or global

```
1 action foo() : Nothing {
2     var s := "unclassified";
3     var p := media->create picture();
4     var loc := senses->current location; // classified
5     s := loc->describe(); // classified
6     p -> draw text(s); // p's mutable state is classified
7     p -> share("facebook");
8 }
```

Figure 6.3: Example of classified information flow

variables (assignments), invoke another action, or invoke a predefined property; (2) conditional statements *if*-then-*else*, (3) loop statements, *for*, *while*, and *foreach*, that iteratively execute a block of statements. The global variables are statically typed and their current value is persisted and accessible across multiple script invocations.

As a statically typed language, TouchDevelop defines a number of data types (e.g., `Number` or `String` for s, or `Picture` for p in Figure 6.3). Each data type provides a number of properties (e.g., $p \rightarrow$ `share`). For the sake of the simplicity, the language does not provide features that allow users to define new types or properties.

6.2.1 Classified Information Flow

In this section, we illustrate several examples to show how scripts written in TouchDevelop may leak private information (referred to as *classified information*). Figure 6.3 shows an example of how classified information flows among values, such as `Number` and `String`. At line 4, variable `loc` becomes classified since it contains the geolocation information obtained via the GPS. Here, we refer to the property `senses->current location` as a *Source* of geolocation information. At line 5, the location is transformed into a string and assigned to `s`, thereby making `s` classified. At line 6, the location string `s` is rendered as text into the picture `p`, causing `p` to be classified. At line 7, the `share` action of `p` leaks the classified information of the user's geolocation to facebook. Here we refer to the property `share` as a *Sink*. One thing to note is that if line 5 were moved to after line 6, then `p` would not be classified. The later update of `s` would not affect `p`.

Now let's look at another example shown in Figure 6.4. At line 5, the message `msg` is added to the message collection `msgs`. The message collection `msgs` keeps a reference to `msg`, which means that `msg` can be accessed from `msgs` at a later time. At line 6, `msg` becomes classified, which causes `msgs` to be classified indirectly. At line 7, `msg2`, the `i`-th message in `msgs`, may contain the information of `msg` or other messages. Thus, `msg2` should also be considered as classified. We refer to this type of information flow as reference-type flow, since it occurs through objects such as message collections that contain references to other objects.

```
1 action foo(msg : Message, msgs: MessageCollection, i: Number) : Nothing {
2     var pic := senses->take camera picture;
3     pic->share('facebook','share a pic');
4     var s := currLoc(); // classified
5     msgs->add(msg);
6     msg->set message(s); // classified
7     var msg2 := msgs->at(i); // classified via reference-type flow
8     msg2->share('facebook');
9     var y := false;
10    if s->contains('Seattle') then {
11        y := true; // classified via implicit flow
12    }
13 }
14
15 action currLoc() returns r : String{
16    var l := senses->current location; // classified
17    r := locations->describe location(l); // classified
18 }
```

Figure 6.4: Implicit and reference-type information flow

Another type of information flow that can potentially leak private information is implicit flow [63, 64]. Implicit flow arises from conditional control structures such as *if* statements where the condition depends on classified information. The statements in the branches of the conditional statement can leak the outcome of the condition, which allows later code to determine the classified information indirectly. Consider the example of implicit flow shown in Figure 6.4. The classified local `s` is used at the *if* statement at line 10. By observing the values of `y`, users can guess whether the geolocation information stored in `s` contains the substring `Seattle`. Thus, to track implicit information flows, we need to consider `y` as classified.

6.3 Capability Identification

The application capabilities tell users what kinds of mobile-device resources (such as personally-sensitive information and wireless network) an application uses, which is useful information for users to decide whether to install the application. These resources can be classified as sources (such as camera or geolocation) and sinks (such as web or facebook sharing). To use these resources, application developers need to use the APIs provided by the device-specific development environment, also called software development kit (SDK). Table 6.1 shows the kinds of sources and sinks provided by the TouchDevelop APIs. Among these sinks, the sink *Sharing* prompts users with the sharing information, which makes it a monitored sink. For the other three kinds of sinks, only the sink *Web* is considered as an non-monitored sink. The reason is that the pictures from the sink *Picture* and emails or phone numbers from the sink *Contacts*

Table 6.1: Capabilities provided by the TouchDevelop APIs

	Capability	Description
Source	Camera	Takes a picture through the camera.
	Location	Gets the geo location, possibly using GPS.
	Picture	Accesses the picture libraries.
	Music	Accesses the music library.
	Microphone	Accesses the microphone.
	Contacts	Accesses emails or phone numbers of contacts.
Sink	Contacts	Saves an email or phone number of a contact to the device.
	Media	Saves pictures to the phone.
	Sharing	Share information through social services, email or short messages.
	Web	Accesses the web, downloading or uploading data.

are all considered as sensitive private information, and if these kinds of private information would flow to *Web*, our approach would identify the flow as an unsafe flow.

Automated Capability Identification. To provide the accurate and complete information of what resources are accessed by applications, our approach provides a static analysis that scans through the application script to automatically identify application capabilities. We have manually annotated all TouchDevelop APIs with source and sink information. We use a fixpoint algorithm to compute the capabilities used by each action of a script. For each action in a script, our approach parses the action into an abstract syntax tree (AST), and automatically scans each statement node in the AST to identify what sources and sinks are used. If a statement in an action a_1 is a call to another action a_2, our approach adds the sources and sinks of a_2 to a_1. A fixpoint is reached if the computed sources and sinks for each action do not change. Since application developers in TouchDevelop can use only the APIs provided by the device-specific SDK for accessing mobile-device resources, our analysis results are guaranteed to be accurate and complete.

6.4 Information Flow Analysis

In this section, we first present an overview of our static information flow analysis, and then follow it up with full technical details.

6.4.1 Overview

Our approach statically computes information flows using abstract interpretation [58]. Our approach maintains the abstract state of the script and updates the state according to the simulated execution of a statement. The state maps local variables to sets of sources. In addition it maps a single mutable location for each kind[1] to a set of sources. Finally, the state maps *sinks* to sources flowing to that sink. Sinks can be thought of as additional mutable locations that accumulate what flows into them. Information flow from a source s_1 to a sink s_2 arises whenever source s_1 appears in the abstract state of sink s_2. The sources in our maps are represented as a set of value elements consisting of constant sources and input parameter names. Input parameter names are used to represent symbolic information that allows us to determine where parameters flow.

Implicit Flows. In order to handle implicit flow arising from control flow statements that branch on classified information, we use an additional special local variable named `pc`. The `pc` variable is assigned (augmented) with source information at conditionals at the entry of both branches. At each basic block, the `pc` is defined by the value of `pc` at the immediate dominator block instead of all predecessor blocks as is the case for normal locals.

Inter-Procedural Analysis. Our approach uses a fix-point algorithm to iteratively compute the summaries of basic blocks in an action and then uses these summaries to compute summaries of actions. At call-sites, summaries are instantiated with concrete values for symbolic parameter names, thereby computing the effect of the call without re-analysis of the action. This approach also handles recursive actions.

Mutable and Immutable Values. We map the TouchDevelop concepts to a simpler model for information flow analysis. We can think of each kind of value as having two separate parts: (1) *an immutable part*, and (2) *a mutable part*. Many types of values have only an immutable part and no mutable parts, e.g., `Number`, `String`, and `GeoLocation`. Other types of values have both immutable parts and mutable parts. E.g., `Picture` has an immutable part that is associated with whether the picture is valid (i.e., whether the pointer is null). The mutable part of a picture consists of the actual pixel colors at each coordinate of the picture.

We track information flow separately for the mutable and immutable parts of values. The immutable part of an object is copied whenever a value is assigned from one local to another, passed as parameter, returned from a method, stored or loaded from a global variable. The immutable part of a value is tracked precisely at each program point and assignments are strong assignments that replaces the original values.

The mutable part of an object is affected only by pre-defined property invocations (i.e., primitive methods). We track the mutable part of values using an abstraction where we have a

[1] Data types in TouchDevelop are called kinds.

single mutable location per kind. Every value of that type shares that same mutable location in the analysis. All updates to the mutable part are weak updates, meaning they are accumulated.

Primitive properties are annotated with information that indicates from which parameters (and thus which kinds) the mutable state is read, and also what mutable parts are written (parameters and return values).

Embedded References. Because values may have embedded references to other values that could be mutable, we also keep track of such embedded references using directed edges from one mutable location to another. The model currently does not accommodate references from immutable parts to mutable parts, but we have not found a need for that. Establishing a reference from one value to another implies a write to the mutable state of the first.

Globals. To simplify the description in the remainder of the chapter, we eliminate global variables from the model. Global variables are treated as extra parameters and return values from each action. One can easily transform a program with globals to a program without globals by adding all globals used in an action (and actions called) as extra parameters, and all globals modified by an action as extra return values. As a result, inside an action, accessing a global is no different than accessing a local variable. We will thus no longer explicitly talk about global variables henceforth.

Parameters. Parameters of an action are treated as ordinary locals inside an action. They are pre-initialized by the action invocation, but otherwise act no differently than normal local variables.

Results. Result variables are treated as ordinary locals inside an action. Upon return, their immutable parts (values) are copied to the caller's locals that receive the results of the invocation.

6.4.2 Simplified Language

We assume that our input program consists of a number of actions, where each action has any number of parameters and any number of results. The body of an action consists of a control flow graph of basic blocks, with a distinguished entry block and a distinguished exit block. Conditionals branching on condition `c` are transformed into non-deterministic branches to the `then` and *`else`* blocks, where the target blocks are augmented with a first instruction of the form `assume(c)` and `assume(`*`not`*` c)`.

The instructions inside a block have the following forms:

$$\begin{aligned} \mathit{Instruction} ::= \;& x := y \mid r := p(x_1..x_n) \\ \mid\; & r_1..r_n := a(x_1..x_m) \mid \mathit{assume}(x) \mid \mathit{assume}(\neg x) \end{aligned}$$

An instruction is either a simple assignment from one local to another, a primitive property invocation of parameters $x_1..x_n$ binding the result to a variable r, an action invocation with parameters $x_1..x_m$ binding the results of the action to $r_1..r_n$, or a special *assume* statement arising from conditional branches. We assume that primitive operations always return a value, even if it is the `Nothing` value.

6.4.3 Summaries of Basic Blocks and Actions

We separate the state into three parts: 1) local variable information, 2) pc information for implicit flow, and 3) mutable state information. The first two are program point specific, but the mutable state is not. The mutable state consists of one classification per kind, and a set of edges between kinds representing possible references from the mutable state of objects of one kind to objects of another kind.

$$
\begin{aligned}
\mathit{Atom} &::= \mathit{Sources}(i) \mid \mathit{Parameter}(i) \mid \mathit{PC}_{in} \\
\mathit{Classification} &::= \mathit{Set\ of\ Atom} \\
\mathit{LocalMap} &::= \mathit{Block} \to \mathit{Local} \to \mathit{Classification} \\
\mathit{SinkMap} &::= \mathit{Block} \to \mathit{Sink}(i) \to \mathit{Classification} \\
\mathit{PCMap} &::= \mathit{Block} \to \mathit{Classification} \\
\mathit{MutableState} &::= \mathit{Kinds}(i) \to \mathit{Classification} \\
\mathit{References} &::= \mathit{Set\ of}(\mathit{Kinds}(i) \times \mathit{Kinds}(i))
\end{aligned}
$$

The fixpoint computation computes the following data structures:

$$
\begin{aligned}
L_{pre}, L_{post} &: \mathit{LocalMap} \\
\mathit{PC}_{pre}, \mathit{PC}_{post} &: \mathit{PCMap} \\
S_{pre}, S_{post} &: \mathit{SinkMap} \\
M_{pre}, M_{post} &: \mathit{Block} \to \mathit{MutableState} \\
R_{pre}, R_{post} &: \mathit{Block} \to \mathit{References}
\end{aligned}
$$

L_{pre} contains the local information on entry to a particular block, whereas L_{post} contains the corresponding information at exit of the block, and similarly for PC_{pre} and PC_{post}. The sink maps S_{pre} and S_{post} contain the classification of the predefined sinks on entry and exit of blocks. M_{pre} and M_{post} contain the mutable state classification and R_{pre} and R_{post} contain the reference links between mutable states.

Block Summary

We initialize L_{pre} for entry blocks of actions to map each parameter local i to the singleton $\{Parameter(i)\}$ and to the empty set for all other locals. Similarly, we initialize PC_{pre} for entry blocks to the singleton $\{PC_{in}\}$ which allows computing symbolic summaries of actions that can be applied in contexts where the PC is classified differently. The sink map S_{pre} for the entry block is empty. These maps will not change during the global fix point of the analysis.

The information for R_{pre} and M_{pre} for the entry block keep track under which assumptions the action has been analyzed. It is initially empty, but may grow as the action is invoked in a context with larger M or R, causing the blocks of the action to be re-analyzed.

For non-entry blocks, the starting state is defined as follows:

$$
\begin{aligned}
L_{pre}(b) &= \bigsqcup_{b' inpred(b)} L_{post}(b') \\
S_{pre}(b) &= \bigsqcup_{b' inpred(b)} S_{post}(b') \\
M_{pre}(b) &= \bigsqcup_{b' inpred(b)} M_{post}(b') \\
R_{pre}(b) &= \bigcup_{b' inpred(b)} R_{post}(b') \\
PC_{pre}(b) &= PC_{post}(dom(b))
\end{aligned}
$$

The locals on entry to a block are simply the union of the post local state of all predecessor blocks, where union is defined point-wise on the map (similarly for the sinks, mutable state, and reference links). For the PC classification is obtained by the post PC classification of the immediate dominator of block b.

Action Summary

We assume each action has a single exit block. The summary of an action is simply the post state of the exit block of the action. For each action, we keep track of the initial M and R under which it was analyzed in the information for its entry block. If we see a call to the action with

a larger M or R, we update that information for the entry block and propagate the changes through the blocks of the action. For example, the summary of action `foo` in Figure 6.4 is:

$$
\begin{aligned}
State = \{ & \\
& L = \{s \rightarrow \{\textbf{Location}\}, pic \rightarrow \{\textbf{Camera}\}, \\
& \quad y \rightarrow \{\textbf{Location}\}, msg \rightarrow \{\textbf{Location}\}, \\
& \quad msg2 \rightarrow \{\textbf{Location}\}\}, \\
& S = \{\textbf{Sharing} \rightarrow \{\textbf{Camera}\}\}, \\
& PC = \{\}, \\
& M = \{Picture \rightarrow \{\textbf{Camera}\}, \\
& \quad Message \rightarrow \{\textbf{Location}\}\} \\
& R = \{< MessageCollection, Message >\} \\
\} &
\end{aligned}
$$

Here the state of locals L shows that the local `s` contains the geolocation data, `pic` contains the camera data, `y` contains geolocation data due to the implicit flow from `s` to `y`, and the local `msg` gets geolocation data from `s` at line 5. The state of mutable locations M shows that the mutable state of `Picture` contains the camera data and the mutable state of `Message` contains the geolocation data. The state of references R contains a pair showing that `MessageCollection` is linked to `Message`. Due to this link, `msg2` reads the mutable data of `msgs` and is considered to contain the geolocation data. The state of sinks S shows that the sharing sink contains camera data. The set PC is empty, since the pc does not carry the camera data after the *if*-`then`-*else* block.

6.4.4 Classified Information Propagation

In this section, we describe how APIs are annotated and how information flow is tracked at the instruction level.

Property Annotations

We assume that every primitive property p is annotated with a set $\textbf{ReadsMutable}_p$ consisting of the parameter indices of parameters whose mutable state is read by p. Similarly, the set $\textbf{WritesMutable}_p$ consists of the indices of parameters whose mutable state is written by p. Additionally, we use index 0 in $\textbf{WritesMutable}_p$ to indicate whether the mutable state of the result depends on the classification of the inputs to property p. By default, we assume that all immutable parts of all parameters are read by a property and that all read parts flow into the

result's immutable part. Additionally, the set $\mathbf{EmbedsLinks}_p$ contains the set of edges between kinds representing possible references established by invoking property p.

A set $\mathbf{Sources}_p$ indicates which predefined sources flow into the result value when invoking property p. Finally, $\mathbf{Sinks}_p$ contains the set of sinks to which information flows on invoking p.

Statement-Based Propagation

The following rules show the propagation of the state for each kind of instruction. We assume L, PC, M and R are the initial states, and L', PC', M' and R' are the post states.

Case $x := y$**:**

$$\begin{aligned} L' &= L[x \mapsto L(y) \cup PC] \\ PC' &= PC \\ M' &= M \\ R' &= R \\ S' &= S \end{aligned}$$

Note how the PC classification flows into the new classification of x. This is needed to keep track of implicit flow.

Case $r := p(x_1..x_n)$**:** First we compute the input classification, which consists of the classification of all input parameters, the classification of all kinds for which there is a parameter annotated with **ReadsMutable**.

$$\begin{aligned} Common = PC \cup \mathbf{Sources}_p \cup \bigcup_i L(x_i) \\ \cup \bigcup_{j \in \mathbf{ReadsMutable}_p} Cl(M, R, kind(x_j)) \end{aligned}$$

The helper function $Cl(M, R, i)$ computes the union of the classification of all kinds j reachable from i via edges in R. Note that $Reach(R, i, i)$ is true for all R.

$$Cl(M, R, i) = \{M(j) \mid Reach(R, i, j)\}$$

With this information, we update the result and the mutable state.

$$
\begin{aligned}
L' &= L[r \mapsto \mathit{Common}] \\
PC' &= PC \\
M'(i) &= \begin{cases} M(i) \cup \mathit{Common} & \text{if } \exists j \in \mathbf{WritesMutable}_p \\ & \text{and } \mathit{Reach}(R, \mathit{kind}(x_j), i) \\ M(i) & \text{otherwise} \end{cases} \\
R' &= R \cup \mathbf{EmbedsLinks}_p \\
S'(i) &= \begin{cases} S(i) \cup \mathit{Common} & \text{if } i \in \mathbf{Sinks}_p \\ S(i) & \text{otherwise} \end{cases}
\end{aligned}
$$

Case $\mathit{assume}(x)$ **or** $\mathit{assume}(\mathit{not}\ x)$**:**

$$
\begin{aligned}
L' &= L \\
PC' &= PC \cup L(x) \\
M' &= M \\
R' &= R \\
S' &= S
\end{aligned}
$$

Assume statements cause the PC classification to be augmented with the classification of the condition.

Case $r_1..r_n = a(x_1..x_m)$**:** First, we update $M_{pre}(\mathit{entry}_a)$ to $M \sqcup M_{pre}(\mathit{entry}_a)$ and $R_{pre}(\mathit{entry}_a)$ to $R \sqcup R_{pre}(\mathit{entry}_a)$. If necessary, propagate changes through blocks of a. We use the state at the exit block of a as the summary of a to be applied at the current invocation. Since the summary contains some symbolic information for parameter classification and pc classification, we first instantiate the exit block information with the invocation site information. Let σ be the substitution

$$\sigma = [PC_{in} \mapsto PC, \mathit{Parameter}(i) \mapsto L(x_i)]$$

Now we compute instantiated versions of the exit block summaries:

$$
\begin{aligned}
L_s &= \sigma(L_{post}(\mathit{exit}_a)) \\
M_s &= \sigma(M_{post}(\mathit{exit}_a)) \\
R_s &= \sigma(R_{post}(\mathit{exit}_a)) \\
S_s &= \sigma(S_{post}(\mathit{exit}_a))
\end{aligned}
$$

Note that no PC information flows out of the action. Let $r'_1..r'_n$ be the result locals in action a. The final states after the invocation of action a is then:

$$\begin{aligned} L' &= L[r_i \mapsto L_s(r'_i)] \\ PC' &= PC \\ M' &= M \sqcup M_s \\ R' &= R \cup R_s \\ S' &= S \sqcup S_s \end{aligned}$$

6.5 Tampered Information

The source to sink information flow we compute so far may not be enough to make good policy decisions about which scripts are good and which scripts are bad. For example, a script taking a picture with the camera and then posting it to facebook may be a reasonable script, especially since posting to facebook will prompt the user and display the text and picture that will be posted. The user thus has a way to *vet* the information being posted.

However, a malicious script could try to encode the user's phone number into the color intensity of some pixels in the posted picture. From an information flow perspective, we would simply see that sources Camera and Contacts flow to Sharing. Users looking at the picture being posted will likely not notice changed pixels containing the hidden phone number.

Can we distinguish somehow between these two cases? Our attempt to do so is based on the following assumption: for sinks that prompt the user to review the information (e.g., emails, sms, phone calls, facebook posts), we want to distinguish if the information being posted is recognizable by the user as containing sensitive information or not. In the case where pixels in the picture taken by the camera are modified based on classified contact information, we want to consider the information in the picture as tampered and thus apply a harsher policy than if the information is not tampered with.

In order to track tampering, we introduce an operator *Tamper* that can be applied to the existing sources.

$$\begin{aligned} Atom ::= & \; Sources(i) \mid Parameter(i) \\ & \mid PC_{in} \mid Tamper(Atom) \end{aligned}$$

Note that the set of atoms is not unbounded, as this is not a free algebra. Indeed, $Tamper(Tamper(s)) = Tamper(s)$ for all s. Additionally, we annotate all properties p with a single bit $\mathbf{Tampers}_p$, in-

dicating whether any input classifications are transformed into tampered output classifications for the result and writes to the mutable store.

The rule for handling the flow at property invocations then needs to be modified insofar as the classification *Common* now becomes:

$$InFlow = PC \cup \bigcup_i L(x_i) \cup \bigcup_{j \in \mathbf{ReadsMutable}_p} Cl(M, R, kind(x_j))$$

$$Common = \mathbf{Sources}_p \cup \begin{cases} InFlow & \text{if } \neg\mathbf{Tampers}_p \\ Tamper(InFlow) & \text{if } \mathbf{Tampers}_p \end{cases}$$

Applying *Tamper* to an entire classification, just means applying the operator pointwise to the set elements.

6.6 User-Aware Privacy Control

By applying the static analysis, we compute information flows on a per action and per script basis and show summaries of which sources flow to which sinks in each action and in the script as a whole. As an example, Figure 6.1 shows the summary of the script named *location and maps*, which can send a text message containing the user's current location or take a picture with the user's current location embedded in it and save the picture into the media storage library of the mobile device. This flow summary shows the information flows of the application: by looking at the information flows at install time, users can understand what private information the application uses and where this private information may escape to. To minimize the efforts of experts in validating applications and users in granting accesses to sources, we further define a policy that classifies flows into safe and unsafe flows.

Classification of Safe and Unsafe Flows. Our policy is based on the assumption described in Section 6.5: we consider a flow as a *safe flow* if it is an untampered flow to a *monitored sink*. Recall that a monitored sink results in an explicit dialog at runtime, presenting the particular information flowing to the sink and requesting permissions from the user before the information escapes from the mobile-device. For example, a post to facebook would prompt the user to review the information before the actual sharing happens. Our approach considers all other flows as unsafe, including untampered flows to non-monitored sinks (Web) and all tampered flows. We may evolve the policy of what constitutes a safe flow based on user feedback, and update the policy when more sources and sinks are added into the system.

Granting Accesses. When running the script for the first time, the user is presented with all sources appearing in unsafe flows along with a radio button group for each source that allows

the user to choose among *anonymized* or *real* information (Figure 6.2). Anonymized information means that the runtime provides the script with anonymized information (a fixed picture or a fixed geolocation etc.), real information means the script gets access to the real information on the users' device, and abort execution means that the runtime stops the execution at the access point. By using anonymized information, a user can safely experiment with an application to determine if it does something useful prior to even considering whether to allow access to real information.

Default Settings. To keep users safe and minimize efforts in granting access, our approach provides default settings. We guarantee that running a script with the default settings does not leak private information, except through monitored sinks where the user is presented untampered information to review. Sources appearing in no flows use real information and are not shown. For sources that appear only in safe flows, the default setting is to use real information; for other sources appearing in flows, the default setting is to use anonymized information.

6.7 Evaluation

This section presents experiments we conducted to evaluate the effectiveness of our extended static information flow analysis. We chose TouchDevelop as a platform for our evaluations due to two major reasons: (1) **source code availability**: the source code of a script is made available as part of the publishing process; (2) **simplicity**: the expressiveness of the TouchDevelop languages enables applications to be created in much fewer lines, reducing the complexity of static analysis; the TouchDevelop language does not allow reflection or native calls to platform APIs, enabling complete annotation of the APIs with source, sink, and flow information; TouchDevelop allows importing of external scripts through only the script bazaar and does not allow generating code at runtime.

6.7.1 Subjects and Evaluation Setup

We integrated our static information flow analysis into the server part of the TouchDevelop environment. Every submitted script is analyzed automatically and the resulting flow information informs the privacy settings when users install scripts. To conduct the experiments, we collected 546 scripts (all publications prior to Oct 6th, 2011) published by 194 TouchDevelop users, excluding scripts published by ourselves. Figure 6.5 shows the number of scripts in different ranges of lines of code (LOC) [2] and the average LOCs in these ranges. Among these scripts, 395 (72.34%) scripts have LOCs ranging from 0-80, and the scripts *Termini 3 Final*[3] and *Ter-*

[2]Meta data and comment statements are excluded for LOC computation.
[3]http://touchdevelop.com/pycw

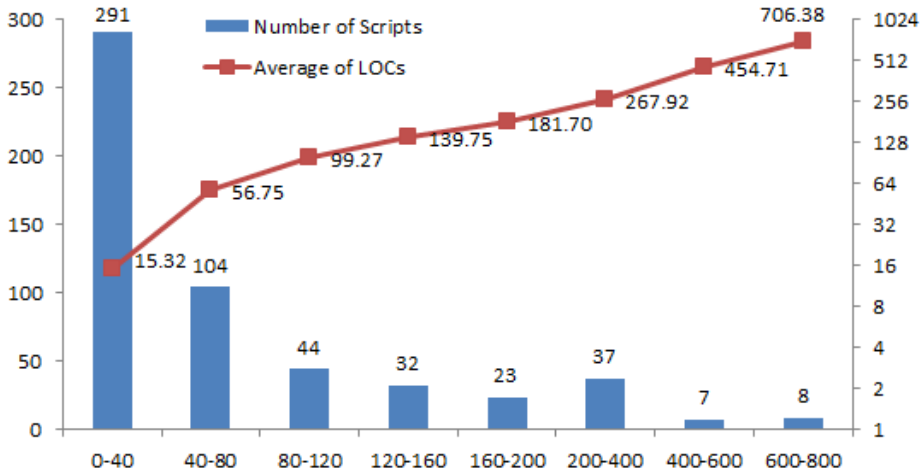


Figure 6.5: Sizes of 546 published scripts in TouchDevelop

mini 3 Beta 1.4[4]) have the maximum LOCs of 738. The major reason why these scripts are of relatively small size is that the expressiveness of the TouchDevelop language enables users to create applications using fewer lines of code than using traditional programming languages for mobile devices. For example, the script *Termini Include Edition 1.0.2*[5] published by the user Pouya Animation[6] creates a UNIX emulator (Terminal) for TouchDevelop in just 407 LOC.

6.7.2 Information Flow Evaluations

To show the effectiveness of our information flow analysis, we posed the following three research questions about the 546 subject scripts:

- **RQ6.1**: What is the advantage of using information flow from sources to sinks to classify scripts, as opposed to the mere presence of both sources and sink (capability usage)?
- **RQ6.2**: How many more scripts can we classify as safe using our tamper analysis, thus eliminating the need to ask users to grant access?
- **RQ6.3**: How many more sources can we classify as safe using our tamper analysis, further reducing the number of sources that require users' decisions?

[4]http://touchdevelop.com/xwgl
[5]http://touchdevelop.com/hllw
[6]https://www.touchdevelop.com/ntqe

Table 6.2: Information flow summary of 546 published scripts

# **Total**	# **Cap** (242)			# **Flow**
	/w **Source**	/w **Sink**	/w **Both**	
546	172	159	89	78

RQ6.1: Information Flow Summary

To address RQ6.1, we compare the number of scripts that are classified as information-leaking using information flows with the number of scripts that are classified as information-leaking using capabilities. Table 6.2 shows the information flow summary of the published scripts. Column "# Total" shows the total number of scripts. Column "# Cap" shows the number of scripts that either have at least one source or one sink. Column "/w Source" shows the number of scripts that have at least one source. Column "/w Sink" shows the number of scripts that have at least one sink. Column "/w Both" shows the number of scripts that have both sources and sinks. Column "# Flow" shows the number of scripts that have computed information flows.

The results show that in 546 published scripts, 242 (44.32%) either have sources (access private information) or have sinks (can leak information from the script). To form an information flow, a script must have at least one source and one sink. As shown in Table 6.2, 457 (83.70%, #Total - #Both) scripts have either no sources or no sinks, which can be classified as non-information-leaking by either using information flow or capabilities usage. For the remaining 89 scripts that have both sources and sinks, our information flow analysis detects that 11 scripts have no information flows. Thus, using potential flow (presence of both source and sink), reduces prompting by 48.26% (from 172 to 89) over the traditional capability approach (presence of sources). Using actual information flows, as computed by our analysis, further reduces prompting by 12.36% (from 89 to 78).

Table 6.3 shows the information flow summary of the published scripts based on source-sink pairs. Each column represents a kind of sink and each row represents a kind of source. The first number in each table cell is the number of scripts for which our analysis determines information flow from the given source to the given sink, whereas the second number in each cell is simply the number of scripts that use the corresponding source and sink. The two numbers presented in each table cell compare our approach of computing actual information flow, to a naïve capability analysis that simply presumes an information flow for each used source-sink pair.

For example, the cell for Camera and Web shows that a naïve capability approach would classify 30 scripts as having information flow from the camera to the web, whereas our information flow analysis proves that none of these scripts actually leak camera information to the

Table 6.3: Information flow vs. source-sink pairs

	Contacts	Media	Sharing	Web	*Any*
Camera	0 / 0	22 / 22	11 / 21	0 / 30	33 / 36
Contacts	1 / 3	0 / 11	30 / 39	0 / 19	30 / 41
Location	0 / 0	6 / 10	12 / 12	27 / 29	30 / 34
Microph.	0 / 0	0 / 2	1 / 1	0 / 3	1 / 6
Music	0 / 0	0 / 1	1 / 1	0 / 1	1 / 3
Picture	0 / 0	18 / 29	2 / 15	14 / 21	24 / 32
Any	1 / 3	29 / 39	44 / 48	40 / 51	78 / 89

web, completely removing the concerns of leaking pictures taken from the camera through the web.

Similarly, the naïve capability approach would consider 19 scripts to leak contact information through the web, while our analysis shows that none of these scripts would do that.

These results show that information flow analysis effectively computes a much finer granularity of the potential flows between sources and sinks used in a script.

RQ6.2: Safe Scripts

To address RQ6.2, we apply our static analysis on the 78 subject scripts that have information flows, referred to as *flow scripts*, and measure the number of flow scripts that have safe flows. We assume only sink *Web* is an non-monitored sink, while all others are monitored sinks. Table 6.4 shows the safe/unsafe flow summary of the 78 scripts that have information flows. Column "# Safe" shows the number of scripts that have safe flows. Column "# Non-Monitored" shows the number of scripts that have information flows from sources into non-monitored sinks. Column "# Tampered" shows the number of scripts that have tampered information flows. Column "# Both" shows the number of scripts that have both safe and unsafe flows. Column "# Mix" shows the number of scripts that have both safe and unsafe flows from a common source (*mix scripts*).

The results show that 45 (57.69%) flow scripts have safe flows and 54 (69.23%) flow scripts have unsafe flows. Among these 54 unsafe flow scripts, 40 flow scripts have flows from sources into non-monitored sinks and 47 have tampered information flows. Based on this safe/unsafe flow summary, we know that 24 (#Safe−#Both), or 30.77% of flow scripts have only safe flows. For these 24 scripts, users are perfectly safe to use the scripts granting full access to private information without prompting or reduced functionality.

Among the 21 flow scripts that have both safe and unsafe flows, none are mix scripts. In all the TouchDevelop scripts, only 2 flow scripts published by ourselves have both safe and unsafe

Table 6.4: Safe/Unsafe flow summary of 78 flow scripts

# **Safe**	# **Unsafe** (54)		# **Both**	# **Mix**
	# **Non-Monitored**	#**Tampered**		
45	40	47	21	0

flows from a common source to sinks. Our current access granting allows users to grant access based on sources only, instead of flows. Users cannot choose real information for one flow and anonymized information for another flow from the same source. As we found only 2 scripts where this limitation matters, it seems to be a good trade-off that avoids giving users too much choice.

RQ6.3: Safe Sources

To address RQ6.3, we look at how many times a user would have to change the default setting for a source if she were to give full access to all scripts. Table 6.5 shows the total number of times a source appears in a given context. Column "Naïve" shows the number of scripts that use this source and any sink. Column "Flow" shows the number of scripts that have information flows from this source to any sinks. Column "Safe" shows the number of scripts for which this source is safe. The last three columns explain why some flows are unsafe. Column "Non-Monitored" shows the number of scripts where information flows from this source to non-monitored sinks. Column "Tamper" shows the number of scripts where information from this source is tampered before it reaches a sink. Column "Both" shows the number of scripts that have common sources in Columns "Non-Monitored" and "Tamper".

Among 33 scripts that have source Camera appearing in flows, 24 scripts (72.73%) have source Camera as a safe source and 9 scripts (27.27%) have source Camera in tampered flows.

Table 6.5: Categorization of sources

	Naïve	**Flow**	**Safe**	Unsafe due to		
				Unvet.	**Tamp.**	**Both**
Camera	36	33	24	0	9	0
Contacts	41	30	25	0	5	0
Location	34	30	0	27	26	23
Microph.	6	1	1	0	0	0
Music	3	1	0	0	1	0
Picture	32	24	6	14	15	11
Total	152	119	56	41	56	34

Similarly, 25 scripts (83.33%) have safe sources of Contacts, leaving only 5 scripts having source Contacts appearing in tampered flows.

In summary, our analysis detects that 47.06% (56) of 119 sources are safe sources. These safe sources are allowed to use real information directly based on our default settings, eliminating the need for access granting. Among the remaining 63 unsafe sources (# Non-Monitored + # Tamper - # Both), 7 (#Non-Monitored - #Both) are solely due to flow to non-monitored sinks, and the remaining 56 sources appear in tampered information flows. These results show that using the naïve classification, a user would have to make 152 changes to settings to use real data in all scripts. Using information flow alone, this number is reduced to 119 changes. Using tamper analysis and monitored sinks in addition to information flow, our approach reduces the burden to 63 changes to settings, an overall reduction of 58.6%.

6.8 Discussion

In this section, we discuss generalizations and limitations of our approach.

Generalization to Other Mobile-Device Platforms. To generalize our approach to other mobile-device platforms, such as Windows Phone, Android, and iOS, several points need to be addressed: (1) these platforms provide a much larger API surface than TouchDevelop and annotating these APIs with source, sink, and flow information is a major effort, (2) the languages used (Java, C#, or assembly code) provide more ways to obscure flow than in our scripting language, in particular through indirect calls, or via reflection. The static analysis would have to be extended to account for these [73, 79]. (3) Indirect flow through mutable storage will require a finer grained heap model than we currently employ (one abstract location per data kind). The static analysis might need to be complemented with dynamic analysis [72, 223] to address this issue.

Limitations of Static Information Flow Analysis. Due to the way our approach handles implicit flows, our approach may produce false positives as described by Kang et al.'s work [119]. However, our evaluation results show that even with these potential false positives, our approach still achieves a significant reduction in access granting for users. To improve our approach when migrating to other mobile-device platforms, our approach can be combined with DTA++ techniques [119].

Another type of implicit flow, covert channels [168], may cause false negatives of our approach. For example, a script can store a classified picture into the media library, and then later share it through facebook via a different application. Our flow analysis would indicate that a picture is stored into the media library (and the user has to agree with that flow), but our approach does not contemplate what could happen to the picture in the library after that. To

address such issues, the operating system would have to provide dynamic taint tracking [72], since such flows involve more than one application or even OS built-in functionality.

6.9 Summary

Applications in mobile-marketplaces may leak private user information without notification. Existing mobile platforms provide little information on how applications use private user data, making it difficult for experts to validate applications and for users to grant applications access to their private data. We have proposed an approach, called user-aware privacy control, that explains the uses of permissions, improving the users' understanding of applications' privacy-sensitive behaviors and reducing efforts for app validation and access-granting. Our approach computes static information flows and classify them as safe/unsafe based on a tamper analysis that tracks whether private data is obscured before escaping through output channels. This flow information enables platforms to provide default settings that expose private data only for safe flows, thereby preserving privacy and minimizing decisions required from the users.

Based on the computation of information flows, our approach provides a behavior-diagnosis interface that shows information flows of the requested permissions. Such a larger scope of information explains how the users' privacy-sensitive information is used by the mobile applications, enabling the behavior-diagnosis cooperation for mobile privacy control. In this cooperation, the tools identify permissions requested by mobile applications for the users to inspect and explain the permissions by showing information flows. Based on thhe users' domain knowledge about the mobile applications' functionality and security requirements, the users inspect the information flows to determine whether the information flows are expected for the mobile applications,

We built our approach into TouchDevelop, an application-creation environment that allows the users to write scripts on mobile devices and install scripts published by other users. We evaluate the effectiveness and performance of our information flow analysis on 546 scripts published by 194 users to evaluate The results show that among the 546 scripts, 172 use a private source, but only 78 scripts (14.29%) flow private information to a sink. Among these 78 scripts, our approach classifies 24 as safe, reducing the need to make access granting choices to a mere 10.1% (54) of all scripts. Alternatively, the users need to grant access to only 63 sources (41.4%) among 152 sources appearing in scripts together with sinks.

CHAPTER 7

Behavior-Diagnosis Cooperation for Security Policy Extraction

7.1 Introduction

Access control is one of the most fundamental and widely used privacy and security mechanisms. Access control is often governed by an Access Control Policy (ACP) [165] that includes a set of rules specifying which principals (such as users or processes) have access to which resources. ACPs are crucial in preventing security vulnerabilities, since decisions (such as *accept* or *deny*) on user requests are based on ACPs. In ACP practice, there exist two major issues that can result in serious consequences such as allowing an unauthorized user to access protected resources: incorrect specification of ACPs and incorrect enforcement of ACP specifications in the system implementation.

The first issue of incorrect specification of ACPs is primarily due to two reasons. First, ACPs contain a large number of complex rules to meet various security and privacy requirements. One way to ensure the correctness of such complex rules is to leverage systematic testing and verification approaches [110, 137] that accept ACPs in a form of formal specification. In practice, ACPs are commonly written in Natural Language (NL) and are supposed to be written in security requirements, a type of non-functional requirements. However, often ACPs are buried in NL documents such as requirement documents. For example, consider the following ACP sentence (i.e., sentence describing ACP rules) for iTrust [197, 13], an open source health-care application: *"The Health Care Personnel (HCP) does not have the ability to edit the patient's security question and password"*. This ACP sentence is not amenable for automated verification, requiring manual effort in extracting the ACP from this sentence into an enforceable format

such as the eXtensible Access Control Markup Language (XACML) [9]. Second, NL software documents could be large in size, often consisting of hundreds or even thousands of sentences (iTrust consists of 37 use cases [113] with 448 use-case sentences), where a portion of the sentences describing ACPs (117 sentences in iTrust) are buried among other sentences. Thus, it is very tedious and error-prone to manually inspect these NL documents for identifying and extracting ACPs for policy modeling and specification.

The second issue of incorrect enforcement of ACP specifications is primarily due to the inherent gap between ACPs specified using domain concepts and the actual system implementation developed using programming concepts. Functional requirements, such as scenario-based requirements (e.g., use cases) that specify sequences of action steps[1], bridge the gap, since they describe functionalities to be implemented by developers using domain concepts. For example, an action step *"The patient chooses to view his or her access log."* in Use Case 8 of iTrust implies that the system shall have the functionality for patient (domain concepts) to view his or her access log. These action steps typically describe that actors (principals) access different resources for achieving some functionalities and help developers determine what system functionalities to implement. Therefore, policy authors can validate action steps against provided ACPs to detect inconsistencies of resource access, also helping the policy authors construct consistent ACPs for the system. In practice, manually inspecting large functional requirements to extract resource-access information is also labor-intensive and tedious. For example, a proprietary IBM enterprise application (that we used in our evaluations) includes 659 use cases with 8,817 sentences.

In general, like other types of NL documents, NL requirements written in English are unstructured and can be ambiguous or include implicit information, posing significant challenges for Natural Language Processing (NLP). However, in software documents such as functional and non-functional requirements, ACP sentences (i.e., NL security requirements for describing ACP rules) tend to follow specific styles such as: *[subject] [can/cannot/is allowed to] [action] [resource]* for role-based ACPs [81]. For example, based on our manual inspection of 217 ACP sentences collected from the iTrust requirements and various security requirements in published articles and web sites [24], about 85% of the ACP sentences follow this style. Similarly, to provide communication values, functional requirements such as use cases are usually written in a relatively simple, consistent, and straightforward style [55, 115].

To tackle the problem, we propose a novel approach, called Text2Policy, which adapts NLP techniques designed around a model (such as the ACP model and the action-step model) to automatically extract model instances from NL software documents and produce formal specifications. Our general approach consists of three main steps: (1) apply linguistic analysis to

[1]We use the term of an *action step* rather than *action* to distinguish the term from an *action* in the access control model described later.

parse NL documents and annotate words and phrases in sentences from NL documents with semantic meanings, (2) construct model instances using annotated words and phrases in the sentences, and (3) transform these model instances into formal specifications.

Specifically, we provide techniques to concretize our general approach for extracting role-based ACPs and action steps from NL software documents and functional requirements, respectively. From the extracted ACPs, our approach automatically generates machine-enforceable ACPs in specification languages such as XACML. These ACPs can be used by automatic testing and verification approaches [110, 137] for checking policy correctness or serve as an initial version of ACPs for policy authors to improve. From each extracted action step, our approach automatically derives an access control request. An example request could be that a principal requests access to a resource with the expected permit or deny decision. Such derived requests can be used for automatic validation against specified or extracted ACPs for detecting inconsistencies.

The goal of security policy extraction is controlling resource accesses for a software system. Based on Text2Policy, our approach provides a behavior-diagnosis interface that shows the security policies in the form of formal models automatically extracted from the sentences in requirements documents, and the users make decisions on whether the security policies are expected based on their domain knowledge about the software system's functionality and security requirements. Also, validating these action steps against the extracted policies explains by instantiating the policies what action steps violating the policies. Thus, our behavior-diagnosis interface further shows the requirements documents where the security policies are extracted as policy-witness scenarios, and the inconsistencies between action steps and the extracted policies as policy-violation scenarios.

The policy-witness and policy-violation scenarios use the instantiations of the extracted security policies as explanations of the security policies, enabling the behavior-diagnosis cooperation for security policy extraction. In this cooperation, the tools extract security policies for the users to inspect and explain the security policies by showing the policy-witness and policy-violation scenarios. Based on the users' domain knowledge about the software system's functionality and security requirements, the users inspect the policy-witness and policy-violation scenarios (instantiations of the security policies) to determine whether these scenarios are expected for the software system, and make decisions to include or reject the security policies.

7.2 ACP and Action-Step Models

In this section, we first introduce the background of the ACP model used for representing ACPs in our approach, and then describe the background of the action-step model used for representing action steps in our approach.

ACP-1: An HCP should not change a patient's account.
ACP-2: An HCP is disallowed to change a patient's account.

Figure 7.1: Example ACP sentences written in NL.

7.2.1 ACP Model and XACML

This section provides the background information about our ACP model and XACML.

ACP Model

An ACP consists of a set of ACP rules. A typical role-based ACP rule consists of four elements: subject, action, resource, and effect [81, 9]. Figure 7.1 shows two example ACP rules. The subject element describes a principal such as a user or process that may request to access resources (e.g., an *HCP* in ACP-1). The action element describes an action (e.g., *change* in ACP-1) that the principal may request to perform. The resource element describes the resource (e.g., *a patient's account* in ACP-1) to which access is restricted. A rule can have one of various effects (i.e., permit, deny, oblige, or refrain). In this chapter, we focus on permit rules and deny rules (i.e., rules with permit or deny effects), which are commonly used in various software systems for granting or blocking accesses to protected resources. A *permit* rule allows a principal to access a resource, whereas a *deny* rule, such as ACP-1 and ACP-2, prevents a principal from accessing a resource.

XACML

The eXtensible Access Control Markup Language (XACML) [9] is an XML-based general-purpose language used to describe policies, requests, and responses for ACPs, recognized as a standard by the Organization for the Advancement of Structured Information Standards (OASIS). XACML is designed to replace application-specific and proprietary ACP languages, thus enabling communication among applications created by different application vendors.

In an application deployed with XACML-based access control, before a principal can perform an action on a particular resource, a Policy Enforcement Point (PEP) sends a request formulated in XACML to the Policy Decision Point (PDP) that stores principal-specific XACML ACP rules. The PDP determines whether the request should be permitted or denied by evaluating the policies whose subject, action, and resource elements match the request. Finally, the PDP formulates its decision in the XACML response language and sends it to the PEP, which enforces the decision.

Currently, XACML has been widely supported by all the main platform vendors and extensively used in a variety of applications [133]. Recent research also provides systematic testing

AS-1: An HCP creates an account.
AS-2: He edits the account.
AS-3: The system updates the account.
AS-4: The system displays the updated account.

Figure 7.2: An example use case.

and verification approaches [110, 137] for ensuring the correct specification of XACML rules. There also exist XACML-based research tools used in various agencies/labs and companies [112]. Thus, we choose XACML as the formal specification to model ACP.

7.2.2 Action-Step Model

Use cases [114] are scenario-based requirements specifications that consist of sequences of action steps for illustrating behaviors of software systems. These action steps describe how actors interact with software systems for exchanging information. Actors are entities outside software systems (such as users) that interact with the systems by providing input to the systems (e.g., in Action Step AS-2 shown in Figure 7.2) or receiving output from the systems (e.g., in AS-4 shown in Figure 7.2). Since action steps describe how actors access or update information (resources) of the systems, each action step can be considered to encode an access control request that an actor requests to access the resources and expect the request to be permitted. Using the access control requests with expected permit decisions derived from action steps, we can automatically validate such requests with expected decisions against specified or extracted ACPs to detect inconsistencies.

We represent the contents of use cases (sequences of action steps) in a formal representation. The content of a NL use case contains a list of sentences, each of which in turn contains one or more action steps initiated by some actor (e.g., an *HCP* in AS-1 shown in Figure 7.2). Each action step has an action associated with a classification, such as the INPUT classification for the action of providing information (e.g., *edits* in AS-2 shown in Figure 7.2) and the OUTPUT classification for the action of receiving information (e.g., *display* in AS-4 shown in Figure 7.2). An action step is also associated to one or more actors and has a set of parameters. These parameters represent the resources created, modified, or used by the actions. In Figure 7.2, AS-2 shows a resource *account* that is modified.

7.3 Challenges and Examples

In this section, we first describe the technical challenges faced by ACP extraction and action-step extraction. We next use examples to illustrate how Text2Policy extracts ACPs and action steps from NL documents and NL use cases, respectively.

7.3.1 Technical Challenges

As a common technical challenge for both ACP extraction and action-step extraction, *TC1-Anaphora* refers to identifying and replacing pronouns with noun phrases based on the context. For example, the pronoun *he* in AS-2 shown in Figure 7.2 needs to be replaced with the *HCP* from AS-1. For ACP extraction, there are two unique technical challenges: (1) *TC2-Semantic-Structure Variance.* ACP-1 and ACP-2 in Figure 7.2 use different ways (semantic structures) to describe the same ACP rule; (2) *TC3-Negative-Meaning Implicitness.* An ACP sentence may contain negative expressions, such as ACP-1. Additionally, the verb in the sentence may have negative meaning, such as *disallow* in ACP-2. For action-step extraction, there are two unique challenges: (1) *TC4-Transitive Actor.* AS-3 implies that an *HCP* (the actor from AS-2) is the initiating actor of AS-3; (2) *TC5-Perspective Variance.* AS-4 implies that an *HCP views* the updated *account*, requiring a conversion to replace the actor and action of AS-4.

To address *TC1-Anaphora*, we adapt the technique *Anaphora Resolution*, specializing the anaphora algorithm introduced by Kennedy et al. [122] to identify and replace pronouns with noun phrases based on the context. To address *TC2-Semantic-Structure Variance*, we propose a technique, called *Semantic-Pattern Matching*, which uses different semantic patterns based on the grammatical functions (subject, main verb, and object) to match different semantic structures of ACP sentences. To address *TC3-Negative-Meaning Implicitness*, we propose an inference technique, called *Negative-Meaning Inference*, which infers negative meaning by using patterns to identify negative expressions and a domain dictionary to identify negative meaning of verbs. To address *TC4-Transitive Actor*, we propose an analysis technique, called *Actor-Flow Tracking.* This technique first tracks non-system actors in action steps. Later, when the analysis encounters action steps that have only system actors, it replaces system actors with tracked non-system actors. To address *TC5-Perspective Variance*, we propose an analysis technique, *Perspective Conversion.* This technique tracks non-system actors of action steps. Later when the analysis encounters action steps that have only system actors and output information from the system, it replaces the system actors with tracked non-system actors and replaces output actions with read actions (such as *view*).

```
<Policy PolicyId="2" RuleCombAlgId="...">
    <Target/>
    <Rule Effect="Deny" RuleId="rule-1">
        <Target>
            <Subjects>
                <Subject>
                    <SubjectMatch MatchId="string-equal">
                        <AttrValue>HCP</AttrValue>
                        <SubjectAttrDesignator AttrId="subject:role"/>
                    </SubjectMatch>
                </Subject>
            </Subjects>
            <Resources>
                <Resource>
                    <ResourceMatch MatchId="string-equal">
                        <AttrValue>patient.account</AttrValue>
                        <ResourceAttrDesignator AttrId="resource-id"/>
                    </ResourceMatch>
                </Resource>
            </Resources>
            <Actions>
                <Action>
                    <ActionMatch MatchId="string-equal">
                        <AttrValue>UPDATE</AttrValue>
                        <ActionAttriDesignator AttrId="action-id"/>
                    </ActionMatch>
                </Action>
            </Actions>
        </Target>
    </Rule>
</Policy>
```

Figure 7.3: Generated XACML ACP for ACP-2 in Figure 7.2

7.3.2 Example of ACP Extraction

Text2Policy adapts NLP techniques that incorporate syntactic and semantic analyses to parse NL software documents, constructs ACP model instances, and produces formal specifications.

In particular, Text2Policy first applies shallow parsing [147] that annotates sentences with phrases, clauses, and grammatical functions of phrases, such as subject, main verb, and object. For example, the shallow-parsing component parses ACP-1 in Figure 7.2 as [subject: *An HCP*] [main verb group: *should not change*] [object: *a patient's account.*]. Text2Policy then uses the domain dictionary to associate verbs with pre-defined semantic classes. For example, in ACP-2, the domain dictionary is used to associate *change* with the UPDATE semantic class, and *disallow* with the NEGATIVE semantic class.

To determine whether a sentence describes an ACP rule (i.e., is an ACP sentence) and extract elements of subject, action, and resource, Text2Policy composes semantic patterns using the identified grammatical functions of phrases and clauses extracted by the shallow-parsing component. For example, ACP-1 can be matched by the semantic pattern *Modal Verb in Main*

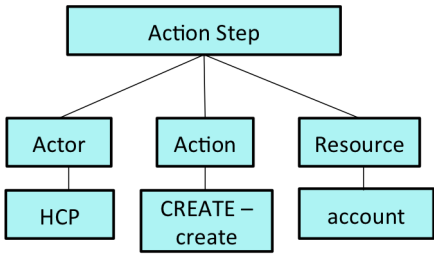


Figure 7.4: An example action step

Verb Group, and the constructed model instance of ACP-1 is [Subject: *HCP*] [Action: *change - UPDATE*] [Resource: *patient.account*].

To infer the effect for an ACP rule, Text2Policy checks whether the corresponding sentence contains any negative expression and whether the main verb group is associated with the NEGATIVE semantic class. For example, Text2Policy identifies the negative expression of *should not change* in ACP-1 and infers the effect of ACP-1 as *deny*.

Using the extracted ACP-model elements and the inferred effect, Text2Policy constructs an ACP model instance for each ACP sentence and generates ACP rules in XACML. Figure 7.3 shows the generated XACML ACP for ACP-2.

7.3.3 Example of Action-Step Extraction

Action-step extraction uses similar linguistic analyses as ACP extraction. First, the techniques of shallow parsing and domain dictionary are used to parse and annotate each sentence in use cases. Next, the technique of anaphora resolution is used to identify and replace pronouns (from the sentence) with the noun phrases based on the context. For example, *He* in AS-2 is replaced with *HCP*. Text2policy then uses a syntactic pattern to check whether the sentence has required elements (subject, main verb group, and object) for constructing an action step, and constructs a model instance if all the elements are found.

Consider the example use case shown in Figure 7.2. Since all sentences include the required elements, Text2policy constructs model instances of these action steps associated with actors (the *system*, *HCP*), action types representing the classification of the actions (e.g., the classification of *display* in AS-4 as OUTPUT), and parameters (the *account*). For example, the model instance of AS-1 is shown in Figure 7.4. In addition, since AS-3 and AS-4 have the *system* as the actor, Text2policy further applies the techniques for *TC4-Transitive Actor* and *TC5-Perspective Variance* on AS-3 and AS-4 to replace the actors and actions. solution algorithm replaces *he* in AS-2 is replaced by *HCP* by the anaphora resolution technique.

Table 7.1: Identified subject, action, and resource elements in sentences matched with semantic patterns for ACP sentences.

Semantic Pattern	Examples
Modal Verb in Main Verb Group	An HCP [*subject*] **can view** [*action*] the patient's account [*resource*]· An admin [*subject*] **should not update** [*action*] patient's account [*resource*]·
Passive Voice followed by To-infinitive Phrase	An HCP [*subject*] **is disallowed to update** [*action*] patient's account [*resource*]· An HCP [*subject*] **is allowed to view** [*action*] patient's account [*resource*]·
Access Expression	An HCP [*subject*] **has read** [*action*] **access to** patient's account [*resource*]· A patient's account [*resouce*] **is accessible** [*action*] **to** an HCP [*subject*]·
Ability Expression	An HCP [*subject*] **is able to read** [*action*] patient's account [*resource*]· An HCP [*subject*] **has the ability to read** [*action*] patient's account [*resource*]·

7.4 Approach

In this section, we describe our general approach for extracting model instances from NL documents and producing formal specification. Our approach consists of three main steps: Linguistic Analysis, Model-Instance Construction, and Transformation.

Figure 7.5 shows the overview of our approach. Our approach accepts NL software documents as input and applies linguistic analysis to parse the NL software documents and annotates their sentences with semantic meanings for words and phrases. Using the annotated sentences, our approach constructs model instances. Based on provided transformation rules, our approach transforms the model instances to formal specifications, which can be automatically checked for correctness and consistencies.

7.4.1 Linguistic Analysis

The linguistic-analysis component includes adapted NLP techniques that incorporate syntactic and semantic NL analyses to parse the NL software documents and annotate the words and phrases in the document sentences with semantic meaning. We next describe the common linguistic-analysis techniques used for both ACP extraction and action-step extraction, and describe the unique analysis techniques proposed for ACP extraction and action-step extraction, respectively.

Common Linguistic-Analysis Techniques

In this section, we describe the common linguistic-analysis techniques used in our general approach: shallow parsing and domain dictionary.

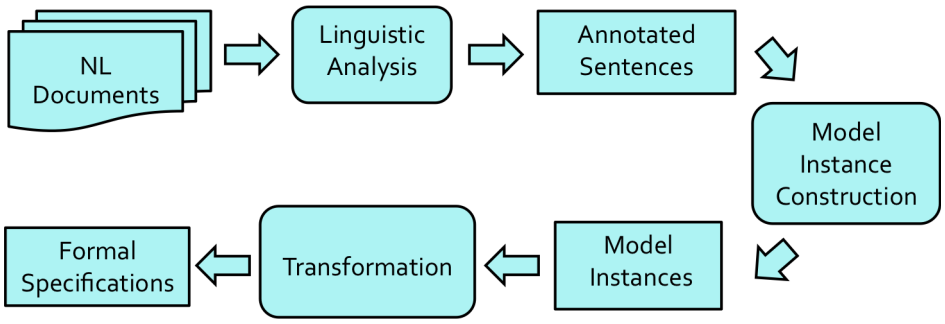


Figure 7.5: Overview of our approach.

Shallow Parsing. Shallow parsing determines the syntactic structures of sentences in NL documents. Research [98, 178] has shown the efficiency of shallow parsing based on finite-state techniques and the effectiveness of using finite-state techniques for lexical lookup, morphological analysis, Part-Of-Speech (POS) determination, and phrase identification. Sinha et al.'s work [170] also shows that the shallow-parsing analysis is effective and efficient for semantic and discourse processing. Therefore, our approach chooses a shallow parser that is fully implemented as a cascade of several Finite-State Transducers (FSTs), described in detail by Boguraev [44].

In the shallow parser, an FST identifies phrases, clauses, and grammatical functions of phrases by recognizing patterns of POS tags and already identified phrases and clauses in the text. The lowest level of the cascade recognizes simple Noun Group (NP) and Verb Group (VG) grammars. For example, ACP-1 is parsed as [NP: *An HCP*] [VG: *should not change*] [NP: *patient's account.*]. Later stages of the cascade try to build complex phrases and identify clause boundaries based on patterns of already identified tokens and phrases. For example, *to change patient's account* in ACP-2 is recognized as a to-infinitive clause. The final set of FSTs marks grammatical functions such as subjects, main verb group, and objects. As an example, the shallow parser finally parses and annotates ACP-1 as [subject: *An HCP*] [main verb group: *should not change*] [object: *patient's account.*].

Domain Dictionary. The domain dictionary is used to associate verbs with pre-defined semantic classes. There are two benefits of associating verbs with semantic classes. The first benefit is to help address *TC3-Negative-Meaning Implicitness*. Consider ACP-2 shown in Figure 7.1. Without the NEGATIVE semantic class associated with the main verb group (*is disallowed*), our analysis would incorrectly infer the effect as *permit* instead of *deny*. The second benefit is to identify verb synonyms, such as *change* and *update*. During validation of action-step information against ACPs, our approach uses verb synonyms to match access requests (transformed from action steps) with an applicable ACP rule.

The domain dictionary is used to associate each verb entry with a semantic class. Besides the NEGATIVE class that we mentioned earlier, a verb entry can be associated with a semantic class that is a kind of operation [169, 170], e.g., OUTPUT (*view* or *display*) and UPDATE (*change* or *edit*). To achieve so, we populate the domain dictionary with an initial set of commonly used verb entries and their respective semantic classes. We then use WordNet [78], a large lexical database of English, to further expand the entries with their synonyms.

Currently, we implement the domain dictionary as an extensible and externalizable XML Blob and the content is populated manually. One major limitation of using an XML Blob is that unmatched verbs (i.e., ones without matched entries in the dictionary) are assigned with the UNCLASSIFIED semantic class. In future work, we plan to extend our technique to query WordNet dynamically when an unmatched verb or adjective is encountered. For example, by querying WordNet for synonyms, we can assign to an unmatched verb the semantic class of its most similar verb among its matched synonyms. Alternatively, we can assign to an unmatched verb the semantic class that is most common among the unmatched verb's k-nearest neighbors.

Anaphora Resolution. To address *TC1-Anaphora*, our approach includes the anaphora-resolution technique to identify and replace pronouns with the noun phrases that they refer to. To resolve anaphora encountered during use-case parsing, we adapt the anaphora algorithm introduced by Kennedy et al. [122] with an additional rule: a pronoun in the position of a subject is replaceable by only noun phrases that also appear as subjects of a previous sentence. As an example, *he* in AS-2 shown in Figure 7.2 is replaced by the *HCP*, the actor of AS-1.

ACP Linguistic Analysis

In this section, we describe unique linguistic-analysis techniques proposed for ACP extraction.

Semantic-Pattern Matching. To address *TC2-Semantic-Structure Variance*, we provide the technique of semantic-pattern matching to identify whether a sentence is an ACP sentence. We compose different semantic patterns based on the grammatical function of phrases identified by shallow parsing. These semantic patterns are more general and more accurate than templates written using low-level syntactical structures, such as POS tags [75]. Our approach uses this technique while identifying subject, action, and resource elements for an ACP rule.

Table 7.1 shows the semantic patterns used in our approach. The text in bold shows the part of a sentence that matches a given semantic pattern. The first pattern, *Modal Verb in Main Verb Group*, identifies sentences whose main verb contains a modal verb. This pattern can identify ACP-1 shown in Figure 7.1. The second pattern, *Passive Voice followed by To-infinitive Phrase*, identifies sentences whose main verb group is passive voice and is followed by a to-infinitive phrase. This pattern can identify ACP-2 shown in Figure 7.1. The third pattern, *Access Expression*, captures different ways of expressing that a principal can have access to a

particular resource. The fourth pattern, *Ability Expression*, captures different ways of expressing that a principal has the ability to access a particular resource. Using the semantic patterns, our approach filters out NL-document sentences that do not match with any of these provided patterns.

Negative-Expression Identification. Negative expressions in sentences can be used to determine whether the sentences have negative meaning. To identify negative expressions in a sentence, our approach composes patterns to identify negative expressions in a subject and main verb group. For example, "*No HCP can edit patient's account.*" has *no* in the subject. As another example, "*An HCP can never edit patient's account.*" has *never* in the main verb group. ACP-1 in Figure 7.1 contains a negative expression in the main verb group. Our approach uses the negative-expression identification while inferring policy effect for an ACP rule.

Use-Case Linguistic Analysis

In this section, we describe a unique linguistic-analysis technique proposed for action-step extraction.

Syntactic-Pattern Matching. To identify whether a sentence is an action-step sentence (i.e., describing an action step), our approach includes the technique of syntactic-pattern matching that identifies sentences with syntactic elements (subject, main verb group, and object) required for constructing an action step. To improve precision in identifying sentences describing users accessing resources, our approach further checks whether the subject is a user of the system and whether the object is a resource defined in the system. For example, our approach ignores the sentence "The prescription list should include medication, the name of the doctor..." [197, 13], since its subject *prescription list* is not a user of the system. Moreover, our approach also uses the technique of negative-meaning inference (described later in this section) to filter out sentences that contain negative meaning, since these negative-meaning sentences tend not to describe action steps.

7.4.2 Model-Instance Construction

After our approach uses linguistic-analysis techniques to parse the input NL documents, words and phrases in the sentences of the NL documents are annotated with semantic meaning. For example, shallow parsing annotates phrases as subjects, main verb groups, and objects. To construct model instances from these sentences, our approach uses the annotated information of words and phrases to identify necessary elements for a given model.

ACP-Model Construction

To construct model instances for ACP rules, our approach identifies subject, action, resource elements based on the matched semantic patterns and infers the policy effect based on the presence or absence of negative expressions in sentences.

Model-Element Identification. Based on the matched semantic patterns, our approach identifies subject, action, resource elements from different syntactic structures in sentences.

Table 7.1 shows the identified subject, action, and resource elements (underlined words) in the sentences matched with semantic patterns. For a sentence that matches the first pattern, *Modal Verb in Main Verb Group*, our approach identifies the subject of the sentence as a subject element, the verb (not the modal verb) in the main verb group as an action element, and the object of the sentence as a resource element. For a sentence that matches the second pattern, *Passive Voice followed by To-infinitive Phrase*, our approach identifies the subject of the sentence as a subject element and identifies action and resource elements from the verb and object in the to-infinitive phrase, respectively. For the first example of the third pattern, *Access Expression*, our approach identifies the subject of the sentence as a subject element, the noun *read* in the main verb group as an action element, and the noun phrase *patient's account* in the prepositional phrase *to patient's account* as a resource element. For the second example of the third pattern, our approach identifies the subject *patient's account* as the resource element, the adjective *accessible* as an action, and the object *HCP* as the subject element. For the sentences that match the fourth pattern, our approach identifies the subject of the sentence as a subject element and identifies action and resource elements from the verb and object in the to-infinitive phrase, respectively.

Policy-Effect Inference. To address *TC3-Negative-Meaning Implicitness*, our approach includes the technique of negative-meaning inference. If an ACP sentence contains negative meaning, we infer the policy effect to be deny (permit otherwise). To infer whether a sentence has negative meaning, the technique of negative-meaning inference considers two factors: negative expression and negative-meaning words in the main verb group. Recall that negative expressions is identified using the technique of negative-expression identification in Section 7.4.1. ACP-1 in Figure 7.1 contains a negative expression in the main verb group. To determine whether there are negative meaning words in the main verb group, our approach checks the semantic class associated with the verb in the main verb group. If the semantic class is NEGATIVE, we consider the sentence has negative meaning. ACP-2 has a negative meaning word, *disallow*, in the main verb group, and therefore its inferred policy effect is deny.

Model-Instance Construction. Using the identified elements (subject, action, and resource) and inferred policy effect, our approach constructs an ACP-model instance for an ACP sentence. Moreover, our approach provides techniques to deal with a possessive noun phrase,

such as *patient's account* or *the account of patient*. Our approach extracts the possessor as an entity and the possessed item as its property. As a complete example, the constructed model instance of ACP-2 is [Subject: *HCP*] [Action: *change - UPDATE*] [Resource: *patient.account.*] [Effect: *deny*]. Here the technique of domain dictionary associates the verb *change* with the semantic class UPDATE.

Action-Step-Model Construction

To construct model instances for action steps described in sentences, our approach identifies actor, action, and parameter elements based on the use-case patterns. Our approach includes two additional new techniques to address *TC4-Transitive Actor* and *TC5-Perspective Variance*.

Model-Element Identification. Our approach uses known patterns of use-case action steps to identify action, actor, and parameter elements for action steps. We devise these patterns based on industry use cases [170], iTrust use cases, and use cases collected from published articles [160]. One of the most used patterns is to identify the subject of a sentence as an actor element, the verb in the main verb group as an action element, and the object of the sentence as a parameter element. These patterns could be easily updated or extended based on the domain characteristics of the use cases for improving the precision of extracting actor, action, and parameter elements.

Model-Instance Construction. Using the identified actor, action, and parameter elements in a sentence, our approach constructs action-step model instances for action steps described in the sentence. For example, the model instance for *A patient views access log* is [Actor: *patient*] [Action: *view - READ*] [Parameter: *access log*]. Here the technique of domain dictionary associates the verb *view* with the semantic class READ.

Actor-Flow Tracking. To address *TC4-Transitive Actor*, we apply data-flow tracking on non-system actors of an action step. We consider subjects (such as the *system* in AS-3) with some specific names as system actors. Non-system actors can usually be obtained from the glossary of requirements documents. Algorithm 3 shows the Actor-Flow Tracking (AFT) algorithm.

We next illustrate the algorithm using the example shown in Figure 7.1. AFT first checks AS-1 and tracks the actor of AS-1 since its actor is a non-system actor (*HCP*) (satisfying the condition at Line 11). AFT then checks AS-2 and tracks the actor of AS-2 (*HCP*, replaced by anaphora resolution) since its actor is also *HCP*. When AFT checks AS-3, AFT finds that AS-3 has only the *system* as its actor (satisfying the condition at Line 15) and replaces the *system* with *HCP* as the actor of AS-3.

Perspective Conversion. To address *TC5-Perspective Variance*, we use a similar algorithm as AFT. The only difference is to replace the condition at Line 15 as *trackActor*

Algorithm 3 Actor-Flow Tracking

Require: ASs for action steps in a use case

```
1:  trackedActor = NULL
2:  for AS in ASs do
3:      Actors = getActors(AS)
4:      onlySystemActor = TRUE
5:      for actor in Actors do
6:          if !isSystemActor(actor) then
7:              onlySystemActor = FALSE
8:              break
9:          end if
10:     end for
11:     if !onlySystemActor then
12:         trackedActor = getNonSystemActor(Actors)
13:         continue
14:     end if
15:     if trackedActor != NULL then
16:         replaceActors(AS, trackedActor)
17:     end if
18: end for
```

$! = NULL\ AND\ getActionType(AS) == OUTPUT$, and to replace the statement at Line 16 as $convertPerspective(AS, trackActor)$. Consider the same example shown in Figure 7.1. When the algorithm reaches AS-4, the tracked actor is *HCP*. Since AS-4 has *system* as its only subject and its action type is OUTPUT (*displays*), our approach converts AS-4 into *An HCP views the updated account* by replacing its actor elements with the tracked actors and its action element with a verb entry whose classification is READ in the domain dictionary, such as *view*. Such conversion helps our approach to correctly extract access requests from action steps.

7.4.3 Transformation

With the formal model of ACPs, our approach can use different transformation rules to transform model instances into formal specifications, such as XACML [9].

ACP Model. Currently, our approach supports the transformation of each ACP rule into an XACML policy rule. Our approach transforms subject, action, and resource elements as the corresponding subject, action, and resource sub-elements of the target element for an XACML policy rule. Our approach then assigns the value of the effect element to the value of the effect attribute of the XACML policy rule to complete the construction of an XACML policy rule. Figure 7.3 shows the extracted XACML rule of ACP-2. More examples can be found on our

Table 7.2: Metrics for addressing research questions.

RQ	Metrics
RQ7.1	$Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F_1\text{-}Score = \frac{2*Precision*Recall}{Precision+Recall}$ TP: True positives, FP: False positives, FN: False negatives
RQ7.2	$Accuracy = \frac{C}{T}$ C: Number of correct ACP rules extracted by Text2Policy T: Total number of ACP rules
RQ7.3	$Accuracy = \frac{C}{T}$ C: Number of correct action-step sentences extracted by Text2Policy T: Total number of action-step sentences

project web site [24]. With more transformation rules, our approach can easily transform the ACP model instances into other specification languages, such as EPAL [34].

Action-Step Model. Currently, our approach supports the transformation of each action step into an XACML request [9] with the expected permit decision. For each action step, our approach transforms actor, action, and parameter elements as subject, action, and resource elements of the request, respectively.

7.5 Evaluations

In this section, we present three evaluations conducted to assess the effectiveness of Text2Policy. For our evaluations, we collected use cases from an open source project iTrust [197, 13], 115 ACP sentences from 18 sources (published papers, public web sites, and iTrust), and 25 use cases from a module in a proprietary IBM enterprise application. We specifically seek to answer the following research questions:

- **RQ7.1**: How effectively does Text2Policy identify ACP sentences in NL documents?
- **RQ7.2**: How effectively does Text2Policy extract ACP rules from ACP sentences?
- **RQ7.3**: How effectively does Text2Policy extract action steps from action-step sentences (i.e., sentences describing action steps)?

Table 7.2 shows metrics used to address our research questions. To address RQ7.1, we used three metrics: precision, recall, and F_1-Score. The first row in Table 7.2 shows formulas for computing these metrics. In these formulas, TP represents the number of correct ACP rules identified by Text2Policy, whereas FP and FN represent the number of incorrect and missing ACP rules, respectively, identified by Text2Policy. To address RQ7.2 and RQ7.3, we used the accuracy metric shown in the second and third rows of Table 7.2, respectively.

Table 7.3: Evaluation results of RQ7.1

Subjects	**# Sent.**	**# ACP Sent.**	**# Ident.**	FP	FN	$Prec$	Rec	F_1
iTrust	448	117	119	16	14	86.6%	88.0%	87.3
IBMApp	479	24	23	0	1	100.0%	95.8%	97.9
Total	927	141	142	16	15	88.7%	89.4%	89.1

7.5.1 Subjects and Evaluation Setup

In our evaluations, we used three categories of subjects for addressing the three research questions. First, we used 37 use cases from iTrust [197, 13]. iTrust is an open source health-care application that provides various features such as maintaining medical history of patients, storing communications with doctors, identifying primary caregivers, and sharing satisfaction results. The requirements documents and source code of iTrust are publicly available on its web site. iTrust requirements specification has 37 use cases, 448 use-case sentences, 10 non-functional-requirement sentences, and 8 constraint sentences. The iTrust requirements specification also has a section, called Glossary, that describes the roles of users who interact with the system.

We preprocessed the iTrust use cases so that the format of the use cases can be processed by Text2Policy. In particular, we removed symbols (e.g., [E1] and [S1]) that cannot be parsed by our approach. We replaced some names with comments quoted in parenthesis. For example, when we see *A user (an LHCP or patient)*, we replaced *A user* with *an LHCP or patient*. We separated sentences by replacing / with *or*. We also separated long sentences that span more than 2 or 3 lines, since such style affects the precision of shallow parsing. The preprocessed documents of the iTrust use cases are available on our project web site [24].

Second, we collected 100 ACP sentences from 17 sources (published articles and public web sites). These ACP sentences and 117 NL ACP rules from the iTrust use cases are the subjects for our evaluation to address RQ2. The document that contains the collected ACP sentences and their original sources can be downloaded from our project web site [24].

Third, we used 25 use cases from a module in a proprietary IBM enterprise application. Due to confidentiality, we refer to this application as *IBMApp*. This module belongs to the financial domain.

We next discuss the results of our evaluations in terms of the effectiveness of Text2Policy in identifying ACP sentences and extracting ACP rules from NL documents and in extracting action steps from use cases.

7.5.2 RQ7.1: ACP-Sentence Identification

In this section, we address the research question RQ1 of how effectively Text2Policy identifies ACP sentences in NL documents. To address this question, we first manually inspected the use cases of iTrust to identify ACP sentences. We then applied Text2Policy to identify ACP sentences and compared those results with our results of manual inspection to identify the numbers of true positives, false positives, and false negatives. We further computed precision and recall values based on these numbers.

Among 448 use-case sentences in the iTrust use cases, we manually identified 117 ACP sentences. Among 479 use-case sentences in the *IBMApp* use cases, we manually identified 24 ACP sentences. We then manually classified these ACP sentences identified by Text2Policy as correct sentences and false positives, and manually identified false negatives.

Table 7.3 shows the results of RQ1 for both the subjects. Column "Subjects" lists the name of the subjects. Columns "# Sent." and "# ACP Sent." show the number of use-case sentences and the number of ACP sentences. Column "# Ident." shows the number of identified ACP sentences, and Columns "FP" and "FN" show the numbers of false positives and false negatives. Based on these numbers, Columns "$Prec$", "Rec", and "F_1" show the computed precision, recall, and F_1-score. For iTrust, the results show that Text2Policy identified 119 sentences with 16 false positives and 14 false negatives. For *IBMApp*, Text2Policy identified 23 sentences with 0 false positive and 1 false negative. The results show that our semantic patterns help identify ACP sentences more precisely on the *IBMApp* use cases. One explanation could be that proprietary use cases are often of higher quality compared to open-source use cases and conform to simple grammatical patterns.

We first provide an example to describe how Text2Policy correctly identifies ACP sentences. One of the ACP sentences that Text2Policy correctly identifies ACP rules is "*HCPs can modify or delete the fields of the office visit information.*" [197, 13]. Our semantic pattern *Modal Verb in Main Verb Group* helps identify that the main verb contains the modal verb *can* and correctly identify the sentence as an ACP sentence.

We next provide some examples to describe how Text2Policy produces false positives and negatives. One false positive produced by Text2Policy is "*The instructions can contain numbers, characters...*" [197, 13], which matches the pattern *Modal Verb in Main Verb Group*. However, this sentence describes a requirement on password setting, instead of an ACP rule. These false positives can be reduced by expanding the domain dictionary to include commonly used nouns that are unlikely to be systems or system actors. The sentence that cannot be identified by Text2Policy is "*The LHCP can select a patient to obtain additional information about a patient.*" [197, 13]. Due to precision in parsing long phrases, the underlying shallow parser fails to identify *to obtain additional information about a patient* as a to-infinitive phrase, causing a false

Table 7.4: Evaluation results of RQ7.2

Subjects	# ACP Sent.	# Extracted	Accu.
iTrust	217	187	86.2%
IBMApp	24	21	87.5%
Total	241	208	86.3%

negative for our approach. These false negatives can be reduced by improving the underlying shallow parser using more training corpus in future work.

7.5.3 RQ7.2: Accuracy of ACP Extraction

In this section, we address the research question RQ2 of how effectively Text2Policy extracts ACP rules from ACP sentences. To address this question, we manually extracted ACP rules from these ACP sentences. We next applied Text2Policy and compared the results with our manually extracted results. We compute the accuracy of the ACP extraction using the number of ACP sentences from which Text2Policy correctly extracts ACPs and the total number of ACP sentences.

Table 7.4 shows the results of RQ2. Column "Subject" lists the name of the subjects. Columns "# ACP Sent." and "# Extracted" show the total number of ACP sentences and the number of ACP sentences from which Text2Policy correctly extracts ACPs. The statistics shown by these two columns are used to compute the accuracy shown in Column "Accu.". Among 217 ACP sentences of iTrust (including 117 from iTrust use cases), Text2Policy correctly extracts ACP rules from 187 ACP sentences, achieving the accuracy of 86.2%. Among 24 ACP sentences in the 25 use cases of *IBMApp*, Text2Policy correctly extracts ACP rules from 21 ACP sentences, achieving the accuracy of 87.5%.

We first provide an example to describe how Text2Policy correctly extracts some ACP rules. One of the sentences from which Text2Policy correctly extracts ACP rules is "*The administrator is not allowed through the system interface to delete an existing entry.*" [197, 13]. Our semantic pattern *Passive Voice followed by To-infinitive Phrase* helps correctly identify this ACP sentence, and correctly extract subject (*administrator*), action (*delete*), and resource (*an existing entry*) elements. Our technique of negative-meaning inference also correctly infers the policy effect to be *deny*.

We next provide examples to describe how Text2Policy fails to extract some ACP rules. One of the sentences from which Text2Policy cannot correctly extract ACP rules is "*Any subject with an e-mail name in the med.example.com domain can perform any action on any resource.*" [10]. The subject of this sentence *Any subject* is a noun phrase followed by two prepositional phrases (*with an e-mail name* and *in the med.example.com domain*). These two prepositional phrases

Table 7.5: Evaluation results of RQ7.3

Subjects	# AS Sent.	# Extracted	Accu.
iTrust	312	258	82.7%
IBMApp	455	370	81.3%
Total	767	628	81.9%

constrain the subject *Any subject*, which is not correctly handled by our current implementation. Moreover, due to the imprecision in parsing long phrases, Text2Policy fails to extract some resources from ACP sentences. In future work, we plan to develop techniques to analyze the effects of prepositional phrases and long phrases for improving the accuracy of ACP extraction.

7.5.4 RQ7.3: Accuracy of Action-Step Extraction

In this section, we address the research question RQ3 of how effectively Text2Policy extracts action steps from action-step sentences. First, we manually extracted actions steps from these action-step sentences. We next used Text2Policy to automatically extract actions steps and compared the results with our manually extracted results. We computed the accuracy of the action-step extraction by using the number of correctly extracted action-step sentences and the total number of action-step sentences.

Table 7.5 shows the results of RQ3. Column "Subject" lists the name of the subjects. Columns "# AS Sent." and "# Extracted" show the total number of action-step sentences and the number of action-step sentences from which Text2Policy correctly extracts action steps. The statistics shown by these two columns are used to compute the accuracy shown in Column "Accu.". Among 312 action-step sentences in the iTrust use cases, Text2Policy correctly extracts action steps from 258 action-step sentences, resulting in an accuracy of 82.7%. Among 455 action-step sentences in the 25 use cases of *IBMApp*, Text2Policy correctly extracts action steps from 370 action-step sentences, resulting in an accuracy of 81.3%.

We next provide examples to describe how Text2Policy fails to extract action steps. One of the action-step sentences from which Text2Policy fails to extract action steps is "*The HCP must provide instructions, or else they cannot add the prescription.*" [197, 13]. The reason is that the current implementation of our approach does not handle the subordinate conjunctions *or else*. Another example sentence is "*The public health agent can send a fake email message to the adverse event reporter to gain more information about the report.*" [197, 13]. For such long sentences with prepositional phrases *to the adverse event reporter to gain more information about the report* after the object of the sentence *a fake email message*, the underlying shallow parser of our approach cannot correctly identify the grammatical functions. We plan to study

more use cases on health-care applications and improve the underlying shallow parser with more patterns to identify grammatical functions of action-step sentences.

7.5.5 Detected Inconsistency

Our approach validates the extracted access requests against the extracted ACPs. Although our approach does not detect violations of the extracted ACPs in our evaluations, our approach identifies a few action steps that do not match any extracted ACPs. To study why these action steps do not match any ACPs, we further apply union on the specifications of action steps to collect the information of what users perform what actions on what resources. From this information, we find that *editor*, one of the system users, is not matched with any subjects in the extracted ACPs. We then check the glossary of the iTrust requirements and the use-case diagram. We confirm that *editor* in fact refers to *HCP*, *admin*, and *all users* in use cases 1, 2, and 4, respectively. Such name inconsistencies can be easily identified by combining validation of ACP rules and using the union information of extracted action steps.

7.6 Discussion

In this section, we discuss applications and limitations of our current approach and propose directions for future work.

Construction of Complete ACPs. From the extracted ACPs, our approach automatically generates formal specifications of ACPs. These formal ACPs can assist the construction of complete ACPs in three ways: (1) these formal ACPs can be used to validate manually specified ACPs for identifying inconsistencies; (2) these formal ACPs can serve as an initial version of ACPs for policy authors to improve, greatly reducing manual effort in extracting ACPs from NL software documents; (3) combined with specified ACPs, these formal ACPs can be fed to automated ACP-verification approaches for checking correctness, such as static verification [110] and dynamic verification via access-request generation [138, 137].

ACP Modelling in the Absence of Security Requirements. In the absence of security requirements, our approach can still provide a solution to assist policy authors to model ACPs for a system. Our approach first extracts deny ACPs and action steps from functional requirements. Besides deriving access requests from action steps, we can also derive a permit ACP rule from each action step. With the extracted and derived ACPs, policy authors have two ways to model ACPs: (1) the policy authors can apply the extracted deny ACPs and add a policy rule to permit all other accesses; (2) the policy authors can combine the extracted deny ACPs and the derived permit ACPs, and add a policy rule to deny all other accesses.

Cooperation Between Tool and Human. The extracted policies can serve as an initial version of ACPs for policy authors to improve, advocating cooperation between the tool and the user [205]: the tool reports policies extracted with low confidence and the user can refine them to get better results. Currently, our implementation is built on an Eclipse-based IDE and can provide visual feedback of extracted policies, e.g., extracted subjects, actions, and resources. We plan to improve the IDE to better support the cooperation between the tool and the user. In addition, to improve the precision of the semantic analysis (such as anaphora resolution), we can apply ambiguity-analysis techniques [210, 51] on the NL software documents to identify nocuous ambiguities, and ask the user to resolve the ambiguities before our approach is applied to extract policies.

ACP-Rule Ordering. Our current approach extracts ACP rules from sentences without considering the ordering of the rules. Doing so may cause security holes in the extracted ACP rules. We plan to study the extracted ACP rules and develop new techniques to extract ordering for the ACP rules.

Context-aware Analysis in Action-Step Extraction. A sequence of action steps may have several state transitions. The techniques of actor-flow tracking and perspective conversion in our approach partially address the context-aware analysis in action-step extraction. For example, a customer may not pay the order if he has not selected an order. We plan to develop techniques to deal with state transitions during action-step extraction.

Other Policy Models. In our evaluations, we encountered some ACP sentences that describe conditions for ACP rules. For example, the ACP sentence "*During the meeting phase, reviewer can read the scores for paper if reviewer has submitted a review for paper.*" [70] contains an if-condition to constrain the ACP rule. Without correct extraction of the condition, the produced specification of ACP rules is incomplete and requires policy authors to manually fix the incompleteness issue. Besides the issue of conditions, our current approach cannot handle multi-level models [94] or workflow models [166] for access control. We plan to extend our approach to support these new models and provide new semantic patterns for identifying new styles. In addition, our approach can be extended to support privacy policies, such as HIPAA privacy policies[2]. Supporting extraction of HIPPA policies requires more sophisticated semantic models to address new challenges, such as condition rules, rule combination, and rule ordering. We plan to investigate techniques to deal with new challenges of extracting HIPAA privacy policies.

[2]http://crypto.stanford.edu/privacy/HIPAA/

7.7 Summary

Access Control Policies (ACP) specify which principals such as users have access to which resources. Ensuring the correctness and consistency of ACPs is crucial to prevent security vulnerabilities. However, in practice, ACPs are commonly written in Natural Language (NL) and buried in large documents such as requirements documents, not amenable for automated techniques to check for correctness and consistency. It is tedious to manually extract ACPs from these NL documents and validate NL functional requirements such as use cases against ACPs for detecting inconsistencies.

To address these challenges, we have proposed an approach, called Text2Policy, which extracts ACPs from NL software documents and produces formal specifications. Our approach incorporates syntactic and semantic NL analyses around models such as ACP and action-step models and extracts model instances from NL software documents. From the extracted ACPs, our approach automatically generates machine-enforceable ACPs (in formal languages such as XACML) that can be automatically checked for correctness. From the extracted action steps, our approach automatically extracts resource-access information, which can be used for automatic validation against specified or extracted ACPs for detecting inconsistencies.

Text2Policy supports the behavior-diagnosis cooperation for security policy extraction. Based on Text2Policy, our approach provides a behavior-diagnosis interface that shows the security policies and explain the security policies by showing the policy-witness and policy-violation scenarios. In this cooperation, the tools extract security policies for the users to inspect and explain the security policies by showing the policy-witness and policy-violation scenarios. Based on the users' domain knowledge about the software system's functionality and security requirements, the users inspect the policy-witness and policy-violation scenarios (instantiations of the security policies) to determine whether these scenarios are expected for the software system, and make decisions to include or reject the security policies.

We have conducted evaluations on iTrust use cases, ACP sentences collected from 18 sources, and 25 proprietary use cases. The results show that Text2Policy effectively identifies ACP sentences with the precision of 88.7% and the recall of 89.4%, extracts ACP rules with the accuracy of 86.3%, and extracts action steps with the accuracy of 81.9%. Such results show that with customized NLP techniques, automated extraction of security policies from NL documents in a specific domain helps effectively reduce manual effort and assist policy construction and understanding.

CHAPTER 8

Behavior-Diagnosis Cooperation for Performance Analysis

8.1 Introduction

Performance problems exist widely in released software [118, 89]. As a type of widespread performance problems, software hangs cause unresponsiveness of software applications [194, 172]. A recent study of hang problems [172] shows that 27.04% of the 233 studied hang faults are caused by time-consuming operations in responsive actions[1], such as expensive computations in the UI thread for GUI applications. Among the expensive operations that cause hang problems, some of these operations are constantly expensive (such as server initializations), whereas some of them depend on the input workloads. These problems are referred to as workload-dependent performance bottlenecks (WDPBs). WDPBs are usually caused by workload-dependent loops (referred to as WDPB loops)[2] that contain certain relatively expensive operations, such as temporary-object creation/destruction [207], file I/O, and UI updates.

To remove WDPBs, a typical solution is to move expensive operations out of the responsive actions, such as spawning separate threads to handle expensive operations in the background for GUI applications [15], or adding program logics to limit the size of workloads (e.g., allowing up to only a specific size k of workloads or processing only the first k items of a workload).

Although WDPBs can be identified with traditional approaches, such as performance testing and single-execution profiling (e.g., call-tree profiling [31, 96, 32] and stack sampling [101]), such traditional approaches are ineffective, suffering from two major issues: the insufficiency issue and the incompleteness issue. First, performance testing mainly relies on black-box random

[1] Actions that are expected to return instantly.

[2] A loop whose iteration count depends on the input workload.

testing or manual input design, often insufficient to identify WDPBs that may not surface on small or even relatively large workloads [143]. A huge amount of existing legacy software lacks workload specifications, and specifications from performance engineers tend to be outdated over time. It often remains unclear to performance testers on how large is large enough for workloads to expose WDPBs (if any indeed exists in the application under test). A recent study [118] shows that 41 out of 109 studied performance faults are due to wrong assumption of workloads. Second, by increasing the input workload, single-execution profiling may reveal the most expensive WDPBs, but it is often incomplete in capturing all the WDPBs that may cause performance problems when the workload size increases. For example, given a large workload, some WDPBs' cost may occupy more than 90% of the total cost, dominating the cost of other WDPBs. In other words, some important WDPBs can be overshadowed by other WDPBs.

To address these two issues suffered by traditional approaches, our research contributes a novel predictive approach, called ΔInfer. ΔInfer *predicts* occurrences of WDPB loops within a GUI application[3] under large future workloads (that have not been generated or executed yet) in contrast to existing approaches on performance testing, which require the generated and executed workloads to directly expose these WDPB loops. In addition, to gain the prediction power, our approach infers complexity models of program locations within the application under analysis from profiles of *multiple* workloads instead of the profile from just a single workload, which existing approaches on single-execution profiling focus on. Our approach then infers complexity models for loops based on the complexity models of program locations at the loop bodies. Such complexity model for a loop captures the relationship between the iteration count of the loop and the workload size, and then is used to predict the iteration count of the loop given a workload.

To identify WDPB loops in GUI applications, our approach addresses two significant challenges: complex contexts and implicit loops. First, in GUI applications, developers usually write code as handlers for various UI events (e.g., button clicks or item selections). When an event is fired, the corresponding handlers would be invoked. Such event-driven nature causes a program location to be invoked in different contexts. Thus, a program location may exhibit quite different execution complexities under different calling contexts, posing challenges for a complexity model to accurately model its complexity. Second, among the most widely-used UI controls, multi-item UI controls (e.g., `ListView` or `TreeView`) [171] may fire events for each item, behaving like an *implicit loop* that invokes the handlers repetitively. Such implicit loops do not have explicit loop statements in the application, posing challenges for manual inspection or static analysis [100, 36, 213] to identify the WDPB loops.

[3]Among applications with WDPBs, our research focuses on identifying WDPBs in GUI applications, since responsiveness in GUI applications is a major source of performance problems [194, 172].

To address the aforementioned challenges, our ΔInfer approach incorporates a novel general concept: *context-sensitive delta inference*, which consists of two major parts: *temporal inference* (inferring differences between executions) and *spatial inference* (inferring differences between program locations).

Temporal Inference. The temporal inference analyzes the *differences of execution counts* of a specific program location among its executions under different workloads to infer complexity models. ΔInfer employs least-squares regressions (such as linear and power-law regressions) [52] to infer a complexity model that uses the workload to predict a program location's execution count.

To address the challenge of complex contexts posed by GUI applications, our approach is context-sensitive: our approach infers complexity models from behaviors exhibited by the executions of a program location under the same calling context (from its caller up to the root function such as a main function or thread-start function), instead of executions of the program location under different calling contexts.

To improve the accuracy of the inferred complexity models, ΔInfer starts with training profiles of workloads selected from the representative usage, and iteratively selects new workloads to obtain new profiles based on the prediction accuracy of the inferred models in previous iterations. The iteration of the model inference and refinement continues until the model accuracy reaches a specified threshold.

Spatial Inference. The spatial inference analyzes *differences of complexity models across program locations* to identify workload-dependent loops as WDPB candidates. If a complexity model for the workload is used to describe a program location's execution count (i.e., $count = f(workload)$), it can be observed that workload-dependent loop raises the complexity model of the program locations inside the loop body to a higher order (such as *constant* to *linear*), and results in a complexity transition. Thus, the *order differences* between complexity models of *different program locations*, i.e., complexity transitions, can be used to effectively identify workload-dependent loops. To identify complexity transitions as workload-dependent loops, the spatial inference abstracts orders from the inferred complexity models and compares the orders for a loop's entry point and program locations inside the loop body. If we denote program locations as methods, then we compare the orders of caller-callee pairs, since callees inside a loop body would exhibit different complexity orders.

To address the challenge of implicit loops posed by GUI applications, ΔInfer uses complexity transitions from certain UI library calls to the application code to identify implicit loops. All the complexity transitions inferred by ΔInfer are considered as WDPB candidates. Based on the complexity models and the average cost per execution obtained from the profiles, ΔInfer predicts costs of the complexity transitions on large workloads to identify WDPBs.

Based on ΔInfer, our approach provides a behavior-diagnosis interface that shows the complexity models of methods besides showing the costs of the executed methods. Our approach infers complexity models of workload-dependent loops from multiple executions of the software on multiple workloads. Such complexity models inferred from the concrete executions explain how the costs of certain methods grow in larger workloads, enabling the behavior-diagnosis cooperation for performance analysis. In this cooperation, the tools compute the costs of the executed methods for the users to inspect and explain the cost growth of the methods by showing the inferred complexity models. Based on the users' domain knowledge about the functionality and performance requirements of the program under analysis, the users inspect the complexity models of the methods to determine whether the cost growth represented by the models are expected for the methods, and make decisions on whether the methods are performance faults.

8.2 Problem Formulation

In this section, we formalize the problem of identifying complexity transitions. For a given application A, we use the term *location*, l, to denote a program location (e.g., a basic block in a method or a method itself) of A, and *cost*, y, to denote a location's performance (e.g., execution count or time). To formulate our context-sensitive analysis and complexity transitions, we first define the *call graph* G for A and the *calling context* c of a location l in A.

Definition 1 *A **call graph** is a directed graph $G(E, V)$, where each vertex $v \in V$ denotes a unique method, and each edge $e(a, b) \in E$ denotes a calling relationship from a to b.*

Without losing the generality, we use the term *belonging method* to denote a method where l is in when l represents a basic block, or a method represented by l. l's belonging method corresponds to a vertex v in G.

Definition 2 *A **calling context**, c, of a location l is a call path from the root of the call graph (usually a main function or thread-start function) to the parent vertex (caller) of vertex v corresponding to l's belonging method.*

To simplify description, we denote a location l under a calling context c as l_c in the rest of the chapter. Using the calling context, we then define the call-tree profiling [31] used in our approach.

Definition 3 *An **execution profile**, P, obtained by executing an application A on a given input, is a call-tree profile that records the execution counts of each location l_c in A.*

An input to A can have a set of parameters that characterize the input from different aspects. Based on the scenarios of A, we identify workload parameters $(W_1, \ldots, W_d)$ that could potentially influence performance, such as the number of lines or the number of characters for the text input to a text editor. For k workloads, we have a vector of values for each workload parameter W_d $(< w_{d,1}, \ldots, w_{d,k} >)$ to denote the values of W_i for these k workloads. After executing the application A on k workloads to obtain k profiles, we have a vector of counters for each location l_c $(< y_{l_c,1}, \ldots, y_{l_c,k} >)$. Based on these vectors, we then define the *k-profile graph* as below.

Definition 4 *A **k-profile graph** is an annotated call graph, $G(E, V)$, where a location l with its corresponding vertex is annotated with a vector of counters for l on k workloads for each of its calling context c.*

For each location l with its corresponding vertex in the k-profile graph, our approach infers **complexity models** using regression learning.

Definition 5 *Given a workload parameter W, a **complexity model** of a location l under the calling context c is a function $f_{l,c}(W)$ that predicts l's execution counts in terms of values of W under the calling context c.*

Based on the definition of a complexity model, we denote the exponent of the highest order term of the complexity model as the *order* of the complexity model, denoted as $O(f_{l,c}(W))$. We next define a *complexity transition*.

Definition 6 *Given a workload parameter W, a **complexity transition** is a pair (n, M), such that*

1. *n is a vertex (method) in the k-profile graph and M is a subset of children vertices (callees) of n;*
2. *$f_{n,c}(W)$ is the complexity model of n under the calling context c, and $f_{l_i,c_i}(W)$ is the complexity model of the location l_i, where l_i is a location in M and the calling context c_i is c concatenated with n.*
3. *$O(f_{l_i,c_i}(W))$ is at least 1 **more than** $O(f_{n,c}(W))$;*
4. *$\forall l_i, l_j \in M, i \neq j$, $O(f_{l_i,c_i}(W)) = O(f_{l_j,c_j}(W))$.*

The definition ensures that a complexity transition captures the workload-dependent loops whose iteration bounds have the same order inside the method n under the calling context c. To simplify our description in the rest of the chapter, we use methods as the locations.

```
1  void CPanel::OnRefreshStatusBar() { // PB_1
2    ...
3    GetOperatedItemIndices(indices);
4    _statusBar.SetText(...); // UI operation
5    ... }
6  void CPanel::GetOperatedItemIndices(CRecordVector<UInt32> &indices) const {
7    GetSelectedItemsIndices(indices);
8     ... }
9  void GetSelectedItemsIndices(CRecordVector<UInt32> &indices) {
10   indices.Clear();
11   for (int i = 0; i < _selectedStatusVector.Size(); i++) // PB_2
12     if (_selectedStatusVector[i]) indices.Add(i);
13   ... }
```

Figure 8.1: Two WDPBs found in the 7-Zip file manager [3]

8.3 Examples

In this section, we use an example to illustrate how ΔInfer identifies WDPBs. Figure 8.1 shows two WDPBs found in the 7-Zip file manager [3] written in C++. The first WDPB (PB_1) is caused by the method `CPanel::OnRefreshStatusBar`, which contains a non-trivial UI update operation (Line 4). PB_1 is invoked when a selection-change event is fired. The second WDPB (PB_2) is caused by the method `GetSelectedItemsIndices`, which contains a workload-dependent loop L (Lines 11-12). Let us assume that the number of files in the current folder is n; if a user clicks a file after selecting all files, the selection-change event will be fired for each file. Such repeated firing will cause PB_1 to be invoked n times (refreshing the status bar n times), and PB_2 to be executed n^2 times. Thus, when n is large, PB_1 and PB_2 would be very expensive and cause the 7-Zip file manager [3] to hang.

With ΔInfer, developers can perform the selection-change action to obtain the profiles on multiple workloads, and use the prediction results of ΔInfer to identify WDPBs on large workloads. To simplify the description, here we simply assume that the selected workload values are 50, 100, and 200 files, and we obtain the profiles P_{50}, P_{100}, and P_{200}.

With these profiles as input, ΔInfer infers complexity models by using regression learning to fit the execution counts of methods to workload sizes. In Figure 8.1, `CPanel::OnRefreshStatusBar` is associated with a linear complexity model, and any method call inside the loop (Lines 11-12) is associated with a quadratic complexity model. By inferring complexity transitions from lower order to higher order (e.g., *constant to linear* and *linear to quadratic*), the complexity transition from some UI library call (not shown in Figure 8.1) to `CPanel::OnRefreshStatusBar()` helps identify the implicit loop, and the complexity transition from `GetSelectedItemsIndices` to

`_selectedStatusVector.Size` helps identify the loop L. These workload-dependent loops are considered as WDPB candidates.

To predict whether PB_1 and PB_2 would cause performance problems on large workloads, ΔInfer predicts the execution counts of PB_1 and PB_2 when the workloads become 10 or 100 times larger (i.e., 500 or 5000). With the predicted execution counts, ΔInfer then uses their average costs (e.g., execution time) per execution to compute their estimated costs on these large workloads. Although PB_2 belongs to the callees of PB_1, the complexity model of PB_2 has a higher order than PB_1's model. Thus, ΔInfer separates the predicted costs of PB_2 from PB_1.

With the predicted costs on large workloads, ΔInfer ranks PB_1 and PB_2 as the top 2 complexity transitions (others are not illustrated here due to space limit), and their combined costs are 10 times of P_{100}'s cost when the workload is 5000. Such costs significantly degrade the performance and cause the file manager to hang. Recall that PB_1 contains a UI-update operation and has a non-trivial cost per execution, and the cost of PB_2 grows much faster than PB_1 due to PB_2's n^2 complexity model, even though the cost per execution for PB_2 is not that large.

8.4 Approach Overview

In this section, we present the overview of ΔInfer. As shown in Figure 8.2, ΔInfer consists of two major parts: temporal inference and spatial inference.

The temporal inference accepts profiles of different workloads as input and infers context-sensitive complexity models. ΔInfer starts by applying regression learning on an initial set of profiles to infer complexity models, and iteratively selects new workloads to refine the inferred models. To validate the inferred models after each iteration, ΔInfer uses a random-validation strategy. ΔInfer repeats the model inference and refinement until the model accuracy reaches a pre-specified threshold.

The spatial inference accepts the inferred complexity models from temporal inference and infers complexity transitions as WDPB candidates. In particular, to make the complexity models comparable, ΔInfer abstracts orders from complexity models, and compares the orders of caller-callee pairs to infer complexity transitions as WDPB candidates. ΔInfer then predicts costs of the complexity transitions on large workloads to identify WDPBs.

8.5 Temporal Inference

This section describes how the technique of temporal inference infers and refines complexity models.

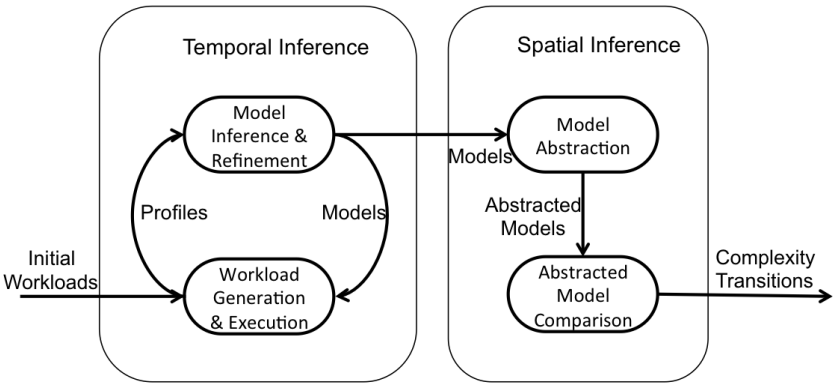


Figure 8.2: Overview of ΔInfer

8.5.1 Workload Generation and Execution

Our approach focuses on detecting scenario-specific WDPBs for an application under analysis. Each scenario is assumed to use a specific configuration of the GUI, such as switching the GUI to the "Wrap Line" mode for a text editor. Based on the chosen scenarios, performance analysts select appropriate performance metrics (such as execution time or energy cost), and characterize the input as performance-relevant workload parameters [1], such as *# lines* in a document as the input to a text editor. When we generate workloads based on a parameter W (referred to as the focused workload parameter), we vary workloads only on W, while the values of other parameters remain the same. For example, when we vary the focused workload parameter *# lines* to generate different workloads, we keep constant the other parameters such as *# of characters in a line*. Doing so can help us avoid the difficulties on inferring models for multiple parameters.

To select an initial set of workloads, performance analysts are expected to define the representative value range (RVR) for the focused workload parameter. For example, the RVR for *# lines* in a document can be $[1, 1280]$. Often the time, performance analysts and developers are well aware of RVRs and can agree on RVRs with a certain variance, but it is difficult to know a triggering workload value for an unknown performance bottleneck. Within the RVR, performance analysts can select an initial value and vary the initial value via arithmetic progression or geometric progression to obtain sorted inputs [213], or can choose the values randomly. For the least-squares regression used by our approach, a guideline for selecting the initial values is to avoid selecting a very small workload, such as 1 or 2 files, as the initial workload for a file manager. We empirically find that such small workloads produce noise in the inferred models, consistent with the finding by Goldsmith et al. [95].

To obtain execution profiles, we instrument the application and execute the application on the chosen workloads. The profiles used by our approach are call-tree profiles, which measure execution counts of program locations in the instrumented application, and distribute the total execution counts of a location for each of its calling contexts [31]. In this chapter, since we focus on GUI applications whose responsive action is in the UI thread, our approach uses the execution profiles of the UI thread as input.

8.5.2 Least-Squares Regression

Given the counter vector of a location l under a calling context c ($< y_{l_c,1}, y_{l_c,2}, \ldots, y_{l_c,k} >$) and the value vector of a workload parameter W ($< w_1, w_2, \ldots, w_k >$), our approach uses least-squares regressions [52], including linear and power law regressions, to infer a complexity model using a set of data points $(w_i, y_{l_c,i})$.

Linear Regression. Linear regression infers a complexity model $y = A + Bw$ and predicts $y_{l_c,i}$ as $\hat{y_{l_c,i}} = A + Bw_i$. The difference $y_{l_c,i} - \hat{y_{l_c,i}}$ is called the residual of the fit at $(w_i, y_{l_c,i})$. Linear regression finds parameters A and B to minimize the sum of squared residuals, $Q(A, B)$, where $Q(A, B) = \sum_{i=1}^{k}(y_{l_c,i} - (A + Bw_i))^2$.

Power-law Regression. Power-law regression infers a complexity model $y = Aw^B$ and predicts $y_{l_c,i}$ as $\hat{y_{l_c,i}} = Aw_i^B$. Power-law regression finds parameters A and B to minimize the sum of squared residuals, $Q(A, B)$, where
$Q(A, B) = \sum_{i=1}^{k}(y_{l_c,i} - (Aw_i^B))^2$.

To measure how good the models fit the data points, our approach computes the correlation coefficient R^2. For the linear regression, the R^2 is defined as below:

$$R^2 = \frac{(\sum_{i=1}^{k} wy - k\bar{w}\bar{y})^2}{(\sum_{i=1}^{k} w^2 - k\bar{w}^2)(\sum_{i=1}^{k} y^2 - k\bar{y}^2)}$$

By replacing w with $ln(w)$ and y with $ln(y)$, we can transform a power-law regression model to a linear regression model: $ln(y) = ln(Aln(w)^B) = ln(A) + Bln(w)$. Thus, R^2 is applicable to power-law regressions by replacing w with $ln(w)$ and y with $ln(y)$.

Using both linear and power-law regressions, regressions can fit a set of data points $(w_i, y_{l_c,i})$ of a location l to two complexity models. Our approach selects the complexity model with better R^2 as the complexity model for l.

8.5.3 Model Validation

The model validation provides the relative prediction error of the inferred models. While the correlation coefficient R^2 measures how the models fit the given data points, the relative prediction error measures how good the prediction accuracy of the models is. For each iteration of

Algorithm 4 $ModelInfer$

Require: P as a set of profiles, VP as a set of validation profiles
Ensure: M as complexity models

$pc = -1$ // previous model count
$e_{pre} = -1$ // previous error
for $ite = 0; ite < max; ite = ite + 1$ **do**
  $kG = AlignProfiles(P)$
  $x = GetWorkloads(P)$
  $M = RegressionLearning(x, kG)$
  $e_M = \{\}$
  for all vp in VP **do**
    $w = GetWorkload(vp)$
    $e_{vp} = 0.0$
    for all m in M **do**
      $r_w = Predict(w, m)$
      $a_m = GetActual(vp, m)$
      $e_{vp} = e_{vp} + \frac{Abs(a_m - r_w)}{a_m}$
    end for
    $e_M = e_M.Add(\frac{e_{vp}}{M.Count})$
  end for
  $e_{total} = \frac{Sum(e_M)}{VP.Count}$
  if $e_{pre} - e_{total} < threshold_{imp}$ **then**
    break // improvement is below the threshold
  end if
  if $e_{total} < threshold_e$ AND $pc == M.Count$ **then**
    break // accuracy is acceptable
  else
    $np = NewWorkload(P, VP, e_M)$
    $P = P.Add(np)$
    $pc = M.Count$
    $e_{pre} = e_{total}$
  end if
end for
return M

model validation, the model validation compares the predicted values of the inferred models to the values of the corresponding locations in all validation profiles, and computes the average relative error.

In our current approach, we use a random-validation strategy for the model validation. We randomly select a set of workload values from a validation value range (VVR) pre-determined based on a guideline (described below), and obtain a set of corresponding validation profiles for determining the accuracy of the models. To prevent the models from overfitting the values within the RVR and better validate the prediction accuracy of the inferred models, VVR must include RVR and the value range should be larger than the RVR. For example, if the RVR is $[1, 500]$, we may set the VVR as $[1, 1000]$. We suggest that the range of VVR should be at least 2 times larger than the RVR. However, a very large validation range would require more iterations to refine the models, and large workload values cause long processing time in obtaining validation profiles. Thus, a very large validation range is not cost-effective for the iterative model inference and refinement.

The advantage of the random-validation strategy is that the validation workloads are distributed across the validation range, preventing models from over-fitting a specific range of values within the whole validation range. Note that the randomly selected workload values cannot be the same as the workload values used to infer the models.

8.5.4 Model Inference and Refinement

Model inference and refinement accept as input a set of profiles on initial workloads and a set of validation profiles, iteratively select new workloads to improve the inferred models, and output the inferred complexity models. These inferred complexity models are then associated with the corresponding vertices in the k-profile graph.

The number of initial workloads and the number of new workloads selected for each iteration can be configured by performance analysts. The configuration mainly depends on the value range of the workload parameter and the time taken to obtain a profile. Adding the new workloads in subsequent iterations improves the average relative error of the inferred models. We terminate the iterations if the average prediction error falls below a predefined prediction-error threshold (e.g., 5%). In certain cases, after adding new profiles, the improvement on the accuracies of the inferred models may be marginal or even negative, and the number of the inferred models whose R^2 is above a predefined R^2 threshold (referred to as $threshold_{R^2}$) may not change. We terminate the iterations for such cases to prevent infinite iterations.

Algorithm 4 shows the details of our iterative algorithm for model inference and refinement. The main part of the algorithm is the iteration cycle of inferring and refining complexity models (Lines 3-30), where the guard condition of the loop at Line 3 ensures that the algorithm termi-

nates after the predefined maximum number of iterations has been reached. Lines 1-2 initialize the model count pc and the prediction error e_{pre}. pc records the number of models whose R^2 are above $threshold_{R^2}$ (e.g., 0.9) in the previous iteration, and e_{pre} records the average prediction error in the previous iteration. Within the iteration cycle, $ModelInfer$ first infers complexity models (Lines 4-6) based on the set of profiles. $ModelInfer$ obtains a k-profile graph by aligning the set of profiles (Line 4) and retrieves the workload values from the profiles (Line 5). $ModelInfer$ then applies regression learning with the workload values and the k-profile graph as input to infer the complexity models (Line 6). Here regression learning returns only the inferred models whose R^2 is above $threshold_{R^2}$. In the next step, $ModelInfer$ computes the errors e_M for each validation profile (Lines 8-17) and the average error e_{total} for all validation profiles (Line 18). Based on the computed errors, $ModelInfer$ terminates the iteration and returns the current models (Line 31) if one of the following two conditions is satisfied: (1) the improvement of the average error is less than the threshold of model improvement $threshold_{imp}$ (Line 19); (2) the average error is less than the threshold of prediction error $threshold_e$ and $M.Count$ is the same as the previous iteration pc (Line 22). If neither of the condition is satisfied, $ModelInfer$ selects a new workload for obtaining a new profile (Lines 25-26), and updates the average error e_{pre} and model count pc for the next iteration. After the iteration terminates, these complexity models are then associated with the corresponding vertices in the k-profile graph.

Aligning Profiles. Given k execution profiles on k workloads, our approach aligns the locations in the profiles using calling contexts, and represents the execution counts of each location l under each calling contex c in k profiles as a vector: $(y_{l_c,1}, y_{l_c,2}, \ldots, y_{l_c,k})$. Our approach then builds the k-profile graph by associating the vectors with each location.

Selecting New Workloads. Our workload-augmentation mechanism ($NewWorkload$ at Line 25) is based on the assumption that a new workload at the area with the highest prediction error improves most the prediction errors of the models. $NewWorkload$ first finds the validation profile p_e that has the highest prediction error based on e_M, and then identifies a profile in P whose workload value w_c is closest to the workload value w_e of p_e. $NewWorkload$ returns the center of w_c and w_e as the new workload value. If the new workload value exceeds the RVR, our approach doubles the ranges of RVR and VVR, and selects a new validation profile in the extended range not overlapping with the original VVR. By doing so, our approach may adaptively evolve the ranges of RVR and VVR to improve the model accuracies.

8.6 Spatial Inference

This section describes how the technique of spatial inference infers complexity transitions by comparing abstracted models and predicts costs of complexity transitions on large workloads.

8.6.1 Abstraction of Model

For each vertex associated with a complexity model in the k-profile graph, model abstraction extracts the exponent of the highest order term from the complexity model as the order of the complexity model: (1) For a complexity model inferred by linear regressions ($y = A + Bw$), the abstracted order of the model is 1 if B is larger than 0; otherwise, the abstracted order of the model is 0. (2) For a complexity model whose orders are decimal values, the abstracted order of the model is the closest integer. (3) For a complexity model whose R^2 is below $threshold_{R^2}$, the abstracted order of the model is 0.

These orders are used as the abstracted models for each complexity model in the k-profile graph, making the complexity models comparable for each caller-callee pair.

Algorithm 5 $TransInfer$

Require: G as a k-profile graph with vertices associated with abstracted models
Ensure: T as complexity transitions

```
1: T = {} // empty set
2: for all v in G.vertices do
3:   for all child in v.Children do
4:     for all c in v.Contexts do
5:       cc = c.append(v) // children context
6:       if child.model(cc).O >= v.model(c).O + 1 then
7:         trans = T.GetTransitions(v, c)
8:         found = false
9:         for all tran in trans do
10:          if tran.model.O == child.model(cc).O then
11:            found = true
12:            tran.AddVertex(child)
13:          end if
14:        end for
15:        if !found then
16:          T.AddTransition(v, c, child)
17:        end if
18:      end if
19:    end for
20:  end for
21: end for
22: return T
```

8.6.2 Inference of Complexity Transitions

Our algorithm, $TransInfer$, accepts as input a k-profile graph G with vertices associated with abstracted models, and outputs a set of inferred complexity transitions T. The details are shown in Algorithm 5.

$TransInfer$ starts by creating an empty set of T, and retrieving a vertex v from $G.vertices$ (Lines 1-2). $TransInfer$ next iterates over each child vertex *child* of v (Line 3). For each calling context c of v (Line 4), $TransInfer$ computes the children context cc by appending v to c (Line 5). $TransInfer$ checks whether the order of the complexity model of *child* under the calling context cc is at least 1 more than the complexity model of v under the calling context c (Line 6). If the condition at Line 6 is not satisfied, $TransInfer$ continues to check the next child vertex (back to Line 3). If the condition at Line 6 is satisfied, $TransInfer$ further checks whether there exist complexity transitions whose complexity model has the same order as the complexity model of *child* (Line 10), and appends *child* to the transition if the condition at Line 10 is satisfied (Line 12). If $TransInfer$ does not find an existing transition whose complexity model matches *child.model*, $TransInfer$ creates a new complexity transition from v to *child* under the calling context c (Line 16). After all children vertices of v are checked, $TransInfer$ continues to check the next vertex v (back to Line 2). The algorithm continues until all the vertices are checked and outputs the set of complexity transitions T (Line 19), which captures workload-dependent loops, including implicit loops.

8.6.3 Cost Prediction of Complexity Transitions

To predict costs of complexity transitions (n, M) on large workloads, our approach computes the average costs per execution avg_{l_c} for each location l_c in M and the predicted execution count $pred_{l_c}$ for each location l_c. To obtain avg_{l_c} for a location l_c, our approach finds l_c on each profile p given for inferring the complexity models, computes the cost per execution $avg_{l_c,p}$ count for each profile p, and then computes avg_{l_c} by computing the average of $avg_{l_c,p}$ for each p. Given a workload w, our approach predicts the execution count $pred_{l_c}$ of a location l_c using its complexity model, and then computes the cost of l_c by multiplying $pred_{l_c}$ with avg_{l_c}. By summing up the costs of each location l_c in M, our approach obtains the predicted cost for (n, M).

8.7 Evaluations

To show the effectiveness of ΔInfer, we conducted evaluations on popular open source GUI applications (Notepad++ [6] and 7-Zip [3]). In our evaluations, we seek to answer the following research questions:

- **RQ8.1**: How effectively does ΔInfer identify WDPBs?
- **RQ8.2**: How effectively does the iterative model refinement improve the accuracy of the inferred complexity models?
- **RQ8.3**: How effectively does context-sensitive analysis improve the precision in identifying complexity transitions?

8.7.1 Subjects and Evaluation Setup

Subject Applications. Since our current implementation supports only Windows applications, we use two popular Windows applications from SourceForge [4][4] as the evaluation subjects: 7-Zip [3] and Notepad++ [6]. 7-Zip is a file archiver with high compression ratio, supporting archive file formats of 7z, zip, and so on. Our evaluations focused on the file manager of 7-Zip (7-Zip FM), a GUI tool that enables users to easily navigate and manipulate files for archiving. This application was rated by 83% of 30,081 users as recommended.[5] The version of 7-Zip used for our evaluations is 9.20, which consists of 86 files and 7,280 LOC. Notepad++ is a text editor and source-code editor for Windows, supporting tabbed editing and several programming languages (e.g., C/C++, Java, and C#). This application was rated by 94% of 14,950 users as recommended. The version of the Notepad++ used for our evaluations is 5.9.0, which consists of 396 files and 155,300 LOC.

These GUI applications represent different types of widely used GUI applications. Their GUIs consist of various types of GUI controls, such as buttons, list views, and text editors. We believe that the characteristics of these applications' WDPBs and performance faults would be representative for many other GUI applications.

Evaluation Setup. As we do not have the developer knowledge of these subject applications, we choose scenarios that would manipulate inputs, so that the performance of the applications will vary based on workloads. Based on the scenarios, we characterize some inputs as performance-relevant workload parameters. The details of the scenarios and focused workload parameters are shown in Table 8.1. Column "ID" shows the scenario ID, Column "Scenario" shows the actions performed in each scenario, and Column "W. Param" shows the workload parameters that we use to vary the focused workload values. **S1-S5** are scenarios for the 7-Zip file manager, and **S6-S10** are scenarios for Notepad++.

For Notepad++, we configure it to use the "Word Wrap" mode, so that we can test its functionality of wrapping words and other functionalities at the same time for each scenario. We executed the instrumented subject applications on each workload, performed the interactions described in each scenario, and collected the call-tree profiles after the executions.

[4]The largest open source applications and software directory

[5]All the rating data was collected on Dec. 23, 2011.

Table 8.1: Scenarios for the evaluations

ID	Scenario	W. Param
(**S1**)	open a folder	# files
(**S2**)	rename a file	# files
(**S3**)	select all items and then click the first item	# files
(**S4**)	create a folder	# files
(**S5**)	delete a file	# files
(**S6**)	open a file	# lines
(**S7**)	enter a character and save the file	# lines
(**S8**)	go to the last line	# lines
(**S9**)	find a word not present in the file	# char
(**S10**)	cut and paste the first character	# lines

Thresholds. In our evaluations, we configure the maximum iteration count max to be 20, the threshold of R^2 for regression learning $threshold_{R^2}$ to be 0.9, the threshold of error improvement $threshold_{imp}$ to be 2%, and the threshold of prediction error $threshold_e$ to be 5%.

Workload Selection. For **S1**-**S8** and **S10**, we set the RVR as $[1, 1280]$ for choosing workload values and the VVR as $[1, 2560]$ for choosing random validation values. For **S9**, the value of the workload parameter *# char* is relatively large in practice, and thus we set the RVR as $[1, 20480]$ and the VVR as $[1, 40960]$. For each scenario, we select 3 values from the RVR to obtain initial training profiles, and randomly select 5 values from the VVR to obtain validation profiles. To observe how initial workloads may affect our algorithm of model inference and refinement, we design two contrast groups for each subject application by selecting different initial workload values. For the 7-Zip file manager, we use $\{20, 40, 80\}$ for **S1**-**S3** and $\{100, 200, 400\}$ for **S4**-**S5**; for Notepad++, we use $\{20, 40, 80\}$ for **S6**-**S7**, $\{100, 200, 400\}$ for **S8** and **S10**, and $\{1000, 2000, 4000\}$ for **S9**. When we choose workload values for the parameter *# lines*, we keep the number of characters on each line to be 200.

We applied ΔInfer to infer complexity models using the training profiles as input and iteratively select new workloads to obtain new profiles for improving the accuracies of the inferred models.

8.7.2 RQ8.1: WDPB Identification

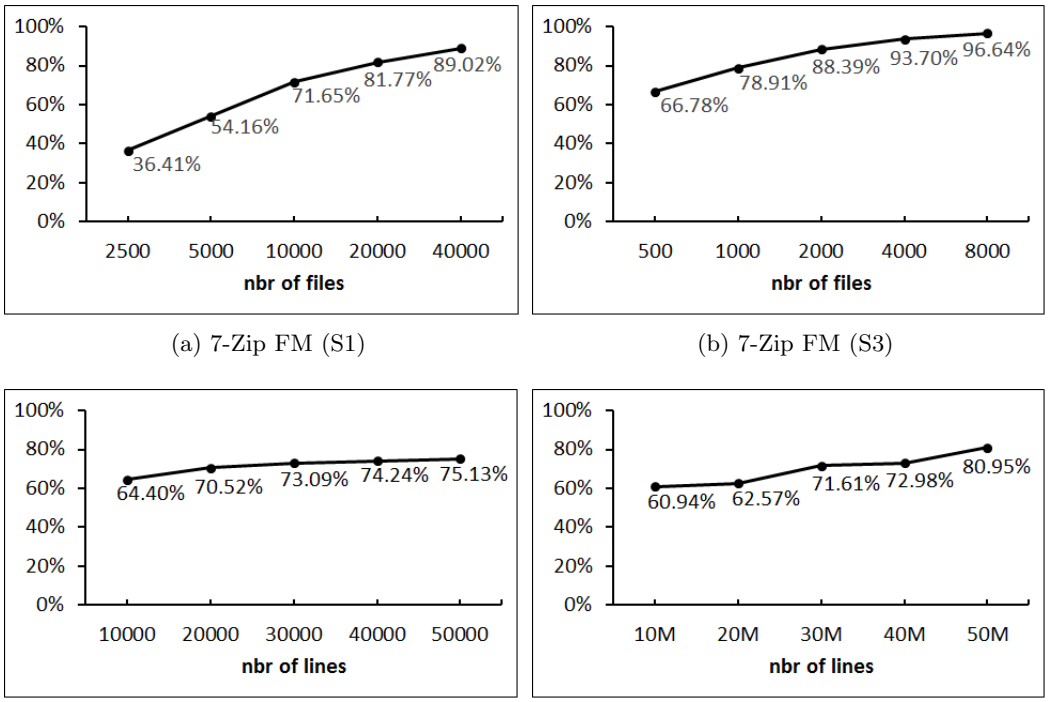
To answer RQ8.1, we first rank the complexity transitions using the predicted costs, and manually inspect the complexity transitions to confirm whether they will cause performance problems on large workloads. We then report the performance faults caused by the identified WDPBs and get the confirmation from developers. Although it is difficult to measure the false negatives of ΔInfer, we propose and measure the cost coverage (i.e., execution time) of the identified WDPBs as the workloads increase. If the identified WDPBs achieve high cost coverage, the probability of ΔInfer missing impactful WDPBs would be low.

7-Zip File Manager. Due to space limit, we describe only two representative WDPBs in this chapter. More details of the WDPBs can be found on our project website [25]. The first WDPB is `RefreshListCtrl`, a common WDPB for the Scenarios **S1**, **S2**, **S3**, and **S5**. In these scenarios, when the user opens/creates a folder or renames/deletes a file, the method `RefreshListCtrl` is invoked to refresh the list-view control. The complexity transition in `RefreshListCtrl` (`RefreshListCtrl`, {`GetItemRelPath`, ...}) captures an linear-workload-dependent loop. Inside the loop, `GetItemRelPath` computes the path prefix of a file using customized string-concatenation operations, which create temporary objects and destroy them after the concatenation. This computation results in intensive creations/destructions of temporary objects.

Another WDPB is a method that invokes `ListView_SortItems`. `ListView_SortItems` is a UI library call that repetitively invokes `CompareItems` to sort the files, behaving like an implicit loop whose complexity model is $nlog(n)$ in theory. ΔInfer identifies this implicit loop with a complexity transition from a constant order to a power-law order. Due to the inefficient implementation of retrieving file properties for comparison, this implicit loop causes a WDPB on large workloads.

Notepad++. We describe two representative WDPBs for Notepad++. The first WDPB is `WrapLines` that appears in every scenario. In these scenarios, when the user opens a file or modifies the content of the file (**S7** and **S10**), the method `WrapLines` is invoked to recompute the word-wrapping data structures by invoking `WrapOneLine`. `WrapOneLine` computes the layout of a line, creating temporary objects and dynamically allocating memory chunks for the computation results; such computation is quite expensive when the workload is large.

The second WDPB is `Document::FindText` for **S9**. ΔInfer finds that `Document::FindText` contains a linear-workload-dependent loop and performs string comparison to find the matching word in the document. Although the cost per iteration is not high, the workload value in terms of *# chars* is easy to become huge in practice. For example, searching a 10MB file for matching a character not present in the document would cause the loop in `Document::FindText` to



(a) 7-Zip FM (S1)

(b) 7-Zip FM (S3)

(c) Notepad++-(S6)

(d) Notepad++-(S9)

Figure 8.3: Cost coverages of identified WDPBs as workloads increase

execute 10M times. Thus, word search is usually considered expensive; and many editors or viewers (such as Adobe PDF Viewer) spawn a separate thread to do so.

Cost Coverage. Figure 8.3 shows the cost coverage of the representative WDPBs on larger workloads. Due to space limit, we choose four scenarios to show the cost coverage of representative WDPBs. For 7-Zip, we choose **S1** to show the coverage of the WDPB caused by `RefreshListCtrl`, and **S3** for the WDPB caused by `RefreshStatusBar`. For Notepad++, we choose **S6** to show the coverage of the WDPB caused by `WrapLines` and **S9** for the WDPB caused by `Document::FindText`. The results show that the identified WDPBs account for more than 75% of all the scenarios when the workloads increase, indicating that the probability of missing impactful WDPBs is low. Moreover, as shown in Figure 8.3b, the cost coverage increases so quickly that it reaches more than 90% on the workload of 8000 files. The reason is that the WDPB of **S3** has the quadratic complexity model.

Fault Confirmation. We reported the detected faults to the project's forum, and the responses of the developers of 7-Zip are quite encouraging. They confirmed the faults caused by `RefreshListCtrl` in **S1**, **S2**, **S3**, and **S5** (4 out of 5 faults reported), and plan to fix

Table 8.2: Results of model inference and refinement

ID	# Ite.	# W.	E. I.	E. E.	M. I.	M. E.
(**S1**)	4	6	35.95	0.62	752	657
(**S2**)	4	6	62.31	0.47	1341	1283
(**S3**)	4	6	29.68	0.85	1234	1223
(**S4**)	4	6	4.71	0.20	1299	1267
(**S5**)	3	5	5.94	0.18	1630	1282
(**S6**)	6	8	536.74	8.62	742	1441
(**S7**)	5	7	455.00	7.69	789	1329
(**S8**)	7	9	17.12	5.51	448	1505
(**S9**)	4	6	138.36	1.83	1287	205
(**S10**)	7	9	7.38	1.86	324	296

the faults in the next version [21]. Although the fault in **S1** was reported by others in the developers' database for faults (*#3193577*), our approach further identifies the WDPBs as the root cause of the fault. Capturing the fault caused by `OnRefreshStatusBar` in **S4** demonstrates the advantages of using predictive models in our approach. Such fault is difficult for traditional performance testing to detect without knowing the triggering workload. This newly detected fault has been dormant since it was introduced in version 4.25 beta (released on 2005-08-01), and still remains in the latest version 9.22 (released on 2011-4-18). After we reported the fault at the forum [22], the developers confirmed the fault and plan to fix it in the next version.

For Notepad++, the performance problem of wrapping words in Scenario **S6** is confirmed as fault *#2909745* in the project's forum. Moreover, our approach further identifies that `WrapLines` causes performance faults in Scenarios **S7**, **S8**, and **S10** after a document is modified. We are waiting for their responses for these not-yet-confirmed faults. For the performance fault of opening a file, the developers confirmed that Notepad++ did not have good performance on large files, and stated that a patch could be applied to improve the performance [20]. We reported the new fault of finding a word in **S9**, and are still waiting for the response.

8.7.3 RQ8.2: Model Inference and Refinement

To answer RQ8.2, we measure the improvement of model accuracies after the iterations of model refinement terminate, and the prediction accuracies of execution counts on large workloads.

Model Refinement. To show the improvement of model accuracies, we measure the average relative error of the inferred models with the initial workloads, measure the average relative error of the inferred models after the iterations of model refinement terminate, and compare the errors. Table 8.2 shows the results of RQ8.2. Column "ID" shows the ID of scenarios. Column "# Ite." shows the number of iterations used to refine the complexity models. Column "#

Table 8.3: Results of cost prediction

ID	10 (%)	20 (%)	50 (%)
(**S1**)	3.18	4.45	6.16
(**S2**)	2.98	4.07	5.55
(**S3**)	*1.40	*1.60	*1.86
(**S4**)	1.65	2.29	3.08
(**S5**)	1.58	2.19	2.95
(**Ave(7-Zip)**)	*2.35	*3.25	*4.44
(**S6**)	18.51	26.38	47.24
(**S7**)	16.84	22.56	36.28
(**S8**)	16.80	24.45	35.23
(**S9**)	11.15	15.97	39.09
(**S10**)	10.79	15.50	24.63
(**Ave(Notepad++)**)	14.82	20.97	36.49

W." shows the number of workloads used to infer the complexity models when the iterations terminate. Column "E. I." shows the average relative errors of the inferred complexity models on the initial workloads, and Column "E. E." shows the average relative errors of the inferred complexity models when the iterations terminate. Column "M. I." shows the number of the inferred complexity models on the initial workloads, and Column "M. E." shows the number of the inferred complexity models when the iterations terminate.

On average, it takes about 5 iterations (using 7 workloads) for ΔInfer to terminate, and improves the average relative error to 2.78%. From the results, we can see that the accuracies of the inferred models are significantly improved. For example, the average relative error of **S6** is improved from 536.74% to 8.62%. The results of Columns "M. I." and "M. E." indicate that certain locations that are incorrectly inferred as workload-dependent can be filtered out after iterative refinements. Different initial workloads may result in different initial accuracies. But with our workload-augmentation mechanism, these differences are not obvious after a few iterations, indicating that our approach is insensitive to the potential variance of intial workloads. In summary, the results show that our algorithm requires a reasonable number of iterations to achieve substantial improvement of model accuracy.

Cost Prediction. To show prediction accuracies on large workloads, we select the workload values that are 10, 20, and 50 times the upper bound of the RVR, obtain the profiles of these workloads, and compare the predicted execution counts and the actual execution counts to compute average relative errors. Table 8.3 shows the results of RQ8.2. Column "ID" shows the ID of scenarios. Columns "10", "20", and "50" show the average relative errors when the

workloads are 10, 20, and 50 times the upper bound of the RVR. For example, for **S1**, we use the workload values $\{12800, 25600, 64000\}$.

On average, when the workload is 50 times of the upper bound of the RVR, the prediction error for the 7-Zip file manager (marked with *) is just 4.44% (excluding **S3**), and the prediction error for Notepad++ is acceptable (36.49%). For **S3** (marked with *), due to the quadratic workload-dependent loop, our profiler reaches its profiling limitations when the workload value exceeds 10,000. Thus, we use workload values $\{4000, 6000, 8000\}$ to estimate the prediction errors. The reason why Notepad++ has a relatively high prediction error is that the developers of Notepad++ optimize the message processing during idle time, causing certain workload-dependent loops to exhibit a bit different complexities under the same contexts. Such result shows that ΔInfer is robust even under such complex situations.

8.7.4 RQ8.3: Context-Sensitive Analysis

Existing approaches [95, 56] that infer complexity models using profiles of *multiple* workloads are context-insensitive. To show effectiveness of context-sensitive analysis and answer RQ3, we compare the number of complexity transitions identified by using context-sensitive analysis and context-insensitive analysis. We first apply ΔInfer to infer complexity transitions using the inferred complexity models. We then apply regression learnings on profiles collected to infer complexity models without calling context, and use these complexity models to infer another set of complexity transitions. We compare these two sets of complexity transitions to show the effectiveness of our context-sensitive analysis.

Table 8.4 shows the results of RQ3. Column "ID" shows the ID of scenarios. Column "ΔInfer" shows the number of complexity transitions inferred by ΔInfer, and Column "# L." shows the number of the inferred complexity transitions that are workload-dependent loops. Column "In-Sen." shows the number of complexity transitions inferred by the context-insensitive analysis, referred to as *ContextIns*. Column "# Miss." shows the number of complexity transitions inferred by ΔInfer but not ContextIns.

The results show that ContextIns identifies much more (32.8 times on average) complexity transitions than ΔInfer, and more than 90% of them do not help identify workload-dependent loops, producing many false positives. Such result is mainly due to the complex contexts posed by the event-driven nature of GUI applications. Based on manual inspection, 86.1% of the complexity transitions inferred by ΔInfer are workload-dependent loops (including implicit loops), and 13.9% of them are false positives. Most of the false positives (15 in **S7**) are complexity transitions inside an internal string-allocator function of `basic_string`. The others involve top-level message handlers, such as `WindowProcedure`. These false positives can be reduced by excluding low-level and top-level system libraries in the analysis. Moreover, the results of

Table 8.4: Comparison to context-insensitive analysis

ID	ΔInfer	# L.	InSen.	# Miss.
(**S1**)	11	10	521	6
(**S2**)	21	19	579	12
(**S3**)	17	16	486	10
(**S4**)	21	19	640	12
(**S5**)	22	20	546	12
(**S6**)	10	10	509	3
(**S7**)	29	14	877	6
(**S8**)	10	10	526	5
(**S9**)	20	20	131	0
(**S10**)	12	11	861	3

Column "# Miss." show that ContextIns misses about 39.9% of the complexity transitions identified by ΔInfer. Based on manual inspection, some of these missing complexity transitions are real WDPBs that cause performance problems. Thus, ContextIns also produces false negatives. We also find that ΔInfer does not miss any WDPB detected by ContextIns. In summary, ΔInfer outperforms ContextIns in terms of greatly reducing false positives and false negatives.

8.8 Discussion

Generalization to Other Types of Applications. Our current approach focuses on detecting WDPBs for GUI applications. However, our approach is applicable to any type of application that requires operations in responsive actions not to block subsequent operations, such as message-queue systems, event-driven servers, and chained filters. Our approach is also applicable to identify energy faults [155] in mobile applications by using profiles of energy costs. Moreover, our approach can be used to assist tasks of performance analytics, such as predicting whether the execution time of complexity transitions exceeds the response requirement on a given workload or estimating a lower bound on the workload that causes the execution time of complexity transitions to violate the response requirement. We plan to investigate such applications in our future work.

Workload Parameters. Our current approach infers models by varying workload values on one focused workload parameter. To identify performance bottlenecks that require combinations of parameters [106], our approach can be used to infer multiple models by varying different parameters separately. Based on the order differences of the inferred models for different workload parameters, we can know that the execution count increases more quickly on which workload parameter; such information is sufficient for our approach to identify complexity

transitions. Multi-dimensional models for multiple workload parameters are often in complex forms or even have no analytic form, making it difficult to uncover such models directly. Existing research [195] requires user-provided information for interdependencies of parameters to infer the models. By adapting the divide-and-conquer strategy to infer one model for one focused parameter, our approach reduces computational complexity without requiring information for the interdependencies of parameters, and yet preserves the effectiveness.

Value-Dependent Performance Bottlenecks. Some performance bottlenecks may depend on specific values of the input, instead of input workloads. Existing approaches [50] have explored the research of this direction, but have limitations due to the lack of runtime information. Moreover, there are other performance bottlenecks that may be triggered by combinations of configuration values as investigated by Hoffmann et al. [106]. These approaches can be leveraged to infer configurations for the scenarios, and our approach can be applied to automatically predict WDPBs afterwards.

Scalability of Scenario-Based Profiling. Our approach instruments an application under analysis and collects profiles for each scenario to detect scenario-specific WDPBs. Scenario-based instrumentation is widely adopted for testing/debugging, and the state of the practice is observed in many popular software products from leading software companies, e.g., PerfTrack [14] based on the Event Tracing for Windows (ETW) [5] platform from Microsoft. Based on such technologies, scenario-based tracing has been used as an automated and scalable solution for complex large-scale software products, e.g., Windows. Moreover, our approach can be fully automated with automatic GUI test scripts, reducing human efforts and improving scalability.

8.9 Summary

Software hangs can be caused by expensive operations in responsive actions (such as time-consuming operations in UI threads). Some of the expensive operations depend on the input workloads, referred to as workload-dependent performance bottlenecks (WDPBs). WDPBs are usually caused by workload-dependent loops (i.e., WDPB loops) that contain relatively expensive operations. Traditional performance testing and single-execution profiling may not reveal WDPBs due to incorrect assumptions of workloads.

We have proposed the ΔInfer approach that predicts WDPB loops under large workloads via inferring iteration counts of WDPB loops using complexity models for the workload size. ΔInfer incorporates the novel concept of *context-sensitive delta inference* that consists of two parts: *temporal inference* for inferring the complexity models of different program locations, and *spatial inference* for identifying WDPB loops as WDPB candidates. ΔInfer enables an instantiation of cooperative testing and analysis to improve performance analysis. The complexity models

inferred by ΔInfer explain how these methods become expensive under larger workloads. Such explanation improves the users' understanding of the application's workload-dependent behaviors, and thereby helps the users make better decisions on whether such costs are expected for the functionality of the application.

ΔInfer enables the behavior-diagnosis cooperation for performance analysis. Based on ΔInfer, our approach provides a behavior-diagnosis interface that shows the complexity models of methods besides showing the costs of the executed methods. In this cooperation, the tools compute the costs of the executed methods for the users to inspect and explain the cost growth of the methods by showing the inferred complexity models. Based on the users' domain knowledge about the functionality and performance requirements of the program under analysis, the users inspect the complexity models of the methods to determine whether the cost growth represented by the models are expected for the methods, and make decisions on whether the methods are performance faults.

We have conducted evaluations on ΔInfer with two popular open source GUI applications, 7-Zip and Notepad++. The results show that ΔInfer infers high-quality complexity models with iterative refinements, performs much better than the context-insensitive analysis, and effectively identifies highly impactful WDPBs that cause 10 performance faults.

CHAPTER 9

Future Work

In this dissertation, we presented a framework, called cooperative testing and analysis, that provides interfaces to support cooperation between software testing/analysis tools and their users, enabling the users to make informed decisions in the cooperation. Under this framework, this research has demonstrated a number of approaches that provide interfaces to enable problem-diagnosis cooperation and behavior-diagnosis cooperation for automated test generation, security analysis, and performance analysis, improving both functional correctness and non-functional qualities (security and performance) of software. There are still many opportunities for extending this work. We next discuss some of the future directions that can be conducted by extending the research in this dissertation.

9.1 Tool Automation for Assisting Cooperation in Test Generation

My future research on improving problem-diagnosis cooperation for test-generation tools includes three major directions. First, we plan to develop visualization techniques that present the identified problems and the residual coverage to improve the users' understanding on the problems. Second, we plan to develop techniques to assist the users in providing guidance by automatically generating guidance suggestions. Third, we plan to extend our research on economical analysis to support estimation of costs for covering a target branch and improve techniques for the estimation of benefits for solving a problem.

Visualization Support for Improving User Understanding on Problems. Covana precisely identifies problems that prevent test-generation tools from achieving high structural coverage, with the focus on OCPs and EMCPs for object-oriented programs. Simply printing the information about the identified problems may not be effective in helping developers understand the problems [134].

To better assist developers in locating identified problems and providing guidance, I started collaboration work with another fellow Ph.D student on developing a novel visualization approach [173]. This ongoing work visualizes the structural coverage achieved by the tools and the problem-analysis results produced by Covana, facilitating developers to understand how the reported problems affect the achieved coverage. We already conducted a preliminary user study involving seven participants to evaluate the effectiveness of our visualization approach, and obtained promising results on the usefulness of our visualization. For example, in our user study on the visualization approach, one user pointed out that our explanations of identified problems required improvement. We plan to further investigate how the work of Marceau et al. [134] can be employed to improve problem explanations to developers.

Suggestion Synthesis for Assisting Users in Providing Guidance. Our preliminary user study of Covana shows that even if developers understand what problems are faced by the tools, developers may have difficulties in implementing their guidance from scratch, and it could be quite time-consuming if they are not familiar with the code. We plan to develop guidance-generation techniques that exploit the semantics and structure of the program. For example, by analyzing the class structures and the already generated object states, we plan to develop techniques that can generate the skeleton of a factory method for a given class, reducing the users' efforts in constructing the factory method. Such skeleton includes a constructor (potentially multiple constructors if the creation of a target object requires other objects), a list of methods that invoke the public setter methods to set certain private fields, and a few methods that modify the other fields of the object, making the object state as close to the target state as possible. Recently, program synthesis approaches [176, 99, 218] have shown promising results in automatically synthesizing executable programs in certain formats. We also plan to explore program synthesis techniques for synthesizing partial solutions to the problems faced by the tools, improving the guidance suggestions.

Improving Economical Analysis. Our current research assumes that the benefit of covering a branch is a fixed constant, i.e., branches are equally critical. Depending on the goal of testing, the benefits of different branches may be different. For example, if the goal is robustest testing, then any branch that leads to uncaught exceptions should have higher benefits to cover. In future work, we plan to improve our models by taking varied benefits of branches into consideration.

In cooperative testing, developers provide guidance to address the problems identified by Covana. If the goal is to cover a critical branch, such as an assertion-violating branch, then developers would like to cover the branch by addressing as few causes as possible to reduce their human efforts. We assume that the cost of solving each problem is the same (i.e., problems are equally difficult). With this assumption, the cost of covering a branch (referred to as branch cost) is the *minimum* number of problems that need to be solved in order to cover the branch. Based on our research on estimation of problem benefits, we plan to develop techniques to estimate branch costs.

Estimating branch costs builds on the estimation of problem benefits because the estimation iteratively uses angelic diagnosis (i.e., introducing choose operators) to bypass a set of problems until the branch is covered. Given a target branch not yet covered, if Covana has already reported a set of problems that cause the target branch not to be covered, EcoCov directly applies angelic diagnosis on all possible subsets of the problems, and identifies the minimum set of necessary problems (those by solving which the branch can be covered) as the branch cost. On the other hand, if the branch statement of the target branch has not been reached yet, Covana is not able to report such a set of candidate problems. In this case, we need to first cover some not-covered branches reported by Covana (i.e., those branches whose branch statements are reached) to reach the target branch. Thus, EcoCov uses a recursive algorithm to compute a path (if exists) from each not-covered branch reported by Covana to the target branch, and then chooses the path with the least number of problems along the path to report as the branch cost. Note that EcoCov provides not only the number of problems as the branch cost but also the set of problems to the developers to guide them towards the target.

The assumption that all the problems have the same difficulty for developers to address (the cost is always a fixed constant) simplifies our analysis. However, a factory method may be more difficult to write than another factory method. For example, creating a red-black tree object whose size is 10 is more difficult than creating a stack object with the same size. Similarly, certain mock objects require complex models to correctly simulate the environment dependencies. Thus, we also plan to conduct a comprehensive study on difficulty levels to address various OCPs and EMCPs. We also plan to conduct user studies to observe how the users address problems of different difficulty levels.

9.2 Detecting Inconsistency between User Expectations and Mobile Application Behaviors

With the rapid growth of smartphones and mobile applications, mobile threats have been increasing dramatically as well. To address the issue of mobile threats, existing markets adopt

two mechanisms to filter and reveal mobile threats for Android users. On one hand, automated detection tools exist to detect malware, such as Bouncer [43]. On the other hand, markets reveal potential mobile threats by listing app permissions at the installation time. However, these mechanisms are shown to have limited successes. Potentially Unwanted Applications (PUAs) such as spyware or trackware confoundedly evade Bouncer to appear on the Android application market [76]. Additionally, several studies [73, 79, 80] have shown that the current Android permission system provides little information to help the users understand permissions. Therefore, the users cannot understand the risk when installing applications.

There exists work that automatically detects mobile threats. However, it is challenging to classify an application as malicious, privacy infringing, or benign. Existing work has looked at permissions [221, 157], code [73, 222], and runtime behavior [72, 209]. However, underlying all of this work is a caveat: *what does the user expect?* Clearly, an application such as a *GPS Tracker* is expected to record and send the phone's geographic location to the network; an application such as a *Phone-Call Recorder* is expected to record audio during a phone call; and an application such as *One-Click Root* is expected to exploit a privilege-escalation vulnerability. Other cases are more subtle.

User expectations are reflected via user perceptions of application behaviors, in combination with user judgments. With the limited information presented by existing approaches, there are gaps between user perceptions and application behaviors. For example, some application behaviors may be user imperceptible (such as sending out users' private information through non-monitored sinks described in Chapter 6), or contradict with user perceptions (such as sending text messages in background while users scroll a list). Also, the users may not be able to make right decisions based on the perceived information. For example, we note that malware may still hide malicious functionality (e.g., eavesdropping) within an application designed to use the corresponding permission (e.g., an application to take voice notes). Revealing the permissions (e.g., audio recording) alone in such applications cannot help the users make right decisions.

With the vision of bridging gaps between user perceptions and application behaviors for improving mobile privacy control, our research on providing behavior-diagnosis interfaces for mobile privacy control [203] is the first step towards the research direction of revealing the user-imperceptible behaviors. Our approach reveals how mobile applications use the users' permissions, improving the users' perception on mobile applications' privacy-related behaviors. One of recent studies shows that users have difficulty to understand the permissions on the installation-time confirmation dialog since these permissions lack contexts and the users cannot figure out what application functionality the permissions correspond to. Thus, to better inform the users about the potential risks, user-perceivable context information (i.e., corresponding app functionality) for permissions is needed. Such context information explains *when* permissions

are used. My ongoing work with others, AppContext [211], employs static analysis to explain under *what contexts* permissions are used and *how* the private data protected by the permissions are used. Such information can complement our information flow analysis, helping the users better understand permission uses of applications.

Moreover, another piece of joint work with others, WHYPER [153], presents a framework that adapts Natural Language Processing (NLP) techniques to establish links between sentences in application descriptions and permissions in the permission list. Such sentences explain *why* an application uses a permission, helping the users better understand the expected functionality of a mobile application.

Based on our research on improving user understanding on both application behaviors and application functionality, we plan to develop an approach that automatically infers models of both application behaviors and application functionality and identifies inconsistencies between the models. To model application functionality, we plan to extend our techniques on NLP to parse application descriptions and texts included in applications' GUIs. Application descriptions provide general information about what applications do and what categories the applications belong to (e.g., games or navigation apps), and GUI texts provide detailed information about what applications do (e.g., a "sendText" button indicating the functionality of sending texts). To model application behaviors, we plan to extend our techniques on information flow analysis [203] and context analysis [211] for better capturing application behaviors related to privacy-sensitive operations. To identify the inconsistencies between these two models, we plan to develop techniques that abstract action-object pairs from the models and compare such abstractions for finding out unmatched pairs as the inconsistencies. We plan to conduct experiments on market applications and pre-identified malware applications to evaluate whether such abstraction achieves high precisions and recalls, and identify limitations of the abstraction for further improvements.

9.3 Identifying Complexity Changes Across Program Versions

Software evolves through its lifetime, undergoing various kinds of changes. To ensure that code changes do not introduce unintended consequences, developers perform regression testing [104, 161, 212, 180], running the existing test suites. However, these existing techniques are mostly for functional correctness and do not address regression testing of quality attributes such as performance [196]. For performance regression testing, the state-of-the-practice is as follows. Developers prepare some test inputs and measure how performance for those inputs changes over different program versions [17, 27, 7]. Such analysis of potential performance regressions is carried out manually and can be error-prone, time-consuming, and sensitive to the input load. Thus, performance regressions are often not detected until the new version is released to the

field [26, 28]. Moreover, observing the running time changes between two versions is difficult to identify performance regressions, since the running time of a piece of code is different for different machines, and even different runs in one machine.

To address the challenges of detecting performance regression across versions, we plan to develop techniques that compare behaviors of old and new code versions to identify behavior changes that indicate performance regressions. For example, a piece of code whose complexity changed from linear to $O(n \log n)$ can be more suspicious for inspection than a piece of code whose complexity was quadratic and remained quadratic. Such changes can be reflected using the changes of the complexity models.

Our approach ΔInfer provides a behavior-diagnosis interface for explaining expensive methods using their complexity models. In future work, we plan to build on ΔInfer [199] using a two-step approach. For the first step, we plan to develop a technique that uses information about code changes to infer complexity models faster than running ΔInfer from scratch on the new code version. This inference requires running a smaller portion of the existing tests, and the initial subset of tests are selected to allow ΔInfer to train and validate the complexity models. These models can then be used to predict the running costs at different changed program locations under different workloads and contexts. For the second step, the technique compares the complexity models from the current code version and the previous code version to identify *likely complexity transitions* that may cause performance regressions.

For example, consider a piece of code whose original complexity model is $a \cdot n + b$, where n is some measure of the input/workload size. There are three common types of complexity transitions: additive (the new model is $a \cdot n + b', b' > b$, likely harmless), multiplicative (the new model is $a' \cdot n + b', a' > a$, potentially problematic), and order (e.g., the new model is $c \cdot n^k, k > 1$, likely the most severe performance degradation). After performing the comparison for different program locations, the technique can rank the list of locations based on the significance of their complexity transitions. It can finally present to the developer for manual inspection the transitions that are confirmed by the test executions. For example, code whose complexity changed from $O(n)$ to $O(n \log n)$ can be more suspicious for inspection than a piece of code whose complexity is $O(n^2)$ but also was $O(n^2)$. Note that in the absence of identifying complexity transitions, i.e., by inferring the models on only the new code version, the developer could incorrectly focus on $O(n^2)$ before $O(n \log n)$. Thus, inspecting complexity transitions between two versions can effectively reveal the unintended performance changes that are difficult for traditional performance testing to detect without knowing the triggering workload. As a real example, the 7-Zip file manager [3] has a performance fault that causes software hangs by refreshing the status bar repetitively when users select or deselect all files in the folder; this fault was introduced in version 4.25 beta, when the complexity of the function refreshing the status bar rose from $O(1)$ to $O(n)$.

9.4 Cooperative Testing and Analysis

Our future research on extending the framework of cooperative testing and analysis plans to focus on two questions: "*What can users do*" and "*How can tools learn from users*"?

First, our current research assumes that the users have the same skill in providing their guidance to help the tools address the encountered problems. However, in practice, the users often have different skills in solving problems, and thus we should improve cooperative testing and analysis by taking such differences into consideration. For example, in the cooperation for improving test generation, EcoCov ranks the problems based on how much coverage the users can obtain by solving a problem. If a user has limited skills in solving difficult OCPs, then EcoCov should revise its ranking by ranking certain EMCPs with high benefits higher, although these EMCPs may not have as high benefits as the top OCPs.

Second, we plan to develop techniques that leverage user-interaction histories to improve automated problem-solving techniques. Consider the cooperation in test generation again. Assume that the users provide guidance to help the tools address several OCPs and EMCPs, and reapply the tools to generate test inputs based on the guidance. If the tools encounter new problems, the tools do not consult the users immediately as the current framework suggests. Instead, the tools should try to leverage the previous guidance given by the users to solve the newly-encountered problems. For example, the tools may extract templates from the provided factory methods, and fill in the templates with the class names in the newly-encountered OCPs. Even better, the tools may try to mutate and evolve the templates to address similar problems. We plan to conduct extensive experiments to evaluate the feasibility of such improvements, starting with the cooperation in test generation.

9.5 Improving Quality of Various Types of Software

During the past few decades, software has been becoming larger, more complex, and more integrated into our daily life, and many new types of software applications have been developed, such as mobile applications and distributed applications. These new types of software applications pose new challenges for ensuring high quality of software. Therefore, there is an ever-increasing demand for better testing and analysis techniques to improve software quality.

The research in this dissertation focuses on sequential object-oriented applications and mobile applications. We plan to adapt my techniques for testing and analyzing other types of applications, such as concurrent applications and database applications.

For example, when testing and analyzing a concurrent program, we can no longer operate on the granularity of a single execution since thread interactions can occur within a method execution, causing different method behaviors given the same method inputs and preventing

software testing/analysis tools from achieving expected effectiveness, such as covering certain branches. For test generation, one possible extension to Covana is to perform symbolic analysis on each concurrent execution. However, this finer granularity can suffer from the state explosion problem more seriously. We plan to conduct further studies on how to extend our research for testing and analyzing concurrent applications. To test and analyze a database application, the model of interactions with a database is critical since most of the current analysis techniques treat database interactions as external method calls. Without a precise model to analyze database interactions, it is difficult to understand the correctness impacts and the performance impacts of interacting with a database. We plan to investigate how existing research on parameterized mock objects [181, 135] can be used to model database interactions, and develop novel techniques to analyze the correctness impacts and the performance impacts simulated by these models.

CHAPTER 10

Assessment and Conclusion

10.1 Conclusion

This dissertation proposes a framework of cooperative testing and analysis, which provides interfaces to support cooperation between software testing/analysis tools and their users, enabling the users to make informed decisions in the cooperation and thus better conducting an SE task. Our framework focuses on providing interfaces to support two types of cooperation with software testing/analysis tools for SE tasks: (1) **Problem-Diagnosis Interface**: for certain SE tasks where automated tools drive the tasks towards a goal of testing and analysis, our framework provides interfaces that precisely report the problems faced by the tools and the benefits of solving these programs. Such interfaces enable the users to cooperate with the tools by addressing problems faced by the tools. (2) **Behavior-Diagnosis Interface**: for certain SE tasks where the users inspect software behaviors to determine whether the behaviors are expected for achieving a goal of testing and analysis, our framework provides interfaces that show a list of behaviors related to the goal for the users to inspect, and present succinct models for better explaining the behaviors. Such interfaces enable the tools to cooperate with the users by analyzing and explaining the software behaviors.

A number of approaches have been developed under this framework to enable effective cooperation between software testing/analysis tools and their users in automated test generation, security analysis, and performance analysis, three major software engineering activities for improving software quality.

We have presented approaches that provide the interface for supporting the problem-diagnosis cooperation for automated test generation, where the users cooperate with test-generation tools by addressing the problems faced by the tools. We have conducted empirical studies to identify problems that prevent the test-generation tools from achieving high structural coverage. We have also proposed approaches that precisely identify these problems for non-covered branches [205, 204, 198, 200] and estimate benefits of solving these problems [132], with the focus on two major types of problems identified in our studies: OCPs and EMCPs. Experimental results show that our approaches effectively identify OCPs and EMCPs with a few false positives and false negatives, and produce highly accurate estimation results on benefits for solving OCPs and EMCPs.

We have presented approaches that provide the interfaces for supporting the behavior-diagnosis cooperation for security analysis, with the focus on mobile privacy control and security policy extraction. To improve mobile privacy control, our approach computes information flows that show what private data types flow to what output channels [203], explaining how mobile applications use permissions. Our approach also provides monitored sinks that support runtime inspection of private information before the information is sent out, and identifies information flows that may escape the users' inspection as unsafe flows. Based on the information flow computation and classification, our approach provides the interface to support the behavior-diagnosis cooperation for mobile privacy control, where the tools identify permissions and explain permission uses to help the users make informed decisions on whether the permission uses are expected. Experimental results show that our approach effectively reduces users' decisions to only 10.1% of all scripts, since our approach does not require users' decisions for information flows where untampered information flow to monitored sinks.

To improve security policy extraction, our approach automatically extracts ACP rules and action steps from NL software documents [201], with the focus on use cases. Based on the extraction of ACP rules and action steps, our approach provides the interface to support the behavior-diagnosis cooperation for security policy extraction, where the tools extract security policies and explain the policies using policy-witness and policy-violation scenarios, helping the users make informed decisions on whether the extracted policies are expected. Experimental results show that our approach extracts ACP rules with the accuracy of 86.3% and extracts action steps with the accuracy of 81.9%.

We have presented an approach that provides the interface for supporting the behavior-diagnosis cooperation for performance analysis. Our approach that infers complexity models for workload-dependent loops from multiple executions on multiple workloads [199]. Based on the inferred complexity models, our approach provides the interface to support the behavior-diagnosis cooperation for performance analysis, where the tools compute costs of the executed methods and explain the cost growth of the methods using the inferred complexity models,

helping the users make informed decisions on whether certain expensive methods are expected in larger workloads. Experimental results show that our approach infers accurate models with a few iterations in sampling more workloads, and performs much better than the context-insensitive analysis. In addition, the results show that our approach effectively identifies highly impactful WDPBs that cause 10 performance faults.

10.2 Risk Analysis

We next present a few risks involved in using the approaches presented in this dissertation. For an SE task on which our framework is applied, our framework assumes that users' subtasks and tools' subtasks are already identified. In other words, our framework does not optimize the task allocation for the users and the tools, but focus on improving the information presented in the interfaces provided by our framework.

Our research on automated test generation is based on the empirical studies that focus on the major types of problems (OCPs and EMCPs) faced by a DSE-based tool, Pex [184], when dealing with complex object-oriented programs. However, when applying test-generation tools on non-object-oriented programs (such as programs written in C) and dynamic-language programs (such as Javascript programs), OCPs are not concerns. The main reason is that these programs allow object fields to be set directly, and thus it is straightforward to generate sequences of method calls to construct object states in such programs. Moreover, certain types of software applications may contain more problems of a specific type. For example, algorithm-based applications tend to have more nested loops, and may have more problems caused by loops [200]. Also, some business applications may contain complex computations of floating-point numbers, posing challenges for constraint solving. Such risks can be alleviated by extending Covana to deal with more types of problems, and more evaluations on various types of software applications.

Our research on mobile privacy control focuses on analyzing TouchDevelop [186] scripts. The risk of generalizing our approach to other mobile-device platforms, such as Android, iOS, and Windows Phone, includes three major points: (1) these other platforms provide a much larger API surface than TouchDevelop and annotating these APIs with sources, sinks, and flow information requires significant efforts; (2) the languages used (Java, C#, or assembly code) provide more ways to obscure flow than in the TouchDevelop language, in particular through indirect calls, or via reflection. Our static analysis needs to be extended with specific techniques to address these issues [73, 79]; (3) indirect flow through mutable storage will require a finer grained heap model than we currently employ (one abstract location per data kind). The static analysis might need to be complemented with dynamic analysis [72, 223] to address such risk.

Our research on extracting ACP rules is evaluated on 37 use cases of iTrust. The iTrust use cases were created based on the use cases in the U.S. Department of Health & Human

Service (HHS) [16] and Office of the National Coordinator for Health Information Technology (ONC) [8], and evolved and revised by about 70 students and teaching assistants as well as instructors each semester since the iTrust requirements were initially created. Although the public availability and activeness make the iTrust use cases suitable for our subjects, we evaluated our approach on only these limited use cases. To reduce the risk of subject representativeness, for the evaluation of ACP extraction, we further collected 100 ACP sentences from other 17 publicly available sources. Furthermore, we also applied our approach on 25 use cases of a module in a proprietary IBM enterprise application that belongs to the financial domain. Such risk of subject representativeness can be further reduced by conducting more experiments on both open-source and proprietary software projects in various domains.

Our research on performance analysis is evaluated on two open-source GUI applications that belong to daily used productivity tools: file archivers and text editors. These two applications are popular open source applications from the SourceForge repository, and their databases of faults and forums are actively maintained. This risk can be further reduced by more studies on more kinds of GUI applications, including both open-source and proprietary GUI applications.

10.3 Lessons Learned

We next summarize some lessons learned through our research and hope that these lessons would help other researchers (including us) for their future research.

Identifying Research Problems from Real Code Bases. Through the research in this dissertation, we found that empirically investigating real code bases was very important in identifying research problems to work on. Our research on Covana started by applying a state-of-the-art test-generation tool, Pex [184], on complex object-oriented programs, and then we empirically studied the causes to not-covered branches for identifying problems that prevent Pex from achieving high coverage. Such empirical data help us identify OCPs and EMCPs as the major problems, driving our research towards the direction of developing techniques for identifying these two major types of problems. Without investigation of the real code bases, we might still constrain ourselves on the algorithm of symbolic execution used by Pex, and have difficulties in developing the research.

Learning from Related Research Areas. This dissertation demonstrated the potential of combining techniques from different research areas. Our research on extraction of ACP rules (Section 7) adapts techniques from the NLP research and proposes new techniques that work upon the NLP techniques. Our research on user-aware privacy control (Section 7) synergistically combines techniques from both security research and software engineering/programming language (SE/PL) research. These problems come with challenges that are difficult for techniques of SE research to address, but the techniques from other research areas can help address some

limitations of the techniques in SE search. Thus, it is always good to learn techniques from related research areas.

Extending Existing Powerful Tools. Building a prototype to evaluate the feasibility of a research idea is used widely in different research areas. In SE research, evaluating a research idea typically requires developing a tool and applying the tool on some code bases. However, developing a fully-functional tool from scratch requires a lot of efforts, because we need to handle many corner cases that are often engineering issues. Thus, building upon existing powerful tools can save a lot of efforts in handling engineering issues, and give us quick feedback on the effectiveness of our research idea. We implemented Covana (Section 4) and EcoCov (Section 5) as extensions to Pex [184], leveraging Pex's DSE engine to perform symbolic execution and collect runtime information for our own analysis. Although we need to write extensions to parse the results produced by Pex, the efforts required for the parsing are manageable and allow us to build a prototype quickly.

Proper Assumptions to Reduce Difficulties of Research Development. When there are no suitable powerful tools for us to build on, identifying proper assumptions of the research problem to reduce difficulties of research development is important for quick prototyping. Our research on user-aware privacy control (Chapter 6) is implemented in TouchDevelop [186]. TouchDevelop has a script bazaar that allows their users to publish and share their scripts, making it a similar mobile platform as Android, Windows Phone, and iOS. However, applications in TouchDevelop are developed in the TouchDevelop language, which is a simple scripting language that does not allow reflection, `eval`, or native calls to platform APIs, making code analysis easier [108]. Since dealing with these difficulties is not the focus of our research, we propose a simplified language that abstracts away such difficulties with proper assumptions, and implement a static analysis tool that can analyze TouchDevelop scripts for our evaluations.

Generalization of Research Ideas. Often the time, we work on a specific research problem and propose techniques to address just the problem. However, when we look back, the techniques that we propose may not be limited to addressing just the specific problem. Our research on Covana (Section 4) focuses on test-generation tools based on DSE tools, but other test-generation tools also face the same problems as the DSE tools, since these tools also need to construct desired object states and deal with external libraries. Thus, the idea proposed in Covana is general to other test-generation tools. Similarly, our research on user-aware privacy control (Section 6) focuses on analyzing TouchDevelop scripts, but the general idea is applicable to other mobile platforms. The essence is to summarize the insight of the idea, and abstract the idea as general as possible.

Reducing Risks by Conducting Evaluations on Both Open-Source and Proprietary Subjects. To evaluate SE research, SE researchers typically use open-source projects that are available on Web. Proprietary software projects are developed in a different way than

the open-source projects, and evaluations on proprietary subjects can reveal different insights. Our research on extracting ACP rules (Chapter 7) is evaluated on both open-source and proprietary use cases. We obtain better evaluation results on the proprietary use cases since these use cases are written in a more clear and coherent style. The open-source use cases are maintained by different users who have different backgrounds and worked on the project for a short time. Thus, these use cases are written in different styles and contain many inconsistencies. For example, our evaluation results show that inconsistent names are used to refer to the same role in a few use cases (Section 7.5.5), but such mistakes are rarely seen in proprietary use cases. However, proprietary subjects typically are not publicly available, and typically require a long process to prepare the legal documents for obtaining the accesses to the proprietary subjects. In this case, doing an summer internship in industrial research labs can provide opportunities for working around proprietary software projects.

REFERENCES

[1] The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. *SIGMETRICS Performance Evaluation Review*, 19(2):5–11, 1991. Reviewer-Al-Jaar, Robert Y.

[2] Java abstract windows toolkit (AWT), 1997. http://docs.oracle.com/javase/6/docs/technotes/guides/awt.

[3] 7-Zip, 1999. http://www.7-zip.org/.

[4] SourceForge, 1999. http://sourceforge.net/.

[5] Event Tracing for Windows (ETW), 2000. http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx.

[6] Notepad++, 2000. http://notepad-plus-plus.org/.

[7] JUnitPerf, 2003. http://www.clarkware.com/software/JUnitPerf.html.

[8] Office of the National Coordinator for Health Information Technology (ONC), 2004. http://www.hhs.gov/healthit/.

[9] eXtensible Access Control Markup Language (XACML), 2005. http://www.oasis-open.org/committees/xacml.

[10] eXtensible Access Control Markup Language (XACML) specification, 2005. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[11] Java Swing, 2006. http://docs.oracle.com/javase/tutorial/uiswing/.

[12] Windows forms and windows presentation foundation, 2006. http://windowsclient.net/.

[13] iTrust: Role-based healthcare, 2008. http://agile.csc.ncsu.edu/iTrust/wiki/.

[14] PerfTrack, 2009. http://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack.

[15] Preventing hangs in windows applications, 2009. http://msdn.microsoft.com/en-us/library/dd744765.

[16] U.S. department of Health & Human Service (HHS), 2010. http://www.hhs.gov/.

[17] Lucene benchmark, 2011. http://people.apache.org/ mikemccand/lucenebench/.

[18] TouchDevelop, 2011. http://research.microsoft.com/TouchDevelop.

[19] Uploading images from smartphones can reveal your childs exact location to strangers, 2011. http://losangeles.cbslocal.com/2011/05/09/uploading-images-from-smart-phones-can-reveal-your-childs-exact-location-to-strangers/.

[20] https://sourceforge.net/projects/notepad-plus/forums/forum/331753/topic/5101818, 2012.

[21] https://sourceforge.net/projects/sevenzip/forums/forum/45797/topic/4941337, 2012.

[22] https://sourceforge.net/projects/sevenzip/forums/forum/45797/topic/4941343/, 2012.

[23] Is Knight's 440 million glitch the costliest computer bug ever?, 2012. http://money.cnn.com/2012/08/09/technology/knight-expensive-computer-bug.

[24] Text2Policy, 2012. http://research.csc.ncsu.edu/ase/projects/text2policy/.

[25] ΔInfer, 2013. https://sites.google.com/site/asergrp/projects/deltainfer/.

[26] GCC performance regression, 2013. http://gcc.gnu.org/ml/gcc-bugs/2013-06/msg00240.html.

[27] Microbenchmarking framework for Java, 2013. https://code.google.com/p/caliper/.

[28] Rails performance regression, 2013. https://github.com/rails/rails/issues/9803.

[29] Sandip Agarwala and Karsten Schwan. Sysprof: Online distributed behavior diagnosis through fine-grain system monitoring. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 8–, 2006.

[30] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[31] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1997.

[32] Glenn Ammons, Jong deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 170–194, 2004.

[33] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 117–133, 2007.

[34] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy architecture language (EPAL 1.2), 2003. http://www.w3.org/Submission/EPAL/.

[35] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*, pages 64–84, 2010.

[36] Thomas Ball, Orna Kupferman, and Mooly Sagiv. Leaping loops in the presence of abstraction. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 491–503, 2007.

[37] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[38] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, 2001.

[39] Shaon Barman, Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 339–352, 2010.

[40] Clark Barrett and Cesare Tinelli. Cvc3. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[41] Kent. Beck. *Test-Driven Development: By Example*. The Addison-Wesley signature series. Addison-Wesley, 2003.

[42] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[43] Google Mobile Blog. Android and security, 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[44] Branimir K. Boguraev. Towards finite-state analysis of lexical cohesion. In *Proceedings of the International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP)*, 2000.

[45] Carolyn Brodie, Clare-Marie Karat, John Karat, and Jinjuan Feng. Usable security and privacy: A case study of developing privacy management tools. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, pages 35–43, 2005.

[46] Carolyn A. Brodie, Clare-Marie Karat, and John Karat. An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, pages 8–19, 2006.

[47] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[48] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[49] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 121–130, 2011.

[50] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199, 2009.

[51] Francis Chantree, Bashar Nuseibeh, Anne de Roeck, and Alistair Willis. Identifying nocuous ambiguities in natural language requirements. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, pages 56–65, 2006.

[52] S. Chatterjee and A.S. Hadi. *Regression Analysis by Example*. Wiley series in probability and mathematical statistics. Wiley-Interscience, 2006.

[53] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *Journal of System Software*, 83(1):60–66, 2010.

[54] Yan Chen, George Danezis, and Vitaly Shmatikov. "These Aren't the Droids Youre Looking For" - retrofitting android to protect data from imperious applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[55] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2000.

[56] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–98, 2012.

[57] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[58] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[59] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 281–290, 2008.

[60] Jonathan de Halleux and Nikolai Tillmann. Moles: Tool-assisted environment isolation with closures. In *Proceedings of the International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 253–270, 2010.

[61] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction And Analysis of Systems (TACAS/ETAPS)*, pages 337–340, 2008.

[62] Sidney W. A. Dekker and David D. Woods. Maba-maba or abracadabra? progress on human-automation co-ordination. *Cognition, Technology & Work*, 4(4):240–244, 2002.

[63] Dorothy E. Denning. A lattice model of secure information flow. *Communications of The ACM*, pages 236–243, 1976.

[64] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of The ACM*, pages 504–513, 1977.

[65] Rao H. Desineni. *A Comprehensive Diagnosis Methodology for Characterizing Logic-behavior of Integrated Circuit Failures.* PhD thesis, 2006. AAI3227998.

[66] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 181–192, 2012.

[67] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 181–192, 2012.

[68] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, 1998.

[69] Hyunsook Do and Gregg Rothermel. Using sensitivity analysis to create simplified economic models for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 51–62, 2008.

[70] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, pages 632–646, 2006.

[71] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS : Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[72] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, 2010.

[73] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 21–21, 2011.

[74] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[75] Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1):91–134, 2005.

[76] F-Secure. Mobile threat report q4 2012, 2012. http://www.f-secure.com/static/doc/labs_global/Research/Mobile%20Threat%20Report%20Q4%202012.pdf.

[77] Michael Feathers. CppUnit, 2006. http://sourceforge.net/projects/cppunit/.

[78] Christiane Fellbaum, editor. *WordNet An Electronic Lexical Database*. The MIT Press, 1998.

[79] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, 2011.

[80] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[81] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[82] Jeanne Ferrante and Karl J. Ottenstein. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:319–349, 1987.

[83] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[84] Robert W Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, 1967.

[85] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 32–41, 2010.

[86] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[87] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[88] B. Freimut, L.C. Briand, and F. Vollei. Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Transaction of Software Engineering (TSE)*, 31(12):1074–1092, 2005.

[89] Qiang Fu, Jian-Guang Lou, Qing-Wei Lin, Rui Ding, Dongmei Zhang, Zihao Ye, and Tao Xie. Performance issue diagnosis for online service systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 273–278, 2012.

[90] Erich Gamma and Kent Beck. JUnit, 2000. http://www.junit.org/.

[91] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the International Workshop on Mobile Cloud Computing and Services (MCS)*, pages 21–26, 2011.

[92] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[93] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[94] Mikhail I. Gofman, Ruiqi Luo, Jian He, Yingbin Zhang, and Ping Yang. Incremental information flow analysis of role based access control. In *Security and Management*, pages 397–403, 2009.

[95] Simon F. Goldsmith, Alex S. Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 395–404, 2007.

[96] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN)*, pages 120–126, 1982.

[97] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 156–166, 2012.

[98] Gregory Grefenstette. Extended finite state models of language. chapter Light Parsing As Finite State Filtering, pages 86–94. Cambridge University Press, 1999.

[99] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 13–24, 2010.

[100] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[101] Robert J. Hall. CPPROFJ: Aspect-capable call path profiling of multi-threaded Java applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 107–116, 2002.

[102] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 145–155, 2012.

[103] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, 2012.

[104] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 312–326, 2001.

[105] Qingfeng He and Annie I. Antón. Requirements-based access Control Analysis and Policy Specification (ReCAPS). *Information and Software Technology*, 51(6):993–1009, 2009.

[106] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212, 2011.

[107] Susan Horwitz. Tool support for improving test coverage. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*, pages 162–177, 2002.

[108] Fraser Howard. Malware with your mocha: Obfuscation and anti-emulation tricks in-malicious javascript, 2011. http://www.sophos.com/security/technical-papers/malware_-with_your_mocha.pdf.

[109] Steve Howard. User interface design and hci: Identifying the training needs of practitioners. *ACM SIGCHI Bulletin*, 27(3):17–22, July 1995.

[110] Vincent C. Hu, D. Richard Kuhn, Tao Xie, and JeeHyun Hwang. Model checking for verification of mandatory access control models and properties. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 21(1):103–127, 2011.

[111] JeeHyun Hwang, Tao Xie, Fei Chen, and Alex X. Liu. Systematic structural testing of firewall policies. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, pages 105–114, 2008.

[112] JeeHyun Hwang, Tao Xie, Vincent C. Hu, and Mine Altunay. ACPT: A tool for modeling and verifying access control policies. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 40–43, 2010.

[113] I Jacobson, M Christerson, P Jonsson, and G Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., 1992.

[114] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., 2004.

[115] Dorota Jagielska, Paul Wernick, Mick Wood, and Steve Bennett. How natural is natural language?: How well do computer science students write use cases? In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 914–924, 2006.

[116] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object capture-based automated testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170, 2010.

[117] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 125–134, 2009.

[118] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88, 2012.

[119] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[120] Clare-Marie Karat, John Karat, Carolyn Brodie, and Jinjuan Feng. Evaluating interfaces for privacy policy rule authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 83–92, 2006.

[121] John Karat, Clare-Marie Karat, Carolyn Brodie, and Jinjuan Feng. Designing natural language and structured entry methods for privacy policy authoring. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*, pages 671–684, 2005.

[122] Christopher Kennedy. Anaphora for everyone: Pronominal anaphora resolution without a parser. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, pages 113–118, 1996.

[123] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 340–349, 2012.

[124] Yunho Kim and Moonzoo Kim. SCORE: a scalable concolic testing tool for reliable embedded software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 420–423, 2011.

[125] James C. King. Symbolic execution and program testing. *Communications of The ACM*, 19(7):385–394, 1976.

[126] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 301–310, 2008.

[127] Andrew J. Ko and Brad A. Myers. Source-level debugging with the whyline. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 69–72, 2008.

[128] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 1569–1578, 2009.

[129] Rahul Kumar and Aditya V. Nori. The economics of static analysis tools. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 707–710, 2013.

[130] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1759–1766, 2008.

[131] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of System Software*, 83(12):2379–2391, 2010.

[132] Sihan Li*, Xusheng Xiao*, Tao Xie, and Nikolai Tillmann. Economic analysis for cooperative structural testing via angelic diagnosis. *The First Two Auhtors Make Equal Contributions. *In Preparation for the International Conference on Software Engineering (ICSE)*, 2014.

[133] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. XEngine: a fast and scalable XACML policy evaluation engine. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 265–276, 2008.

[134] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 499–504, 2011.

[135] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 149–153, 2009.

[136] David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. Cooperative work in software testing. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 93–96, 2008.

[137] Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 163–172, 2008.

[138] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 667–676, 2007.

[139] Math.NET, 2008. http://www.mathdotnet.com/.

[140] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[141] James Bret Michael, Vanessa L. Ong, and Neil C. Rowe. Natural-language processing support for developing policy-governed software systems. In *Proceedings of the International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 263–274, 2001.

[142] MICROSOFT. What is User Account Control?, 2011. http://windows.microsoft.com/en-US/windows-vista/What-is-User-Account-Control.

[143] Ian Molyneaux. *The Art of Application Performance Testing - Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[144] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[145] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[146] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[147] Mary S. Neff, Roy J. Byrd, and Branimir K. Boguraev. The talent system: Textract architecture and data model. *Natural Language Engineering*, 10(3-4):307–326, 2004.

[148] James W. Newkirk, Alexei A. Vorontsov, Michael C. Two, and Philip A. Craig. NUnit, 2004. http://www.nunit.org/.

[149] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[150] OASIS. Privacy policy profile of XACML v2.0., 2005. http://docs.oasis-open.org/xacml/2.0/privateprofile/access_control-xacml-2.0-privacy_profile-specos.pdf.

[151] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, 2002. Planning Report 02-3.

[152] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[153] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 527–542, 2013.

[154] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 815–825, 2012.

[155] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, pages 5:1–5:6, 2011.

[156] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 277–284, 1999.

[157] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of Android Apps. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, pages 241–252, 2012.

[158] QuickGraph, 2008. http://www.codeproject.com/KB/miscctrl/quickgraph.aspx.

[159] Franziska Roesner. User-driven access control: A new model for granting permissions in modern operating systems. *Qualifying Examination Project, University of Washington*, 2011.

[160] Colette Rolland and Camille Ben Achour. Guiding the construction of textual use case specifications. *Data & Knowledge Engineering*, 25(1-2):125–160, 1998.

[161] Gregg Rothermel, Roland J. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering (TSE)*, 27(10):929–948, 2001.

[162] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. Mckinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 63–74, 2009.

[163] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2002.

[164] J H Saltzer and M D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[165] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Revised Versions of Lectures Given During the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures (FOSAD)*, pages 137–196, 2001.

[166] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 139–149, 2006.

[167] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[168] Shiuh-Pyng Shieh and Virgil D. Gligor. Auditing the use of covert storage channels in secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 285–295, 1990.

[169] Avik Sinha, Stanley M. Sutton Jr., and Amit Paradkar. Text2Test: Automated inspection of natural language use cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 155–164, 2010.

[170] Avik Sinha, Amit M. Paradkar, Palani Kumanan, and Branimir Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 327–336, 2009.

[171] Hongzhi Song, Yu Qi, Xuhong Tian, and Dongfeng Xu. Navigating and visualizing long lists with fisheye view and graphical representation. In *Proceedings of the Workshop on Digital Media and its Application in Museum & Heritage (DMAMH)*, pages 123–128, 2007.

[172] Xiang Song, Haibo Chen, and Binyu Zang. Why software hangs and what can be done with it. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 311–316, 2010.

[173] Yoonki Song, Xusheng Xiao, Tao Xie, Emerson Murphy-Hill, Nikolai Tillmann, and Jonathan de Halleux. Visualization of test information to assist structural test generation. *In Preparation for the International Conference on Software Engineering (ICSE)*, 2014.

[174] Kavitha Srinivas and Harini Srinivasan. Summarizing application performance from a components perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 136–145, 2005.

[175] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 223–234, 2009.

[176] Saurabh Srivastava, Sumit Gulwani, and JeffreyS. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.

[177] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 163–178, 2012.

[178] Mark Stickel and Mabry Tyson. FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In *Proceedings of Finite-State Devices for Natural Language Processing*, pages 383–406, 1997.

[179] SvnBridge: Use tortoisesvn with team foundation server, 2009. http://www.codeplex.com/SvnBridge.

[180] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. eXpress: Guided path exploration for efficient regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, 2011.

[181] Kunal Taneja, Yi Zhang, and Tao Xie. Moda: automated test generation for database applications via mock objects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 289–292, 2010.

[182] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 189–206, 2011.

[183] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 193–202, 2009.

[184] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[185] Nikolai Tillmann, Michal Moskal, and Jonathan de Halleux. TouchDevelop - programming cloud-connected mobile devices via touchscreen. *Microsoft Technical Report MSR-TR-2011-49*, 2011.

[186] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the SIGPLAN Symposium on New Ideas in Programming and Reflections on Software (ONWARD!)*, pages 49–60, 2011.

[187] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[188] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–368, 2006.

[189] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP)*, Oakland, CA, May 2011.

[190] Daniel von Dincklage and Amer Diwan. Explaining failures of program analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 260–269, 2008.

[191] Collin Wang, Heng Ma, and David J. Cannon. Human-machine collaboration in robotics: Integrating virtual tools with a collision avoidance concept using conglomerates of spheres. *Journal of Intelligent and Robotic Systems*, 18(4):367–397, 1997.

[192] Lanjia Wang, Haixin Duan, and Xing Li. Port scan behavior diagnosis by clustering. In *Proceedings of the International Conference on Information and Communications Security (ICICS)*, pages 243–255, 2005.

[193] Wuhong Wang and Herner Bubb. A theoretical framework for ecological function allocation in human-machine interface. In *Proceedings of the International Symposium on Knowledge Acquisition and Modeling (KAM)*, pages 257–261, 2008.

[194] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. Hang analysis: fighting responsiveness bugs. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 177–190, 2008.

[195] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 190–199, 2012.

[196] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering (TSE)*, 26(12):1147–1156, 2000.

[197] Laurie Williams and Yonghee Shin. Work in progress: Exploring security and privacy concepts through the development and testing of the iTrust medical records system. In *Proceedings of the Annual Frontiers in Education Conference*, pages 30–31, 2006.

[198] Xusheng Xiao. Problem identification for structural test generation: first step towards cooperative developer testing. In *Proceedings of International Conference on Software Engineering (ICSE), ACM Student Research Competition*, pages 1179–1181, 2011.

[199] Xusheng Xiao, Shi Han, Tao Xie, and Dongmei Zhang. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 90–100, 2013.

[200] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 246–256, 2013.

[201] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 12:1–12:11, 2012.

[202] Xusheng Xiao, Suresh Thummalapenta, and Tao Xie. Advances on improving automation in developer testing. In *Advances in Computers*, volume 85, pages 165–212. Academic Press, 2012.

[203] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan De Halleux, and Michal Moskal. User-aware privacy control via extended static-information-flow analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 80–89, 2012.

[204] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Covana: Precise identification of problems in Pex. In *Proceedings of the International Conference on Software Engineering (ICSE), Demonstration*, pages 1004–1006, 2011.

[205] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–620, 2011.

[206] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2006.

[207] Guoqing (Harry) Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.

[208] xUnit, 2007. http://www.codeplex.com/xunit.

[209] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, page 29, 2012.

[210] Hui Yang, Anne de Roeck, Vincenzo Gervasi, Alistair Willis, and Bashar Nuseibeh. Extending nocuous ambiguity analysis for anaphora in natural language requirements. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, pages 25–34, 2010.

[211] Wei Yang, Xusheng Xiao, Sihan Li, Benjamin Andow, William Enck, and Tao Xie. Appcontext: Analyzing contexts of permission uses in smartphone applications. *In Preparation for the International Conference on Software Engineering (ICSE)*, 2014.

[212] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Journal of Software Testing, Verification, and Reliability (STVR)*, 22(2):67–120, 2012.

[213] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 67–76, 2012.

[214] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 43–52, 2011.

[215] Sai Zhang. Palus: A hybrid automated test generation tool for java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1182–1184, 2011.

[216] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanc Muslu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396, 2014.

[217] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 353–363, 2011.

[218] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234, 2013.

[219] Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 63–72, 2011.

[220] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318, 2009.

[221] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious Apps in official and alternative Android markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[222] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 95–109, 2012.

[223] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.

[224] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29:366–427, 1997.