2014

Improving Quality of Software with Foreign Function Interfaces using Static Analysis

Siliang Li
Lehigh University

IMPROVING QUALITY OF SOFTWARE WITH FOREIGN FUNCTION INTERFACES USING STATIC ANALYSIS

by

Siliang Li

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Engineering

Lehigh University

May 2014

Copyright page

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____
Date

_____
Dr. Gang Tan (Dissertation Director)

_____
Accepted Date

Committee Members:

_____
Dr. Gang Tan (Committee Chair)

_____
Dr. Mooi-Choo Chuah (Committee Member)

_____
Dr. Brian D. Davison (Committee Member)

_____
Dr. Yu David Liu (Committee Member)

DEDICATION

兹将本论文献给我的父亲李德全和母亲王秀芝，以及我所有的家人。感谢你们给予我爱，支持，和奉献！

I dedicate this dissertation to my loving parents, without whom I would not be where I am today. They are the bedrock of my life. I am eternally grateful for their love, understanding, support and patience. Thank you for my upbringing and teachings.

Additionally, I dedicate this dissertation to my extended family in China, especially my Aunts. Throughout the years, they supported me and, most important of all, helped care for my parents in my absence, allowing me worry-free focus on my studies.

Finally, I dedicate this work to all my friends whom have supported me. Special thanks to Youliang Yang, whom has always been there for me, cheered me through good times and bad.

Thank you all with much love.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation and sincerest gratitude to my dissertation advisor and mentor Dr. Gang Tan for taking a chance in me and taking me under his wings over six years ago when I first embarked upon this exciting and challenging journey. It was under Dr. Tan's superb guidance, exemplary scholarship, and tireless dedication that profoundly inspired me to be a mature thinker and independent researcher. His unwavering support, constant encouragement and limitless patience propelled me forward to make this dissertation possible. Without him, I would not have achieved the goals for my dissertation.

Secondly, I would like to thank my dissertation committee of Dr. Mooi-Choo Chuah, Dr. Brian D. Davison, and Dr. David Yu Liu for their insightful consultation and invaluable advice during the past four years from my initial research ideas to its final implementation. At Lehigh University, Dr. Chuah was the first faculty member that I came to know and respect. Throughout the years, she gave me a great deal of support and guidance that helped me overcome difficult times. Dr. Davison has always inspired me and taught me to approach research problems with broader and more practical considerations. It was because of his indispensable inputs that made my dissertation more encompassing. I was also fortunate enough to have collaborated with Dr. Liu from State University of New York, Binghamton. I am truly grateful for his dedication and outstanding expertise on the subject, which made this dissertation complete and comprehensive. My research would not be possible without all the committee members' support.

Additionally, I would like to thank the friendly and helpful staff at the Computer Science and Engineering Department and at the P.C. Rossin College of Engineering and Applied Science at Lehigh University. Their dedication and assistance made my

experience at Lehigh pleasant and smooth.

Finally, I would like to thank my friends and colleagues at the SOS lab at Lehigh University. Their collective friendship and companionship gave me a great sense of support and community, making my dissertation journey rewarding and memorable.

TABLE OF CONTENTS

List of Tables x

List of Figures xi

Abstract 1

1 Introduction 2
- 1.1 Foreign Function Interface . . . 2
- 1.2 Software composed of FFIs . . . 3
- 1.3 Issues with FFIs and software quality . . . 4
  - 1.3.1 Issues with FFIs . . . 4
  - 1.3.2 Software quality issues . . . 7
- 1.4 Challenges and motivations . . . 9
- 1.5 Previous work and open problems . . . 10
- 1.6 Thesis statement . . . 12
- 1.7 Static analysis . . . 12
- 1.8 Overview . . . 13
  - 1.8.1 Exception analysis in the JNI . . . 13
  - 1.8.2 Atomicity enforcement in the JNI . . . 14
  - 1.8.3 Reference count analysis in the Python/C interface . . . 14
- 1.9 Contributions . . . 15

2 Exception Analysis in the Java Native Interface 18
- 2.1 Introduction . . . 18
- 2.2 Background: the JNI . . . 21
  - 2.2.1 How JNI exceptions can be thrown . . . 23
  - 2.2.2 Checked exceptions vs. unchecked exceptions . . . 23
  - 2.2.3 Interface code vs. library code . . . 24
- 2.3 Defining bug patterns . . . 24
  - 2.3.1 Inconsistent exception declarations . . . 25
  - 2.3.2 Mishandling JNI exceptions . . . 26
- 2.4 Overview of TurboJet . . . 30
- 2.5 Scalable and precise exception analysis . . . 32
  - 2.5.1 Separating interface and library code . . . 33
  - 2.5.2 Fine-grained tracking of exception states . . . 34
  - 2.5.3 An FSM specification of exception-state transitions . . . 36
  - 2.5.4 Path-sensitive analysis . . . 37
  - 2.5.5 Context-sensitive analysis . . . 41
  - 2.5.6 Transfer functions for JNI functions . . . 46
  - 2.5.7 Merging symbolic states . . . 46

2.6 Finding bugs of mishandling JNI exceptions . . . 46
2.6.1 Determining unsafe operations . . . 48
2.6.2 Warning recovery . . . 54
2.7 Prototype implementations and evaluation . . . 55
2.7.1 Accuracy . . . 57
2.7.2 Efficiency . . . 61
2.7.3 Comparison with previous studies . . . 63
2.8 An Eclipse plug-in tool . . . 66
2.9 Summary . . . 68

3 Native Code Atomicity for Java 69
3.1 Introduction . . . 69
3.2 Background and assumptions . . . 71
3.3 The formal model . . . 73
3.3.1 Abstract syntax . . . 73
3.3.2 Constraint generation: an overview . . . 75
3.3.3 Intraprocedural constraint generation . . . 77
3.3.4 Constraint closure . . . 81
3.3.5 Atomicity enforcement . . . 82
3.4 Prototype implementation . . . 85
3.5 Preliminary evaluation . . . 87
3.6 Summary . . . 91

4 Reference Counting in Python/C Programs with Affine Program Analysis 92
4.1 Introduction . . . 92
4.2 Background: the Python/C interface and reference counting . . . 95
4.2.1 Python/C reference counting and its complexities . . . 96
4.3 Pungi overview . . . 99
4.4 Affine abstraction . . . 101
4.4.1 Bug definition with non-escaping references . . . 101
4.4.2 SSA transform . . . 104
4.4.3 Affine translation . . . 106
4.4.4 Escaping references . . . 115
4.5 Affine analysis and bug reporting . . . 117
4.6 Implementation and limitations . . . 120
4.7 Evaluation . . . 121
4.8 Summary . . . 125

5 Related Work 126
5.1 FFIs . . . 126
5.2 Work related to TurboJet . . . 127
5.3 Work related to JATO . . . 128
5.4 Work related to Pungi . . . 130

6 Future Work **133**

7 Concluding Remarks **135**

Bibliography **136**

Appendix **148**

A Whitelist **148**

B Interprocedural exception analysis **149**

Brief Biography **154**

Curriculum Vitae **155**

LIST OF TABLES

1.1 Popular real-world JNI applications. 4
1.2 Popular Python/C packages in the Fedora LINUX operating system. . . 5

2.1 Accuracy evaluation of TurboJet on finding inconsistent exception declarations. 55
2.2 Accuracy evaluation of TurboJet for finding inconsistent exception declarations. 56
2.3 Experimental results for assessing effectiveness of warning recovery and static taint analysis 60
2.4 Efficiency evaluation of TurboJet. 62
2.5 Comparing TurboJet with an alternative exception analysis on finding inconsistent exception declarations. 65
2.6 Comparison with the previous study [86] (of the 35 errors in the previous study, 11 are due to explicit throws and 24 due to implicit throws). 66

4.1 Statistics about selected benchmark programs. 122
4.2 All warnings reported by Pungi, which include true reference over- and under-counting errors and false positives. 123
4.3 Comparison of errors found between Pungi and CPyChecker. 125

LIST OF FIGURES

1.1 A conceptual definition of an FFI. 2

2.1 A simple JNI example. This example also demonstrates how a native method can violate its exception declaration. 21
2.2 An example of mishandling JNI exceptions. 22
2.3 Two more examples of mishandling JNI exceptions. 29
2.4 System architecture of TurboJet. 31
2.5 Examples for illustrating the need for path and context sensitivity. . . . 35
2.6 An incomplete FSM specification of exception-state transitions. 37
2.7 Syntax of Java types that TurboJet tracks. 39
2.8 Example of TurboJet path sensitivity. 42
2.9 Exception analysis using ESP. 43
2.10 Exception analysis using TurboJet. 44
2.11 High-level steps of warning generation for mishandling JNI exceptions. 47
2.12 An example program and its pointer graph. The program takes a Java integer array and computes the sum of all positive elements. The nodes with shading are tainted nodes. 53
2.13 A typical example of false positives. 59
2.14 An example of TurboJet plug-in's warning on inconsistent exception declarations. 67
2.15 An example of TurboJet plug-in's warning on mishandling JNI exceptions. 67

3.1 A running example 75
3.2 Java-Side Intraprocedual Constraint Generation 78
3.3 Native-Side Intraprocedural Constraint Generation 79
3.4 Class-Level Constraint Generation 80
3.5 Execution time of the benchmark programs under different locking schemes. 90

4.1 An example Python/C extension module called `ntuple` (its registration table and module initializer code are omitted). 95
4.2 An overview of Pungi. 100
4.3 A contrived example of a buggy Python/C function. 103
4.4 Part of the control-flow graph for the code in Fig. 4.1 after SSA. 105
4.5 Syntax of affine programs. 108
4.6 Affine translation $\mathcal{T}(-)$ for typical C constructs. 109
4.7 Translation of the example in Fig. 4.3. 111
4.8 An example of interprocedural affine translation. 113
4.9 An example of escaping references. 116
4.10 An example of affine analysis. 119

ABSTRACT

A Foreign Function Interface (FFI) is a mechanism that allows software written in one host programming language to directly use another foreign programming language by invoking function calls across language boundaries. Today's software development often utilizes FFIs to reuse software components. Examples of such systems are the Java Development Kit (JDK), Android mobile OS, and Python packages in the Fedora LINUX operating systems. The use of FFIs, however, requires extreme care and can introduce undesired side effects that degrade software quality. In this thesis, we aim to improve several quality aspects of software composed of FFIs by applying static analysis.

The thesis investigates several particular characteristics of FFIs and studies software bugs caused by the misuse of FFIs. We choose two FFIs, the Java Native Interface (JNI) and the Python/C interface, as the main subjects of this dissertation. To reduce software security vulnerabilities introduced by the JNI, we first propose definitions of new patterns of bugs caused by the improper exception handlings between Java and C. We then present the design and implement a bug finding system to uncover these bugs. To ensure software safety and reliability in multithreaded environment, we present a novel and efficient system that ensures atomicity in the JNI. Finally, to improve software performance and reliability, we design and develop a framework for finding errors in memory management in programs written with the Python/C interface. The framework is built by applying affine abstraction and affine analysis of reference-counts of Python objects. This dissertation offers a comprehensive study of FFIs and software composed of FFIs. The research findings make several contributions to the studies of static analysis and to the improvement of software quality.

CHAPTER 1

INTRODUCTION

1.1 Foreign Function Interface

A Foreign Function Interface (FFI) is a programming language interface that allows software code written in one programming language (*i.e.*, *host* language) to use software code written in another programming language (*i.e.*, *foreign* language) in the form of invoking function calls or method calls across the boundaries of the two languages. Fig. 1.1 illustrates a conceptual definition of an FFI. It acts as a gluing layer that connects the code together for two software components written in different programming languages.

FFIs offer flexible and convenient interoperability between languages. For example, a programmer can design and implement a software component in a high-level language, like Java, while leveraging on an existing software component, such as one written in C. She can do so by simply snapping together the different components with the help of an FFI.

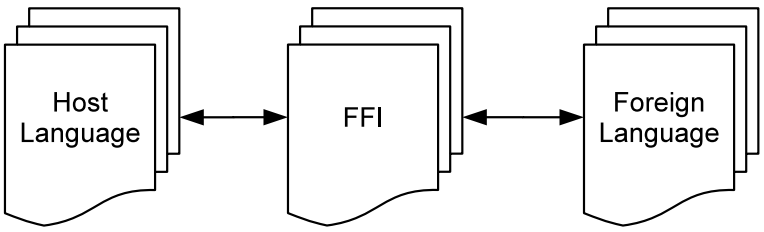


Figure 1.1: A conceptual definition of an FFI.

1.2 Software composed of FFIs

Today's software development demands faster time-to-market as well as more reliability and higher system performance. The use of FFIs allows programmers to achieve these goals. Programmers can focus on introducing and developing new functionalities while leveraging FFIs to connect together existing libraries and system software that offer the more common and standard functionalities. For the most part, these libraries and system software are proven over the years for their reliability and performance.

Many well-known software systems are composed of FFIs. Examples include the Java Development Kit (JDK), Android mobile OS, and the Python packages in the Fedora LINUX operating systems. The JDK is packaged with core system libraries such as file compression utilities and math libraries. These libraries are written in C/C++. In order to interoperate with these libraries, there is a large set of Java Native Interface (JNI) code that glues together the Java code and the C/C++ code. The popular Android mobile OS is another example of systems composed of FFIs. It includes a Java Virtual Machine implementation (*i.e.*, Dalvik) and a kernel written in C/C++. To interoperate between the virtual machine and the kernel, JNI code is used in the Android mobile OS. Table 1.1 shows a set of other popular real-world JNI applications.

Another example of systems composed of FFIs is the Python packages in the Fedora LINUX operating system. These packages are written in Python. Like the JDK, they often take advantage of the standard and system libraries that already exist and written in C/C++. Python code interoperates with the C/C++ code by the use of the Python/C interface. Table 1.2 lists some of these popular Python/C packages [2].

JNI Package	Description
java-gnome	A package that allows Java to use the GNOME desktop environment [34].
jogl	A library that allows OpenGL to be used in Java [36] .
libec	A library for elliptic curve cryptography.
libreadline	A package that provides JNI wrappers for accessing the GNU readline library [52].
posix	A package that provides JNI wrappers for accessing OS system calls [71].
spread	An open-source toolkit that provides a high performance messaging service across local and wide area networks; it comes with its default JNI bindings [83].

Table 1.1: Popular real-world JNI applications.

1.3 Issues with FFIs and software quality

While the use of FFIs brings much convenience and efficiency in software development, it at the same time introduces plenty of side effects that can degrade software quality. We next take a look at some key intrinsic characteristics of FFIs and the effects these characteristics have on quality of software composed of FFIs.

1.3.1 Issues with FFIs

Programming with FFIs is an error-prone process. A programmer needs to know thoroughly about both host programming languages and foreign programming languages, especially the differences between the two. Some of the differences are obvious, such as program semantics, language syntax, and type systems. But some are subtle, such as exception handling and memory management. She also needs to have a good understanding of the proper use of FFIs. All of these pose challenges in writing correct and bug-free code.

Python/C Package	Description
krbV	A library that allows python programs to use Kerberos 5 authentication and security.
pycrypto	A collection of both secure hash functions and various encryption algorithms.
pyxattr	A extension module wrapper for libattr, which allows for various file and directory operations.
rrdtool	Round Robin Database Tool to store and display time-series data.
dbus	A message bus system, a simple way for applications to talk to one another.
gst	A wrapper that allows GStreamer applications to be written in Python.
canto	An Atom/RSS feed reader for the console that is meant to be quick, concise, and colorful.
duplicity	An utility that backs up files and directory.
netifaces	A library to retrieve information about network interfaces.
pyaudio	A library that provides Python bindings for PortAudio, the cross-platform audio I/O.
pyOpenSSL	A Python wrapper module around the OpenSSL library.
ldap	A library that provides an object-oriented API for working with LDAP within Python programs.
yum	A Python library for software package manager that installs, updates, and removes packages on RPM-based systems.

Table 1.2: Popular Python/C packages in the Fedora LINUX operating system.

The discussion below reviews some typical differences between host programming languages and foreign programming languages in the context of FFIs, and the type of bugs that can be caused by these differences. The host programming languages suggested here are the typical high-level languages like Java, C#, and Python. The foreign languages can be native languages, like C, C#, and assembly.

Exception handling

Modern programming languages like Java and C# have managed environments where errors and exceptional situations are handled within the environments. Managed envi-

ronments provide better security assurance to software systems, where faulty code in software won't easily become security vulnerabilities. With the use of FFIs, however, this security model can be broken. By using the JNI, for example, code written in C and C++ can throw Java exceptions. But, these exceptions are outside the normal control of the managed environment of Java, that is, the JVM. Unless programmers diligently ensure exceptional situations are handled properly, hackers can craft clever security attacks by taking advantage of the faulty code.

Thread and concurrency model

Modern programming languages also often have support for multithreaded programming. To ensure thread-safety, many of these languages provide mechanisms to permit only one thread to execute a protected region of code at a time. This notion is known as *atomicity*. Individual programming languages may have different mechanisms in ensuring atomicity. In general, to ensure atomicity is difficult and challenging. In the context of FFIs, this can be even more complex, involved, and error-prone.

Memory management

Java, C#, and Python have memory management systems. The memory management systems are responsible for making allocated memory available and cleaning up memory that are no longer needed. Programmers are freed from the task of manually managing memory resources in these languages. C/C++ does not have an automatic memory management system. When programming with FFIs whose foreign languages are C/C++, programmers need to be keenly aware of this difference and take appropriate actions to ensure memory resources are managed correctly. For instance, memory resources

allocated in a foreign language can be out of reach of the memory management of a host language and cannot be reclaimed automatically. Programmers cannot assume the resources are cleaned by the managed environment and have to make sure the memory resources are reclaimed manually.

1.3.2 Software quality issues

In this section, we discuss software quality issues that we are concerned with and outline our objectives in addressing these issues. By quality, we are primarily interested in the following set of attributes: security, reliability, safety, and performance.

Security

A secure software system composed of an FFI means that the system cannot be exploited in its FFI layer for security attacks. That is, having an FFI in a software system should not make the system less secure. We intend to explore bug patterns introduced by FFIs that can introduce security vulnerabilities and show how one can protect software composed of FFIs against these types of attacks.

Reliability

Software should not become less reliable by employing an FFI in the system. Specifically, exceptional situations in the FFI should not cause the system to behave unexpectedly. We are interested in identifying sources of these potential reliability issues and types of bugs in the system.

Safety

By safety, we mean multithreaded safety in software systems composed of FFIs. Multithreaded safety has been studied extensively before but primarily for single-language systems. These safety issues commonly involve data integrity, thread synchronization, race condition, and deadlock. In recent years, there have been studies on thread atomicity, which is a stronger guarantee than race-free condition. But, few if any of these studies have explored on multi-lingual systems involving FFIs, partly due to the complexity and difficulty in studying such systems. We are interested in taking up this challenge and investigate thread atomicity in software systems composed of FFIs. The safety is expected to be achieved without outsized performance overhead.

Performance

A software system's performance should not be impacted by having FFIs. After all, many programmers choose to use FFIs for the purpose of achieving higher performance; native code components in general perform better than those written with host programming languages. However, that assumption is not guaranteed and can be defeated if care is not taken while programming with FFIs. We aim to identify performance issues introduced by having FFIs, in particular, those related to memory management. We are interested in finding the commonly known problems like memory leak and dangling pointers in software composed of FFIs.

1.4 Challenges and motivations

FFIs present unique challenges in both identification of the bug patterns and the design of suitable solutions to find these types of bugs.

Bug definitions are not always easy because of their subtlety, little previous experience, and complexities involved. The identification of bug patterns is to accurately describe the unique situation when a bug is caused by an FFI. These types of bugs are subtle and not easily discernable. Even when they do surface, they can be mistakenly seen as the same as those in software without FFIs and hence overlooked and never uncovered. There had been little research done in the systematic identification of bugs caused by FFIs, whether empirical or experimental. This requires keen knowledge of both the host and the foreign programming languages, in particular in areas like type systems, exception handlings, memory management, and thread models. Furthermore, to precisely define what a bug is requires in-depth understanding of the interactions of the two programming languages through the use of an FFI and the impact from the FFI itself during the interactions.

Finding bugs and hence solutions to software quality issues can also be challenging in the context of FFIs. The challenges stem from finding suitable solutions that are effective, accurate, and efficient. Typically, there are two means for finding software bugs: dynamic analysis and static analysis. This is also true for finding bugs in FFIs. In dynamic analysis, bugs are discovered during an execution or the runtime of the software. And, this is typically achieved by manually written tests. Because of the particular layer where an FFI exists in code and the functional purpose of the FFI, it is rather difficult to trigger a bug to manifest in the FFI. To ensure software free from quality issues caused by the FFI, which is to find all the possible bugs in the FFI, would require lots of manual

tests hence incur a large amount of overhead in time and efforts. Static analysis, on the other hand, requires no runtime overhead. With static analysis, bugs are uncovered by scanning the code[1] of software, which can be done relatively quickly. In addition, when static analysis is designed appropriately, it can find all possible bugs in software. The downside of static analysis, however, is that it may find bugs that are not true bugs, that is, false positives.

1.5 Previous work and open problems

There have been increased interests in recent years in the study of programming interoperability, in particular, the use of FFIs, FFIs' effects on software, and ways to mitigate some of the negative effects FFIs bring to software quality. In this section, we highlight the most significant work in recent years and overview open problems that still need to be addressed.

A few recent studies reported hundreds of interface bugs in JNI programs ([26, 40, 86]) by using type systems, experimental studies, and empirical studies, respectively. Errors occur often in interface code because FFIs generally provide little or no support for safety checking, and also because writing interface code requires resolving differences (*e.g.*, memory models and language features) between two languages. Past work on improving FFIs' safety can be roughly classified into several categories: (1) Static analysis has been used to identify specific classes of errors in FFI code [25, 26, 87, 40]; (2) In another approach, dynamic checks are inserted at the language boundary and/or in the native code for catching interface errors (see [45]) or for isolating errors in native

[1]Static analysis can scan either source code or binary code. In this dissertation, static analysis is performed on software source code.

code so that they do not affect the host language's safety [85] and security [80]; (3) New interface languages are designed to help programmers write safer interface code (*e.g.*, [31]).

There are still many open problems despite recent advancements in this research area. First and foremost, identifications of bugs or bug patterns that exist in software composed of FFIs have just begun. The types of bugs reported thus far consist only a small set of problems and touch only a few software quality issues; we believe there are more types of bugs that are unique in these systems, as demonstrated in our later chapters. Definitions of bugs in FFIs cannot simply assume the same bugs that exist in single-language systems are applicable here. They require a fresh look of software and demand in-depth understanding and analysis of FFIs. Secondly, there are only a few proposed solutions that are suitable or general enough to find bugs in software composed of FFIs. As discussed earlier, finding accurate, efficient, and scalable solutions requires designing and building new methods and frameworks. In the case with static analysis approach, the challenges continue to be the ability of having sound systems (*i.e.*, ones that leave no false negatives) while achieving low false positives. Third, bug finding tools designed in research settings, especially in the area of FFIs, are still off-limits in practice for the most part for programmers who develop software using FFIs. This is because: 1) the tools may not be that easy and practical to be used and 2) the bugs reported are not always of practical importance. Our research aims to address some of these open problems, and to provide more comprehensive solutions and discussions in this area. We hope to provide a concrete set of tools that programmers can use in finding quality related bugs in their software development.

1.6 Thesis statement

Software systems often use components written in Foreign Function Interfaces (FFIs) for the purpose of development efficiency and high system performance. Such software, however, can suffer in quality because of possible misuse of FFIs. In this thesis, we present systems designed and built by applying static analysis of source code of software composed of FFIs to identify bugs that can cause software quality issues.

The systems presented in this dissertation are effective and accurate; experimental results demonstrate that our systems can find bugs with very low false negatives and low false positives. The systems are also scalable and efficient; they can scan large software systems quickly and efficiently.

1.7 Static analysis

Static analysis is the study of software code (source code, binary code, or both). We choose static analysis as the main methodology to tackle the open problems discussed earlier because of the following reasons. First, differing from dynamic analysis, which requires an actual execution of a software system, static analysis does not require the running of the software system. It introduces no runtime overhead compared to dynamic analysis. Second, many types of quality related issues can only be feasibly found by static analysis but not by dynamic analysis because static analysis can examine every control flow path of a program and offer assurance in finding all these issues. That is, when applied in a sound and safe manner, static analysis produces no false negatives. Finally, applying static analysis in software composed of FFIs requires an in-depth study

and novel improvements of static analysis, which evokes great intellectual curiosity, interest and creativity. Static analysis has been studied extensively in the past in program verification but primarily in software systems composed of a single programming language. To apply static analysis in FFIs is a new exciting adventure and requires: 1) in-depth knowledge of static analysis, 2) rigorous studies and research in the area of programming languages and software engineering, and 3) proposals of novel improvements and applications of static analysis to address the unique and challenging issues in software composed of FFIs. Because of these reasons, we choose to use static analysis as the primary approach for finding bugs and for improving quality of software systems composed of FFIs.

1.8 Overview

We now present an overview of the research conducted in this dissertation.

1.8.1 Exception analysis in the JNI

Software composed of the Java Native Interface (JNI) often has security and reliability issues. We introduce an exception analysis framework TurboJet to address these issues. This framework is built by applying several static analyses and our proposed improvements on these techniques. The framework finds exception-related bugs in JNI applications. Specifically, it finds bugs of inconsistent exception declarations and bugs of mishandling JNI exceptions. TurboJet is carefully engineered to achieve both high efficiency and accuracy. We have applied TurboJet on a set of benchmark programs and

identified many errors. We have also implemented a practical Eclipse plug-in based on TurboJet that JNI programmers can use to find errors in their code.

1.8.2 Atomicity enforcement in the JNI

Atomicity enforcement in a multithreaded application can be critical to the application's safety and reliability. In this project, we take the challenge of enforcing atomicity in a multilingual application, which is developed in multiple programming languages. Specifically, we describe the design and implementation of JATO, which enforces the atomicity of a native method when a Java application invokes the native method through the JNI. JATO relies on a constraint-based system, which generates constraints from both Java and native code based on how Java objects are accessed by threads. Constraints are then solved to infer a set of Java objects that need to be locked in native methods to enforce the atomicity of the native method invocation. We also propose a number of optimizations that soundly improve the performance. Evaluation through JATO's prototype implementation demonstrates it enforces native-method atomicity with reasonable runtime overhead.

1.8.3 Reference count analysis in the Python/C interface

Python programmers can write software (also known as Python "extension modules") in C/C++ with the help of the Python/C interface. As discussed earlier, native code in the Python/C interface are outside Python's system of memory management; therefore extension programmers are responsible for making sure these objects are reference counted correctly. This is an error prone process when code becomes complex. In

this project, we propose Pungi, a system that statically checks whether Python objects' reference counts are adjusted correctly in Python/C interface code. Pungi transforms Python/C interface code into affine programs with respect to our proposed abstractions of reference counts. Our system performs static analysis on transformed affine programs and reports possible reference counting bugs. Our prototype implementation found over 150 bugs in a set of Python/C programs.

1.9 Contributions

This dissertation makes several contributions in areas of improving software quality, static analysis, and research of FFIs. Specifically, this dissertation gives clear definitions of bug patterns that affect quality of software composed of FFIs, proposes a set of concrete tools that can be used to find these bugs, and identifies important bugs in real-world software applications.

First, it is among the first to clearly identify a set of software bugs in FFIs. Because of the challenges in understanding the intricacies of programming languages, their complex interactions through the use of FFIs, and the context under which bugs can arise, the bugs that we are interested in are subtle and not easily discernable but all have critical implications. Clear and accurate definitions of the bugs provide the basis of comprehending the problems and identifying the solutions needed. In this thesis, we give precise definitions of a number of bug patterns. For example, in the analysis of exception handling in the JNI, we define a particular bug pattern of mishandled Java exceptions that can cause security vulnerabilities in the JNI. This bug pattern is unique and has not been studied previously. For this type of bugs to manifest, two conditions must

satisfy: 1) a pending exception at a program point in the JNI code, and 2) the operation immediately follows the program point is a dangerous operation. This definition of the bug pattern gives us a clear objective on what our tool needs to identify in order to find these types of bugs.

Second, we demonstrate that finding the bugs that we are concerned with is tractable, can be achieved by using static analysis, and can be done with accuracy and efficiency. We strive for *soundness* in the design of all our systems. By soundness, we mean a system should produce no false negatives. We rigorously and conservatively inspect all aspects of a program. We factor these considerations into the selections among different static analyses and the designs of final systems. Experimental results show that all our systems are able to achieve satisfactory accuracy. Our systems are also efficient and scalable. The systems are designed with reasonable engineering trade-offs. For example, by focusing only on the code that actually has effects on the reference-count changes of Python objects in the Python/C interface, the two-stage system we propose omits code that has no relevance in the bug finding, allowing the system to analyze entire software systems very quickly.

Third, we introduce a number of improvements on the static analyses that were chosen in our systems. The improvements are novel and essential in addressing the unique challenges we face in finding the particular types of bugs in software systems. The improvements enable us to achieve accurate bug finding as well as efficiency. For example, in the finding of bugs caused by mishandled Java exceptions and inconsistent Java exception declarations, we propose a context-sensitive analysis as improvement on a previous proposed program analysis [17]. This improvement allows us to find the types of bugs more accurately. Another example is in the finding of reference-count errors in the Python/C interface. A previous study [42] proposed a system that uses affine-analysis

for finding reference-count bugs. Our system improves upon that study by introducing escape-reference-count analysis as well as using Static Single Assignment (SSA). This greatly enhances the capability and accuracy of our bug finding system.

Lastly, our research uses real software systems in our experiments and finds bugs that are of practical importance. Our experiments are carried out on FFI packages like the JDK, popular JNI libraries, and the Python libraries in the Fedora LINUX operating systems. By using these real software systems, we demonstrate the usefulness of our systems. More importantly, we show our systems are practical. The bugs found by our systems have either been reported to the respective software bug repositories or compared and verified with those bugs that have been previously reported.

CHAPTER 2

EXCEPTION ANALYSIS IN THE JAVA NATIVE INTERFACE

2.1 Introduction

The Java Native Interface (JNI) allows Java programs to interface with low-level C/C++/assembly code (*i.e.*, native code). A native method is declared in a Java class by adding the `native` modifier. For example, the `ZipFile` class in Fig. 2.1 declares a native method named `open`. The actual implementation of the `open` method is implemented in C. Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. We define a *JNI application* as a set of Java class files together with the native code that implements the native methods declared in the class files.

Specifically in this chapter, we study the differences of exception-handling mechanisms between Java and native code. Java has two features related to exceptions that help improve program reliability.

- *Compile-time exception checking.* A Java compiler enforces that a checked exception must be declared in a method's (or constructor's) `throws` clause if it is thrown and not caught by the method (or constructor). While the usefulness of checked exceptions for large programs is not universally agreed upon by language designers, proper use of checked exceptions improve program robustness by enabling the compiler to identify unhandled exceptional situations during compile time.
- *Runtime exception handling.* When an exception is pending in Java code, the Java

Virtual Machine (JVM) automatically transfers the control to the nearest enclosing try-catch block that matches the exception type.

Native methods can also throw, handle, and clear Java exceptions through a set of interface functions provided by the JNI. Consequently, an exception may be pending when the control is in a native method. For the rest of this chapter, we will use the term *JNI exceptions* for those exceptions that are pending on the native side, while using the term *Java exceptions* for those pending on the Java side. JNI exceptions are treated differently from Java exceptions.

- A Java compiler does not perform compile-time exception checking on native methods, in contrast to how exception checking is performed on Java methods.
- The JVM does not provide runtime exception handling for JNI exceptions. An exception pending on the native side does not immediately disrupt the native-code execution, and only after the native code finishes execution will the JVM mechanism for exceptions start to take over.

Because of these differences, it is easy for JNI programmers to make mistakes. First, since there is a lack of compile-time exception checking on native methods, the exceptions declared in a native-method type signature might differ from those exceptions that can actually happen during runtime. Second, since there is no support for runtime exception handling in native methods, JNI programmers have to write their own code for implementing the correct control flow for handling and clearing exceptions—an error-prone process. Both kinds of mistakes might lead to unexpected crashes in Java programs and even security vulnerabilities. More detailed discussion and concrete examples of such mistakes will be presented in Section 2.3.

The major contribution of this chapter is TurboJet, a complete framework that performs exception analysis on JNI applications. Specifically, TurboJet statically analyzes and examines JNI code for finding inconsistencies between native-method exception declarations and implementations. Furthermore, it statically analyzes native-method implementations to check whether exceptions are handled properly. The benefit of static analysis is that it examines every control-flow path in a program and in general does not miss mistakes. It is especially appropriate for catching defects in exceptional situations because they are hard to trigger with normal input and events. Having said that, we note that the focus of TurboJet is not the creation of new static-analysis algorithms, but rather the building of a scalable system that performs efficient and accurate analysis over large Java packages of mixed Java and native code. In this process, TurboJet adapts several static-analysis algorithms, which are combined to achieve favorable results.

Our experimental results show that TurboJet delivers high accuracy in catching bugs with relatively low false-positive rates and short execution time. For 16 benchmark programs with nearly 100K lines of code, TurboJet finds 147 bugs caused by mishandling inconsistent exceptions and 17 bugs of inconsistent exception declarations, with the false positive rates 23% and 32%, respectively. The total time took to examine all benchmark programs is only 14 seconds.

The rest of this chapter is structured as follows. We present in Section 2.2 some background information. In Section 2.3, we discuss in detail the bug patterns studied in this chapter, followed by an overview of TurboJet in Section 2.4. In Section 2.5, we present details of TurboJet's exception-analysis framework. We then discuss how TurboJet detects bugs of mishandling JNI exceptions in Section 2.6. Prototype implementation and evaluation are presented in Section 2.7. We have developed an Eclipse plug-in based on TurboJet; its details are discussed in Section 2.8. We summarize in

Section 2.9.

2.2 Background: the JNI

The JNI is Java's foreign function interface that allows Java code to interoperate with native code. Figs. 2.1 and 2.2 provide two example JNI programs. A Java class declares a native method using the `native` keyword. The native method can be implemented in C code. Once a native method is declared, Java code can invoke the native method in the same way as how a Java method is invoked.

Java code

```
class ZipFile {
  // Declare a native method;
  // no exception declared
  private static native long open
    (String name, int mode,
     long lastModified);

  public ZipFile (...) {
    // Calling the method
    // may crash the program
    ...; open(...); ...
  }

  static {System.loadLibrary("ZipFile");}
}
```

C code

```
void Java_ZipFile_open (JNIEnv *env, ...) {
  ...
  // An exception is thrown
  ThrowIOException(env);
  ...
}
```

Figure 2.1: A simple JNI example. This example also demonstrates how a native method can violate its exception declaration.

Interactions between Java and C can be involved and are made possible through the use of JNI functions. In general, Java can pass C code a set of Java object references.

Java code

```
class Vulnerable {
  //Declare a native method
  private native void bcopy(byte[] arr);
  public void byteCopy(byte[] arr) {
    //Call the native method
    bcopy(arr);}
  static {
    System.loadLibrary("Vulnerable");
  }
}
```

C code

```
//Get a pointer to the Java array,
//then copy the Java array
//to a local buffer
void Java_Vulnerable_bcopy
  (JNIEnv* env, jobject obj, jbyteArray jarr) {
  char buffer[512];
  //Check for the length of the array
  if ((*env)->GetArrayLength(jarr)>512) {
    JNU_ThrowArrayIndexOutOfBoundsException(env,0);
  }
  jbyte *carr =
    (*env)->GetByteArrayElements(jarr,NULL);
  //Dangerous operation
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements(arr,carr,0);
}
```

Figure 2.2: An example of mishandling JNI exceptions.

In addition, a JNI environment pointer of type "`JNIEnv *`" is passed to C. The environment pointer contains entries for JNI function pointers, through which C code can invoke JNI functions. For instance, the C code in Fig. 2.2 invokes `GetArrayLength` through the environment pointer to get the length of a Java array. Through these JNI functions, native methods can read, write, and create Java objects, raise exceptions, invoke Java methods, and so on.

2.2.1 How JNI exceptions can be thrown

Both Java code and native code may throw exceptions. There are several ways that an exception may become pending on the native side:

- Native code can throw exceptions directly through JNI functions such as `Throw` and `ThrowNew`. We call such throws *explicit throws*.
- Many JNI functions throw exceptions to indicate failures. For instance, `NewCharArray` throws an exception when the allocation fails. Such throws are considered as *implicit throws*.
- A native method can call back a Java method through JNI functions such as `CallVoidMethod`. The Java method may throw an exception. After the Java method returns, the exception becomes pending on the native side. We treat this type of throws also as implicit throws.

Both explicit and implicit throws result in pending exceptions on the native side.

2.2.2 Checked exceptions vs. unchecked exceptions

In Java, there are two kinds of exceptions: *checked exceptions* and *unchecked exceptions*[1]. Checked exceptions are used to represent those error conditions that a program can recover from and are therefore statically checked by Java's type system. That is, a checked exception that can escape a Java method (or constructor) has to be a subclass of the exception classes declared in the method's (or constructor's) `throws` clause.

[1]Java's unchecked exceptions include `Error`, `RuntimeException` and their subclasses; all other exception classes are checked exceptions.

By contrast, unchecked Java exceptions are not statically checked. Following Java's practice, TurboJet issues warnings about inconsistent exception declarations only for checked exceptions. Nevertheless, TurboJet's exception analysis tracks both checked and unchecked exceptions (unlike our previous work [50], which tracks only checked exceptions). The tracking of unchecked exceptions is necessary for finding the second kind of bugs (*i.e.*, mishandling JNI exceptions).

2.2.3 Interface code vs. library code

Native code associated with a JNI package can roughly be divided into two categories: *interface code* and *library code*. The library code is the code that belongs to a common native library. The interface code implements Java native methods and glues Java with C libraries through the JNI. For example, the implementation of the Java classes under `java.util.zip` has a thin layer of interface code that links Java with the popular zlib C library. Typically, the size of interface code is much smaller than the size of library code; this fact is taken advantage of by TurboJet for better efficiency, as we will show. It is also worth mentioning that some JNI packages include only interface code. For example, native code in `java.io` directly invoke OS system calls for IO operations and does not go through a native library.

2.3 Defining bug patterns

TurboJet looks for two kinds of mistakes in native methods of JNI applications: inconsistent exception declarations and mishandling JNI exceptions. We next give a detailed

discussion of these two kinds of mistakes and present concrete examples.

2.3.1 Inconsistent exception declarations

Definition 1 *A native method has an inconsistent exception declaration if one exception that can be thrown from its implementation is not a subclass of the exceptions declared in the method's Java signature.*

Since a Java compiler does not perform static exception checking on native methods, this type of bug might lead to unexpected crashes in Java programs. Let us use the program in Fig. 2.1 as an example. The `ZipFile` class declares a native method named `open`. The Java-side declaration of `open` does not have any `throws` clause, leading programmers and the compiler to think that no checked exceptions can be thrown. However, the C-code implementation does throw an `IOException`; `ThrowIOException` is a utility function that throws a Java `IOException`, violating the declaration. This cannot be detected by a Java compiler. Consequently, when the `ZipFile` constructor invokes the native `open` method, the constructor is not required to handle the exception or declare it. But in reality a thrown `IOException` in `open` crashes the program, out of the expectation of programmers and the compiler. This example was actually a real bug in `java.util.zip.Zipfile` of Sun's Java Development Kit (JDK); it was fixed only recently in JDK6U23 (JDK 6 update 23) by adding a "*throws IOException*" clause to `open`.

This type of bug can be difficult to debug when they occur. The JVM will report a Java stack trace through which programmers can know the exception originates in a native method. However, since the exception may be thrown due to nested function calls

in native code, it is still difficult to locate the source of the exception without knowing the native call stack. JNI debuggers (*e.g.*, [44, 19]) can help, but not all bugs manifest themselves during particular program runs.

2.3.2 Mishandling JNI exceptions

As we have discussed, JVM provides no runtime support for exception handling when a native method is running. A pending JNI exception does not immediately disrupt the native code execution. Because of the subtle difference between the runtime semantics of JNI exceptions and Java exceptions, it is easy for JNI programmers to make mistakes.

Fig. 2.2 presents a toy example that shows how mishandling JNI exceptions may lead to security vulnerabilities. In the example, the Java class `Vulnerable` declares a native method, which is realized by a C function. The C code checks for some condition (a bounds check in this case), and when the check fails, a JNI exception is thrown. The function `JNU_ThrowArrayIndexOutOfBoundsException` is a JNI utility function. It first uses `FindClass` to get a reference to class `ArrayIndexOutOfBoundsException` and then uses `ThrowNew` to throw the exception. It appears that the following `strcpy` is safe as it is after the bounds check. However, since the JNI exception does not disrupt the control flow, the `strcpy` will always be executed and may result in an unbounded string copy. Consequently, an attacker can craft malicious input to the public Java `byteCopy()` method, and overtake the JVM.

The error in Fig. 2.2 can be fixed quite easily—just insert a return statement after the exception-throwing statement. In general, when a JNI exception is pending, native code

should do one of the following two things:

- Perform some clean-up work (*e.g.*, freeing buffers) and return the control to the Java side. Then the exception-handling mechanism of the JVM takes over.
- Handle and clear the exception on the native side using certain JNI functions. For example, `ExceptionClear` clears the pending exception; `ExceptionDescribe` prints information associated with the pending exception; `ExceptionOccurred` checks if an exception is pending.

In short, JNI programmers are required to implement the control flow of exception handling correctly. This is a tedious and error-prone process, especially when function calls are involved. For example, imagine a C function, say `f`, invokes another C function, say `g`, and the function `g` throws an exception when an error occurs. The `f` function has to explicitly deal with two cases after calling `g`: the successful case, and the exceptional case. Mishandling it may result in the same type of errors as the one in the example. An empirical study has identified 35 cases of mishandling JNI exceptions in 38,000 lines of C code [86].

The error pattern of mishandling exceptions is not unique to the JNI. Any programming language with managed environments that allows native components to throw exceptions faces the same issue. Examples include the Python/C API interface [72] and the OCaml/C interface [47].

Furthermore, mishandling Java exceptions in JNI-based C code is a special case of mishandling errors in C. Since the C language does not provide language support for exceptions, C functions often use integer return codes to report errors to callers.

Callers must use these return values to check for error conditions and perform appropriate actions such as handling errors or propagating errors to their callers. This process is tedious and can be many programming mistakes, as reported by a study on how errors are handled in an embedded software system [11]. Previous systems [75, 11] used static analysis to automatically detect error-handling errors in C code. These systems rely on manual patterns to determine where errors are generated, propagated, and handled. In the JNI context, where Java exceptions are generated, propagated, and handled can be identified automatically based on relevant JNI-function invocations; this makes our problem more tractable.

Defining the pattern of mishandling JNI exceptions. When a JNI exception is pending, native code should either return to the Java side after performing some clean-up tasks, or handle and clear the exception itself. Intuitively, there is a set of "safe" operations that are allowed when an exception is pending. For example, a return-to-Java operation is safe, so are calling JNI functions such as `ExceptionOccurred` or `ExceptionClear`. Given this intuition, we next define the defect pattern of mishandling JNI exceptions:

Definition 2 *JNI-based native code has a defect of* mishandling JNI exceptions *if there is a location in the code that can be reached during runtime with a state such that*

1. *a JNI exception is pending,*
2. *and the next operation is an unsafe operation.*

By this definition, the example in Fig. 2.2 has a defect because it is possible to reach the location before `strcpy` with a pending exception, and `strcpy` is in general unsafe.

(a)

```
int len=(*env)->GetArrayLength(arr);
int *p=(*env)->GetIntArrayElements(arr,NULL);
for (i=0; i<len; i++) {
  sum += p[i];
}
```

(b)

```
jcharArray elemArr=(*env)->NewCharArray(len);
(*env)->SetCharArrayRegion(elemArr,0,len,chars);
```

Figure 2.3: Two more examples of mishandling JNI exceptions.

We make a few clarifications about the definition. First, a JNI exception can be either a checked or an unchecked exception. Second, it leaves open the notion of "unsafe operations". Our strategy of determining unsafe operations will be described in Section 2.6.1. Finally, readers may wonder whether it is wise to look for defects of mishandling JNI exceptions in the first place. After all, one could argue that the example in Fig. 2.2 is a case of buffer overflows and a bug finder targeting buffer overflows would suffice. This is indeed true for that example. However, we would argue that there are many other scenarios that do not involve buffer overflows, but are similar to the example in terms of incorrect control flow following JNI exceptions. We give two examples in Fig. 2.3.

In Fig. 2.3(a), `GetIntArrayElements` may throw an exception and return NULL when it fails to get a pointer to an input Java integer array. In this case, the subsequent array access in `p[i]` results in a null-pointer dereference.

In Fig. 2.3(b), `NewCharArray` may throw an exception when allocation fails. The JNI reference manual states that "it is extremely important to check, handle, and clear a pending exception before calling any subsequent JNI functions. Calling most JNI functions with a pending exception—with an exception that you have not explicitly

cleared—may lead to unexpected results." Therefore, it is unsafe to invoke the next JNI function `SetCharArrayRegion` in the example.

These examples demonstrate that mishandling JNI exceptions may result in a variety of runtime failures. Rather than constructing a set of bug finders targeting each of these failures, it is beneficial to have a single framework targeting the general pattern of mishandling JNI exceptions.

2.4 Overview of TurboJet

Fig. 2.4 presents a high-level diagram that depicts the system flow of TurboJet. At a high-level, TurboJet takes a JNI application as input and generates warnings for native methods in terms of inconsistent exception declarations and mishandling JNI exceptions. A JNI application is composed of both Java class files and native code that implements the native methods declared in the class files. The application is fed into an exception-analysis component, which performs static analysis to determine possible pending JNI exceptions at each program point. We use the term *exception effects* to refer to the set of exceptions that may escape a method. The exception-effect information is then used by two warning generators that search for bugs of (1) inconsistent exception declarations, and (2) mishandling JNI exceptions.

The exception-analysis component is designed to be both efficient and precise. These goals are achieved by a two-stage analysis: the first stage is a coarse-grained analysis, which quickly prunes a large chunk of irrelevant code; the second stage performs fine-grained analysis that employs both path sensitivity and context sensitivity to perform precise exception analysis on the remaining code. Details of the exception-

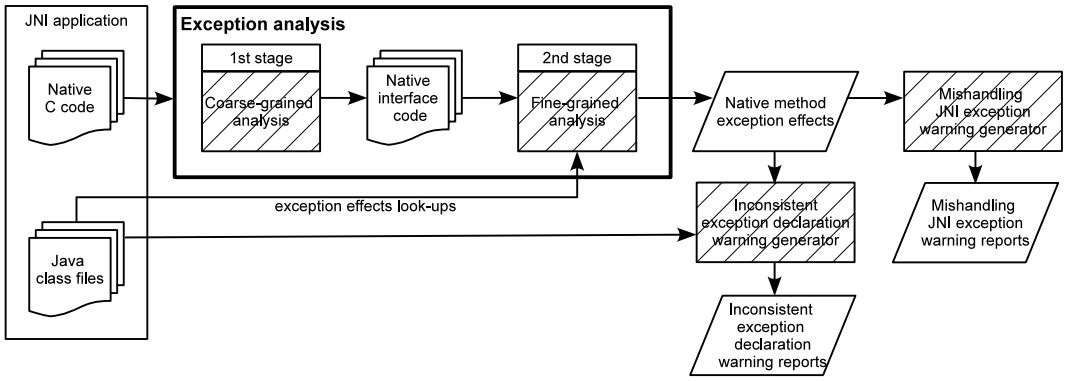


Figure 2.4: System architecture of TurboJet.

analysis component is discussed in Section 2.5.

The warning generator for inconsistent exception declarations is straightforward and directly use the information generated by exception analysis. The exception analysis determines possible pending JNI exceptions at every program point. The possible pending checked exceptions at the end of a native method is considered the actual exception effects of the native method. The actual exception effects are then compared with the declared exception effects extracted from Java class files. If there is a mismatch, the warning generator issues a warning. In particular, if a checked exception can possibly be pending at the end of a native method and if the class of the exception is not a subclass of one of the exception classes listed in the native method's signature, then a warning is issued.

The warning generator for mishandling JNI exceptions also uses the information from exception analysis and includes two additional components: (1) a static analysis for identifying unsafe operations; (2) a "warning recovery" mechanism that attempts to generate just one warning message for each distinct bug. Section 2.6 presents the technical details of the warning generator for mishandling JNI exceptions.

2.5 Scalable and precise exception analysis

The major stages in TurboJet's exception analysis is depicted in Fig. 2.4. As mentioned before, the goal of exception analysis is to find out the set of possible pending JNI exceptions at every program point of native methods. Its core is a two-stage design, motivated by the following two observations.

1. Library code in a JNI package does not affect Java's exception state. As discussed in the background section, native code in a JNI package consists of interface and library code. Intuitively, a general-purpose library such as the zlib C library works independently of Java. Only interface code, which bridges between Java and the library, inspects and changes Java's state via JNI functions. In particular, only interface code will throw, clear, or handle JNI exceptions.
2. Accurately computing exception effects of native methods requires fine-grained static analysis. The need for this will be explained in more detail. But at a high level, the analysis needs path sensitivity since JNI code frequently correlates its exception state with other parts of the execution state; it needs context sensitivity because JNI code often invokes utility functions that groups several JNI function calls.

TurboJet's exception analysis uses a two-stage design to take advantage of these observations. The first stage is a coarse-grained analysis that aims to separate interface code from library code automatically. It is conservative and quickly throws away code that is irrelevant for calculating exception effects. Afterward, the second stage can focus on the remaining smaller set of interface code. The second stage is much slower as it needs the full set of sensitivity (path and context sensitivity) for accurate analysis. This

stage takes into considerations of JNI function calls and performs exception-effect look-ups in Java class files since native code can call back Java methods, which may throw exceptions. The second stage is at the heart of TurboJet's exception analysis.

2.5.1 Separating interface and library code

The first stage of TurboJet's exception analysis is a crude, flow-insensitive analysis with the goal of separating interface and library code. This analysis is helped by JNI's design philosophy that interaction with the JVM in native code is through a set of well-defined JNI interface functions.[2] In particular, native code interacts with Java through the JNI functions accessible from a global JNI environment pointer passed from Java. This design makes identification of interface code straightforward.

Accordingly, a native function is defined as part of the interface code if

1. it invokes a JNI function through the environment pointer, or
2. it invokes another native function that is part of the interface code.

If a native function does not belong to the interface code (by the above definition), then its execution will not have any effect on the JVM's exception state. We have implemented a straightforward worklist algorithm that iterates through all functions and identifies invocations of JNI functions via the JNI environment pointer. The algorithm is used to calculate the set of interface code.

[2]There are exceptions; for instance, native code can have a direct native pointer to a Java primitive-type array and read/rewrite the array elements through the pointer. Nevertheless, JVM's exception state (the focus of this chapter) can be changed only by JNI functions for a well-defined native method.

We note that the above definition of interface code covers the case of calling back a Java method since a Java call back is achieved through invoking a JNI function such as `CallVoidMethod`. This implies that any native function that performs a Java call back is part of the interface code. Calling-back-into-Java is rare in some JNI packages, but can be common in others. For instance, the `sun.awt` JNI package of JDK6 has almost one hundred call-back functions.

Having the first-stage exception analysis helps TurboJet achieve high efficiency, as our experiments demonstrate (see Section 2.7). Briefly, without the first-stage analysis, TurboJet's analysis time would be increased by more than an order of magnitude on the set of JNI programs we studied. Another option that could avoid the first-stage analysis is to manually separate interface and library code. Manual separation is straightforward to perform in practice as a JNI application is usually organized in a way that its library code is in a separate directory. Nevertheless, it has two downsides. First, the manual process may make mistakes and code that does affect Java's state would be wrongly removed. Second, since manual classification most likely uses a file as the unit, it would include irrelevant functions into interface code when they are in files that mix relevant and irrelevant functions. Our first-stage analysis uses functions as the classification unit and provides automatic classification.

2.5.2 Fine-grained tracking of exception states

The second stage of exception analysis is a fine-grained analysis that is path-sensitive and context-sensitive. By applying the fine-grained analysis on the remaining interface code, TurboJet can precisely identify the exception effects at every program point.

(a)

```
int *p = (*env)->GetIntArrayElements(arr, NULL);
if (p == NULL) /*exception thrown*/
  return;
for (i=0; i<10; i++) sum += p[i];
```

(b)

```
void ThrowByName(JNIEnv *env, const char* cn){
  jclass cls = (*env)->FindClass(cn);
  if(cls != NULL){
    (*env)->ThrowNew(cls);
  }
}
```

Figure 2.5: Examples for illustrating the need for path and context sensitivity.

Before presenting our solution, we first discuss requirements of performing accurate tracking.

Requirements of TurboJet's exception analysis

The first requirement is that the analysis needs path sensitivity because JNI programs often exploit correlation between exception states and other execution states. For instance, many JNI functions return an error value and at the same time throw an exception to signal failures (similar situation occurs in the Python/C API). As a result of this correlation between the exception state and the return value, JNI programs can either invoke JNI functions such as `ExceptionOccured` *or* check the return value to decide on the exception state. Checking the return value is the preferred way as it is more efficient. Fig. 2.5(a) presents such an example involving `GetIntArrayElements`, which returns the null value and throws an exception when it fails.

Invoking `GetIntArrayElements` results in two possible cases: an exception is thrown and `p` equals `NULL`; no exception is thrown and `p` gets a non-null value. That is, the value of `p` is correlated with the exception state. To infer correctly that the state

before the loop is always a no-exception state, a static analysis has to be path sensitive, taking the branch condition into account.

The second requirement is that the analysis also needs context sensitivity because JNI programs often use utility functions to group several JNI function calls. JNI programming is tedious; a single operation on the Java side usually involves several steps in native code. For instance, the function in Fig. 2.5(b) uses a two-step process for throwing an exception: first obtaining a reference to the class of the exception and then throwing the exception using the reference. For convenience, JNI programmers often use these kinds of utility functions to simplify programming. What exception is pending after an invocation of the function in Fig. 2.5(b) depends on what the context passes in as the argument.

Clearly, it is not possible for TurboJet to infer in every case the exact exception effects. For instance, suppose a JNI program calls back a Java method and TurboJet cannot determine which Java method it is. In such cases, TurboJet has to be conservative and assumes any kind of exceptions can be thrown. In the analysis, this is encoded by specifying the exception state afterwards becomes `java.lang.Exception`, the root class of all checked exceptions.

2.5.3 An FSM specification of exception-state transitions

TurboJet uses an FSM (Finite State Machine) to track how exception states can be changed by JNI functions. For instance, an exception becomes pending after `Throw`; after `ExceptionClear`, the current pending exception is cleared. Fig. 2.6 presents an FSM specification. It is incomplete as only a few JNI functions are included for brevity.

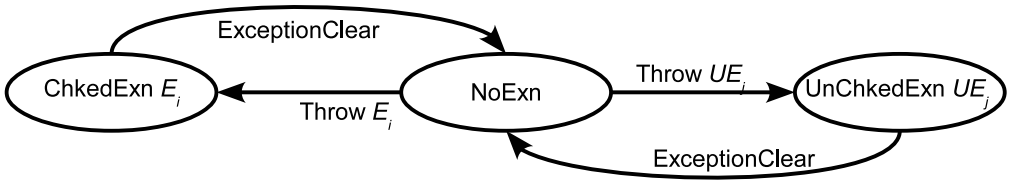


Figure 2.6: An incomplete FSM specification of exception-state transitions.

The following table summarizes the meaning of the states in the FSM:

`NoExn`	No JNI exception is pending
`ChkedExn` E_i	A checked exception E_i is pending
`UnChkedExn` UE_j	An unchecked exception UE_j is pending

The FSM has a state for each specific type of exceptions, including both checked and unchecked exceptions.

2.5.4 Path-sensitive analysis

Static-analysis algorithms that capture full path sensitivity (*e.g.*, [73]) are rather slow since they track all execution states. Fortunately, we are interested only in exception states and transitions between exception states are described by an FSM. TurboJet adopts ESP, proposed by Das *et al.* [17]. It is well-suited for capturing partial path sensitivity.

Given a safety property specified by an FSM, ESP symbolically evaluates the program being analyzed, tracks and updates *symbolic states*. A symbolic state consists of a *property state* and an *execution state*. A property state is a state in the FSM. An

execution state models the rest of the program's state and can be configured with different precision. The framework provides a conservative, scalable, and precise analysis to verify program safety properties. Readers may refer to the ESP paper for more detailed description. In the end, ESP infers information of the following format *at each control-flow edge*:

$$\{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

It contains n symbolic states and each symbolic state has its own property state (ps) and execution state (es). Intuitively, it means that there are n possible cases when the program's control is at the control-flow edge. In the i-th case, the property state is ps_i and the execution state is es_i.

In the context of exception analysis, TurboJet uses the FSM specified in Fig. 2.6. That is, a property state is an exception state. For the execution state, TurboJet tracks two kinds of information: (1) constant values of variables of simple types; and (2) Java-side information of variables that hold Java references. We next explain how these two kinds of information are tracked.

Tracking constant values. For variables of simple types, TurboJet tracks their constant values using an interprocedural and conditional constant propagation [88]. In particular, it tracks integer constants and string constants. String constants are tracked since JNI programs often use them for finding a class, finding a method ID, and for other operations.

Take the program in Fig. 2.5(a) as an example. After `GetIntArrayElements`,

$$
\begin{array}{rcl}
t & ::= & \texttt{jobj}^n(s_c) \mid \texttt{jcls}^n(s_c) \mid \texttt{jthw}^n(s_c) \\
 & \mid & \texttt{jmid}^n(s_c, s_m, s_t) \mid \texttt{jfid}^n(s_c, s_f, s_t) \\
s & ::= & \text{“}Str\text{”} \mid \top \\
n & ::= & 0 \mid 1 \mid *
\end{array}
$$

Figure 2.7: Syntax of Java types that TurboJet tracks.

there are two symbolic states encoded as follows.

$$
\begin{array}{l}
\{\langle \texttt{NoExn}, \{\, p = \top \,\}\rangle, \\
\ \langle \texttt{UnChkedExn OutOfMemoryError}, \{\, p = 0 \,\}\rangle\}
\end{array}
$$

There are two cases. In the first case, there are no exception pending and p is not a constant. In the second case, there is an unchecked exception `OutOfMemoryError` pending, and $p = 0$. This information correlates the exception state with the value of p and enables the analysis to take advantage of the following if-branch to infer that before the loop it must be the case that no exception is pending.

Following the standard constant propagation, we use $\bot$ for uninitialized variables and $\top$ for non-constants.

Tracking Java types of C variables. JNI programs hold references to Java-side objects and use references to manipulate the Java state. These references are of special types in the native side. For example, a reference of type `jobject` holds a reference to a Java object; a reference of type `jmethodID` holds a reference to a Java method ID.

To accurately track exception states, it is necessary to infer more Java-side information about these references. For instance, since the JNI function `Throw` takes a `jthrowable` reference, TurboJet has to know the class of the Java object to infer the

exact exception class it throws. TurboJet uses a simple type system to capture this kind of information. The type system is presented in Fig. 2.7. The following table summarizes the meaning of each kind of types.

$\texttt{jobj}^n(s_c)$	A reference to a Java object whose class is s_c
$\texttt{jcls}^n(s_c)$	A reference to a Java class object with the class being s_c
$\texttt{jthw}^n(s_c)$	A reference to a Java `Throwable` object whose class is s_c
$\texttt{jmid}^n(s_c, s_m, s_t)$	A reference to a Java method ID in class s_c with name s_m and type s_t
$\texttt{jfid}^n(s_c, s_f, s_t)$	A reference to a Java field ID in class s_c with name s_f and type s_t

Each s represents either a constant string, or an unknown value (represented by $\top$). When n is 0, it means the reference is a null value; when n is 1, it is non-null; when n is $*$, it can be either null or non-null.

As an example, the following syntax denotes a non-null Java method ID in class "Demo" with name "callback" and type "()V". The type means that the function takes zero arguments and returns the `void` type.

$$\texttt{jmid}^1(\text{“Demo”}, \text{“callback”}, \text{“()V”})$$

Fig. 2.8 presents a more elaborate example demonstrating how path sensitivity works in TurboJet. The JNI program invokes a callback method in class `Demo`. We assume

that the callback method throws a checked exception `IOException`, abbreviated as `IOExn` in the figure. Before invoking a Java method, there is a series of preparation steps in the program: (1) finding a reference to the class `Demo`; (2) allocating an object; (3) obtaining the method ID of the callback function. Since each step may throw an exception, proper checking of null return values after each call is inserted in the code.

For each control-flow edge, Fig. 2.8 contains annotation that specifies what TurboJet's exception analysis infers. Notice that after `FindClass`, there are two symbolic states representing two cases. The following if-statement checks the return value and as a result the number of symbolic states is reduced to one on both branches.

After `CallVoidMethod`, the only exception state is "`ChkedExn IOException`". TurboJet can infer this accurately because it knows exactly which Java method the code is calling, thanks to the information associated with `mid` before the call. Since TurboJet also takes Java class files as input, it uses the method ID to perform a method look-up in class files and extracts its type signature. The type signature tells what exceptions are declared in the method's `throws` clause. In this case, it is assumed to declare an `IOException`.

2.5.5 Context-sensitive analysis

ESP-style path-sensitivity works well on a single function. However, the context of its interprocedural version is based on the property state alone [17] and is not sufficient for JNI programs. The need for more fine-grained contexts is because of a common programming pattern in JNI programs: for convenience, JNI programs typically use a set of utility functions for accessing fields and throwing exceptions. These utility

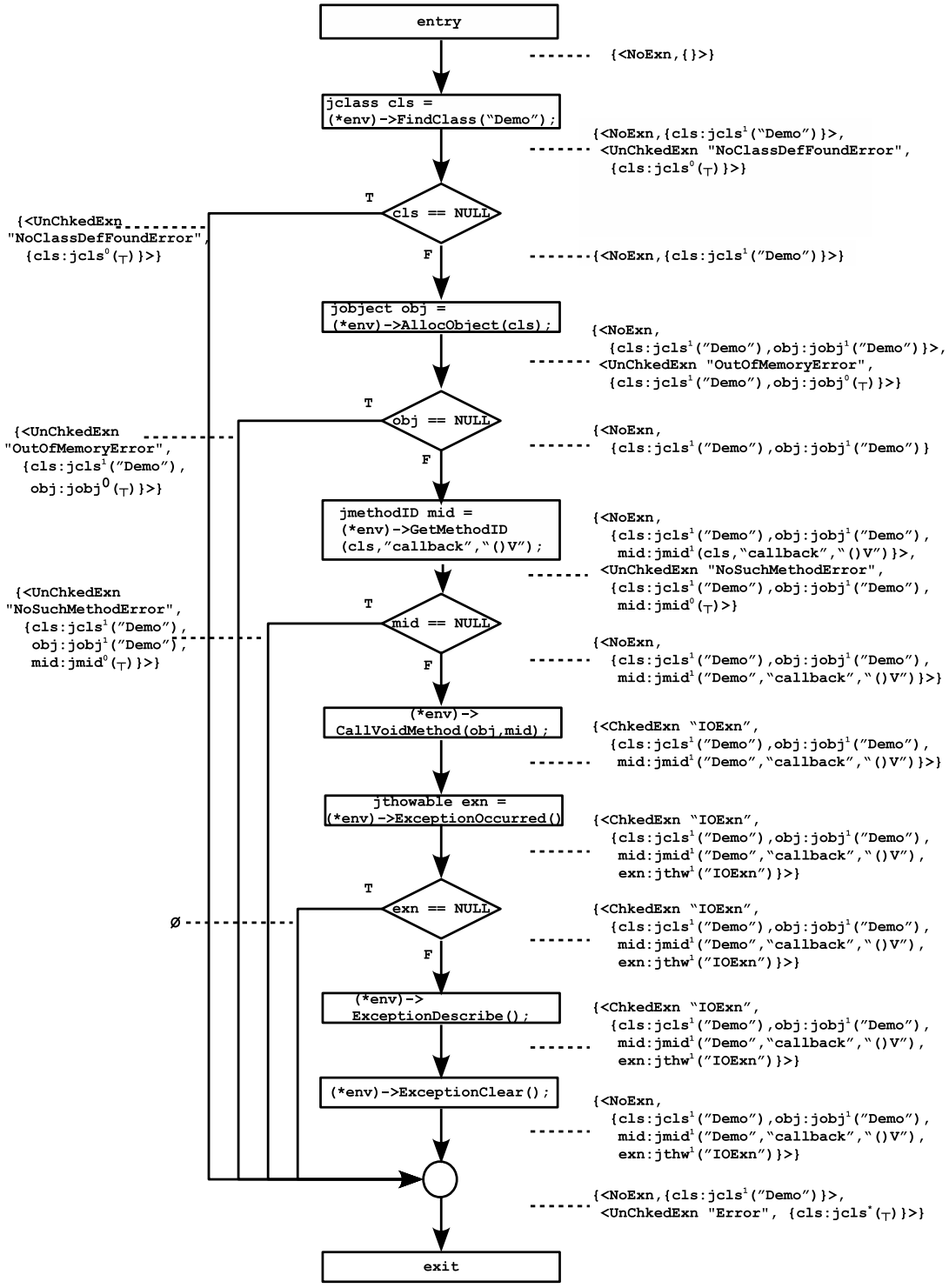


Figure 2.8: Example of TurboJet path sensitivity.

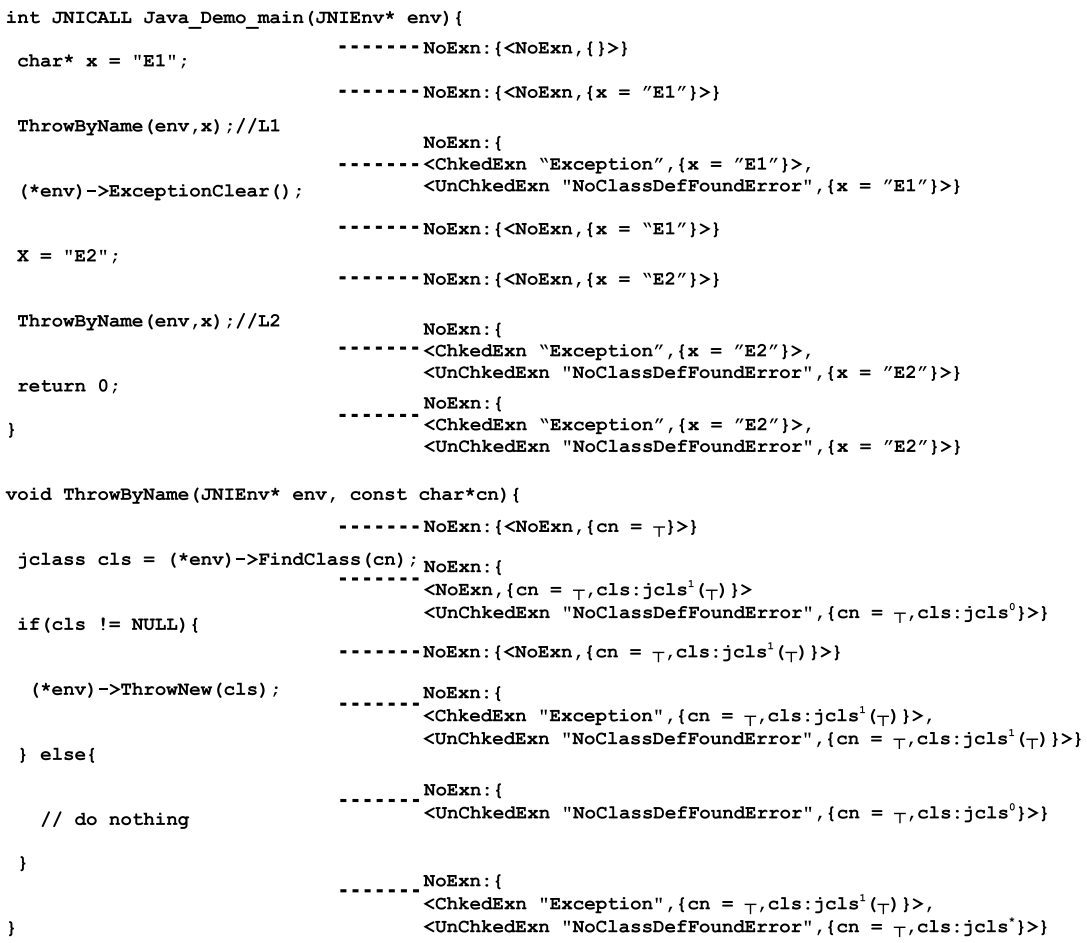


Figure 2.9: Exception analysis using ESP.

functions take names of fields, methods, classes, or exceptions as parameters and are called multiple times with different names. To accurately track Java types in native code, our system has to take the context into account.

Figs. 2.9 and 2.10 present a typical JNI program for demonstrating the need. In the example, `ThrowByName` is a utility function, which takes the name of an exception as a parameter and uses a sequence of JNI functions to throw the exception. The function is used by the `Java_Demo_main` function in two contexts, denoted by `L1` and `L2`. At

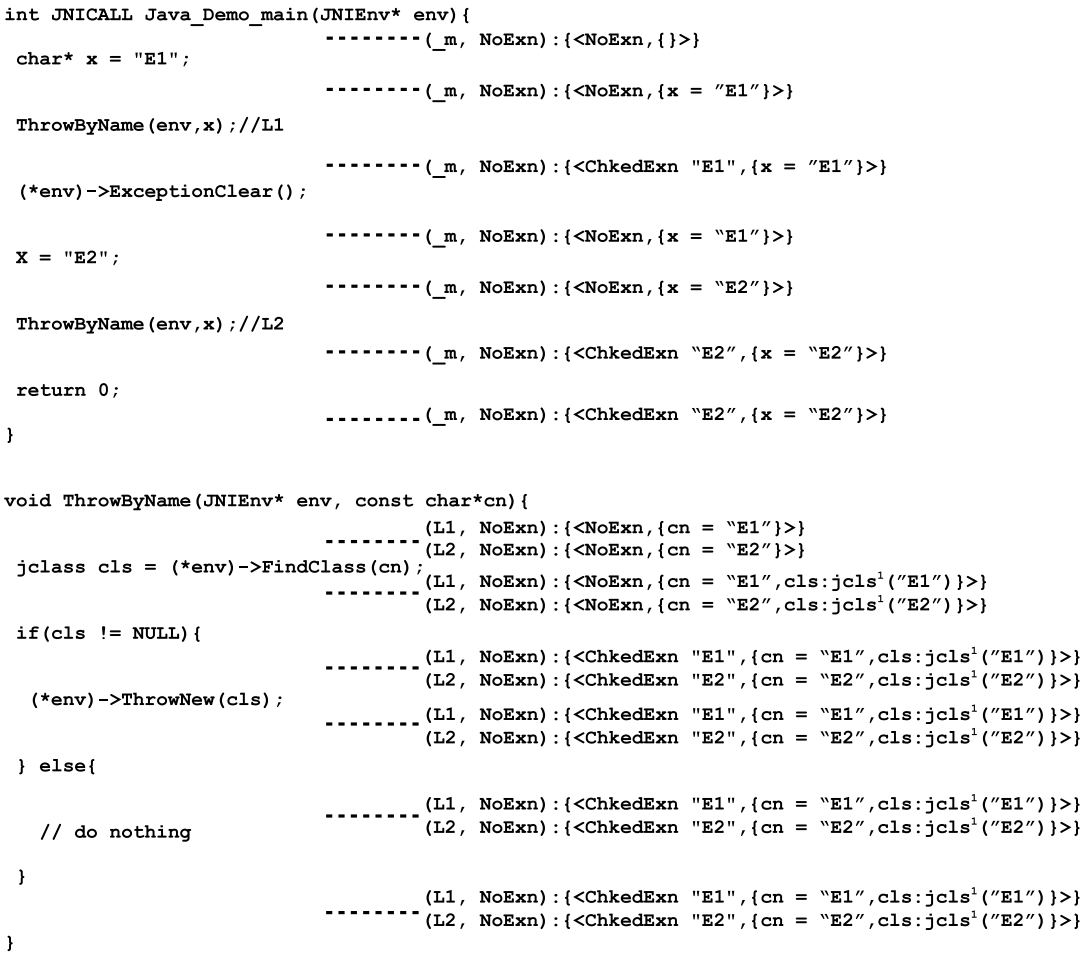


Figure 2.10: Exception analysis using TurboJet.

`L1`, exception `E1` is thrown and exceptions `E2` at `L2`.

ESP as shown in Fig. 2.9 merges the execution states of the two call sites when analyzing `ThrowByName` since their property states are the same; both are in the `NoExn` state. As a result, the value of parameter `cn` in `ThrowByName` gets $\top$. Consequently, the analysis determines that `ThrowByName` has the "`ChkedExn Exception`" effect. But in reality, `E1` can only be thrown at `L1`, and `E2` at `L2`. Some unchecked exception may be thrown as well. For brevity, the figure includes only

"`NoClassDefFoundError`".

TurboJet as shown in Fig. 2.10 improves the context by using the call-string approach (of length one) [78]. The context now becomes (n_c, ps), where n_c is the caller node and ps the property state. As a result, the format of information at a control-flow edge becomes

$$(n_c, ps) : \{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

It represents the set of symbolic states when the context is (n_c, ps).

With the added contextual information, TurboJet is able to infer the exception effects accurately for the program in Fig. 2.9. In the figure, we use "_m" to represent a special (unknown) context that invokes the `Java_Demo_main` function. Another note about the figure is that the "`NoClassDefFoundError`" unchecked exception disappears because for each context the analysis determines the exact name of the exception and our system is able to infer `FindClass` always succeeds for a specific name.

The overall interprocedural algorithm is given in the appendix, using notation similar to the ESP paper. The complexity of this algorithm is $Calls \times |D|^2 \times E \times V^2$, where $Calls$ is the number of call sites in the program, $|D|$ is the number of exception states, E is the number of control-flow edges in the control-flow graph, and V is the number of variables in the program. The time complexity is more than that of ESP, but still remains polynomial.

2.5.6 Transfer functions for JNI functions

Invoking a JNI function updates the symbolic state. The effects on the symbolic state are defined by a set of transfer functions. These transfer functions are defined according to their specification in the JNI manual [51].

2.5.7 Merging symbolic states

At a merge point of a control flow graph, TurboJet groups symbolic states in the following fashion. It merges two symbolic states if the exception state of the first is a subclass of the exception state of the second. Since the subclass relation is reflexive, TurboJet's merge function extends ESP's, which merges two symbolic states if their exception states are the same. Grouping symbolic states by using a parent class of the exceptions is conservative. This may introduce imprecision. The advantage is that by having less symbolic states, the analysis runs faster.

2.6 Finding bugs of mishandling JNI exceptions

In this section, we discuss how TurboJet utilizes the information generated by exception analysis to find bugs of mishandling JNI exceptions. By definition 2 (on page 28), two conditions should be met for such kind of bugs: (1) a JNI exception is pending; and (2) the next operation is an unsafe operation. The first condition is determined by the result from exception analysis.

Fig. 2.11 presents pseudo code that highlights the major steps of warning generation

Input: Program P
Output: A list of warning locations in P
Notation:

1. $op(l)$ stands for the operation at location l in P
2. $\mathrm{ss}_\circ(l)$ stands for the symbolic state at the entry of node for l

BEGIN: $\mathrm{ss}_\circ$ = exceptionAnalysis(P);
for each location l in P **do**
  if ($\langle ps, es \rangle \in \mathrm{ss}_\circ(l)$) and ($ps \neq$ `NoExn`) and (unSafe($op(l)$)) **then**
    Issue a warning for location l;
    P=warningRecovery(P);
    goto BEGIN
  end if
end for

Figure 2.11: High-level steps of warning generation for mishandling JNI exceptions.

for mishandling JNI exceptions. First, it performs exception analysis on an input program. We use notation $\mathrm{ss}_\circ$ to stand for the result of exception analysis. It is a function mapping locations of P to the symbolic states calculated by exception analysis. In particular, $\mathrm{ss}_\circ(l)$ is the symbolic state at the entry of location l. It is calculated as the join of symbolic states of all edges that flow into the node for l in the control-flow graph.

After exception analysis, a warning is issued for a location if the exception analysis indicates a possible pending exception *and* next to that location an unsafe operation is identified. Finally, after a warning is issued, it performs "warning recovery" which transforms the old program and re-computes exception analysis. We next discuss how TurboJet decides whether an operation is unsafe and how it performs warning recovery.

2.6.1 Determining unsafe operations

Our strategy for determining unsafe operations is an algorithm of whitelisting plus static taint analysis.

Whitelisting

The whitelist is comprised of those operations that are absolutely safe to use when an exception is pending. In general, after an exception is thrown, a JNI program either (1) cleans up resources and returns the control to Java, or (2) handles and clears the exception itself using JNI functions such as `ExceptionClear`. A lists the set of operations that are on the whitelist.

Static taint analysis

A pure whitelist strategy, however, would result in too many false positives (see the experiments section). As an example, the following code is considered safe, but a warning would be issued by the whitelisting strategy since "`a=a+1`" is not on the whitelist.

```
int *p = (*env)->GetIntArrayElements(arr,NULL);
if ((*env)->ExceptionOccurred()) {
        a=a+1; return a;}
```

We cannot put plus and assignment operations on the whitelist because that would allow statements like "`a=(*p)+1`" to escape detection.

Our idea is to use static taint analysis to decide the safety of operations that are not on the whitelist. In static taint analysis, a *taint source* specifies where taint is generated.

A *taint sink* specifies unsafe ways in which data may be used in the program. A *taint propagation* rule specifies how taint is propagated in the program.

Our system uses taint sources to model where faults may happen and use static analysis to track fault propagation. In general, a fault is an accidental condition that can cause a program to malfunction. One benefit of static taint analysis is that it can accommodate various sources of faults. For example, in an application that can receive network packets that are controllable by remote attackers, all network packets can be considered being tainted because their contents may be arbitrary values. In the JNI context, data passed from Java to C can be considered being tainted because attackers can write a Java program and affect those data, as the example in Fig. 2.2 shows. Our framework is flexible about how taint is generated and propagated and can thus accommodate various fault models.

In our implementation, our fault model is comprised of the following parts:

- Certain JNI functions may fail to return desired results and are sources of faults. For instance, `NewIntArray` may fail to allocate a Java integer array.
- Certain JNI functions may return direct pointers to Java arrays or strings that can be controlled by attackers. For example, `GetIntArrayElements` may return a direct pointer to a Java integer array when successful. Therefore, the fault model also considers the results of these functions as sources of faults.
- Some library function calls may fail. For example, `malloc` may fail to allocate a buffer. In general, for those external functions that may generate faults, our system requires manual annotation to specify they are fault sources.

All faults in our fault model are associated with pointers. Intuitively, a tainted pointer means that it points to data that may be affected by faults specified in the fault model. For example, the pointer result of `GetIntArrayElements` is tainted because its value may be null or a pointer to a Java buffer (controlled by attackers). Given this intuition, a taint sink (*i.e.*, unsafe ways of using taint) in our setting is an operation where a tainted pointer value is dereferenced for either memory reading or memory writing. Note that the operation of copying a tainted pointer variable to another pointer variable does not constitute a taint sink, as there are no dereferences. On the other hand, the second variable also becomes tainted because of the copy and a dereference of it would be unsafe.

An operation is unsafe if it is not on the whitelist and it may dereference tainted pointers. Now come back to the example earlier in this section. `p` is marked as being tainted as it is the result of `GetIntArrayElements`. As a result, the operation "`a=a+1`" is considered safe because it does not involve the use of any tainted data. On the other hand, "`a=(*p)+1`" would be unsafe. Note that finding an unsafe operation is not a sufficient condition for issuing a warning. By the algorithm in Fig. 2.11, a warning is issued only when performing an unsafe operation with a pending exception.

To track taint propagation statically, we have implemented an interprocedural, flow-insensitive algorithm. The flow-insensitivity makes our static taint analysis scalable. The algorithm consists of two steps:

1. A pointer graph is constructed for the whole program. For every pointer in the program, there is a node in the graph to represent the value of the pointer. The edges in the graph approximate how pointer values flow in the program. Our pointer graph is similar to CCured's pointer graph [66], except that our graph has

different kinds of edges, which will be discussed shortly.

2. Nodes in the graph that correspond to taint sources are marked as being tainted, and then our algorithm propagates taint along edges of the graph. After this process, any marked nodes are considered being tainted.

Next we discuss more details of our pointer graph. For every pointer in the program, we have a node in the graph to represent the value of the pointer. For example, if a program has a declaration

```
int *p;
```

then its pointer graph has a node for the address of `p`, or `&p`, and another node for `p`. This is because both `p` and `&p` are pointers. Similarly, if the program has

```
int **q;
```

then the graph has a node for the address of `q`, a node for `q`, and a node for `*q`; all three are pointer values.

There are two kinds of edges in the graph. The first kind is *flow-to* edges. A directed flow-to edge from node one to node two exists if the pointer value of node one may directly flow to the pointer value for node two. For example, an assignment from pointer one to pointer two would result in a flow-to edge.

The second kind of edges is *contains* edges. It allows our pointer graph to be sound in the presence of aliases. A contains edge exists from node one to node two if the storage place pointed to by the pointer for node one may contain the pointer for node two. For example, given the declaration "`int *p`", there is a contains edge from the

node for `&p` to the node for `p`. In this example, a contains edge is like a points-to edge in alias analysis. Contains edges allow us to handle C structs. A pointer to a C struct contains all the pointers inside the struct.

Fig. 2.12 presents an example program and its pointer graph. Dotted edges are contains edges; solid edges are flow-to edges. The node "`ret(GIAE)`" represents the pointer value returned by `GetIntArrayElements`. The flow-to edge from this node to the node for "`p`" exists because of the assignment on line 3. The flow-to edge from `&p` to `q` is because of the assignment on line 2.

Our algorithm also propagates flow-to edges along contains edges. This is because when a pointer value flows to another pointer value, the storage places pointed to by these two values may be aliases. As a result, implicit flows exist between the aliases. An example of the propagation of flow-to edges along contains edges in Fig. 2.12 is the propagation of the flow-to edge from `&p` to `q` along the contains edges that are from `&p` to `p` and from `q` to `*q`. As a result, the flow-to edges from `p` to `*q` and from `*q` to `p` are added.

Having constructed the pointer graph, it is easy to compute what pointer values in the example program may be tainted. First, our fault model specifies that the return value of `GetIntArrayElements` may be tainted. This taint is then propagated in the pointer graph along flow-to edges, which in turn taints the nodes for `p` and `*q`. Given this result, operations on line 6 and 7 are unsafe as they dereference tainted pointer "`*q`".

For external library functions without source code, the algorithm accepts hand annotations about taint propagation as their models. A library function without annotation is assumed to have the default taint propagation rule: the output is tainted if and only if any of its arguments is tainted. The default rule can be overwritten with hand annotations.

```
1  int *p;
2  int **q = &p;
3  p = (*env)->GetIntArrayElements(arr,NULL);
4  int len = (*env)->GetArrayLength(arr);
5  for (i=0; i<len; i++) {
6     if ((*q)[i]>0) {
7        sum += (*q)[i];}
8  }
```

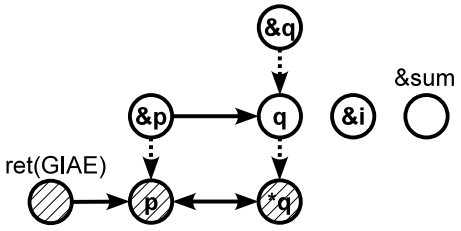


Figure 2.12: An example program and its pointer graph. The program takes a Java integer array and computes the sum of all positive elements. The nodes with shading are tainted nodes.

We use CIL's attribute language for such manual annotation. For instance, an annotation can say that the output is always tainted and independent of the arguments; in this case, the function is a taint source. A different annotation might say that whether the output is tainted depends on a specific argument.

Calling any external functions that take tainted pointers as parameters is considered unsafe—they might dereference those pointers. This is a source of inaccuracy in our analysis. We could enhance the precision by allowing hand annotation describing how parameters are used in external functions; this is left as future work. We treat JNI functions as external library functions. Therefore, if a JNI function is not on the white list, an invocation of the JNI function is considered unsafe if it takes tainted pointers as parameters.

Our algorithm for constructing pointer graphs handles most C features, including function calls and returns, structs, unions, and others. One limitation of the pointer-

graph construction at this point is it does not construct flow-to edges for inlined assemblies.

2.6.2 Warning recovery

The purpose of warning recovery is to suppress duplicate warnings. To illustrate its necessity, we use the example in Fig. 2.12. At both lines 6 and 7, an exception thrown by `GetIntArrayElements` may be pending and operations at these two places are unsafe since they dereference the tainted pointer "`*q`". A naïve system would issue warnings for both lines 6 and 7. However, both warnings are due to the error of mishandling JNI exceptions possibly thrown at line 3, and can be considered duplicates. Ideally, only one warning should be issued.

To suppress this kind of duplicates, we have implemented a warning-recovery strategy. First, our system remembers information about which exceptions can reach which location. For line 6 of the example problem, the exception raised at line 3 can reach line 6. Our system records location information by augmenting exception states. In particular, `ChkedExn` E becomes "`ChkedExn` E from $\{ l_1, ..., l_n \}$". If the exception state at a location l is "`ChkedExn` E from $\{ l_1, ..., l_n \}$", it means that exception E thrown at l_1 to l_n might reach l. We augment "`UnChkedExn` E" similarly.

Next, after a warning is issued, our system inserts an `ExceptionClear` statement immediately after all locations included in the abstract state. For example, after a warning for line 6 is issued, an `ExceptionClear` statement is inserted after line 3, whose exception reaches line 6.

After inserting the `ExceptionClear` statements, our system re-computes excep-

tion analysis and continues issuing warnings. Because of the inserted statements, duplicate warnings due to the same exception source are suppressed. This strategy allows us to suppress the warning at line 7.

Note that our system inserts `ExceptionClear` purely for warning recovery, not for transforming the program into a correct or semantically equivalent one.

2.7 Prototype implementations and evaluation

JNI Package	LOC	Inconsistent exception declarations		
	(K)	Warnings	True Bugs	FP(%)
java.io	2	3	1	67
java.lang.math	11	0	0	0
java.lang.non-math	3	1	1	0
java.net	10	4	0	100
java.nio	0.4	1	1	0
java.security	0.01	0	0	0
java.sql	0.02	0	0	0
java.util.timezone	0.7	0	0	0
java.util.zip	14	2	2	0
java.util.other[3]	0.02	0	0	0
java-gnome	6	5	3	40
jogl	0.6	2	2	0
libec	19	1	1	0
libreadline	2	2	2	0
posix	2	1	1	0
spread	28	3	3	0
TOTAL	99	25	17	32

Table 2.1: Accuracy evaluation of TurboJet on finding inconsistent exception declarations.

To evaluate TurboJet, we conducted experiments to evaluate its accuracy, effectiveness and efficiency. Our static analysis is implemented in the CIL framework [65], a tool set for analyzing and transforming C programs. Before our analysis is invoked, the CIL

JNI Package	LOC	Mishandling JNI exceptions		
	(K)	Warnings	True Bugs	FP(%)
java.io	2	0	0	0
java.lang.math	11	0	0	0
java.lang.non-math	3	8	5	38
java.net	10	88	60	32
java.nio	0.4	8	8	0
java.security	0.01	0	0	0
java.sql	0.02	0	0	0
java.util.timezone	0.7	0	0	0
java.util.zip	14	24	20	17
java.util.other	0.02	0	0	0
java-gnome	6	17	13	24
jogl	0.6	5	5	0
libec	19	0	0	0
libreadline	2	0	0	0
posix	2	40	36	10
spread	28	0	0	0
TOTAL	99	190	147	23

Table 2.2: Accuracy evaluation of TurboJet for finding inconsistent exception declarations.

front end converts C to the CIL intermediate language. The conversion compiles away many complexities of C, thus allowing our system to concentrate on a relatively clean subset of C. CIL's parser does not support C++ programs, so our analysis is limited to C programs only. There are also a few limitations in CIL's parser, and we had to tweak a few programs' syntax so they are acceptable to CIL's parser. Our system has a total about 6,600 lines of OCaml code, including 2,000 lines for constructing pointer graphs. In the exception analysis, we used a may-alias analysis module and a call-graph module included in the CIL. Our system also used JavaLib [35] to access information from Java class files. All experiments were carried out on a Linux Ubuntu 9.10 box with Intel Core2 Duo CPU at 3.16GHz and with 512MB memory.

We compiled a set of JNI packages for experimentation. The packages and their

numbers of lines of C source code are listed in Tables 2.1 and 2.2. Statistics for Java class files are not included because TurboJet needs only Java type signatures for its analysis. The packages with names starting with `java` are extracted from Sun's JDK6U03 (release 6 update 3). These packages cover all native code in Sun's JDK under the `share` and `solaris` directories. Other packages are also selected as they are well-known Java applications that consist of JNI code and native components. Brief descriptions of these benchmark programs are shown in Table 1.1.

The experiments were designed to answer the following set of questions:

1. How accurate is TurboJet at uncovering errors of inconsistent exception declarations and mishandling JNI exceptions? Does it generate too many false positives along the process?
2. How efficient is TurboJet in terms of analysis time? Does it scale to large programs?
3. Is the two-stage exception analysis necessary?

2.7.1 Accuracy

Tables 2.1 and 2.2 present the number of warnings and true bugs of both types of bugs in the set of benchmark programs.

Results on inconsistent exception declarations

Conceptually, there are two categories of bugs. The first category contains those bugs when a native method does not declare any exceptions but its implementation can ac-

tually throw some exceptions. The second category contains those bugs when a native method declares some exceptions but the implementation can actually throw a checked exception that is not a subclass of any of the declared exceptions. All the bugs we identified in the benchmark programs belong to the first category. For example, in the `libec` package, the implementation of native method `generateECKeyPair` throws `java.security.KeyException`. However, the method's Java signature does not declare any exception. A similar bug is found in the `java.nio` package of Sun's JDK, where native method `force0` throws `java.io.IOException` but the method's Java-side signature does not declare any exception. We manually checked against JDK6U23, a new version of the JDK, for the bugs identified in JDK6U03 and found that one bug in `ZipFile` has been fixed in JDK6U23. But other bugs in the JDK remain.

False positives. There are 8 false positives. All false positives are caused by the imprecision when tracking the execution state. Fig. 2.13 presents a typical example. A string constant that represents the name of a Java exception class is stored in a C struct and used later for throwing an exception. Since TurboJet tracks only constant values of C variables of simple types, when it encounters the `ThrowByName`, it cannot tell the exact exception class. Consequently, it will report the method that contains the code can possibly throw `java.lang.Exception`, the top checked-exception class. However, the native method's Java-side signature declares a more precise exception class, `java.io.IOException`. We could further improve TurboJet's static analysis to reduce these false positives. But since the total number of false positives is quite small, we did not feel it is necessary.

```
...
data->name = "java.io.IOException";
...
if(data->name != NULL){
   ThrowByName(env,data->name);
}
...
```

Figure 2.13: A typical example of false positives.

Results on mishandling JNI exceptions

The warnings of mishandling JNI exceptions in the table are the results of applying strategies of whitelisting, static taint analysis and warning recovery. The overall false positive rate among all benchmark JNI programs is 23%. Our system achieves a relatively high precision. Of the 147 true bugs, 129 of them are because of implicit throws, while the rest are because of explicit throws. That is, the majority of the errors are because programmers forgot to check for exceptions after calling JNI functions. This result is consistent with our expectation.

False positives. False positives are mainly of two kinds. The first kind includes those places where external library functions are invoked with tainted pointer parameters. For soundness, our system issued warnings for them because they might dereference the tainted pointers. For future work, this kind of false positives could be removed by either including the source code of the library functions, or by adding additional manual annotation about how parameters are used. The second kind of false positives are because of flow insensitivity of our static taint analysis. Our design favored scalability and we believe the overall FP rate supports this tradeoff.

To assess the effectiveness of warning recovery and static taint analysis, we have carried out two additional sets of experiments. First, we tested a version of the sys-

JNI packages	V1	FP(%)	V2	FP(%)
java.io	0	0	0	0
java.lang.math	0	0	0	0
java.lang.non-math	17	71	38	87
java.net	132	55	318	81
java.nio	16	50	30	73
java.security	0	0	0	0
java.sql	0	0	0	0
java.util.timezone	0	0	0	0
java.util.zip	35	43	48	58
java.util.other	0	0	0	0
java-gnome	24	46	83	84
jogl	5	0	5	0
libec	0	0	0	0
libreadline	0	0	0	0
posix	66	46	108	67
spread	0	0	0	0
TOTAL	295	50	630	77

Table 2.3: Experimental results for assessing effectiveness of warning recovery and static taint analysis

tem without warning recovery; we denote this version by V1. We also tested a version with neither warning recovery nor static taint analysis; we denote this version by V2. Table 2.3 presents the results. Without warning recovery, the overall FP rate would be 50%, as compared to 23%. Further removing the component of static taint analysis would hike the FP rate to 77%. These experiments demonstrated that warning recovery and static taint analysis are very effective in terms of reducing the number of false positives.

False negatives

TurboJet is designed to be conservative, though it is possible for TurboJet to have false negatives due to errors in its implementation. So we first conducted manual audit in half

of the packages of our benchmark programs to obtain the ground truth, and compared it with the results reported by TurboJet. In this way, we found and fixed several bugs in early implementations of TurboJet. Clearly, this is not a proof that TurboJet is sound. For the claim of soundness, we would need to (1) take a formalized semantics of C and the JNI, (2) formalize TurboJet, and (3) show that it can catch all relevant bugs. Although there has been efforts on formalizing C [69, 46] and the JNI [84], they target only subsets of C and the JNI and omit many important language features. Therefore, a formal proof of soundness remains a difficult task. On the other hand, there are many places where the system would be obviously unsound if we did not make a specific design choice. One example we have discussed before is issuing warnings when an external function takes tainted pointers as parameters. Without doing that, the system would be unsound as the external function might dereference the pointers.

2.7.2 Efficiency

Table 2.4 presents the analysis time of the TurboJet system (for detecting both types of bugs) on the benchmark programs. As we can see, TurboJet's coarse-grained exception analysis (first stage) is very efficient, taking only 420 μs for all packages. We group the time for the second-stage exception analysis and the time for warning generation into the column named "2nd stage time". It dominates the total time taken by TurboJet. In all, it takes about 14 s for all packages.

Effectiveness of the two-stage system. Table 2.4 also presents statistics that are used to evaluate the effectiveness of the two-stage design of TurboJet's exception analysis. For each package, the third column (LOC (K) retained/reduced) shows the lines of source

JNI Package	*1st. time*	*r&r*	*2nd. time*	*tt. time*	*i&l man.*	*w/o 1st.*
java.io	30	0.7/1	3.07	3.07	2/0	26.7
java.lang.math	30	0/11	0	30μs	2/9	2.09
java.lang.non-math	30	0.5/3	0.1	0.1	3/0	37.11
java.net	100	0.5/9	3.43	3.43	10/0	101.13
java.nio	10	0.04/0.3	0.50	0.50	0.4/0	0.31
java.security	0	0/0.08	0	0	0.08/0	0.05
java.sql	0	0/0.02	0	0	0.02/0	0.00171
java.util.timezone	0	0/0.7	0	0	0.7/0	2.12
java.util.zip	20	0.4/14	1.13	1.13	0.8/13	72.88
java.util.other	0	0/0.02	1.00	1.00	0.02/0	0.10
java-gnome	15	0.3/5	1.71	1.71	0.6/5	45.27
jogl	5	0.3/0.3	1.09	1.09	0.3/0.3	17.37
libec	20	0.1/19	0.61	0.61	0.4/19	70.21
libreadline	20	0.1/2	0.05	0.05	0.7/1	19.31
posix	20	0.3/2	0.08	0.08	2/0	11.28
spread	60	0.3/27	0.51	0.51	2/25	12.97
TOTAL	360	4/94	13.28	13.28	25/72	418.9

Note:
1st stage time in μs (*1st. time*)
LOC (K) retained/reduced (*r&r*)
2nd stage time in s (*2nd. time*)
Total time in s (*tt. time*)
Interface/library LOC (K) by manual separation (*i&l man.*)
Time in s without 1st stage (*w/o 1st.*).

Table 2.4: Efficiency evaluation of TurboJet.

code retained and reduced by the coarse-grained analysis. The sum of the two numbers is the total lines of source code of the package. The numbers show that it is very effective in terms of separating the code that affects Java's state. It reduces the amount of code necessary to be analyzed by almost 97%. Only a small portion of code is left for the fine-grained analysis.

The column "interface/library LOC (K) by manual separation" shows the lines of interface code and library code determined by a manual separation. For each package, the sum of the two numbers determined by the manual separation is the size of the package

and is therefore the same as the sum in the third column (LOC (K) retained/reduced). The manual separation is a quick, rough classification using files as the unit. That is, if a file contains some JNI function calls, we put it under the category of interface code. When compared to the column of "LOC (K) retained/reduced" by the first stage, we see our automatic algorithm is better at separating interface code and library code; the main reason is that it uses functions as the classification unit.

Finally, the column "Time without 1st stage" presents the analysis time if the whole package is fed directly into the rest of TurboJet without going through the first stage. This resulted in a dramatic slowdown in runtime.

The experiments demonstrate that our system is efficient and scalable. For nearly 100K lines of code, it took about 14 s to examine all of them.

2.7.3 Comparison with previous studies

To further validate the design of TurboJet, we compared our work with our two previous studies that used alternative design approaches for exception analysis.

Comparison with an alternative exception analysis

In our previous work on finding bugs of mishandling JNI exceptions [49], we implemented an interprocedural exception analysis. It calculates whether an exception is pending at every program point, but does not determine the set of possible pending exceptions. Since its goal is simpler, it uses a specially designed lattice to get a primitive form of path sensitivity. Furthermore, it is not context sensitive. TurboJet's exception

analysis was started with that algorithm but we quickly discovered its imprecision led to too many false positives when calculating exception effects. The previous system has an acceptable false-positive rate only because its exception analysis is combined with other kinds of analysis (in particular, static taint analysis); this strategy worked well when identifying bugs because of mishandling JNI exceptions but would lead to high false-positive rates when calculating exception effects. As a result, we gradually moved to a path-sensitive and context-sensitive system. The need for context sensitivity was also pointed out by J-Saffire [26], which is why they used polymorphic type inference rather than a monomorphic version.

Table 2.5 presents comparison results between TurboJet and a version of TurboJet that uses the crude exception analysis as in [49]. `CrudeExn` stands for the version with the crude exception analysis. The table presents false-positive rates and running times of the two versions for finding inconsistent exception declarations.

It is clear from the results that TurboJet's exception analysis is more accurate. The improvement in terms of reducing false positive rate is more than 50%. Since `CrudeExn` only determines whether or not there is an exception pending from the native side, a warning is issued if a native method's type signature declares an exception class that is a strict subclass of `java.lang.Exception` and `CrudeExn` determines that an exception is pending from the native side. As a result, it causes higher false-positive rates.

Comparison with an empirical study

A previous empirical study on Sun's JDK 1.6 found a number of errors of mishandling JNI exceptions [86]. The study used grep-based scripts to examine 38,000

JNI package	FP (%)		Time	
	CrudeExn	*TurboJet*	*CrudeExn*	*TurboJet*
java.io	80	67	1.25	3.07
java.lang.math	0	0	5μ	30μ
java.lang.non-math	50	0	0.05	0.1
java.net	100	100	1.25	3.43
java.nio	75	0	0.14	0.50
java.security	0	0	0	0
java.sql	0	0	0	0
java.util.timezone	0	0	0	0
java.util.zip	60	0	0.68	1.13
java.util.other	0	0	0.81	1.00
java-gnome	50	40	0.87	1.71
jogl	75	0	0.45	1.09
libec	67	0	0.44	0.61
libreadline	50	0	0.01	0.05
posix	83	0	0.01	0.08
spread	50	0	0.22	0.51
TOTAL	67	32	6.18	13.28

Table 2.5: Comparing TurboJet with an alternative exception analysis on finding inconsistent exception declarations.

lines of C code and found 35 counts of mishandling JNI exceptions. Table 2.6 shows the comparison between the results of TurboJet and the empirical study on the set of JDK directories that are common in both studies. TurboJet uncovered all of the errors that were found in the previous study; it also discovered more errors. We like to note, however, that in deciding whether a warning is a true bug, we exercised a more conservative approach and adhered to the JNI's specification more closely than the one taken in the previous study. For example, TurboJet assumes JNI function `GetFieldID` can throw `NoSuchFieldError`, `ExceptionInInitializerError`, and `OutOfMemoryError` exceptions, and issues a warning when insufficient error checking occurs. The previous study took a more liberal measure on such cases; for example, it ignored the possibility of `OutOfMemoryError` exception in the case of `GetFieldID`. This discrepancy con-

tributes to the majority of the difference in errors shown in the table (roughly about two thirds). Another benefit of our tool is its much lower false-positive rate, which means much less manual work to sift through warnings for identifying true bugs.

Results	The previous study	TurboJet
Warnings	556	132
True bugs	35	93
FP %	93.7	29.5

Table 2.6: Comparison with the previous study [86] (of the 35 errors in the previous study, 11 are due to explicit throws and 24 due to implicit throws).

2.8 An Eclipse plug-in tool

We implemented TurboJet as an Eclipse plug-in tool. This tool identifies both types of bugs described in this chapter, and provides a developer-friendly graphic user interface. The plug-in is built as an additional checker that extends Codan [16], an open source project that performs a variety of static code analyses and checking on C/C++ code in Eclipse.

The TurboJet plug-in has a user-interface frontend and a code-analysis backend. For the frontend, the plug-in leverages Codan to extract information from source code, to configure settings (including the path to Java class files and which types of bugs should be identified), and to perform code navigation. The plug-in passes information on both the Java class files and C source code to the backend, which invokes the exception analysis system.

Like any other checker in Codan, the TurboJet plug-in can be easily configured inside Eclipse. A JNI developer may disable the checker, configure bug levels (*e.g.*, warn-

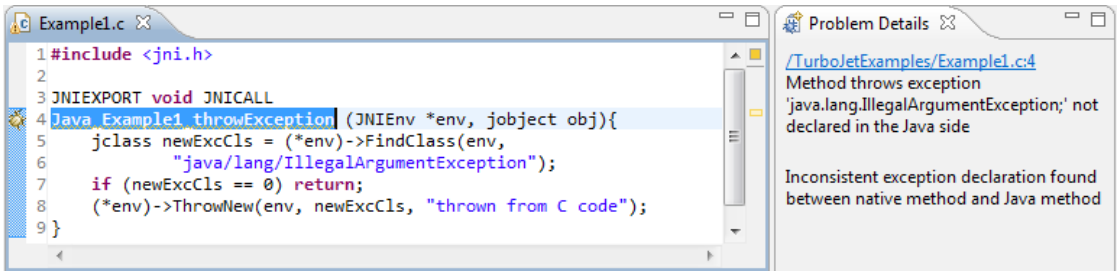


Figure 2.14: An example of TurboJet plug-in's warning on inconsistent exception declarations.

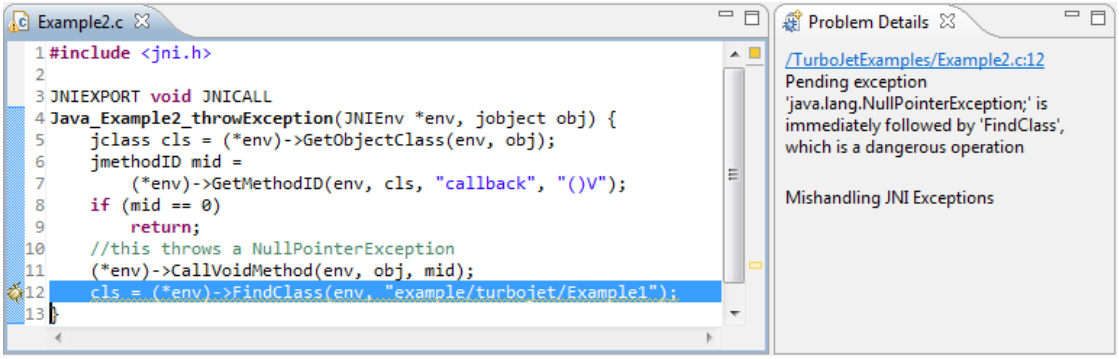


Figure 2.15: An example of TurboJet plug-in's warning on mishandling JNI exceptions.

ings or errors), and select which particular type of bug (*i.e.*, mishandling JNI exceptions or inconsistent exception declarations) to check.

Fig. 2.14 and Fig. 2.15 show examples of the TurboJet plug-in's warnings on inconsistent exception declarations and mishandling JNI exceptions, respectively. The problematic code is highlighted with warnings. Developers are provided with warning messages, and can navigate to the buggy code by double-clicking on the messages.

The TurboJet Eclipse plug-in is available for download [3].

2.9 Summary

Exceptions are commonly used in FFIs as ways for the foreign language to report error conditions to the host language. FFI programmers often make mistakes related to exceptions since FFIs do not provide support for exception checking and exception handling.

We have designed and implemented a novel static analysis framework for finding bugs in exceptional situations in JNI programs. TurboJet is a system that statically analyze native code (1) to extend Java's rules for checked exceptions to native code and (2) to identify errors of mishandling JNI exceptions. It is both scalable and has high precision thanks to its carefully engineered trade-offs. Our experimental results demonstrated the effectiveness of our techniques. We have also built a practical Eclipse plug-in that can be used by programmers to catch errors in their JNI code.

CHAPTER 3

NATIVE CODE ATOMICITY FOR JAVA

3.1 Introduction

Atomicity in programming languages is a fundamental concurrency property: a program fragment is *atomic* if its execution sequence—regardless of how the latter interleaves with other concurrent execution sequences at run time—exhibits the same "serial" behavior (*i.e.*, as if no interleaving happened). Atomicity significantly simplifies the reasoning about concurrent programs because invariants held by the atomic region in a serial execution naturally holds for a concurrent execution. Thanks to the proliferation of multi-core and many-core architectures, there is a resurgence of interest in atomicity, with active research including type systems and program analyses for atomicity enforcement and violation identification (*e.g.*, [24, 58, 30, 20]), efficient implementation techniques (*e.g.*, [29]) and alternative programming models (*e.g.*, [4, 53, 10]).

As we adopt these research ideas to serious production settings, one major hurdle to cross is to support atomicity across foreign function interfaces (FFIs). Almost all languages support an FFI for interoperating with modules in low-level languages (*e.g.*, [51, 72, 47]). For instance, numerous classes in `java.lang.*` and `java.io.*` packages in the Java Development Kit (JDK) use the Java Native Interface (JNI), the FFI for Java. Existing atomicity solutions rarely provide direct support for FFIs. More commonly, code accessed through FFIs—called *native code* in JNI—is treated as a "black box."

The "black box" assumption typically yields two implementations, either leading to

severe performance penalty or unsoundness. In the first implementation, the behavior of the native code is over-approximated as "anything can happen," *i.e.*, any memory area may be accessed by the native code. In that scenario, a "stop-the-world" strategy is usually required to guarantee soundness when native code is being executed—all other threads must be blocked. In the second implementation, the runtime behavior of native code is ignored, an unsound under-approximation. Atomicity violations may occur when native code happens to access the same memory area that it interleaves with. Such systems, no matter how sophisticated their support for atomicity for non-native code, technically conform to weak atomicity [57] at best. The lack of atomicity support for native code further complicates the design of new parallel/concurrent programming models. For example, several recent languages [4, 41, 10] are designed to make programs atomic by default, promoting the robustness of multi-core software. Native code poses difficulties for these languages: the lack of atomicity support for it is often cited [4] as a key reason for these languages to design "opt-out" constructs from their otherwise elegant implicit atomicity models.

We present JATO for atomicity enforcement across the JNI. It is standard knowledge that atomicity enforcement requires a precise accounting of the relationship between threads and their accessed memory. JATO is built upon the simple observation that despite rather different syntax and semantics between Java and native code, the memory access of both languages can be statically abstracted in a uniform manner. JATO first performs a static analysis to abstract memory access from both non-native code and native code, and then uses a lock-based implementation to guarantee atomicity, judiciously adding protection locks to selected memory locations. With the ability to treat code on both sides of the JNI as "white boxes" and perform precise analysis over them, our solution is not only sound, but also practical in terms of performance as demonstrated by

a prototype implementation. This system makes the following contributions:

- We propose a novel static analysis to precisely identify the set of Java objects whose protection is necessary for atomicity enforcement. The analysis is constructed as a constraint-based inference, which uniformly extracts memory-access constraints from JNI programs.
- We report a prototype implementation, demonstrating the effectiveness and the performance impact on both micro-benchmarks and real-world applications.
- We propose a number of optimizations to further soundly improve the performance, such as no locks on read-only objects.

3.2 Background and assumptions

The JNI allows Java programs to interface with low-level native code written in C, C++, or assembly languages. It allows Java code to invoke and to be invoked by native methods. A native method is declared in a Java class by adding the **native** modifier to a method. For example, the following `Node` class declares a native method named `add`:

```
class Node {int i=10; native void add (Node n);}
```

Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. Note that the Java side may have multiple Java threads running, each of which may invoke some native method.

The implementation of a native method receives a set of Java-object references from the Java side; for instance, the above `add` method receives a reference to `this` object

and a reference to the `n` object. A native-method implementation can interact with Java through a set of JNI interface functions (called JNI functions hereafter) as well as using features provided by the native language. Through JNI functions, native methods can inspect/modify/create Java objects, invoke Java methods, and so on. As an example, it can invoke `MonitorEnter` to lock a Java object and `MonitorExit` to unlock a Java object.

Assumptions. In any language that supports atomicity, it is necessary to define the *atomic region*, a demarcation of the program to indicate where an atomic execution starts and where it ends. One approach is to introduce some special syntax and ask programmers to mark atomic regions – such as atomic blocks. JATO's assumption is that each native method forms an atomic region. This allows us to analyze unannotated JNI code directly. Furthermore, we believe that this assumption matches Java programmers' intuition nicely. Java programmers often view native methods as black boxes, avoiding the reasoning about interleaving between Java code and native code. Finally, the assumption does not affect expressiveness. For instance, an atomic region with two native method invocations can be encoded as creating a third native method whose body contains the two invocations. If there is Java code fragment in between the two invocations, the encoded version can model the Java code by inserting a Java callback between the two invocations. Overall, the core algorithm we propose stays the same regardless of the demarcation strategy of atomic regions.

When enforcing native-method atomicity, JATO focuses on those Java objects that cross the Java-native boundary. It ignores the memory regions owned by native methods. For instance, native code might have a global pointer to a memory buffer in the native heap and lack of protection of the buffer might cause atomicity violations. Enforcing

this form of atomicity can be performed on the native-side alone (*e.g.*, [8]). Furthermore, native code cannot pass pointers that point to C buffers across the boundary because Java code does not understand C's type system; native code has to invoke JNI functions to create Java objects and pass references to those Java objects across the boundary. Because of these reasons, JATO focuses on language-interoperation issues and analyzes those cross-boundary Java objects.

3.3 The formal model

In this section, we use an idealized JNI language to describe the core of JATO: a constraint-based lock inference algorithm for ensuring the atomicity of native methods.

3.3.1 Abstract syntax

The following BNF presents the abstract syntax of an idealized JNI language where notation $\overline{X}$ represents a sequence of X's. Its Java subset is similar to Featherweight Java (FJ) [33], but with explicit support for field update and let bindings. For simplicity, the language omits features such as type casting, constructors, field initializers, multi-argument methods on the Java side, and heap management on the native side.

$$
\begin{array}{rcll}
P & ::= & \overline{\textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}\ \{F\ M\ N\}} & \textit{classes} \\
F & ::= & \overline{\mathsf{c}\ \mathsf{f}} & \textit{fields} \\
M & ::= & \overline{\mathsf{c}\ \mathsf{m}(\mathsf{c}\ x)\{e\}} & \textit{Java methods} \\
N & ::= & \overline{\textbf{native}\ \mathsf{c}\ \mathsf{m}(\mathsf{c}\ x)\{t\}} & \textit{native methods} \\
e & ::= & x \mid \textbf{null} \mid e.\mathsf{f} \mid e.\mathsf{f} := e \mid e.\mathsf{m}(e) \mid \textbf{new}_{\ell}\ \mathsf{c} \mid \textbf{let}\ x = e\ \textbf{in}\ e & \textit{Java terms} \\
t & ::= & x \mid \textbf{null} \mid \texttt{GetField}(t, fd) \mid \texttt{SetField}(t, fd, t) & \textit{native terms} \\
 & & \mid \texttt{NewObject}_{\ell}(\mathsf{c}) \mid \texttt{CallMethod}(t, md, t) \mid \textbf{let}\ x = t\ \textbf{in}\ t & \\
\textit{bd} & ::= & e \mid t & \textit{method body} \\
fd & ::= & \langle \mathsf{c}, \mathsf{f} \rangle & \textit{field ID} \\
md & ::= & \langle \mathsf{c}, \mathsf{m} \rangle & \textit{method ID}
\end{array}
$$

A program is composed of a sequence of classes, each of which in turn is composed of a sequence of fields F, a sequence of Java methods M, and a sequence of native methods N. In this JNI language, both Java and native code are within the definition of classes; real JNI programs have separate files for native code. As a convention, metavariable $\mathsf{c}(\in \mathbb{CN})$ is used for class names, f for field names, m for method names, and x for variable names. The root class is `Object`. We use e for a Java term, and t for a native term. A native method uses a set of JNI functions for accessing Java objects. `GetField` and `SetField` access a field via a field ID, and `CallMethod` invokes a method defined on a Java object, which could either be implemented in Java or in native code. Both the Java-side instantiation expression (**new**) and the native-side counterpart (`NewObject`) are annotated with labels $\ell(\in \mathbb{LAB})$ and we require distinctness of all ℓ's in the code. We use notation $\mathcal{L}_P : \mathbb{LAB} \mapsto \mathbb{CN}$ to represent the mapping function from labels to the names of the instantiated classes as exhibited in program P. We use

```
class Node extends Object {
  int i=10;
  native void add (Node n) {
    x1=GetField(this,<Node,i>);
    x2=GetField(n,<Node,i>);
    SetField(this,<Node,i>,x1+x2);}}
class Thread2 extends Thread {
  Node n1, n2;
  Thread2(Node n1, Node n2) {
    this.n1=n1; this.n2=n2;}
  void run() {n2.add(n1);}}
```

```
class Main extends Object {
  void main() {
    n1=new Node_{ℓ1}();
    n2=new Node_{ℓ2}();
    th=new Thread2_{ℓth}(n1,n2);
    th.start();
    n1.add(n2);
  }
}
```

Figure 3.1: A running example

$mbody(\mathsf{m}, \mathsf{c})$ to compute the method body of m of class c, represented as $x.bd$ where x is the parameter and *bd* is the definition of the method body. The definition of this function is identical to FJ's namesake function when m is a Java method. When m is a native method, the only difference is that the method should be looked up in N instead of M. We omit this lengthy definition in this short presentation.

Throughout this section, we will use a toy example to illustrate ideas, presented in Fig. 3.1. We liberally use void and primitive types, constructors, and use "$x = e_1; e_2$" for **let** $x = e_1$ **in** e_2. Note that the `Node` class contains a native method for adding integers of two `Node` objects and updating the receiver object. The goal in our context is to insert appropriate locks to ensure the execution of this native method being atomic.

3.3.2 Constraint generation: an overview

Atomicity enforcement relies on a precise accounting of memory access, which in JATO is abstracted as constraints. Constraints are generated through a type inference algorithm, defined in two steps: (1) constraints are generated intraprocedurally, both for Java methods and native methods; (2) all constraints are combined together through a

closure process, analogous to interprocedural type propagation. The two-step approach is not surprising for object-oriented type inference, because dynamic dispatch approximation and concrete class analysis are long known to be intertwined in the presence of interprocedural analysis [70]: approximating dynamic dispatch – *i.e.*, determine which methods would be used to enable interprocedural analysis – requires the knowledge of the *concrete classes* (*i.e.*, the class of the runtime object) of the receiver, but interprocedural analysis is usually required to compute the concrete classes of the receiver object. JATO performs step (1) to intraprocedurally generate constraints useful for dynamic dispatch approximation and concrete class analysis, and then relies on step (2) to perform the two tasks based on the constraints. The details of the two steps are described in Section 3.3.3 and Section 3.3.4, respectively.

One interesting aspect of JATO is that both Java code and native code will be abstracted into the same forms of constraints after step (1). JATO constraints are:

$$
\begin{array}{lll}
\mathcal{K} ::= \overline{\kappa} & & \textit{constraint set} \\
\kappa ::= \alpha \xrightarrow{\theta} \alpha' \mid \alpha \leq \alpha' \mid [\alpha.\mathsf{m}]^{\alpha'} & & \textit{constraint} \\
\theta ::= \mathsf{R} \mid \mathsf{W} & & \textit{access mode} \\
\alpha ::= \ell \mid \phi \mid \texttt{thisO} \mid \texttt{thisT} & & \textit{abstract object/thread} \\
\quad \mid \alpha.\mathsf{f} \mid \alpha.\mathsf{m}^{+} \mid \alpha.\mathsf{m}^{-} & &
\end{array}
$$

An *access constraint* $\alpha \xrightarrow{\theta} \alpha'$ says that an (abstract) object α accesses an (abstract) object α', and the access is either a read ($\theta = \mathsf{R}$) or a write ($\theta = \mathsf{W}$). Objects in JATO's static system are represented in several forms. The first form is an instantiation site label ℓ. Recall earlier, we have required all ℓ's associated with the instantiation expressions (**new** or `NewObject`) to be distinct. It is thus natural to represent abstract objects with instantiation site labels. Our formal system's precision is thus middle-of-the-road: we

differentiate objects of the same class if they are instantiated from different sites, but reins in the complexity by leaving out more precise features such as nCFA [79] or n-object context-sensitivity [61]. The other forms of α are used by the type inference algorithm: label variables $\phi \in \mathbb{LVAR}$, `thisO` for the object enclosing the code being analyzed, `thisT` for the thread executing the code being analyzed, $\alpha.\mathsf{f}$ for an alias to field f of object α, and $\alpha.\mathsf{m}^+$ and $\alpha.\mathsf{m}^-$ for aliases to the return value and the formal parameter of a method invocation to method name m of α, respectively.

The additional two forms of constraints, $\alpha \leq \alpha'$ and $[\alpha.\mathsf{m}]^{\alpha'}$, are used for concrete class analysis and dynamic dispatch approximation, respectively. Constraint $\alpha \leq \alpha'$ says that α may flow into α'. At a high level, one can view this form of constraint as relating two aliases. (As we shall see, the transitive closure of the binary relation defined by $\leq$ is *de facto* a concrete class analysis.) Constraint $[\alpha.\mathsf{m}]^{\alpha'}$ is a *dynamic dispatch placeholder*, denoting method m of object α is being invoked by thread α'.

3.3.3 Intraprocedural constraint generation

We now describe the constraint-generation rules for Step (1) described in Section 3.3.2. Fig. 3.2 and Fig. 3.3 are rules for Java code and native code, respectively. The class-level constraint-generation rules are defined in Fig. 3.4. *Environment* Γ is a mapping from x's to α's. *Constraint summary* $\mathcal{M}$ is a mapping from method names to constraint sets. Judgment $\Gamma \vdash e : \alpha \backslash \mathcal{K}$ says expression e has type α under environment Γ and constraints $\mathcal{K}$. Since no confusion can exist, we further use $\Gamma \vdash t : \alpha \backslash \mathcal{K}$ to represent the analogous judgment for native term t. Judgment $\vdash_{\texttt{cls}}$ **class** $\mathsf{c} \ldots \backslash \mathcal{M}$ says the constraint summary of class c is $\mathcal{M}$. Operator $\triangleright$ is a mapping update: given a mapping U, $U \triangleright [u \mapsto v]$ is identical to U except element u maps to v in $U \triangleright [u \mapsto v]$.

$$\text{(T-Read)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K}}{\Gamma \vdash e.\mathsf{f} : \alpha.\mathsf{f} \backslash \mathcal{K} \cup \{\texttt{thisT} \xrightarrow{\mathsf{R}} \alpha\}}$$

$$\text{(T-Write)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash e.\mathsf{f}\texttt{:=}e' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{f},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \alpha\}}$$

$$\text{(T-Msg)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash e.\mathsf{m}(e') : \alpha.\mathsf{m}^{+} \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{m}^{-},\ [\alpha.\mathsf{m}]^{\texttt{thisT}}\}}$$

$$\text{(T-Thread)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad javaT(\Gamma, e) \text{ is of a thread class}}{\Gamma \vdash e.\texttt{start}() : \alpha \backslash \mathcal{K} \cup \{[\alpha.\texttt{run}]^{\alpha}\}}$$

$$\text{(T-New)}\ \Gamma \vdash \textbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \emptyset \qquad \text{(T-NewThread)}\ \frac{\mathsf{c} \text{ is of a thread class} \qquad \phi \text{ fresh}}{\Gamma \vdash \textbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \{\ell \leq \phi,\ \phi \leq \ell\}}$$

$$\text{(T-Var)}\ \Gamma \vdash x : \Gamma(x) \backslash \emptyset \qquad \text{(T-Null)}\ \Gamma \vdash \textbf{null} : \ell_{\texttt{null}} \backslash \emptyset$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \rhd [x \mapsto \alpha] \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}'}$$

Figure 3.2: Java-Side Intraprocedual Constraint Generation

Observe that types are abstract objects (represented by α's). Java nominal typing (class names as types) is largely orthogonal to our interest here, so our type system does not include it. Taking an alternative view, one can imagine we only analyze programs already typed through Java-style nominal typing. For that reason, we liberally use function $javaT(\Gamma, e)$ to compute the class names for expression e.

On the Java side, (T-Read) and (T-Write) generate constraints to represent the read-/write access from the current thread (`thisT`) to the object whose field is being read-/written (α in both rules). The constraint $\alpha' \leq \alpha.\mathsf{f}$ in (T-Write) abstracts the fact that e' flows into the field f of e, capturing the data flow. The flow constraint generated by (T-Msg) is for the flow from the argument to the parameter of the method. That rule

$$\text{(TN-Read)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad fd = \langle \mathsf{c}, \mathsf{f} \rangle}{\Gamma \vdash \texttt{GetField}(t, fd) : \alpha.\mathsf{f} \backslash \mathcal{K} \cup \{\texttt{thisT} \xrightarrow{\mathsf{R}} \alpha\}}$$

$$\text{(TN-Write)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash t' : \alpha' \backslash \mathcal{K}' \qquad fd = \langle \mathsf{c}, \mathsf{f} \rangle}{\Gamma \vdash \texttt{SetField}(t, fd, t') : \alpha' \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{f},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \alpha\}}$$

$$\text{(TN-Msg)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash t' : \alpha' \backslash \mathcal{K}' \qquad md = \langle \mathsf{c}, \mathsf{m} \rangle}{\Gamma \vdash \texttt{CallMethod}(t, md, t') : \alpha.\mathsf{m}^{+} \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{m}^{-},\ [\alpha.\mathsf{m}]^{\texttt{thisT}}\}}$$

$$\text{(TN-Thread)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad md = \langle \mathsf{c}, \texttt{start} \rangle \qquad \mathsf{c} \text{ is a thread class}}{\Gamma \vdash \texttt{CallMethod}(t, md) : \alpha \backslash \mathcal{K} \cup \{[\alpha.\texttt{run}]^{\alpha}\}}$$

$$\text{(TN-New)}\ \Gamma \vdash \mathbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \emptyset$$

$$\text{(TN-NewThread)}\ \frac{\mathsf{c} \text{ is a thread class} \qquad \phi \text{ fresh}}{\Gamma \vdash \texttt{NewObject}_{\ell}(\mathsf{c}) : \ell \backslash \{\ell \leq \phi,\ \phi \leq \ell\}}$$

$$\text{(TN-Var)}\ \Gamma \vdash x : \Gamma(x) \backslash \emptyset \qquad\qquad \text{(TN-Null)}\ \Gamma \vdash \mathbf{null} : \ell_{\texttt{null}} \backslash \emptyset$$

$$\text{(TN-Let)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \rhd [x \mapsto \alpha] \vdash t' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ t' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}'}$$

Figure 3.3: Native-Side Intraprocedural Constraint Generation

in addition generates a dynamic dispatch placeholder. (T-Thread) models the somewhat stylistic way Java performs thread creation: when an object of a thread class is sent a `start` message, the `run` method of the same object will be wrapped up in a new thread and executed. (T-New) says that the label used to annotate the instantiation point will be used as the type of the instantiated object. (T-NewThread) creates one additional label variable to represent the thread object. The goal here is to compensate the loss of precision of static analysis, which in turn would have affected soundness: a thread object may very well be part of a recursive context (a loop for example) where one instantiation point may be mapped to multiple runtime instances. The static analysis needs to

$$\text{(T-Cls)}\ \frac{\begin{array}{c}\vdash_{\text{cls}} \textbf{class}\ \mathsf{c}_0 \ldots \backslash \mathcal{M} \\ [\text{this} \mapsto \texttt{this0},\ x \mapsto \texttt{this0}.\mathsf{m}^-] \vdash bd : \alpha \backslash \mathcal{K} \text{ for all } mbody(\mathsf{m}, \mathsf{c}) = x.bd \\ \mathcal{K}' = \mathcal{K} \cup \{\alpha \leq \texttt{this0}.\mathsf{m}^+\}\end{array}}{\vdash_{\text{cls}} \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}_0\ \{F\ M\ N\} \backslash (\mathcal{M} \rhd \overline{\mathsf{m} \mapsto \mathcal{K}'})}$$

$$\text{(T-ClsTop)}\ \vdash_{\text{cls}} \textbf{class}\ \texttt{Object} \backslash []$$

Figure 3.4: Class-Level Constraint Generation

be aware if all such instances access one shared memory location – a soundness issue because exclusive access by one thread or shared access by multiple threads have drastically different implications in reasoning about multithreaded programs. The solution here is called *doubling* [41, 90], treating every instantiation point for thread objects as two threads. Observe that we do not perform doubling for non-thread objects in (T-New) because there is no soundness concern there. The rest of the three rules should be obvious, where $\ell_{\texttt{null}}$ is a predefined label for **null**. For the running example, the following constraints will be generated for the two classes written in Java:

$$\texttt{Main} : \{\texttt{main} \mapsto \{\ell_{th} \leq \phi_2,\ \phi_2 \leq \ell_{th},\ [\ell_{th}.\texttt{run}]^{\ell_{th}},\ \ell_2 \leq \ell_1.\texttt{add}^-,\ [\ell_1.\texttt{add}]^{\texttt{thisT}}\}\}$$
$$\texttt{Thread2} : \{\texttt{run} \mapsto \{\ell_1 \leq \ell_2.\texttt{add}^-,\ [\ell_2.\texttt{add}]^{\texttt{thisT}}\}\}$$

The native-side inference rules have a one-on-one correspondence with the Java-side rules – as related by names – and every pair of corresponding rules generate the same form of constraints. This is a crucial insight of JATO: by abstracting the two worlds of Java syntax and native code syntax into one unified constraint representation, the artificial boundary between Java and native code disappears. As a result, thorny

problems such as callbacks (to Java) inside native code no longer exists – the two worlds, after constraints are generated, are effectively one. The constraints for the `Node` class in the running example are:

$$\texttt{Node} : \{\mathsf{add} \mapsto \{\texttt{thisT} \xrightarrow{\mathsf{R}} \texttt{thisO},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \texttt{thisO},\ \texttt{thisT} \xrightarrow{\mathsf{R}} \texttt{thisO}.\mathsf{add}^{-}\}\}$$

3.3.4 Constraint closure

Now that the constraint summary has been generated on a per-class per-method basis, we can discuss how to combine them into one global set. This is defined by computing the *constraint closure*, defined as follows:

Definition 3 (Constraint Closure) *The closure of program P with entry method md, denoted as $[\![P, md]\!]$ is the smallest set that satisfies the following conditions:*

- ***Flows**: $\leq$ is reflexive and transitive in $[\![P, md]\!]$.*
- ***Concrete Class Approaching**: If $\{\alpha' \leq \alpha\} \cup \mathcal{K} \subseteq [\![P, md]\!]$, then $\mathcal{K}\{\alpha'/\alpha\} \subseteq [\![P, md]\!]$.*
- ***Dynamic Dispatch**: If $[\ell.\mathsf{m}]^{\ell_0} \in [\![P, md]\!]$, then $\mathcal{M}(\mathsf{m})\{\ell/\texttt{thisO}\}\{\ell_0/\texttt{thisT}\} \subseteq [\![P, md]\!]$ where $\mathcal{L}_P(\ell) = \mathsf{c}$ and $\vdash_{\texttt{cls}}$ **class** $\mathsf{c} \ldots \backslash\mathcal{M}$.*
- ***Bootstrapping**: $\{[\ell_{\texttt{BO}}.\mathsf{m}]^{\ell_{\texttt{BT}}}, \ell_{\texttt{BP}} \leq \ell_{\texttt{BO}}.\mathsf{m}^{-}\} \subseteq [\![P, md]\!]$ where $md = \langle \mathsf{c}, \mathsf{m} \rangle$.*

The combination of **Flows** and **Concrete Class Approaching** is *de facto* a concrete class analysis, where the "concrete class" in our case is the object instantiation sites

(not Java nominal types): the **Flows** rule interprocedurally builds the data flow, and the **Concrete Class Approaching** rule substitutes a flow element with one "up stream" on the data flow. When the "source" of the data flow – an instantiation point label – is substituted in, concrete class analysis is achieved. Standard notation $\mathcal{K}\{\alpha'/\alpha\}$ substitutes every occurrence of α in $\mathcal{K}$ with α'. **Dynamic Dispatch** says that once the receiver object of an invocation resolves to a concrete class, dynamic dispatch can thus be resolved. The substitutions of `thisO` and `thisT` are not surprising from an interprocedural perspective. The last rule, **Bootstrapping**, bootstraps the closure. $\ell_{\texttt{BO}}, \ell_{\texttt{BT}}, \ell_{\texttt{BP}}$ are pre-defined labels representing the bootstrapping object (the one with method md), the bootstrapping thread, and the parameter used for the bootstrapping invocation.

For instance, if P is the running example, the following constraints are among the ones in the closure from its main method, *i.e.*, $[\![P, \langle \mathsf{c}_{\texttt{main}}, \mathsf{m}_{\texttt{main}} \rangle]\!]$:

$$\ell_{\texttt{BT}} \xrightarrow{\mathsf{R}} \ell_1 \qquad \ell_{\texttt{BT}} \xrightarrow{\mathsf{W}} \ell_1 \qquad \ell_{\texttt{BT}} \xrightarrow{\mathsf{R}} \ell_2$$
$$\ell_{th} \xrightarrow{\mathsf{R}} \ell_2 \qquad \ell_{th} \xrightarrow{\mathsf{W}} \ell_2 \qquad \ell_{th} \xrightarrow{\mathsf{R}} \ell_1$$

That is, the bootstrapping thread performs read and write access to object ℓ_1 and read access to object ℓ_2. The child thread performs read access to object ℓ_1 and read and write access to object ℓ_2. This matches our intuition about the program.

3.3.5 Atomicity enforcement

Based on the generated constraints, JATO infers a set of Java objects that need to be locked in a native method to ensure its atomicity. JATO also takes several optimizing steps to remove unnecessary locks while still maintaining atomicity.

Lock-all. The simplest way to ensure atomicity is to insert locks for all objects that a native method may read from or write to. Suppose we need to enforce the atomicity of a native method md in a program P, the set of objects that need to be locked are:

$$Acc(P, md) \stackrel{\text{def}}{=} \{\, \alpha \mid (\alpha' \xrightarrow{\theta} \alpha) \in [\![\, P, md \,]\!] \wedge (\alpha \in \mathbb{LAB} \vee labs(\alpha) \subseteq \{\, \ell_{\texttt{BO}}, \ell_{\texttt{BP}} \,\}) \,\}$$

The first predicate $(\alpha' \xrightarrow{\theta} \alpha) \in [\![\, P, md \,]\!]$ says that α is indeed read or written. The α's that satisfy this predicate may be in a form that represents an alias to an object, such as $\ell.\mathsf{f}_1.\mathsf{f}_2.\mathsf{m}^+$, and it is clearly desirable to only inform the lock insertion procedure of the real instantiation point of the object (the $\alpha \in \mathbb{LAB}$ predicate) – *e.g.*, "please lock the object instantiated at label ℓ_{33}." This, however, is not always possible because the instantiation site for the object enclosing the native method and that for the native method parameter are abstractly represented as $\ell_{\texttt{BO}}$ and $\ell_{\texttt{BP}}$, respectively. It is thus impossible to concretize any abstract object whose representation is "built around them". For example, $\ell_{\texttt{BO}}.\mathsf{f}_3$ means that the object is stored in field f_3 of the enclosing object $\ell_{\texttt{BO}}$, and access to the stored object requires locking the enclosing object. This is the intuition behind predicate $labs(\alpha) \subseteq \{\, \ell_{\texttt{BO}}, \ell_{\texttt{BP}} \,\}$, where $labs(\alpha)$ enumerates all the labels in α.

For the running example, the set of objects to lock for the native `add` method – $Acc(P, \langle \texttt{Node}, \texttt{add} \rangle)$ – is $\{\, \ell_{\texttt{BO}},\ \ell_{\texttt{BP}} \,\}$, meaning both the enclosing object and the parameter needs to be locked.

Locking all objects in $Acc(P, md)$ is sufficient to guarantee the atomicity of md. This comes as no surprise: every memory access by the native method is guarded by a lock. The baseline approach here is analogous to a purely dynamic approach: instead of statically computing the closure and the set of objects to be locked as we define here,

one could indeed achieve the same effect by just locking at run time for every object access.

In the lock-all approach, JATO inserts code that acquires the lock for each object in the set as computed above and releases the lock at the end. The lock is acquired by JNI function `MonitorEnter` and released by `MonitorExit`.

Lock-on-write.

In this strategy, we differentiate read and write access, and optimize based on the widely known fact that non-exclusive reads and exclusive writes are adequate to guarantee atomicity. The basic idea is simple: given a constraint set $\mathcal{K}$, only elements in the following set needs to be locked, where $size$ computes the size of a set:

$$lockS(\mathcal{K}) \stackrel{\text{def}}{=} \{\ell \mid size(\{\ell' \mid \ell' \xrightarrow{\mathsf{W}} \ell \in \mathcal{K}\}) \neq 0 \wedge size(\{\ell' \mid \ell' \xrightarrow{\theta} \ell \in \mathcal{K}\}) > 1\}$$

It would be tempting to compute the necessary locks for enforcing the atomicity of native method md of program P as $lockS([\![\, P, md \,]\!])$. This unfortunately would be unsound. Consider the running example. Even though the parameter object `n` is only read accessed in native method `add`, it is not safe to remove the lock due to two facts: (1) in the main thread, `add` receives object ℓ_1 as the argument; (2) in the child thread, object ℓ_1 is mutated. If the lock to the parameter object `n` were removed, atomicity of `add` could not be guaranteed since the integer value in the parameter object may be mutated in the middle of the method. Therefore, it is necessary to perform a global analysis to apply the optimization.

The next attempt would be to lock objects in $lockS([\![\, P, \langle \mathsf{c}_{\texttt{main}}, \mathsf{m}_{\texttt{main}} \rangle \,]\!])$. Clearly, this is sound, but it does not take into the account that native method md only accesses

a subset of these objects. To compute the objects that are accessed by md, we define function $AccG(P, md)$ as the smallest set satisfying the following conditions, where $md = \langle \mathsf{c}, \mathsf{m} \rangle$ and $\mathcal{K}_0 = [\![P, \langle \mathsf{c}_{\mathtt{main}}, \mathsf{m}_{\mathtt{main}} \rangle]\!]$:

- If $\{[\ell.\mathsf{m}]^{\ell_0}, \ell_1 \leq \ell.\mathsf{m}^-\} \subseteq \mathcal{K}_0$ where $\mathcal{L}_P(\ell) = \mathsf{c}$, then $Acc(P, md)\{\ell/\ell_{\mathtt{BO}}\}\{\ell_0/\ell_{\mathtt{BT}}\}\{\ell_1/\ell_{\mathtt{BP}}\} \subseteq AccG(P, md)$.
- If $\ell \leq \alpha \in \mathcal{K}_0$ and $\alpha \in AccG(P, md)$, then $\ell \in AccG(P, md)$.

In other words, $Acc(P, md)$ almost fits our need, except that it contains placeholder labels such as $\ell_{\mathtt{BO}}$, $\ell_{\mathtt{BT}}$, and $\ell_{\mathtt{BP}}$. $AccG$ concretizes any abstract object whose representation is dependent on them. With this definition, we can now define our strategy: locks are needed for native method md of program P for any object in the following set:

$$AccG(P, md) \cap lockS([\![P, \langle \mathsf{c}_{\mathtt{main}}, \mathsf{m}_{\mathtt{main}} \rangle]\!]).$$

Lock-at-write-site. Instead of acquiring the lock of an object at the beginning of a native method and releasing the lock at the end, this optimization inserts locking around the code region of the native method that accesses the object. If there are multiple accesses of the object, JATO finds the smallest code region that covers all accesses and acquires/releases the lock only once.

3.4 Prototype implementation

We implemented a prototype system based on the constraint-based system described in the previous section. Java-side constraint generation in JATO is built upon Cypress [94],

a static analysis framework focusing on memory access patterns. Native-side constraint generation is implemented in CIL [65], an infrastructure for analyzing and transforming C code. The rest of JATO is developed in around 5,000 lines of OCaml code.

One issue that we have ignored in the idealized JNI language is the necessity of performing Java-type analysis in native code. In the idealized language, native methods can directly use field IDs in the form of $\langle c, f \rangle$ (and similarly for method IDs). But in real JNI programs, native methods have to invoke certain JNI functions to construct those IDs. To read a field of a Java object, native method must take three steps: (1) use `GetObjectClass` to get a reference to the class object of the Java object; (2) use `GetFieldID` to get a field ID for a particular field by providing the field's name and type; (3) use the field ID to retrieve the value of the field in the object.

For instance, the following program first gets `obj`'s field `nd`, which is a reference to another object of class `Node`. It then reads the field `i` of the `Node` object.

```
jclass cls = GetObjectClass(obj);
jfieldID fid = GetFieldID(cls, "nd", "Node");
jobject obj2 = GetField(obj, fid);
jclass cls2 = GetObjectClass(obj2);
jfieldID fid2 = GetFieldID(cls2, "i", "I");
int x1 = GetIntField(obj, fid2);
```

The above steps may not always be performed in consecutive steps; caching field and method IDs for future use is a common optimization. Furthermore, arguments provided to functions such as `GetFieldID` may not always be string constants. For better precision, JATO uses an interprocedural, context-sensitive static analysis to track constants and infer types of Java references [50]. For the above program, it is able to decide

that there is a read access to `obj` and there is a read access to `obj.nd`. To do this, it is necessary to infer what Java class `cls` represents and what field ID `fid` represents.

3.5 Preliminary evaluation

We performed preliminary evaluation on a set of multithreaded JNI programs. Each program was analyzed to generate a set of constraints, as presented in Section 3.3. Based on the closure of the generated constraints, a set of objects were identified to ensure atomicity of a native method in these programs. Different locking schemes were evaluated to examine their performance.

All experiments were carried out on an iMac machine running Mac OS X (version 10.7.4) with Intel core i7 CPU of 4 cores clocked at 2.8GHz and with 8GB memory. The version of Java is OpenJDK 7. For each experiment, we took the average among ten runs.

We next summarize the JNI programs we have experimented with. The programs include: (1) a parallel matrix-multiplication (`MM`) program, constructed by ourselves; (2) a Fast-Fourier-Transform (`FFT`) program, adapted from JTransforms [89] by rewriting some Java routines in C; (3) the `compress` program, which is a module that performs multithreaded file compression provided by the MessAdmin [59] project; (4) the `derby` benchmark program is selected from SPECjvm2008 [82] and is a database program. Both `compress` and `derby` are pure Java programs, but they invoke standard Java classes in `java.io` and `java. util.zip`, which contain native methods.

The analysis time and LOC for both Java side and C side on each program are listed

below. It is observed that majority of the time is spent on Java side analysis, particularly on Java-side constraint-generation.

Program	LOC (Java)	Time (Java)	LOC (C)	Time (C)
`MM`	275	3.34s	150	10μs
`FFT`	6,654	8.14s	3,169	0.01s
`compress`	3,197	27.8s	5,402	0.05s
`derby`	919,493	81.04s	5,402	0.05s

All programs are benchmarked under the three strategies we described in the previous section. L-ALL stands for the lock-all approach. L-W stands for the lock-on-write approach. L-WS stands for the case after applying the lock-on-write and lock-at-write-site optimizations.

Matrix multiplication. The programs takes in two input matrices, calculates the multiplication of the two and writes the result in an output matrix. It launches multiple threads and each thread is responsible for calculating the result of one element of the output matrix. The calculation of one element is through a native method. In this program, three two-dimensional arrays of `double` crosses the boundary from Java to the native code.

For this program, JATO identifies that the native method accesses the three cross-boundary objects. These objects are shared among threads. The input matrices and their arrays are read-accessed whereas the resulting matrix and its array are read- and write-accessed.

Fig. 3.5(a) presents the execution times of applying different locking schemes. The size of the matrices is 500 by 500 with array elements ranging between 0.0 and 1000.0.

L-WS has the best performance overall.

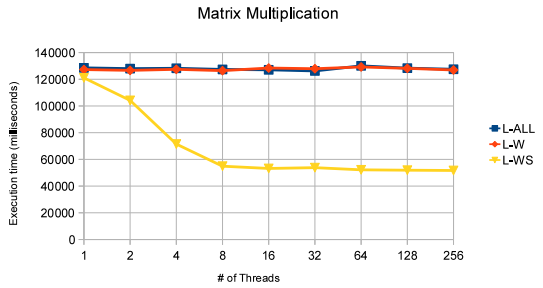
FFT. The native method of this program takes in an array of `double` to be transformed and sets the transformed result in an output array. The input array is read-accessed whereas the output array is write-accessed. The arrays are shared among threads. Fig. 3.5(b) shows the results of `FFT`. Similar to the program of matrix multiplication, L-W improved upon L-ALL, and L-WS performs the best among the three.

compress. This program compresses an input file by dividing the file into smaller blocks and assigning one block to one thread for compression. The actual compression is performed in the native side using the `zlib` C library. JATO identifies that a number of objects such as `Deflater` are shared among threads and read/write accessed at the Java side. One exception is `FileInputStream`, where it is only read-accessed in Java but is write-accessed at the native side. In term of the number of locks inserted, there is little difference between lock-all and lock-on-write.

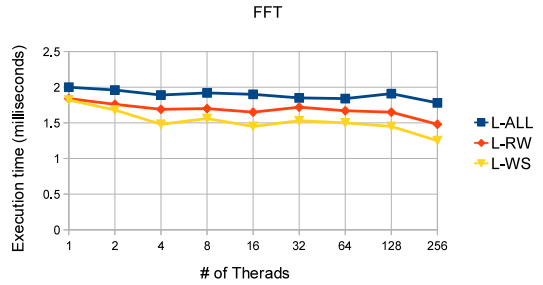
Fig. 3.5(c) presents the results of `compress`. The file size is about 700MB and the block size is 128K. The performance gain of L-W over L-ALL is negligible. We see there is some minor improvement using L-WS. This is because in the native code, write-access code regions to the locked objects are typically small.

derby. It is a multithreaded database. Some `byte` arrays and `FileInputStream` objects are passed into the native code. They are read-accessed between threads from the Java side. On the native side, both kinds of objects are write-accessed.

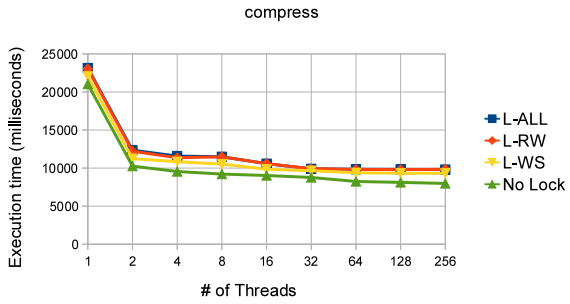
Fig. 3.5(d) shows the result of running derby. The experiment was run for 240 seconds with 60 seconds warm-up time. The peak ops/min occurs when the number of



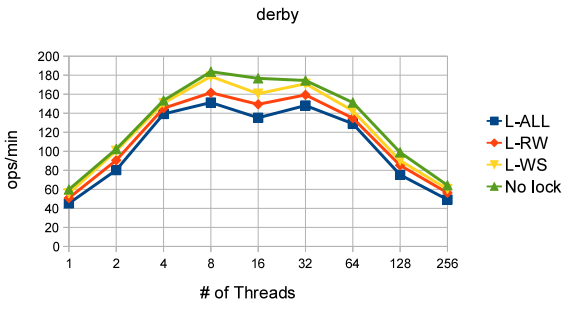
(a) MM



(b) FFT



(c) compress



(d) derby

Figure 3.5: Execution time of the benchmark programs under different locking schemes.

threads is between 8 to 32. We can see that in L-WS approach, the performance gains at its peak is about 35% over L-ALL.

For `compress` and `derby`, we also experimented with the no-lock scheme in which no locking is inserted in native methods. Although the uninstrumented programs run successfully, there is no guarantee of native-method atomicity as provided by JATO. The programs of matrix multiplication and `FFT` would generate wrong results when no locks were inserted for native-method atomicity. For the matrix-multiplication program, even though the native method of each thread calculates and updates only one element of the output matrix, it is necessary to acquire the lock of the output matrix before operating on it: native methods use JNI function `GetArrayElements` to get a pointer to the output matrix and `GetArrayElements` may copy the matrix and return a pointer to the copy [51].

3.6 Summary

JATO is a system that enforces atomicity of native methods in multithreaded JNI programs. Atomicity enforcement algorithms are generalized to programs developed in multiple languages by using an inter-language, constraint-based system. JATO takes care to enforce a small number of locks for efficiency.

CHAPTER 4

REFERENCE COUNTING IN PYTHON/C PROGRAMS WITH AFFINE PROGRAM ANALYSIS

4.1 Introduction

The Python programming language has become widely adopted in the software development community over the years because of many appealing features of the language itself and a robust ecosystem [60]. Similar to many other languages, Python provides a Foreign Function Interface (FFI), called the Python/C interface. The interface allows Python programs to interoperate with native modules written in C/C++. Through the interface, Python programs can reuse legacy native libraries written in C/C++ or use native code to speed up their performance-critical parts. Python provides a comprehensive set of Python/C API functions. Through these functions, native modules can create Python objects, manipulate objects, raise and handle Python exceptions, and perform other actions [72].

Another important feature of Python is its memory management. Python allocates objects on its heap. When objects are no longer in use, Python's memory manager garbage collects these objects from the heap. The standard implementation of Python uses the reference-counting algorithm. The representation of every Python object has a reference-count field. When Python code is running, the Python runtime automatically adjusts the reference counts during program execution and maintains the invariant that an object's reference count be the same as the number of references to the object. Specifically, the reference count of an object is incremented when there is a new reference to the object or decremented when a reference disappears. When an object's reference

count becomes zero, its space is reclaimed from the heap by the garbage collector.

Native modules incorporated in a Python program, on the other hand, are outside the control of Python's garbage collector. When those native modules manipulate Python objects through the Python/C interface, reference counts are not adjusted automatically by the Python runtime and it is the native code's responsibility to adjust reference counts in a correct way (through `Py_INCREF` and `Py_DECREF`, discussed later). This is an error-prone process. Incorrect adjustments of reference counts result in classic memory errors such as memory leaks and use of dangling references.

In this chapter, we describe a system called Pungi, which performs static analysis to identify reference-counting errors in native C modules of Python programs. Pungi abstracts a native module to an affine program, which models how reference counts are changed in the native module. In an affine program, the right-hand side of an assignment can only be an affine expression of the form $a_0 + \Sigma_{i=1}^{n} a_i x_i$, where a_i are constants and x_i are program variables. A previous theoretical study [43] has shown that an affine program is sufficient to model reference-count changes in the case of *shallow aliasing* (which assumes multi-level references to be non-aliases). That study, however, is mainly concerned with computational complexity and does not consider many practical issues, including function calls with parameter passing and references that escape objects' scopes. Furthermore, its proposed affine-abstraction step has not been implemented and tested for effectiveness. In fact, its affine-abstraction step is non-intuitive by requiring reversing the control flow of programs. Moreover, it does not describe how to analyze the resulting affine program to identify reference-counting errors. More detailed discussion of that work and its comparison with Pungi will be presented when we discuss the design of Pungi.

Major contributions of Pungi are described as follows:

- We propose a set of ideas that make the affine-abstraction step more complete and practical. In particular, we show how to perform affine abstraction interprocedurally and how to accommodate escaping references. We further show that the affine-abstraction step can be simplified by first performing a Static-Single Assignment (SSA) transform on the input program.
- We propose to use path-sensitive, interprocedural static analysis on the resulting affine programs to report possible reference-counting errors. We show this step is precise and efficient.
- We have built a practical reference-count analysis system that analyzes Python/C extension modules. Our system over 150 errors in 13 benchmark programs, with a modest false-positive rate of 22%.

The main limitation of Pungi is the assumption of shallow aliasing, which allows direct references to be aliases but multi-level references are assumed to reference distinct objects. For instance, if a Python program has a reference to a list object, then all objects within the list are assumed to be distinct objects. Pungi's assumption of shallow aliasing and its other assumptions may cause it to have false positives and false negatives. However, our experience shows that Pungi remains an effective tool given that it can find many reference-counting errors and its false-positive rate is moderate.

The rest of this chapter is structured as follows. Section 4.2 includes the background information about the Python/C interface and reference counting. In Section 4.3, we provide an overview of Pungi. The detailed design of Pungi is presented in Section 4.4 and 4.5. Pungi's implementation and a summary of its limitations are in Section 4.6.

```
1  static PyObject* create_ntuple(PyObject *self,
2        PyObject *args) {
3   int n, i, err;
4   PyObject *tup = NULL;
5   PyObject *item = NULL;
6   // parse args to get input number n
7   if (!PyArg_Parse(args, "(i)", &n)) return NULL;
8   tup = PyTuple_New(n);
9   if (tup == NULL) return NULL;
10  for (i=0; i<n; i++) {
11    item = PyInt_FromLong(i);
12    if (item == NULL) {Py_DECREF(tup); return NULL;}
13
14    err = PyTuple_SetItem(tup, i, item);
15    if (err) { // no need to dec-ref item
16      Py_DECREF(tup); return NULL;}
17  }
18  return tup;
19 }
```

Figure 4.1: An example Python/C extension module called `ntuple` (its registration table and module initializer code are omitted).

Experimental results are discussed in Section 4.7. We summarize on this chapter in Section 4.8.

4.2 Background: the Python/C interface and reference counting

The Python/C interface allows a Python program to incorporate a native library by developing a native extension module. The extension module provides a set of *native functions*. Some of the native functions are registered to be *entry native functions*, which can be imported and directly called by Python code; the rest are helper functions. An entry native function takes Python objects as input, uses Python/C API functions to create/-manipulate objects, and possibly returns a Python object as the result.

Fig. 4.1 presents a simple C extension module called `ntuple`. It implements

one function `create_ntuple`, which takes an integer `n` and constructs a tuple `(0,1,...,n-1)`. In more detail, references to Python objects have type "`PyObject *`".[1] Parameter `args` at line 2 is a list object, which contains the list of objects passed from Python. The call to the API function `PyArg_Parse` at line 7 decodes `args` and puts the result into integer `n`; format string "`(i)`" specifies that there should be exactly one argument, which must be an integer object. API function `PyTuple_New` creates a tuple with size `n`. The loop from line 10 to line 17 first creates an integer object using `PyInt_FromLong` and updates the tuple with the integer object at the appropriate index. For brevity, we have omitted the extension module's code for registering entry native functions and for initialization.

After the `ntuple` extension module is compiled to a dynamically linked library, it can be imported and used in Python, as shown below.

```
>>> import ntuple
>>> ntuple.create_ntuple(5)
(0, 1, 2, 3, 4)
```

4.2.1 Python/C reference counting and its complexities

As mentioned, native extension modules are outside the reach of Python's garbage collector. Native code must explicitly increment and decrement reference counts (we abbreviate reference counts as refcounts hereafter). Specifically,

- `Py_INCREF(p)` increments the refcount of the object referenced by `p`.

[1]The Python/C interface defines type `PyObject` and a set of subtypes that can be used by extension code, such as `PyIntObject` and `PyStringObject`. Pungi does not distinguish these types in its analysis and treats them as synonyms. Therefore, we will just use `PyObject` in the rest of the chapter.

- `Py_DECREF(p)` decrements the refcount of the object referenced by `p`. When the refcount becomes zero, the object's space is reclaimed and the refcounts of all objects whose references are in object `p` get decremented.

Correct accounting of refcounts of objects, however, is a complex task. We next discuss the major complexities.

Control flow. Correct reference counting must be performed in all control flow paths, including those paths resulting from error conditions or interprocedural control flows. Take code in Fig. 4.1 as an example. At line 11, an integer object is allocated, but the allocation may fail. In the failure case, the code returns immediately, but it is also important to perform `Py_DECREF` on the previously allocated `tup` object; forgetting it would cause a memory leak. Similarly, at line 16, a `Py_DECREF(tup)` is necessary. Clearly, taking care of reference counts of all objects in all control-flow paths is a daunting task for programmers.

Borrowed and stolen references. It is common in native code to use the concept of *borrowed references* to save some reference-counting work. According to the Python/C manual [72], when creating a new reference to an object in a variable, "if we know that there is at least one other references to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily".

For instance, if function `foo` calls `bar` and passes `bar` a reference to an object:

```
void foo () {
  PyObject *p = PyInt_FromLong (...);
  bar (p); }
```

```
void bar (PyObject *q) { ... }
```

Within the scope of `bar`, there is one more reference (namely, `q`) to the object allocated in `foo`. However, it is safe not to increment the refcount of the object inside `bar`. The reason is that, when the control is within `bar`, we know there is at least one more reference in the caller and that reference outlives local reference `q`. Therefore, it is safe to allow more references than the refcount of the object. In this situation, the callee "borrows" the reference from the caller, meaning that the callee creates a new reference without incrementing the refcount.

Moreover, certain Python/C API functions allow callers of those functions to borrow references. For instance, `PyList_GetItem` returns a reference to an item in a list. Even though it returns a new reference to the list item, `PyList_GetItem` does not increment the refcount of the list item. This is safe when the list is not mutated before the new reference is out of scope; in this case, the reference stored in the list will outlive the new reference.[2] The Python/C reference manual lists the set of API functions with this behavior.

Dual to the situation that callers may borrow references from some API functions, certain API functions can "steal" references from the callers. For instance, in a call `PyTuple_SetItem(tuple,i,item)`, if `tuple[i]` contains an object, the object's refcount is decremented; then `tuple[i]` is set to `item`. Critically, `item`'s refcount is not incremented even though a new reference is created in the tuple. This practice is safe if we assume the `item` reference is never used after the set-item operation, which is often the case. Another behavior is that `PyTuple_SetItem(tuple,i,item)` may fail, in which case

[2] If the list may be mutated, then the caller should increment the refcount of the retrieved object after calling `PyList_GetItem`.

`Py_DECREF(item)` is automatically performed by the API function. This is why at line 16 in Fig. 4.1 there is no need to decrement the refcount on `item`.

API reference-count semantics. We have already alluded to the fact that Python/C API functions may have different effects on the refcounts of involved objects. Certain functions borrow references and certain functions steal references. Certain functions allocate objects. For instance, the calls to `PyTuple_New` and `PyInt_FromLong` in Fig. 4.1 allocate objects and set the refcounts of those objects to be one when allocation succeeds. And certain functions do not affect the refcounts of objects. When programmers use those API functions, they can often be confused by their effects on refcounts and make mistakes.

All of the above factors make correct reference counting in native code extremely difficult. As a result, reference-counting errors are common in Python/C native extensions.

4.3 Pungi overview

Fig. 4.2 shows the main steps in Pungi. It takes a Python/C extension module as input and reports reference-counting errors. Pungi analyzes only C code, but does not analyze Python code that invokes the C code.

The first step performed by Pungi is to separate *interface code* from *library code* in the extension module. As observed by a previous static-analysis system on the Java Native Interface [50], code in an FFI package can be divided into interface and library code. The library code is part of the package that belongs to a common native library. The interface code glues the host language such as Python with the native library. A

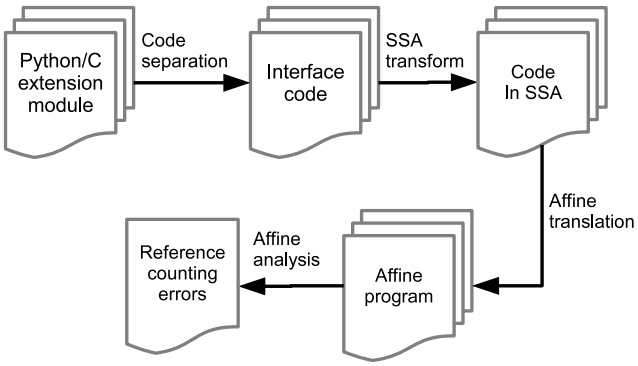


Figure 4.2: An overview of Pungi.

native function is part of the interface code if 1) it invokes a Python/C API function, or 2) it invokes another native function that is part of the interface code. For example, the PyCrypto package has a thin layer of interface code that links Python with the underlying cryptography library. Typically, the size of interface code is much smaller than the size of library code. Therefore, Pungi performs a static analysis to separate interface code and library code so that the following steps can ignore the library code. Pungi implements a simple worklist algorithm to find functions in the interface code. If a native function does not belong to the interface code, then its execution should not have any effect on Python objects' refcounts.

After separation, *affine abstraction* converts the interface code to an affine program. The conversion is performed in two steps: Static Single Assignment (SSA) transform and affine translation. First, the SSA transform is applied on the interface code. The SSA transform makes the following affine-translation step easier to formulate; each variable is assigned only once, making it easy to track the association between variables and Python objects. In affine translation, the interface code in the SSA form is translated into an affine program. In the affine program, variables are used to track properties of Python objects, such as their refcounts. Statements are affine operations that model

how properties such as refcounts are changed in the interface code. Assertions about refcounts are also inserted into affine programs; assertion failures suggest reference-counting errors. Details of the process of affine abstraction are presented in Section 4.4.

After affine abstraction, Pungi performs an interprocedural and path-sensitive analysis that analyzes the affine program and statically checks whether assertions in the affine program hold. If an assertion might fail, a warning about a possible reference-counting error is reported. Details of affine analysis are presented in Section 4.5.

4.4 Affine abstraction

For better understanding, we describe Pungi's affine abstraction in two stages. We will first present its design with the assumption that object references do not escape their scopes. We will then relax this assumption and generalize the design to allow escaping object references (*e.g.*, via return values or via a memory write to a heap data structure).

4.4.1 Bug definition with non-escaping references

One natural definition of a reference-counting error is as follows: at a program location, there is an error if the refcount of an object is not the same as the number of references to the object. However, this bug definition is too precise and an analysis based on the definition would generate too many false positives in real Python/C extension modules. This is due to the presence of borrowed and stolen references we discussed. In both cases, it is safe to make the refcount be different from the number of actual references.

Pungi's reference-counting bug definition is based on a notion of object scopes and the intuition that the expected *refcount change* of an object should be zero at the end of the object's scope (when references to the object do not escape its scope). To define an object's scope, we distinguish two kinds of objects:

- An object is a *Natively Created (NC) object* if it is created in a Python/C extension module. In Fig. 4.1, objects referenced by `tup` and `item` are NC objects. An NC object's scope is defined to be the immediate scope surrounding the object's creation site. For instance, the scope of the object referenced by `tup` is the function scope of `create_ntuple`.
- An object is a *Python Created (PC) object* when its reference is passed from Python to an entry native function through parameter passing. Note that we call objects whose references are passed to a native function *parameter objects*, but those parameter objects are PC objects only if that native function is an entry function. In Fig. 4.1, the `self` and `args` objects are PC objects. We define the scope of a PC object to be the function scope of the entry native function that receives the reference to the PC object because Pungi analyzes only native code,

Definition 4 *In the case of non-escaping object references, there is a reference-counting error if, at the end of the scope of an NC or PC object, its refcount change is non-zero. If the change is greater than zero, we call it an error of reference over-counting. If the change is less than zero, we call it an error of reference under-counting.*

We next justify the bug definition. In the discussion, we use rc to stand for the refcount change of an object. Suppose the object is an NC object. If $rc > 0$, it results in a memory leak at the end of the scope because (1) the refcount remains positive

```
1 void buggy_foo () {
2   PyObject * pyo = PyInt_FromLong(10);
3   if (pyo == NULL) return;
4   return;
5 }
```

Figure 4.3: A contrived example of a buggy Python/C function.

and (2) the number of references to the object becomes zero (as object references do not escape the scope). Take the contrived code in Fig. 4.3 as an example. The object creation at line 2 may result in two cases. In the failure case, the object is not created and `PyInt_FromLong` returns `NULL`. In the successful case, the object is created with refcount one; in this case, the net refcount change to the object is one before returning, signaling a reference over-counting error. The correct code should have `Py_DECREF(pyo)` before line 4.

If $rc < 0$ for an NC object, then there is a use of a dangling reference because at some point of the native function execution, the refcount of the object becomes zero and the object is deallocated as a result; the next `Py_DECREF` dereferences the dangling reference.

Suppose the object is a PC object of an entry native function. We can safely assume at the beginning of the function the object's refcount is the same as the number of references to the object because the object is passed from Python, whose runtime manages refcounts automatically. If $rc > 0$ at the end of the entry native function, then after the execution of the function the object's refcount must be greater than the number of references to the object (because object references do not escape). This leads to a potential memory leak. If $rc < 0$, this leads to a dangling reference when the native function is invoked with an object whose refcount is one. Since Pungi analyzes only native code, not Python code; it has to be conservative.

One limitation of the bug definition is that it misses some dangling-reference errors that happen in the middle of native functions. For example, a native function can first decrement the refcount of a PC object and then increment the refcount. Although at the end the refcount change is zero, the object gets deallocated after the decrement if the object's original refcount is one; the following increment would use a dangling reference. This is a limitation of Pungi and we leave it to future work.

4.4.2 SSA transform

Inspired by a previous theoretical study, Pungi uses an affine program to model how refcounts are changed in the interface code of a Python/C extension module. The previous study, however, requires reversing the control-flow graph: the changes at a program location are computed based on changes that follow the location in the control-flow graph (meaning that changes for program locations later in the control-flow graph have to be computed first). The resulting affine program's control flow reverses the control flow of the original program. This process is non-intuitive and it is also unclear how to generalize it to cover function calls with parameter passing.

We observe that the fundamental reason why reversing the control-flow graph is necessary is that variables may be assigned multiple times to reference different objects. Based on this observation, Pungi simplifies the affine abstraction step by first applying the Static Single Assignment (SSA) transform to the interface code. The SSA transform inserts ϕ nodes into the program at control-flow join points and renames variables so that they are statically assigned only once. As we will show, the benefit is that Pungi does not need to reverse the control-flow graph when performing the affine-translation step; further, we can also generalize the affine translation to cover function calls with

parameter passing.

Pungi's SSA transform performs transformation on only variables of type "`PyObject *`" because only Python objects are of interests to Pungi. Variables of other types are not SSA transformed. For the example in Fig. 4.1, variable `i` is not SSA transformed even though it is statically assigned twice. On the other hand, the `item` variable is initialized at the beginning of the code and assigned in the loop. Therefore, one ϕ node is inserted before the conditional test $i < n$ and the `item` variable is split to multiple ones. The critical parts of the control-flow graph after the SSA transform are visualized in Fig. 4.4

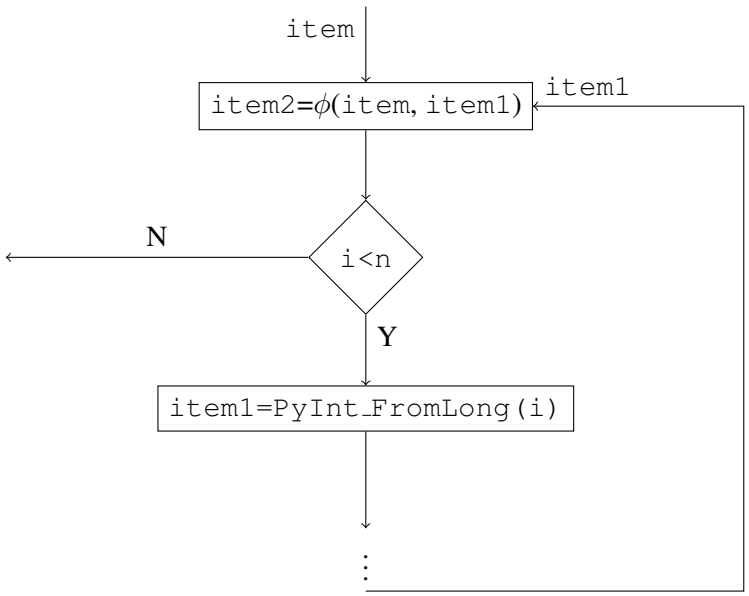


Figure 4.4: Part of the control-flow graph for the code in Fig. 4.1 after SSA.

4.4.3 Affine translation

The affine-translation step translates C interface code in the SSA form to an affine program that models the refcount changes of Python objects. We next explain the intuition behind the translation, before presenting the translation algorithm.

Intuition about the affine translation. Let us assume a function takes n input references: $p_1, p_2, ..., p_n$, each of which is a reference to some Python object. Shallow aliasing allows some of these references to be aliases. For instance, p_1 and p_2 may reference the same object, in which case the refcount of the object can be changed via either p_1 or p_2.

With the assumption of shallow aliasing, Lal and Ramalingam [43] proved the following key properties:

(i) the refcount change to an object is the sum of the amount of changes made via references in $p_1, p_2, ..., p_n$ that point to the object.

(ii) the refcount change to an object via a reference is independent from the initial aliasing situation and therefore can be computed assuming an initial aliasing situation in which $p_1, p_2, ..., p_n$ are non-aliases.

We next illustrate via an example as follows:

```
p3 = p1;
Py_INCREF(p1);
Py_DECREF(p3);
Py_INCREF(p1);
```

```
Py_DECREF(p2);
```

Let us first assume p1, p2, and p3 reference distinct objects initially. Let rc_i be the refcount change made by the program to the object that pi initially points to. Since it is a simple program, we can easily see that $rc_1 = 1, rc_2 = -1, rc_3 = 0$. The reason why rc_3 is zero is because p3 is updated to be p1 in the first statement; so there is no refcount change to the object that p3 initially references.

Now suppose the program is actually run in an initial aliasing situation where p1 and p2 are aliases referencing object a and p3 references a different object b. In this case, according to the stated properties (i) and (ii), we can compute that the refcount change to object a is $rc_1 + rc_2$, which is zero, and the refcount change to object b is rc_3, which is also zero.

The follow-up question is how to compute rc_i for an arbitrary program. The computation is modeled by an affine program, which is discussed next.

Affine program syntax. The syntax of our affine programs is presented in Fig. 4.5. In the syntax, we use meta-symbol x for variables and i for integer constants. An affine program consists of a set of mutually recursive functions; we assume the first function is the main function. A function declaration contains a name and a body. The body contains the declaration of a list of local variables and a block, which is a list of statements.

A statement in an affine program contains various forms of assignments, of which the right-hand sides are affine expressions. The condition c in an if-statement or a while-statement can be either a predicate, which compares a variable to a constant, or a question mark. The question mark introduces non-determinism into an affine program and is used when translating an if-statement or a while-statement with complex conditions

$$
\begin{array}{rrcl}
(\textit{Program}) & \textit{Prog} & ::= & f_1; f_2; ...; f_n \\
(\textit{Function}) & f & ::= & \text{fname}()\{\textsf{locals } x_1, ..., x_k;\ b\} \\
(\textit{Block}) & b & ::= & s_1; ...; s_n \\
(\textit{Statement}) & s & ::= & x = i \mid x = x + i \mid x = x + y \\
 & & \mid & \textsf{if } c \textsf{ then } \{b_1\} \textsf{ else } \{b_2\} \mid \textsf{while } c \textsf{ do } \{b\} \mid \textsf{assert } p \\
 & & \mid & (x_1, ..., x_n) = \text{fname}() \mid \textsf{return } (x_1, ..., x_n) \\
(\textit{Condition}) & c & ::= & p \mid ? \\
(\textit{Predicate}) & p & ::= & x == i \mid x \neq i \mid x > i \mid x < i \mid x \geq i \mid x \leq i
\end{array}
$$

Figure 4.5: Syntax of affine programs.

in C code. The statement "assert p" makes an assertion about predicate p. During affine translation, the translator inserts assertions about objects' refcount changes into the affine program.

There are also function-call and function-return statements. An affine function takes zero parameters and returns a tuple. As we will discuss, a native C function with n object-reference parameters is translated to an affine function that has zero parameters and returns a tuple with n components, which are the refcount changes of the n parameter objects.

Intraprocedural affine translation. The translation from C interface code into an affine program is syntax directed, translating one function at a time. We next explain how Pungi translates a C function.

Suppose the C function takes n parameters $p_1, ..., p_n$, each of which is a reference to a Python object. We assume unique numeric labels have been given to parameter objects and object creation sites in the C function. Assume there are m labels in total, ranging from 1 to m. Among those labels, the first n labels are given to the n parameter objects and the rest to objects created in the function.

C construct	map updates	affine translation
function entry	forall $i \in [1..n]$ $\text{map}(p_i) = i$	forall $i \in [1..m]$ $rc_i = 0$
$x = y$	map(x) = map(y)	none
`Py_INCREF`(x)		$rc_{\text{map}(x)}$ ++
`Py_DECREF`(x)		$rc_{\text{map}(x)}$ --
x = `PyInt_FromLong`$_l$(...)	$\text{map}(x) = l$	if ? then { $rc_l = 1$; $on_l = 1$ } else { $rc_l = 0$; $on_l = 0$ }
if (x == NULL) then $s1$ else $s2$		if $on_{\text{map}(x)} == 0$ then { $\mathcal{T}(s1)$ } else { $\mathcal{T}(s2)$ }
`return`		forall $i \in \text{OutScope}([1..m])$ assert ($rc_i == 0$) return ($rc_1, \ldots, rc_n$)
`f`($x_1, \ldots, x_k$)		($tmp_1, \ldots, tmp_k$) = f(); $rc_{\text{map}(x_1)}$ += tmp_1; ...; $rc_{\text{map}(x_k)}$ += tmp_k;

Figure 4.6: Affine translation $\mathcal{T}(-)$ for typical C constructs.

There are two important aspects about the affine translation. First, the translation maintains a *variable-object map* that maps from C variables to object labels; it tracks which object a C variable references at a program location. Second, for a Python object with label i, the affine program after translation uses a set of affine variables to track properties of the object. The most important one is the rc_i variable, which tracks the refcount change to the object. (Other affine variables will be described later.)

Fig. 4.6 presents the translation rules for typical C constructs. The first column of the table presents a C construct, the second column presents the updates to the variable-object map, and the last column contains the translation result.

At the function entry, the variable-object map is initialized to map from parameters to labels of parameter objects. Recall that with shallow aliasing the refcount change to an object is independent from the initial aliasing situation; this is why initially parameters are mapped to unique labels, essentially assuming they are non-aliases. In terms of

translation for the function entry, refcount changes for all objects are initialized to be zero.

Reference assignment $x = y$ results in an update to the variable-object map: afterwards, x references the same object as y. `Py_INCREF`(x) is translated to an affine statement that increments the rc variable of the object that x currently references. We use "$rc_i{+}{+}$" as an abbreviation for $rc_i = rc_i{+}1$. Similarly, `Py_DECREF`(x) is translated to a decrement on the corresponding rc variable.

The translation also translates Python/C API function calls. Such a translation required us to carefully read the Python/C reference manual about the refcount effects of API functions (and sometimes even required us to read the source code of the Python interpreter when the manual is unclear). One complication when translating API functions is the need to deal with error conditions, which are common in the Python/C interface. In the example in Fig. 4.3 on page 103, `PyInt_FromLong` is supposed to allocate an integer object, but the allocation may fail. The subsequent code tests whether the object is null and proceeds with two cases. Error conditions are typically signaled in the Python/C interface by returning a null reference. To deal with error conditions, Pungi introduces another affine variable for an object during translation: an object non-null variable, called the on variable. It is one when the object is non-null and zero when the object is null. Fig. 4.6 presents the translation of `PyInt_FromLong` with object label l. It is translated into a non-deterministic if-statement. In the case of an allocation success, the rc variable is set to be one and the on variable is also one (meaning it is non-null); in the failure case, both variables are set to be zero.

Pungi translates an if-statement in C code in a heuristic way. It recognizes a set of boolean conditions (testing for null, testing for nonnull, etc.) in the if-statement

```
buggy_foo () {
  locals rc1, on1;
  rc1 = 0;
  if (?) {rc1 = 1; on1 = 1} else {rc1 = 0; on1 = 0};
  if (on1 == 0) { assert (rc1 == 0); return ();}
  assert (rc1 == 0); return ();
}
```

Figure 4.7: Translation of the example in Fig. 4.3.

and translates those conditions accurately. Fig. 4.6 presents one such case when the condition is to test whether an object reference is null; the translated code tests the corresponding object's on variable. For complex boolean conditions, Pungi just translates them to question marks.

A return statement is translated to assertions about object refcount changes followed by the returning of a tuple of refcount changes of the parameter objects. We delay the discussion why the tuple of refcount changes is returned when we discuss the interprocedural translation. An assertion is inserted for every object that is about to go outside its scope. This is according to the bug definition we discussed in Section 4.4.1. The auxiliary $\mathrm{OutScope}$ function returns a set of labels whose corresponding objects are about to go outside their scopes. NC (Natively Created) objects created in the function being translated belong to this set. Parameter objects are also in this set if the function is an entry native function; that is, when they are PC (Python Created) objects.

We present in Fig. 4.7 the translation result for the function in Fig. 4.3. Since the original function takes no parameters, the resulting affine function returns an empty tuple. From the affine function, we can see that the last assertion fails, which implies a reference-counting error in the original function.

We note that the SSA transform makes the presented affine-translation possible.

Without the SSA, the variable-object map would possibly be updated differently in two different branches of a control-flow graph; then the translation would face the issue of how to merge two maps at a control-flow join point. After the SSA transform, an object-reference variable is statically assigned once and conflicts in variable-object maps never arise. The previous study [43] addressed the issue of variables being assigned multiple times by reversing the control flow during the affine translation. By performing the SSA transform first, Pungi simplifies the affine translation in the intraprocedural case and allows function calls that pass parameters.

Interprocedural translation. As we have seen, the affine function translated from a native function returns the refcount changes of parameter objects by assuming those parameter objects are distinct. This assumption, however, may not be true as the native function may be called with aliases and different call sites may have different aliasing situations. Fortunately, because of property (ii) in Section 4.4.3 (on page 106), it is possible to make *post-function-call refcount adjustments* according to the aliasing situation of a specific call site. The last entry in Fig. 4.6 describes how a function call is translated. First, the corresponding function is invoked and it returns the refcount changes of the parameter objects assuming they are distinct. After the function call, the rc variables of the parameter objects are adjusted according to the variable-object map of the caller.

Fig. 4.8 presents an example. Fig. 4.8(a) is some Python/C interface code, which has two functions. Function `foo` is assumed to be a native entry function. It invokes `bar` at two places. Fig. 4.8(b) is the translated affine program. Note that the post-function refcount adjustments are different for the two call sites. For the first call `f(x1,x1)`, the two refcounts are both added to `rc1`; for the second call `f(x2,x2)`, the two refcounts are both added to `rc2`.

```
void foo (PyObject *x1, PyObject *x2) {
  if (...) bar(x1, x1) else bar(x2, x2);
  return; }
void bar (PyObject *p1, PyObject *p2) {
  Py_IncRef(p1); Py_DecRef(p2); }
```

(a) Untransformed Python/C interface code

```
void foo () {
  locals rc1, on1, rc2, on2, tmp1, tmp2;
  rc1 = 0; rc2 = 0;
  if (?) {
    (tmp1, tmp2) = bar();
    rc1 += tmp1; rc1 += tmp2;
  } else {
    (tmp1, tmp2) = bar();
    rc2 += tmp1; rc2 += tmp2;}
  assert (rc1 == 0); assert (rc2 == 0);
  return (); }
bar () {
  locals rc1, on1, rc2, on2;
  rc1 = 0; rc2 = 0; rc1++; rc2--;
  return (rc1, rc2); }
```

(b) Transformed affine program

Figure 4.8: An example of interprocedural affine translation.

Another note about the interprocedural translation is that, if the SSA form of a native function has ϕ nodes, then the native function is translated to multiple affine functions with one affine function created for one ϕ node. In particular, for a ϕ node, the translation finds the set of nodes in the control-flow graph that are dominated by the ϕ node and are reachable from the ϕ node without going through other ϕ nodes. This set of nodes is then translated to become the body of the affine function created for the ϕ node. Afterwards, the affine function is lifted to be a function at the global scope (that is, lambda-lifting [37]). As an example, Pungi translates the following function to exactly the same affine program in Fig. 4.8(b). This is because after the SSA transform, there is a ϕ node inserted before line 4 and an additional affine function is created for that ϕ node.

```
1 void foo (PyObject *x1, PyObject *x2) {
2   PyObject *p1, *p2;
3   if (...) {p1=x1; p2=x1} else {p1=x2; p2=x2};
4   Py_IncRef(p1); Py_DecRef(p2);
5   return;
6 }
```

For the `ntuple` program in Fig. 4.1, since a ϕ node is inserted before the testing for loop condition (see Fig. 4.4), an affine function is created for the loop body; it makes a recursive call to itself because there is a control-flow edge back to the ϕ node because of the loop.

4.4.4 Escaping references

References to an object may escape the object's scope. In this case, the expected refcount change to the object is greater than zero. Object references may escape in several ways. A reference may escape via the return value of a function. Fig. 4.9(a) presents such an example. When the integer object is successfully created, the function returns the `pyo` reference. In this case, the refcount change to the integer object is one. A reference may also escape to the heap. The code in Fig. 4.1 on page 95 contains such an example. At line 14, The `item` reference escapes to the heap in the tuple object when the set-item operation succeeds. In that case, the refcount change to the object created at line 11 is also one.

To deal with escaping references, we revise the bug definition as follows:

Definition 5 *There is a reference-counting error if, at the end of the scope of an NC or PC object, its refcount change is not the same as the number of times references to the object escape. If the refcount change is greater than the number of escapes, we call it an error of reference over-counting. If the change is less than the number of escapes, we call it an error of reference under-counting.*

The previous bug definition with non-escaping references is a specialization of the new definition when the number of escapes is zero. The new definition essentially uses the number of escapes to approximate the number of new references created outside the object's scope. One limitation is that an object reference may escape to the same heap location multiple times and a later escape may overwrite the references created in earlier escapes. This would result in missed errors, although this happens rarely in practice as suggested by our experience with real Python/C extension modules.

```
PyObject* foo () {
  PyObject *pyo=PyInt_FromLong(10);
  if (pyo==NULL) {
    return NULL;
  }
  return pyo;
}
```

(a) Untransformed Python/C interface code

```
foo () {
  locals rc1,ev1,on1;
  rc1=0; ev1=0;
  if (?) {rc1=1; on1=1}
  else {rc1=0; on1=0};
  if (on1==0) {
    assert (rc1==ev1);
    return;
  }
  if (on1==1) ev1++;
  assert (rc1==ev1);
  return (rc1,ev1);
}
```

(b) Transformed affine program

Figure 4.9: An example of escaping references.

Given the new bug definition, the affine-translation step is adjusted in the following ways. First, an *escape variable*, ev, is introduced for each Python object and records the number of escapes. It is initialized to be zero at the beginning of a function. Second, the translator recognizes places where an object's references escape and increments the object's escape variable in the affine program by one. Third, assertions are changed to assert an objects' refcount change be the same as the number of escapes. Finally, a function not only returns the refcount changes of its parameter objects, but also returns the numbers of escapes of the parameter objects. The post-function-call adjustments adjust both the refcount changes and the numbers of escapes of the arguments.

Fig. 4.9(b) presents the translated result of the code in Fig. 4.9(a). Variable `ev1` is introduced to record the number of escapes for the integer object created. This example also illustrates that the number of escapes may be different on different control-flow paths.

One final note is that in Pungi, with the assumption of shallow aliasing, callers of functions that return a reference are assumed to get a reference to a new object. That is, a function call that returns a reference is treated as an object-creation site.

4.5 Affine analysis and bug reporting

The final step of Pungi is to perform analysis on the generated affine program and reports possible reference-counting errors. There are several possible analysis algorithms on affine programs, such as random interpretation [27]. Pungi adapts the ESP algorithm [17] to perform affine analysis. The major reason for choosing ESP is that it is both path-sensitive and interprocedural. The analysis has to be path sensitive to rule

out impossible paths. The affine program in Fig. 4.9(b) shows a typical example. In the statement "`if (on1==0) ...`", the analysis must be able to remember the path condition `on1==0` to rule out the impossible case where `rc1==1` and `on1==1`. Without that capability, the analysis would not be able to see that the first assertion always holds. The analysis also must be interprocedural as the affine program in Fig. 4.8 illustrates.

ESP symbolically evaluates the program being analyzed, tracks and updates symbolic states. At every program location, it infers a set of possible symbolic states of the following form:

$$\{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

In ESP, a symbolic state consists of a property state ps and an execution state es. The important thing about the split between property and execution states is that ESP is designed so that it is path- and context-sensitive only to the property states. Specifically, at a control-flow join point, symbolic states merge based on the property state; the execution states of all states that have the same property state are merged. By splitting property and execution states in different ways, we can control the tradeoff between efficiency and precision of the algorithm.

A particular analysis needs to decide how to split between property and execution states in ESP. We next discuss how they are defined in Pungi but leave the detailed algorithm to the ESP paper. When analyzing an affine program, Pungi's property state is the values of refcount-change variables and escape variables. The execution state is the values of all other variables.

Fig. 4.10 presents the analysis result at key program locations for the affine program in Fig. 4.9. As we can see, after the first if-statement, there are two symbolic states, representing the two branches of the if-statement. Then path sensitivity allows the anal-

```
foo () {
  locals rc1,ev1,on1;
  rc1=0; ev1=0;
  // {<[rc1=0,ev1=0], []>}
  if (?) {
    rc1=1; on1=1
    // {<[rc1=1,ev1=0], [on1=1]>}
  } else {
    rc1=0; on1=0
    // {<[rc1=0,ev1=0], [on1=0]>}
  };
  // {<[rc1=1,ev1=0], [on1=1]>, <[rc1=0,ev1=0], [on1=0]>}
  if (on1==0) {
    // {<[rc1=0,ev1=0], [on1=0]>}
    assert (rc1==ev1);
    return;
  }
  // {<[rc1=1,ev1=0], [on1=1]>}
  if (on1==1) ev1++;
  // {<[rc1=1,ev1=1], [on1=1]>}
  assert (rc1==ev1);
  return (rc1,ev1);
}
```

Figure 4.10: An example of affine analysis.

ysis to eliminate impossible symbolic states after the testing of `on1==0` in the second if-statement.

We note that ESP was originally designed with a finite number of property states, while values of refcount changes and escapes can be arbitrarily large. In our implementation, we simply put a limit on those values (10 in our implementation) and used a top value when they go out of the limit.

4.6 Implementation and limitations

We have built a prototype implementation of Pungi. The implementation is written in OCaml within the framework of CIL [65], which is a tool that allows analysis and transformation of C source code. Pungi's prototype implementation cannot analyze C++ code because CIL can parse only C code. Passes are inserted into CIL to perform the separation of interface code from library code, the SSA transform, the affine translation, and the affine analysis. Our implementation of the SSA transform follows the elegant algorithm by Aycock and Horspool [6]. The total size of the implementation is around 5,000 lines of OCaml code.

Pungi also needs to identify entry native functions because assertions about parameter objects are inserted only to entry functions. Native extensions typically have a registration table to register entry functions to Python statically. Pungi searches for the table and extracts information from the table to identify entry functions. Since Python is a dynamically typed language, a native extension module can also dynamically register entry functions. Therefore, Pungi also uses some heuristics to recognize entry functions. In particular, if a function uses `PyArg_Parse` (or several other similar functions) to decode arguments, then it is treated as an entry function.

Limitations. Before we present the evaluation results of Pungi, we list its major limitations. We will discuss our plan to address some of these limitations when discussing future work. Some of these limitations have been discussed before, but we include them below for completeness.

First, Pungi assumes shallow aliasing. Whenever an object reference is retrieved from a collection object such as a list, read from a field in a struct, or returned from a

function call, the reference is assumed to point to a distinct object; such a site is treated as an object-creation site.

Second, Pungi reports errors assuming Python invokes entry native functions with distinct objects. This is reflected by the fact that an assertion of the form $rc = ev$ is inserted for every parameter object of an entry native function. This assumption can be relaxed straightforwardly and please see discussion in future work.

Third, Pungi's bug definition may cause it to miss some dangling reference errors in the middle of functions, because assertions are inserted only at the end of functions.

Finally, a native extension module can call back Python functions through the Python/C API, resulting in a Ping-Pong behavior between Python and native code. An accurate analysis of such situations would require analyzing both Python and C code. On the other hand, we have not encountered such code in our experiments.

4.7 Evaluation

We selected 13 Python/C programs for our evaluation. These programs are common Python packages in Fedora OS and they use the Python/C interface to invoke the underlying C libraries. Brief descriptions of these benchmark programs are shown in Table 1.2. One major reason we selected those programs for evaluation is that a previous tool, CPyChecker [56], has reported its results on those programs and we wanted to compare Pungi's results with CPyChecker's. All evaluation was run on a Ubuntu 9.10 box with 512MB memory and 2.8GHz CPU.

Table 4.1 lists the selected benchmarks, their sizes in terms of thousands of lines of

Benchmark	Total (KLOC)	Interface code (KLOC)	Time (s)
krbV	7.0	3.7	0.78
pycrypto	16.6	7.0	1.32
pyxattr	1.0	1.0	0.09
rrdtool	31.4	0.6	0.01
dbus	93.1	7.0	0.66
gst	2.7	1.8	0.03
canto	0.3	0.2	0.001
duplicity	0.5	0.4	0.001
netifaces	1.1	1.0	0.09
pyaudio	2.9	2.7	0.03
pyOpenSSL	9.6	9.3	1.27
ldap	3.8	3.4	0.23
yum	3.0	2.4	0.20
TOTAL	173	40.5	4.7

Table 4.1: Statistics about selected benchmark programs.

code (KLOC), sizes of their interface code (recall that the first step Pungi performs is to separate interface from library code), and the amount of time Pungi spent on analyzing their code for reference-counting errors. The time is an average of ten runs. As we can see, Pungi is able to analyze a total of 173K lines of code in a few seconds, partly thanks to the separation between interface and library code.

The main objective in our evaluation is to know how effective our tool is in identifying the reference-counting errors as defined. This includes the number of bugs Pungi reports, the false positive rate, and the accuracy of our tool compared to CPyChecker.

Errors found

For a benchmark program, Table 4.2 shows the number of warnings issued by Pungi, the numbers of true reference over- and under-counting errors, and the number of false positives. For the 13 benchmark programs, Pungi issued a total of 210 warnings, among which there are 142 true reference over-counting errors and a total of 22 true reference

Benchmark	All Warnings	Reference Over-counting	Reference Under-counting	False Positives (%)
krbV	85	74	0	11 (13%)
pycrypto	10	6	1	3 (30%)
pyxattr	4	2	0	2 (50%)
rrdtool	0	0	0	0 (0%)
dbus	3	1	0	2 (67%)
gst	30	12	13	5 (17%)
canto	6	0	4	2 (33%)
duplicity	4	2	0	2 (50%)
netifaces	8	2	1	5 (63%)
pyaudio	35	28	2	5 (14%)
pyOpenSSL	9	3	1	5 (56%)
ldap	15	11	0	4 (27%)
yum	1	1	0	0 (0%)
TOTAL	210	142	22	46 (22%)

Table 4.2: All warnings reported by Pungi, which include true reference over- and under-counting errors and false positives.

under-counting errors. We manually checked all true errors to the best of our ability via a two-person team. Common errors reported by both CPyChecker and Pungi have been reported to the developers by the CPyChecker author and some of those errors have been fixed in later versions of the tested benchmarks. Most of the additional true errors found by Pungi were easy to confirm manually.

There are 46 false positives and the overall false-positive rate is moderate, about 22%. We investigated those false positives and found most false positives are because of the following reasons:

- Object references in structs. With the assumption of shallow aliasing, Pungi treats the assignment of an object reference to a field in a struct as an escape of the reference, and treats the reading an object reference from a field of a struct as returning a reference to a new object. For example, in the following code `p` and `q` would reference two distinct objects in Pungi's analysis.

```
f->d = p;
q = f->d;
```

As a result, Pungi loses precision when tracking refcounts in such cases. This may cause both false positives and false negatives and it contributes to the majority (22 in total) of all the false positives seen in packages such as `pycrypto` and `ldap`.

- Type casting. Pungi treats references of `PyObject` type (and its subtytpes such as `PyLongObject`, `PyIntObject`, and `PyStringObeject`) as references to Python objects. In some package code, a Python object reference is cast into another type such as an integer and then escapes to the heap. Pungi's affine translation cannot model this casting and would incorrectly issue a reference over-counting warning. 20 false positives in packages such as `gst` and `pyOpenSSL` were caused by this reason.

Comparison with CPyChecker

Table 4.3 shows the comparison of errors found between Pungi and CPyChecker. We looked into the differences and found that Pungi found all errors reported by CPyChecker. In addition, Pungi found 50 more errors than CPyChecker. The reason is because Pungi employs more precise analysis that applies the SSA and analyzes loops as well as function calls. CPyChecker's analysis is intraprocedural and ignores loops. We categorize the causes in the table. In the column Common, we put the number of errors that are reported by both Pungi and CPyChecker. Column MA (Multiple Assignments) shows the number of errors that Pungi found but missed by CPyChecker because CPyChecker's implementation cannot deal with the case when variables are statically assigned multiple times with different object references; Pungi can deal with this by the

Benchmark	Pungi				CPyChecker
	Common	MA	Proc	Loop	Errors found
krbV	39	33	1	1	39
pycrypto	6	0	1	0	6
pyxattr	2	0	0	0	2
rrdtool	0	0	0	0	0
dbus	1	0	0	0	1
gst	21	2	0	2	21
canto	4	0	0	0	4
duplicity	2	0	0	0	2
netifaces	3	0	0	0	3
pyaudio	25	3	1	1	25
pyOpenSSL	1	3	0	0	1
ldap	8	3	0	0	8
yum	1	0	0	0	1
TOTAL	112	43	3	4	112

Table 4.3: Comparison of errors found between Pungi and CPyChecker.

SSA transform. Column Proc shows the number of errors Pungi found but missed by CPyChecker because it cannot perform interprocedural analysis. Column Loop shows the number of errors Pungi found but missed by CPyChecker because it cannot analyze loops. The comparison shows that Pungi compares favorably to CPyChecker.

4.8 Summary

We have described Pungi, a static-analysis tool that identifies reference-counting errors in Python/C extension modules. It translates extension code to an affine program, which is analyzed for errors of reference counting. Pungi's affine abstraction is novel in that it applies the SSA transform to simplify affine translation and in that it can deal with the interprocedural case and escaping references. The prototype implementation found over 150 bugs in over 170K lines. We believe that Pungi offers an efficient and accurate tool for statically analyzing code that uses reference counting to manage resources.

CHAPTER 5

RELATED WORK

This dissertation work belongs to the general category of improving 1) software quality, 2) software and system engineering, 3) static analysis, and 4) FFIs' safety, reliability, and security. In this section, we discuss systems that are closely related to this dissertation.

5.1 FFIs

Almost all widely used programming languages support a Foreign Function Interface (FFI) for interoperating with program modules developed in low-level code (*e.g.*, [51, 72, 47, 9, 13, 23]). Early work on FFIs were mostly concerned with how to design an FFI and how to provide efficient implementation.

FFI-based software is often error-prone. In recent years, researchers studied how to improve upon FFIs' safety, reliability, and security. FFI-based code is often a rich source of software errors: a few recent studies reported hundreds of interface bugs in JNI programs ([26, 40, 49]). Errors occur often in interface code because FFIs generally provide little or no support for safety checking, and also because writing interface code requires resolving differences (*e.g.*, memory models and language features) between two languages. Past work on improving FFIs' safety can be roughly classified into several categories: (1) Static analysis has been used to identify specific classes of errors in FFI code [25, 26, 87, 40, 49]; (2) In another approach, dynamic checks are inserted at the language boundary and/or in the native code for catching interface errors (see [45]) or for isolating errors in native code so that they do not affect the host language's safety [85] and security [80]; (3) New interface languages are designed to help programmers write

safer interface code (*e.g.*, [31]). TurboJet takes the static-analysis approach, which is well-suited for finding bugs in exceptional cases.

5.2 Work related to TurboJet

We next compare our exception analysis work (TurboJet) in more detail with three closely related work on using static analysis to find bugs in JNI programs [26, 40, 49]. Our previous system [49] and a system by [40] identify situations of mishandling JNI exceptions in native code. Both systems compute at each program location whether there is a possible JNI exception pending. However, they do not compute specific classes of pending exceptions. To do that, TurboJet has to use a much more complicated static analysis. The analysis tracks information of C variables in native code and correlates them with exception states; it also takes Java method signatures into account for tracking exceptions that may be pending when invoking a Java method. Both are necessary for computing exception effects. J-Saffire [26] identifies type misuses in JNI-based code but does not compute exception effects for native methods. J-Saffire also finds it necessary to track information of C variables that hold Java references. To deal with context sensitivity required for analyzing JNI utility functions, J-Saffire performs polymorphic type inference that is based on semi-unification, while TurboJet uses a context-sensitive dataflow analysis. Since TurboJet's context sensitivity uses call strings of length one, there are cases when J-Saffire's type inference can infer more precise information than TurboJet's. For instance, if a string constant is passed two levels down in a function-call chain and used as an argument to `FindClass`, then TurboJet cannot infer the exact Java type of the resulting reference. In practice, however, we did not find this results in noticeable imprecision in TurboJet.

Many systems perform exception analysis for languages that provide built-in support for exceptions. For example, the Jex tool [74] and others (*e.g.*, [55, 14]) can compute what kinds of exceptions can reach which program point for Java programs. This information is essential for understanding where exceptions are thrown and caught. TurboJet computes this information for JNI programs

Our system TurboJet employs static taint analysis to track "bad" data in exceptional situations. Taint analysis, either static (*e.g.*, [54, 38, 91, 5, 92]) or dynamic (*e.g.*, [67, 93, 68]), has been successfully applied to preventing a range of attacks (*e.g.*, format string attacks [77]). Static taint analysis does not incur runtime overhead as dynamic analysis does, but may report false errors. A promising hybrid strategy proposed by [15] performs static taint analysis as much as it can, but leaves difficult cases for dynamic analysis. One technical difference between our static taint analysis and previous ones is that it depends on a pointer graph. The pointer graph (described in Section 2.6.1) both approximates taint propagation and is used to cope with aliases; previous systems [54, 15] have separate taint propagation and pointer-analysis modules.

5.3 Work related to JATO

The benefits of static reasoning of atomicity in programming languages were demonstrated by Flanagan and Qadeer [24] through a type effect system. Since then, many static systems have been designed to automatically insert locks to enforce atomicity: some are type-based [58, 41]; some are based on points-to graphs [30]; some reduce the problem to an ILP optimization problem [20]. Among them, JATO's approach is more related to [41]. Unlike that approach where the focus is on the interaction between lan-

guage design and static analysis, JATO focuses on static analysis in a mixed language setting.

Atomicity can either be implemented via locks (*e.g.*, the related work above) or by transactional memory (TM) [29]. Related to our work JATO are two concepts articulated in TM research: *weak atomicity* and *strong atomicity* [57]. In a system that supports weak atomicity, the execution of an atomic program fragment exhibits serial behaviors *only* when interleaving with that of other atomic program fragments; there is no guarantee when the former interleaves with *arbitrary* executions. To support the latter, *i.e.*, strong atomicity, has been a design goal of many later systems (*e.g.*, [12]). Most existing strong atomicity algorithms would disallow native methods to be invoked within atomic regions, an unrealistic assumption considering a significant number of Java libraries are written in native code for example. Should they allow for native methods but ignore their impact these approaches would revert back to what they were aimed at solving: weak atomicity.

In a software transactional memory setting where the atomicity region is defined as atomic blocks, *external actions* [28] are proposed as a language abstraction to allow code running within an atomic block to request that a given pre-registered operation (such as native method invocation) be executed outside the block. In the "atomicity-by-default" language AME [4], a `protected` block construct is introduced to allow the code within the block to opt out of the atomicity region. Native methods are cited as a motivation for this construct. Overall, these solutions focus on how to faithfully model the non-atomicity of native methods, not how to support their atomicity.

5.4 Work related to Pungi

Reference counting is a general approach in tracking the number of references belong to a object allocated in memory. It is typically used in implementing dynamic garbage collection and memory management in programming languages and operating systems. There have been several systems designed and built in finding reference count bugs, such as [21].

Malcom has constructed a practical tool called CPyChecker [56], which is a gcc plug-in that can find a variety of errors in Python's native extension modules, including reference-counting errors. CPyChecker traverses a finite number of paths in a function and reports errors on those paths. It does not perform interprocedural analysis and ignores loops, while Pungi covers both. CPyChecker also produces wrong results when a variable is statically assigned multiple times, while Pungi uses SSA to make variables assigned only once. Experimental comparison between Pungi and CPyChecker is presented in the evaluation section.

Emmi *et al.* have used software model checking to find reference-counting errors in an OS kernel and a file system [22]. Their system's focus, assumptions, and techniques are quite different from Pungi's. The focus of their system is to find reference-counting errors in the presence of multiple threads. It assumes there is an array of reference-counted resources and assumes each resource in the array is used uniformly by a thread. Therefore, their system can use a technique called temporal case splitting to reduce the reference-counting verification of multiple resources and multiple threads to the verification of a single resource and a single thread. In the context of Python/C, however, objects passed from Python are not used uniformly by native code: an object's refcount may be adjusted differently from how other objects' refcounts are adjusted. Pungi uses

an affine program to capture the effects of reference counts on objects. Another note is that the system by Emmi *et al.* assumes simple code for adjusting refcounts and has not dealt with any aliasing situation (including shallow aliasing).

Python/C interface code can also be generated by tools such as SWIG [7] and Cython [1]. They would reduce the number of reference-counting errors as most of the interface code is automatically generated. However, these tools do not cover all possible cases of code generation; in particular, they do not handle every feature of C/C++. As a result, a lot of interface code is still written manually in practice.

The work of Pungi is an example of finding errors in Foreign Function Interface (FFI) code. Errors occur often in FFI code [26, 87, 40, 49] because writing interface code requires resolving language differences such as memory management between two languages. Past work on improving FFIs' safety can be put into several categories. First, some systems use dynamic checking to catch errors (*e.g.*, [45]), to enforce atomicity [48], or to isolate errors in native code so that they do not affect the host language's safety and security [80, 85]. Second, some researchers have designed new interface languages to help programmers write safer interface code (*e.g.*,[31]). Finally, static analysis has been used to identify specific classes of errors in FFI code, including type errors [25, 26] and exception-handling errors [49, 50]. Pungi belongs to this category and finds reference-counting errors in Python/C interface code.

Pungi uses affine programs to abstract the reference-counting aspect of Python/C programs and performs analysis on the resulting affine programs. Affine analysis has been used in program verification in the past (*e.g.*,[27, 63, 39, 62, 18]).

Our system Pungi is inspired by the idea presented in [42]. This paper proposes a technique to perform reference count analysis for the case of shallow pointers, which

disallows multi-level pointers (*e.g.*, pointers to pointers, or pointers to objects that contains pointers). The property this paper verifies is to check the number of reference increments is the same as the number of reference decrements in every control-flow path. This property is suitable for whole-program analysis but wouldn't be sufficient for modular analysis, which is needed when we analyze a C function that interoperates with Python.

Using affine analysis in program verification has also been studied in the past, such as [27, 64, 39, 62, 18].

CHAPTER 6

FUTURE WORK

Many interesting problems remain to be solved in improving quality of software composed of FFIs.

As a natural extension to the study of multithreaded safety in an FFI, we can further investigate how to ensure locking inserted by JATO does not cause deadlocks (even though we didn't encounter such cases during our experiment), probably using the approach of a global lock order as in Autolocker [58]. Moreover, we believe that JATO's approach can be generalized to other FFIs such as the OCaml/C interface [47] and the Python/C interface [72].

We also plan to generalize Pungi to relax some of its assumptions. Pungi assumes shallow aliasing and assumes parameter objects to entry native functions are distinct objects. One possibility is to report errors for any possible aliasing situation, by adding nondeterminism into native functions. As one example, suppose an entry native function takes two parameter objects referenced by `p1` and `p2`, respectively. Suppose the function can be called either with `p1` and `p2` referencing two distinct objects or with `p1` and `p2` referencing the same object. We can insert the following code at the beginning of the native function before translation: "`if (?) {p1=p2}`", which nondeterministically initializes `p1` and `p2` for the two aliasing situations. As another example, after an object is retrieved from a list, we can nondeterministically assume the object can be a new object, or any existing object. This approach can be further improved if Python and C code are analyzed together and some alias analysis is used to eliminate impossible aliasing situations. Another possible approach to relax the shallow aliasing assumption is to keep and maintain a set of finite access paths to each Python object, as suggested

in [76].

Exploring beyond using static analysis alone is another promising direction. Besides static analysis, dynamic analysis and even hybrid analysis that combines both static analysis and dynamic analysis can be considered for finding bugs in software composed of FFIs. For example, in taint analysis, there are several studies that use static analysis to target the most possible taint targets and taint sinks, while using dynamic analysis to find the possible taint path, which can be quite efficient. The combination of using static analysis and dynamic analysis is a common practice in the real world. Programmer commonly use static analysis tools, such as FindBugs [32], to find some bugs. They then use unit tests (*i.e.*, a form of dynamic analysis) to find other types of bugs that the static analysis tools cannot find. We believe systems that are designed with a hybrid approach can be more powerful and comprehensive in solving software quality issues.

In our study, we focus on using static analysis exclusively on program source code. Another possible approach is to apply static analysis on program binary code. Static analysis of software source code needs to consider lots of language-specific features, such as referencing a C struct's field (as discussed in Section 2.7.1). Static analysis of binary code in general does not face this problem. It can potentially offer language-agnostic solutions hence more flexible solutions to some of the problems we have explored in this dissertation. However, the study of static analysis in binary code is still in its early stage; there are many basic problems to be resolved and issues to be addressed. These include complexity of binary code, lack of high-level semantics, and code obfuscation [81]. These challenges at the same time make application of static analysis in binary code both an interesting research topic and a potentially rewarding research venture.

CHAPTER 7

CONCLUDING REMARKS

Foreign Function Interfaces (FFIs) bring convenience and efficiency to software development, but at the same time can introduce software quality issues.

In this dissertation, we use two FFIs – the Java Native Interface (JNI) and the Python/C interface – to show bug definitions and examples of bug patterns that can cause software quality issues. We present complete bug finding systems that are built with static analysis. In the process, we propose improvements of several static analyses and demonstrate that our novel designs of finding bugs are effective and efficient. Our systems have found a number of real bugs in popular FFI software applications. With these research findings and proposed solutions, we are making a solid step toward better software quality with FFI code. We believe that the techniques presented in this dissertation are applicable to other environments such as Objective-C, OCaml, and .NET.

BIBLIOGRAPHY

[1] Cython. http://cython.org/.

[2] Fedora build systems. http://arm.koji.fedoraproject.org/koji/index.

[3] Turbojet eclipse plug-in.
`https://www.dropbox.com/sh/gxhg5rx29pb6092/vOWZw3fsZK`.

[4] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, 2008.

[5] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy (S&P)*, pages 143–159, Washington, DC, USA, 2002. IEEE Computer Society.

[6] John Aycock and Nigel Horspool. Simple generation of static single-assignment form. In *Proceedings of the 9th International Conference on Compiler Construction*, pages 110–124, 2000.

[7] David M. Beazley. *SWIG Users Manual: Version 1.1*, June 1997.

[8] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *24th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 81–96, 2009.

[9] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, 2001.

[10] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *38th ACM Symposium on Principles of Programming Languages (POPL)*.

[11] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *International Conference on Software engineering (ICSE)*, pages 242–251, 2006.

[12] B. CarlStrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *27th ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[13] (Ed.) Manuel Chakravarty. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2005.

[14] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for Java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, New York, NY, USA, 2001. ACM.

[15] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *15th ACM Conference on Computer and Communications Security (CCS)*, pages 39–50, 2008.

[16] `http://wiki.eclipse.org/CDT/designs/StaticAnalysis`, 2011.

[17] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *23th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[18] Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. Abstract domains of affine relations. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 198–215, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Christopher Elford. Integrated debugger for Java/JNI environments. `http://software.intel.com/en-us/articles/integrated-debugger-for-javajni-environments/`, October 2010.

[20] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 291–296, 2007.

[21] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 352–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 352–367, 2009.

[23] Kathleen Fisher, Riccardo Pucella, and John H. Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1):3–19, 2001.

[24] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *24th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[25] Michael Furr and Jeffrey Foster. Checking type safety of foreign function calls. In *26th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.

[26] Michael Furr and Jeffrey Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.

[27] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *30th ACM Symposium on Principles of Programming Languages (POPL)*, pages 74–84, New York, NY, USA, 2003. ACM.

[28] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, December 2005.

[29] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.

[30] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *In First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT) '06*, June 2006.

[31] Martin Hirzel and Robert Grimm. Jeannie: Granting Java Native Interface devel-

opers their wishes. In *22th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.

[32] David Hovemeyer and William Pugh. Finding bugs is easy. In *the Companion to the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[33] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java - a minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[34] The java-gnome user interface library. http://java-gnome.sourceforge.net, 2011.

[35] http://javalib.gforge.inria.fr, 2011.

[36] `https://jogl.dev.java.net`, 2011.

[37] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *2nd International Conference on Functional Programming Languag es and Computer Architecture*, pages 190–203, New York, September 1985. Springer-Verlag.

[38] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy (S&P)*, pages 258–263, 2006.

[39] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[40] Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

[41] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *25th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2010.

[42] Akash Lal and G. Ramalingam. Reference count analysis with shallow aliasing. *Inf. Process. Lett.*, 111(2):57–63, December 2010.

[43] Akash Lal and G. Ramalingam. Reference count analysis with shallow aliasing. *Information Processing Letters*, 111(2):57–63, December 2010.

[44] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn McKinley. Debug all your code: A portable mixed-environment debugger for Java and C. In *24th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 207–226, 2009.

[45] Byeongcheol Lee, Martin Hirzel, Robert Grimm, Ben Wiedermann, and Kathryn S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *31th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 36–49, 2010.

[46] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

[47] Xavier Leroy. *The Objective Caml system*, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[48] Siliang Li, David Yu Liu, and Gang Tan. JATO: Native code atomicity for Java. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS '12)*, 2012.

[49] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *16th ACM Conference on Computer and Communications Security (CCS)*, pages 442–452, 2009.

[50] Siliang Li and Gang Tan. JET: Exception checking in the Java Native Interface. In *26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–358, 2011.

[51] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[52] Java-Readline. `http://java-readline.sourceforge.net`, 2003.

[53] Yu David Liu, Xiaoqi Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity. In *CC'08: International Conference on Compiler Construction*, March 2008.

[54] Benjamin Livshits and Monica Lam. Finding security vulnerabilities in Java applications with static analysis. In *14th Usenix Security Symposium*, pages 271–286, 2005.

[55] Donna Malayeri and Jonathan Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2006.

[56] David Malcolm. cpychecker. https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html.

[57] Milo M. K. Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[58] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 346–358, 2006.

[59] messAdmin. `http://messadmin.sourceforge.net/`.

[60] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18, 2013.

[61] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[62] Markus Müller-Olm and Oliver Rüthing. On the complexity of constant propagation. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 190–205, London, UK, UK, 2001. Springer-Verlag.

[63] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 330–341, 2004.

[64] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.

[65] George Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[66] George Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.

[67] James Newsome and Dawn Song. Dynamic taint analysis for automatic dedection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium(NDSS)*, 2005.

[68] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, pages 372–382, 2005.

[69] Michael Norrish. *Formalising C in HOL*. PhD thesis, University of Cambridge, 1998.

[70] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *6th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 146–161, 1991.

[71] Posix for Java. `http://bmsi.com/java/posix/`, 2009.

[72] Python/C API reference manual. `http://docs.python.org/c-api/index.html`, April 2009.

[73] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[74] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution

of exception structure in object-oriented systems. *ACM Transactions on Programming Languages and Systems*, 12(2):191–221, 2003.

[75] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *30th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, 2009.

[76] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *Science of Computer Programming*, 58(12):264 – 289, 2005. Special Issue on the Static Analysis Symposium 2003 SAS03 10th International Static Analysis Symposium.

[77] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[78] M. Sharir and A. Pnueli. Two approaches to inter-procedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall Inc., 1981.

[79] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. CMU-CS-91-145.

[80] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the JVM. In *17th ACM Conference on Computer and Communications Security (CCS)*, pages 201–211, 2010.

[81] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung

Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*, Hyderabad, India, December 2008.

[82] SPECjvm2008. `http://www.spec.org/jvm2008/`.

[83] JNI Binding to FlushSpread and Spread. `http://gsd.di.uminho.pt/members/jop/spread-jni`, 2004.

[84] Gang Tan. JNI Light: An operational model for the core JNI. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 114–130, 2010.

[85] Gang Tan, Andrew Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[86] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *17th Usenix Security Symposium*, pages 365–377, 2008.

[87] Gang Tan and Greg Morrisett. ILEA: Inter-language analysis across Java and C. In *22th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 39–56, 2007.

[88] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[89] Piotr Wendykier and James G. Nagy. Parallel colt: A high-performance java li-

brary for scientific computing and image processing. *ACM Trans. Math. Softw.*, 37(3):31:1–31:22, September 2010.

[90] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *25th ACM Conference on Programming Language Design and Implementation (PLDI)*.

[91] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th Usenix Security Symposium*, pages 179–192, Berkeley, CA, USA, 2006. USENIX Association.

[92] Wang Xinran, Jhi Yoon-Chan, Zhu Sencun, and Liu Peng. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 289–298, Washington, DC, USA, 2008. IEEE Computer Society.

[93] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *15th Usenix Security Symposium*, pages 121–136, 2006.

[94] Haitao Steve Zhu and Yu David Liu. Scalable object locality analysis with cypress principle. Technical report, SUNY Binghamton, May 2012.

APPENDIX A

WHITELIST

- A list of JNI functions that are safe to call with a pending exception (specified in the JNI Manual [51]):

  `ExceptionOccurred`,

  `ExceptionDescribe`,

  `ExceptionClear`,

  `ExceptionCheck`,

  `ReleaseStringChars`,

  `ReleaseStringUTFchars`,

  `ReleaseStringCritical`,

  `Release⟨Type⟩ArrayElements`,

  `ReleasePrimitiveArrayCritical`,

  `DeleteLocalRef`,

  `DeleteGlobalRef`,

  `DeleteWeakGlobalRef`,

  `MonitorExit`,

  `PushLocalFrame`,

  `PopLocalFrame`.

- The return operation and memory free operation.

APPENDIX B

INTERPROCEDURAL EXCEPTION ANALYSIS

We first describe some notations. A symbolic state $S = D \times X$, where a property state $D = \{$ `NoExn`, `ChkedExn` $E_1, \ldots,$ `ChkedExn` E_n, `UnChkedExn` $UE_1, \ldots,$ `UnChkedExn` $UE_m \}$, and an execution state X is a map from variables to values in a constant-propagation lattice or Java types. Given a symbolic state s, $ps(s)$ is its property state and $es(s)$ is its execution state.

A global control-flow graph = $[N, E, F]$, where N is a set of nodes, E is the set of edges, and F is the set of functions. Notation $src(e)$ denotes edge e's source node and $dst(e)$ the destination node. For node n, notation $In_0(n)$ stands for its first incoming edge and $In_1(n)$ the second (if there is one). A merger node is assumed to have two incoming edges. For a non-branch node, $Out_T(n)$ stands for its only outgoing edge. For a branch node, $Out_T(n)$ stands for the true branch and $Out_F(n)$ the false branch. We assume each function has a distinguished entry node, denoted by $entryNode(f)$. Notation $fn(n)$ denotes the function that node n belongs to. When n stands for a function-call node, $callee(n)$ denotes the callee function.

α denotes a function that merges a set of symbolic states:

$$\alpha(ss) = \{ \langle d, \bigsqcup_{s \in ss[d]} es(s) \rangle \mid d \in roots(ss) \}$$

where

$$ss[d] = \{\, s \mid s \in ss \ \wedge\ ps(s) <: d \,\}$$

$$roots(ss) = \{\, d \mid ss[d] \neq \emptyset \ \wedge\ \forall \langle d', es' \rangle \in ss.\ \neg(d <: d') \,\}$$

and $<:$ denotes the subclass relation

We use F to denote transfer functions for nodes. For instance, F_{Merge} is the transfer function for merger nodes.

- $F_{Merge}(n, ss_1, ss_2) = \alpha(ss_1 \bigcup ss_2)$.
- $F_{Call}(f, ss, \bar{a}) = \alpha(\{s' | s' = f_{Call}(f, s, \bar{a}) \wedge s \in ss\})$, where f_{Call} binds parameters of f to the symbolic-evaluation results of arguments $\bar{a}$ in symbolic state s; it also takes care of scoping by removing bindings for variables not in the scope of f.
- $F_{Branch}(n, ss, v) = \alpha(\{\, s' | s' = f_{Branch}(n, s, v) \wedge s \in ss \wedge es(s') \neq \bot \,\})$, where f_{Branch} takes advantage of the fact that the result of the branching condition is v and adjusts the symbolic state s.
- $F_{Exit}(f, ss) = \alpha(\{\, s' \mid s' = f_{Exit}(f, s) \wedge s \in ss \,\})$, where f_{Exit} takes care of scoping by removing variables that are only in the scope of f from the symbolic state.
- We use F_{JNI} to denote the transfer functions for JNI functions and we use F_{Other} to denote the transfer function for all other nodes.

The algorithm is formally described in Algorithms 1 and 2.

Algorithm 1: Auxiliary procedures

procedure *Add*(e, n_c, d, ss)

  if *Info*$(e, n_c, d) \neq ss$

    Info$(e, n_c, d) := ss$;

    Worklist := *Worklist* $\bigcup \{[dst(e), n_c, d]\}$;

  end if

end procedure

procedure *AddTrigger*$(n, n_c, d, ss, \bar{a})$

  $ss' := F_{Call}(fn(n), ss, \bar{a})$;

  $e := Out_T(n)$;

  $ss' := \alpha(ss' \bigcup$ *Info*$(e, n_c, d))$;

  Add(e, n_c, d, ss');

end procedure

procedure *AddToSummary*(n, n_c, d, ss)

  $ss' = F_{Exit}(fn(n), ss)$;

  if *Summary*$(fn(n), n_c, d) \neq ss'$

    Summary$(fn(n), n_c, d) := ss'$;

    for each $n'_c, d' \in D$ such that *Info*$(In_0(n_c), n'_c, d') \neq \emptyset$ **do**

      Worklist := *Worklist* $\bigcup \{[n_c, n'_c, d']\}$;

    end for

  end if

end procedure

Algorithm 2: Interprocedural exception analysis

Input:

Global control-flow graph=$[N, E, F]$;

$f_{entry} \in F$ is an entry function to be analyzed

Globals:

Worklist : $2^{N \times N \times D}$;

Info : $(E \times N \times D) \rightarrow 2^S$;

Summary : $(F \times N \times D) \rightarrow 2^S$;

procedure $solve$

$\forall\, e, n, d, \mathit{Info}(e, n, d) = \emptyset$;

$\forall\, f, n, d, \mathit{Summary}(f, n, d) = \emptyset$;

$e := Out_T(entryNode(f_{entry}))$;

$\mathit{Info}(e, \_\mathrm{m}, \texttt{NoExn}) := \{[\texttt{NoExn}, \top]\}$;

$\mathit{Worklist} := \{[dst(e), \_\mathrm{m}, \texttt{NoExn}]\}$;

while *Worklist* $\neq \emptyset$

Remove $[n, n_c, d]$ from *Worklist*;

$ss_{in} := \mathit{Info}(In_0(n), n_c, d)$;

switch (n) **do**

case $n \in$ *Merge*

$ss_{out} := F_{Merge}(n, ss_{in}, \mathit{Info}(In_1(n), n_c, d))$;

case $n \in$ *Branch*

$\mathit{Add}(Out_T(n), n_c, d, F_{Branch}(n, ss_{in}, T))$;

$\mathit{Add}(Out_F(n), n_c, d, F_{Branch}(n, ss_{in}, F))$;

case $n \in$ *JNIFun*

$\mathit{Add}(Out_T(n), n_c, d, F_{JNI}(n, ss_{in}, n_c, d))$;

case $n \in \mathit{Call}(\bar{a})$

$ss_{out} := \emptyset$;

for each $d' \in D$ such that $ss_{in}[d'] \neq \emptyset$ **do**

$sm := \mathit{Summary}(callee(n), n, d')$;

if $sm \neq \emptyset$ **then**

$ss_{out} := ss_{out} \bigcup sm$;

end if

$\mathit{AddTrigger}(entryNode(callee(n)), n, d', ss_{in}[d'], \bar{a})$;

end for

$\mathit{Add}(Out_T(n), n_c, d, \alpha(ss_{out}))$;

case $n \in$ *Exit*

$\mathit{AddToSummary}(n, n_c, d, ss_{in})$;

case $n \in$ *Other*

$ss_{out} := F_{Other}(n, ss_{in}, n_c, d)$;

$\mathit{Add}(Out_T(n), n_c, d, ss_{out})$;

end while

return *Info*

end procedure

BRIEF BIOGRAPHY

Siliang Li was born in the city of Dalian in Liaoning Province of the Peoples Republic of China. There, he attended the Dalian No. 44 junior high school and No. 24 senior high school. At age 17, he came to the United States under a Sino-US cultural exchange program. He lived in Stockton, California and attended Amos Alonzo Stagg High School. After which, Siliang attended and graduated with a B.S. in Computer Engineering from the University of Arizona and an M.S. in Computer Engineering from Lehigh University.

Professionally, Siliang is a co-author of several peer-reviewed publications that have appeared in prestigious venues like ACM Computer and Communication Security (CCS), ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA), European Conference on Object-Oriented Programming (ECOOP), and Science of Computer Programming (SCP). He was also invited as reviewer for the journal publication Science of Computer Programming (SCP) and as member of OOPSLA'14 Artifact Evaluation Committee (AEC). Siliang has taught courses in programming languages, discreet mathematics and artificial intelligence.

While working on his dissertation, Siliang has worked full-time at Knight Capital Group, Inc. in Jersey City, NJ, then Blackrock, Inc. in New York City, NY.

SILIANG LI

Computer Science & Engineering
19 Memorial Drive West
Bethlehem, PA 18015

Phone: (215) 354-6843
Email: siliang.li@lehigh.edu
Web: http://www.cse.lehigh.edu/~sil206

RESEARCH INTERESTS	Programming languages, software engineering, and static analysis.
PROFESSIONAL EXPERIENCE	Associate, BlackRock Inc., New York, NY (2013 - Present) Associate, Knight Capital Group, Jersey City, NJ (2012 - 2013) Senior Software Engineer, Werum Software & Systems, Parsippany, NJ (2009 - 2012) Software Engineer, Managing Editor Inc., Jenkintown, PA (2007 - 2009) Graduate Teaching Assistant, Lehigh University, Bethlehem, PA (2007 - 2008) Graduate Assistant, University of Rhode Island, Kingston, RI (2003 - 2006)
EDUCATION	Ph.D., Computer Engineering, Lehigh University, May 2014 M.S., Computer Engineering, Lehigh University, Jan 2008 B.S., Computer Engineering, University of Arizona, May 2003
REFEREED PUBLICATIONS	Exception Analysis in the Java Native Interface. S. Li and G. Tan. Science of Computer Programming (SCP), Volume 89, Part C, 1 September 2014, Pages 273–297 Finding Reference-Counting Errors in Python/C Programs with Affine Analysis. S. Li and G. Tan. Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP '14), Uppsala, Sweden, July 2014 JATO: Native Code Atomicity for Java. S. Li, Y. D. Liu, and G. Tan. Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS '12), Kyoto, Japan, Dec 2012 JET: Exception Checking in the Java Native Interface. S. Li and G. Tan. Proceedings of 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '11), Portland, Oregon, Oct 2011 Finding Bugs in Exceptional Situations of JNI Programs. S. Li and G. Tan. Proceedings of 16th ACM Computer and Communication Security (CCS '09), Chicago, IL, Nov 2009
AWARDS	SIGPLAN Student Travel Grant (2011) CCS Travel Grant Award (2009) Graduate Student Research Scholarship (2008 - 2011)
PROFESSIONAL SERVICE	OOPSLA Artifact Evaluation Committee (2014) Reviewer for Science of Computer Programming (2012 - 2013) Organizer for New Jersey Programming Languages and System Seminars in Oct 2009

PRESENTATIONS "Finding Reference-Counting Errors in Python/C Programs with Affine Analysis" at ECOOP '14, Uppsala, Sweden, Jul 2014

"JATO: Native Code Atomicity for Java" at APLAS '12, Kyoto, Japan, Dec 2012

"JATO: Native Code Atomicity for Java" at Graduate Research Seminar, Lehigh University Nov 2012

"JET: Exception Checking in the Java Native Interface" at OOPSLA '11, Portland, Oregon, Oct 2011

"JET: Exception Checking in the Java Native Interface" at Graduate Student Research Seminar, Lehigh University, Sep 2011

"Finding Bugs in Exceptional Situations of JNI Programs" at CCS '09, Chicago, IL, Nov 2009

"Finding Bugs in Exceptional Situations of JNI Programs" at Graduate Student Research Seminar, Lehigh University, Oct 2009