RICE UNIVERSITY

Operating System-Style Protections for Language-Based Systems

by

Algis Rudys

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dan S. Wallach, Chair, Assistant Professor
Computer Science

Keith Cooper, Professor
Computer Science

Robert Cartwright, Professor
Computer Science

Mike Fagan, Faculty Fellow
Computational and Applied Math

HOUSTON, TEXAS

MARCH, 2007
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI

UMI Microform 3256734
Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346
Operating System-Style Protections for Language-Based Systems

Algis Rudys

Abstract

Process-based separation has long been the prevalent model for providing security and isolation to protection domains in computer systems. However, the recent rise of component-based systems, which execute multiple plug-ins in the same process, has exposed a weakness of processes. At the same time, the recent spate of vulnerabilities in software has revealed the usefulness of language-based schemes to supplement the protections offered by processes. I propose a language-based protection model to replace processes as the basis for providing security and isolation.

In this thesis, I present three different language-based mechanisms which add particular operating system-style protection semantics to the language. Soft termination provides a mechanism for guaranteed, safe termination of a task without interfering with other modules. Garbage collector memory accounting provides an accurate accounting of the memory used by each individual task running in the language-based system. Soft boundaries is a set of static analyses to verify that a specified task separation policy is followed by a particular codebase. These mechanisms provide the security and isolation that process-based separation provides, while tackling the problems of component-based architectures and malicious code head-on.
Acknowledgments


I would first like to thank my advisor, Dan Wallach. He has been a mentor and friend, and has consistently provided unhesitating advice and assistance. He has given guidance, support, and criticism through my years as a graduate student, and without him, I would not have made it as far as I did.

Thank you to my thesis committee, Dan Wallach, Keith Cooper, Robert Cartwright, and Mike Fagan, for volunteering their time and expertise. Thanks also to my masters thesis committee, Dan Wallach, Robert Cartwright, and Scott Rixner, without whom I would not be here. I would also like to thank John Clements, Jiangchun “Frank” Luo, Liwei Peng, David Price, and Jonathan Bannet for their contributions to this project. Thank you also to Matthias Felleisen, Shriram Krishnamurthy, Natarajan Shankar, and Drew Dean for their comments, and thanks to Willy Zwaenepoel, Dave Johnson, Walid Taha, and Keith Cooper for helpful discussions. In addition, thanks to the many anonymous reviewers who contributed to improving this work.

I’d also like to thank my family and friends for their support and encouragement. Thanks to my parents for their love, and for supporting me through college and provid-
ing an example to me. Thanks to my siblings for their help and advice. Thanks also to Andrew Ladd, Brian Chen, Mark Flaum, and the whole gang for their encouragement and, occasionally, a little much-needed badgering.
Contents

Abstract ii
Acknowledgments iii
List of Figures x
List of Tables xii

1 Introduction 1
1.1 Inadequacies of Process Separation ...................... 2
1.1.1 Component Architectures ............................. 3
1.1.2 When Processes Do Not Protect ...................... 4
1.2 Tasks in Language-Based Systems ....................... 5
1.3 Bugs and System Availability .......................... 6
1.4 Bug Elimination ........................................ 6
1.5 Bug Containment ......................................... 8
1.5.1 Containment Through Enforcement ................... 8
1.5.2 Containment Through Analysis ....................... 9
1.6 Modifying Language Semantics ......................... 9

2 Related Work 11
2.1 Language-Based Protections ............................ 11
2.1.1 Bytecode Rewriting .................................. 12
2.2 Persistent Systems ...................................... 14
2.3 Resource Accounting .................................... 15
2.3.1 Operating System-Based Resource Accounting ........ 15
2.3.2 Language-Based Resource Accounting ................ 16
2.3.3 Garbage collection ................................... 17
3 Soft Termination

3.1 System design ............................................. 20
  3.1.1 Naïve Termination ..................................... 20
  3.1.2 Hard Termination ..................................... 21
  3.1.3 Soft Termination ..................................... 22
  3.1.4 Soft Termination with Task, System, and Blocking Code 31

3.2 Java Implementation ...................................... 40
  3.2.1 Termination Check Insertion .......................... 41
  3.2.2 Control Flow ......................................... 42
  3.2.3 Blocking Calls ....................................... 44
  3.2.4 Invoking Termination ................................. 45
  3.2.5 Optimizations ........................................ 46
  3.2.6 Synchronization ...................................... 47
  3.2.7 Thread Scheduling .................................... 48
  3.2.8 System Code Safety ................................... 48
  3.2.9 Memory Consistency Models ......................... 48
  3.2.10 Defensive Security .................................. 48

3.3 Performance ................................................ 49
  3.3.1 Microbenchmarks ..................................... 49
  3.3.2 Application Benchmarks .............................. 51
  3.3.3 Termination Check Overhead ......................... 52

3.4 Summary .................................................... 54

4 Garbage Collector-Based Memory Accounting ............. 55

4.1 Design ..................................................... 55
  4.1.1 Instrumented allocation .............................. 56
  4.1.2 Process abstractions ................................. 57
4.1.3 Garbage collection-based accounting \hspace{1cm} 58
4.2 Implementation and Results \hspace{1cm} 62
  4.2.1 Boehm microbenchmark \hspace{1cm} 64
  4.2.2 Application benchmarks \hspace{1cm} 65
  4.2.3 Multitasking microbenchmark \hspace{1cm} 66
4.3 Policy Semantics \hspace{1cm} 67
4.4 Summary \hspace{1cm} 69

5 Transactional Rollback \hspace{1cm} 71
  5.1 System Design \hspace{1cm} 71
    5.1.1 Rollback and Transactions \hspace{1cm} 72
    5.1.2 Architecture of Transactional Rollback \hspace{1cm} 72
    5.1.3 Design Discussion \hspace{1cm} 76
  5.2 Java Implementation \hspace{1cm} 79
    5.2.1 Locking Code Insertion \hspace{1cm} 80
    5.2.2 Lock State Sharing \hspace{1cm} 83
    5.2.3 Arrays \hspace{1cm} 84
    5.2.4 Constructors \hspace{1cm} 84
    5.2.5 Native Methods \hspace{1cm} 85
    5.2.6 Open Files \hspace{1cm} 86
    5.2.7 Optimizations \hspace{1cm} 86
    5.2.8 Integration into a Real-World System \hspace{1cm} 87
  5.3 Performance \hspace{1cm} 89
    5.3.1 Microbenchmarks \hspace{1cm} 89
    5.3.2 Application Benchmarks \hspace{1cm} 90
    5.3.3 Optimization Profiling \hspace{1cm} 92
  5.4 Summary \hspace{1cm} 93

6 Preserving Task Isolation \hspace{1cm} 95
  6.1 Taxonomy of Inter-Task Bugs \hspace{1cm} 95
6.1.1 Pointer Leakage ........................................... 96
6.1.2 Time of Check to Time of Use Bugs .................. 96
6.1.3 Denial of Service via Thread Capture ................. 97

6.2 Design of Task Separation ............................. 98
6.2.1 Protecting Data with Soft Boundaries ................. 98
6.2.2 Protecting Task Threads with Soft Boundaries ...... 100
6.2.3 Design Motivations .................................... 102

6.3 Java Implementation .................................. 104
6.3.1 Specifying Boundary Semantics ..................... 105
6.3.2 Implementation of API Checker ..................... 106
6.3.3 Implementation of Critical Data Checker .......... 109
6.3.4 Implementation of Thread Tracker .................. 110
6.3.5 Implementation Issues .............................. 111

6.4 Results ............................................... 113
6.4.1 Four real world programs ............................ 113
6.4.2 Results from API checking ........................... 117
6.4.3 Critical Data Analysis ............................... 120
6.4.4 Thread tracker results ............................... 123

6.5 Summary ........................................... 126

7 Issues with Modifying Language Semantics ............. 127

7.1 Code-to-Code Transformation Issues ................ 127
7.1.1 Implementations .................................... 128
7.1.2 Implementation Issues .............................. 129
7.1.3 Limitations ........................................ 135
7.1.4 Optimizations .................................... 138

7.2 Language Run-Time System Design Issues .............. 140
7.2.1 Garbage Collector-Based Memory Accounting ....... 140
7.2.2 Implementation Issues .............................. 141
7.2.3 Limitations ........................................ 141
7.3 Static Analysis of Code .............................................. 141
7.3.1 Soft Boundaries Implementation .......................... 142
7.3.2 Implementation Issues and Limitations ............... 142
7.4 Summary ............................................................... 143

8 Future Work .............................................................. 144
8.1 Building a Complete System .................................... 144
8.1.1 Complete Resource Accounting ......................... 144
8.1.2 Real-World Implementation Issues .................... 145
8.1.3 Module System Integration ................................. 146
8.2 Analysis of Soft Boundaries .................................... 146
8.3 Techniques of Implementation ............................... 147

9 Conclusions .............................................................. 149

Bibliography .............................................................. 152
Figures

3.1 Simple language used for our analysis. ........................................ 23
3.2 Evaluation context for reduction of the simple language. ............... 24
3.3 An operational semantics for our language. ................................. 24
3.4 Lexical scoping rules for expanding $[V / x] M$ for any expression $M$. 25
3.5 The steps in evaluating a sample program written in our language ........ 26
3.6 The soft termination transformation. ........................................... 27
3.7 An extended language for analysis. ........................................... 31
3.8 Evaluation context for reduction of this extended language. .......... 32
3.9 Operational semantics for the extended language. ......................... 32
3.10 The soft termination transformation for this extended language. .... 33
3.11 The soft termination transformation applied to any function definition. 41
3.12 The soft termination transformation applied to any loop. .............. 42
3.13 The soft termination transformation applied to a blocking call. ....... 43
3.14 Soft termination overhead for microbenchmarks. .......................... 50
3.15 Soft termination overhead for application benchmarks. ................. 52

4.1 Memory usage in language-based systems .................................... 56
4.2 Memory usage in traditional operating systems ............................ 57
4.3 Using the garbage collector for memory management: first pass ...... 59
4.4 Using the garbage collector for memory management: second pass .... 59
4.5 Run-time overhead in Boehm’s artificial GC benchmark ................ 63
4.6 Run-time overhead in a multitasking microbenchmark ................... 64

5.1 Object structure before a transaction has begun operation on the object. 77
Tables

3.1 Average total number of termination checks performed for each benchmark. 53

4.1 Statistics for the performance of the multitasking microbenchmark . . . . . . 65

6.1 Profile of applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.2 API errors and warnings generated by the API checker . . . . . . . . . . . . 117
6.3 Results of applying the thread tracker . . . . . . . . . . . . . . . . . . . . . . . . 123
Chapter 1

Introduction

The process separation model has long been the prevalent mechanism for isolating protection domains from one another in computer systems. In the process separation model, each protection domain executes in its own hardware-enforced address space with its own threads and other system resources allocated to it. Any interaction among processes must be mediated by the kernel. Process separation is an effective means of providing added robustness as well as security to a system. However, two recent developments in computer systems and security have revealed inadequacies in the process-based protection model.

The first trend is the increasing prevalence of component architectures. Language-based systems like Java and .Net, which allow, and even encourage, running code from multiple protection domains within the context of a single process, have gained widespread use. In addition, many programs support plug-in components to add functionality to the program. The primary reason for using component architectures is increased performance; if all of the code is running within the same address space, there is no need to pay a performance penalty for the operating system to mediate communications among the components. However, these plug-in components may have bugs which could affect other components running concurrently, and process separation cannot provide isolation at the level of components.

The second trend is the recent explosion of security vulnerabilities and their subsequent mass exploitation by worms and viruses. The most exploited bug is the buffer overflow, underscoring the fact that process separation cannot protect a system from the weaknesses of unsafe programming languages. At the same time, the widespread promulgation of spyware and adware have exposed a related problem, that process separation does not adequately protect the system from malicious processes.
Language-based techniques are being increasingly used to improve the security and robustness of systems. There have been a variety of proposals to use static analysis to find security vulnerabilities and other bugs in system code. Likewise, with safe languages like Java and C\(^2\) gaining popularity, system designers are realizing the robustness advantages of type-safety. Although these are promising first steps, they are being designed to supplement the process separation model, not replace it.

I propose an alternative, language-based separation model. Static analysis and type safety as well as other language-based mechanisms could be used to provide semantics for components similar to the semantics provided by an operating system to its processes. Because components in modern component architectures need to communicate with one another, such interactions are allowed, avoiding any performance hit for using an inter-process communications mechanism. However, the system can assure that only allowed interactions take place, providing isolation benefits to the individual components.

In Section 1.1, I explore the inadequacies of the process separation model. Section 1.2 introduces the notion of tasks, a concrete representation of a protection domain for language-based systems, and Section 1.3 discusses system availability for tasks in a language-based system. Section 1.6 introduces some general-purpose techniques for modifying a language's semantics. I show how these techniques can be used to eliminate bugs in Section 1.4. Finally, Section 1.5 shows how language-based techniques can be used to contain bugs, mitigating their impact.

1.1 Inadequacies of Process Separation

The process separation model has long been the prevalent mechanism for providing isolation to different protection domains in a computer system. However, for all of its strengths, there are some glaring holes in this model. First and foremost, process separation provides no protection from code running in the same process. Second, a recent spate of vulnerabilities has exposed some more subtle weaknesses in the process separation model.
1.1.1 Component Architectures

The process separation model fundamentally protects everything in a process (code, threads, and memory) from all code outside the process. A growing trend in computer systems is use of a component architecture, with a high degree of interaction among the various components. Because using inter-process communications is inefficient, many such systems have taken to running all of the components in the same process. Moreover, language-based systems like Java and .Net are designed to support and encourage running code from multiple protection domains within the context of the single process.

There are numerous other examples. On the client side, it is common to run plug-ins and view dynamic content within a web browser; all of this code routinely executes within the same process as the web browser. Java web servlets and Enterprise Java Beans, Microsoft IIS’s ISAPI\(^1\) and FastCGI\(^2\) all support running server plug-ins within the same process as the server itself. In all of these cases, bugs in plug-in components might affect other components running concurrently, possibly destabilizing the entire system.

However, the process model only provides separation between processes; component architectures must provide their own mechanism to ensure safety. Web browsers provide little protection from plug-ins; a simple infinite loop in a Java applet will still freeze the latest web browsers. On the server side, component architectures tend to use some ad hoc mechanism to protect the system. Although Java provides some degree of protection, it is primarily intended to protect the environment in which Java is running, not to protect one Java component from another. For ISAPI, on the other hand, designers are advised to test plug-ins to ensure integrity. Although a safe mode is provided, it merely executes the plug-in in a separate process, forfeiting any advantage of using ISAPI in the first place.

Of course, simply running the component architecture within a language-based system like Java is not a sufficient solution; components may still interfere with one-another. Numerous mechanisms have been proposed for constructing process-like boundaries to separate the components running within the system [34, 7, 68]. These systems suffer from the

\(^{2}\)See http://www.fastcgi.com/.
same limitations and inefficiencies as operating system processes, requiring some degree of system mediation (and hence, run-time overhead) to allow the various components to communicate and share data. As a result, considering the time and effort that has gone into maximizing the efficiency of inter-process communications for operating systems, using processes might be the better option.

1.1.2 When Processes Do Not Protect

Recent security vulnerabilities in traditional operating systems have demonstrated that the process separation model has its flaws. By exploiting the use of unsafe programming languages, an attacker can provide unexpected input to achieve some unintended effect. In the worst case, this could be the execution of arbitrary code. Because processes run with at least the privileges of the user, and sometimes as the administrator, any code executed for the attacker is executed at this privilege level.

The most exploited class of security vulnerabilities is buffer overflows, which is made possible by the lack of bounds checking in C and C++. While some solutions have been devised to make buffer overflow harder to exploit [32, 135], a number of advanced techniques have successfully circumvented many of these [103]. There is a 100% solution to buffer overflows: build memory bounds checking into the language. Unfortunately, most deployed code is written in languages that do not provide such a facility.

There is also a related problem of software Trojan horses. While Trojan horses have been around for a long time, it is only recently that the vast majority of users have been exposed to this class of software, in the form of adware and spyware. Adware and spyware generally either pose as legitimate software (such as web browser toolbar plug-ins) or are installed along with legitimate software. The prevalence of adware and spyware demonstrates that the process model cannot address the problem of malicious code.

A solution to this problem that has been around for a long time is virtualization [65, 8]. In virtualization, suspect code is executed within the context of a virtual machine monitor. Virtualization can be used to keep detailed logs of events and replay them within the virtual machine [44, 63]. Unfortunately, using virtualization can entail a significant overhead in
terms of time and resources. In addition, communications between the virtualized system and the host is highly suboptimal.

1.2 Tasks in Language-Based Systems

I use the term task to represent the encapsulation of some computation that needs to be performed where, as a matter of policy, it needs to be isolated from other tasks. That is, a task is the representation of a protection domain. Tasks are defined by the code that is executed. This is because the code is what actually comes into the system. Memory, data objects, and threads are all resources used and manipulated by the code, and are in effect independent of the code.

Note in particular that tasks and threads are totally distinct concepts. Tasks use threads to execute the code of the task, but nothing ties a particular thread to a particular task. A thread executing the code of one task at one moment may execute the code of some different task the next moment. Similarly, there might be multiple threads simultaneously executing the code of a single task.

Related to this idea of tasks, current research is being done in the language theory community on building high-level language-friendly module systems [53]. Such systems are primarily intended to simplify the development process. They allow for the development of independent, reusable software components (modules). Each component can specify abstract dependencies. The components can be combined, with a dynamic linker automatically resolving dependencies.

Components as used in module systems serve a similar purpose to tasks as used in this thesis. In both cases, code with distinct functionality, and in many cases independently developed, needs to function together without issue. In this thesis, tasks have the additional property that they may have different security requirements, as well. In a sense, this thesis can be seen as providing protection semantics for module systems.
1.3 Bugs and System Availability

When running multiple concurrent tasks in a single language run-time system, it is critical for the system to make guarantees about its availability. Access controls are not sufficient to protect against denial of service attacks. More generally, most existing security mechanisms built into language-based systems focus on safety policies, that is, security policies that are strictly a function of a task's prior execution history and the current request being made. Such policies are designed to guarantee that "nothing bad ever happens."

However, to preserve system availability, we need policies that talk about the future, i.e., "something good eventually happens." Such liveness policies [3] can cover topics as diverse as preventing deadlocks or preventing exhaustion of aggregate resources including CPU, memory, and network usage. In these cases, we cannot conclusively state, for any given lock operation or memory allocation, whether the program is in a "safe" state. However, we do wish to guarantee that the system won't get stuck.

When designing systems to have high availability, there are two general approaches we can take. One approach is to dynamically monitor system usage to detect when bad conditions occur, and taking corrective action when necessary. In this case, there must be some corrective action that can be taken. The other approach is to statically analyze the system and prove that the it cannot ever reach a bad state (or, that it will always, eventually, reach a good state). In the latter case, there must be some tolerance for false positives, since the static analysis will inevitably find some cases that can never happen in the actual code.

1.4 Bug Elimination

The primary strength of a language-based systems and a language-based protection model is its ability to eliminate entire classes of bugs and security vulnerabilities. The use of safe languages guarantees that certain classes of bugs cannot happen. Static analysis can be used to find even more bugs.

The most important attribute of language-based protections is the use of type-safe languages. Type-safe languages can guarantee certain properties of the resulting programs. The most important of these is memory safety. By guaranteeing that a program never writes
past the bounds of a data structure, the language guarantees that buffer-overflow vulnerabilities cannot exist. Similarly, since almost all type-safe languages are garbage collected, there is no issue with using or deallocating already-deallocated memory blocks.

A related guarantee that becomes possible is pointer safety. In a type-safe language, it is not possible to forge a pointer into memory. As a result, pointers become capabilities allowing access to data structures in the program. This allows for the safe execution of malicious code within the same address space. Even the malicious code is limited by the API which the application presents. Programs written in unsafe languages, on the other hand, must always be mindful of component plug-ins forging pointers to potentially sensitive data structures.

In addition, static analysis has gained widespread use for detecting and eliminating bugs. Wagner et al. took a large step in applying static analysis to security when they utilized integer range analysis to detect buffer overflows [135]. Chess [26] and Evans et al. [48] independently performed more precise analysis and uncovered a wider class of bugs, including formatting bugs. CQUAL is another system for applying static analysis to annotated code to find security bugs [124, 76]. Finally, Engler et al. [46] presented a framework for finding bugs by detecting patterns in the code, and labelling inconsistencies as bugs.

Type theory has also been applied to improving the security and reliability of code. Jif [119] is a system for labelling the flow of information in Java programs, and using type theory to validate the labelling. Confinement types [19, 144] provide protection against pointer leakage in Java by preventing classes outside a package from dereferencing a class or casting instances of that class to a superclass. Contracts [52, 17] allow a programmer to specify pre-conditions and post-conditions on functions and parameters and automatically verify these conditions at run-time. Finally, Sewell and Vitek [123] provide a theory based on π-calculus for describing and limiting the interactions among tasks; the limits are enforced at run-time.

Chapter 6 discusses a suite of static analyses I’ve build to detect bugs and security vulnerabilities across protection domains. I provide a mechanism for specifying interfaces into protection domains and finding any attempts to bypass these interfaces. I also provide
a way to mark state as being sensitive, and have a mechanism to detect any attempts to modify this state. These analyses could be used to detect when malicious code (such as adware, spyware, or a virus) is attempting to access a protected interface or attempting to change a critical system setting (say, in the virus scanner).

1.5 Bug Containment

The big strength of process-based protections, however, is their ability to contain bugs. One misbehaving process is limited in its ability to affect the operation of other processes. Mechanisms extending from resource usage bounds to termination have been devised to keep wayward processes in check. Fortunately, equivalent mechanisms can be built to contain bugs in language-based systems. There are two strategies for using language-based techniques to contain bugs, enforcement mechanisms and analysis.

1.5.1 Containment Through Enforcement

The most basic enforcement mechanism provided by operating systems is termination. If a process is misbehaving in any way, the system administrator can terminate the process in a safe and guaranteed manner. In chapter 3, I present soft termination, a similar mechanism for language-based systems. Implemented as a transformation over the code, soft termination prevents a component from making forward progress, guaranteeing its termination, while allowing the component to perform a limited degree of clean-up.

Resource accounting is the other critical mechanism by which operating systems prevent processes from misbehaving. By accounting for memory usage and enforcing limits, operating systems can prevent one process from starving other processes of critical resources. The problem with resource accounting in component architectures is that components may share resources such as memory, and threads may cross from one component to another; determining ownership is tricky.

In language-based systems, a memory accounting mechanism could by built into the garbage collector at minimal cost, providing an accounting of memory reachable from each component [141]. All memory reachable from a component is kept live by the component,
and should be charged to the component. The design and implementation of such a system is described in Chapter 4. A CPU accounting system could interface with the thread scheduler to similar effect; a thread executing within the code of a component would be charged to that component. Of course, allowances would need to be made for memory allocated or code executed within the system on behalf of a component.

1.5.2 Containment Through Analysis

I have also developed analysis-based techniques to minimize exposure to bugs. One example of this, discussed in Chapter 6, is thread capture analysis, which detects when a critical thread calls out to another protection domain, which (due to bugs or malicious intent) may never return the thread. The analysis limits the thread capture to non-critical threads. We also examine program interfaces for methods that are unwittingly exported to other components.

1.6 Modifying Language Semantics

A number of techniques are available to us in implementing protection semantics. Semantics can either be added below the language level through alterations to the language run-time system, at the language level through systematic alterations to code being run on the system, or above the language level through static analysis of code run in the language run-time system. All of these techniques have a place in the semantics I’m proposing for language-based systems. Chapter 7 describes some implementation issues surrounding these techniques, as well as their benefits and drawbacks.

The most straightforward way of effecting a change in a language’s semantics is to simply modify the language run-time system. Because the language’s semantics are determined by how the run-time system interprets the language, changing the language run-time system directly changes the language’s semantics. In addition, since you have full access to the internals of the language system’s implementation, this gives you the most flexibility in making changes.
However, the price you pay is one of compatibility; your new semantics are only applicable on the one run-time system implementation which has been modified. For this reason, I want to avoid using this technique as much as possible. Unfortunately, some semantics changes can only be implemented at this level, because the language run-time system doesn’t provide interfaces for many internal processes.

The next level for implementing semantics modifications is at the language level, through systematic code-to-code transformations. This provides a great deal of flexibility in changing the languages semantics. In addition, code-to-code transformations are portable across different implementations of a language-based system. Code transformations are a well-understood way of describing semantics changes in a language-independent manner, meaning the transformations can fairly easily be translated to other languages entirely. However, transformations are still limited to operations available at the language level. If the language run-time system does not provide an interface to some internal mechanism, then code-to-code transformations cannot manipulate it.

The highest level for implementing semantics modifications is by performing static analysis over a program. The static analysis can view every action that is or might be taken during the course of a program. However, more detailed analysis is extremely complicated; any realistic static analysis would necessarily be conservative, and return false-positives. In addition, static analysis is run before a program is actually executed, meaning it is limited to reporting errors. On the other hand, errors can be found and fixed before the program is ever deployed or executed.
Chapter 2

Related Work

This thesis inherits from existing research in a number of areas. I will begin by giving an overview of language and operating systems security research, which ties together all the work done in this thesis. Then, I will discuss work related to the specific subproblems we solved and the techniques we used. Finally, I will discuss some similar systems for enhancing support for modular code in language-based systems.

2.1 Language-Based Protections

Lampson was the first to define protection domains [82]. He identified three ways in which one could adversely affect a program: modifying the program’s internal data, accessing data that should be kept private, and denying the program the ability to run; protection domains isolate programs from such interference. Many operating systems have adopted his principle in the form of processes that run in independent execution spaces [37, 112]. Capability based systems extend the boundary idea to explicitly limit resource access within a system [59]. In this thesis, protection-domains are applied within language-based systems to support inter-task separation.

Designers of multi-user computer realized that safe languages could also be used to enforce cross-domain protections. One of the earliest such instances was the Burroughs B5000 [90]. Other examples have included Smalltalk [64], Pilot [110], Cedar [129], Lisp Machines [20], and Oberon [143]. Stack inspection extends safe language to provide access control protections for the language [137].

As applications grew in size, complexity, and functionality, researchers began exploring ways to support operating system-style protections in language-based systems. Much of the recent work in this area has been done with respect to the Java programming lan-
guage [67]. Systems such as MVM [34], KaffeOS [7], and JKernel [68] work by explicitly implementing process-style separations inside Java. MVM and KaffeOS set up independent memory areas for individual tasks, and disallow memory references or method calls from one task to another. JKernel utilizes bytecode transformations to enforce the same restrictions. MVM has been extended with a fast and efficient inter-task communication mechanism to mitigate the performance impact of hard boundaries [101]. Luna leverages the Java type system to protect internal task data and threads [69]. The programmer tags each variable and object to be one of type sharable or unsharable; sharable objects are not interchangeable with their unsharable counterparts, and only sharable objects can be passed to other tasks.

A number of other mechanisms have been proposed for enforcing protection between protection domains without eliminating communication between the domains. Provos et al. [108] present an ad-hoc scheme of separating privileged and unprivileged functionalities of the same program into separate programs. Stack inspection is used in safe languages, most famously Java, to provide access control to sensitive system functions [137], while a recent Scheme system called MrEd provides support for thread termination and some resource management [57].

Confinement types [19, 144] provide protection against pointer leakage in Java by preventing classes outside a package from dereferencing a class or casting instances of that class to a superclass. Contracts [52, 17] allow a programmer to specify preconditions and postconditions on functions and parameters and automatically verify these conditions at run-time. Finally, Sewell and Vitek [123] provide a theory based on $\pi$-calculus for describing and limiting the interactions among tasks; the limits are enforced at run-time.

### 2.1.1 Bytecode Rewriting

Java bytecode rewriting in particular has been applied in far too many other systems to provide a comprehensive list here. It is used in SAFKASI [137] to implement the security-passing-style, with the same semantics as stack inspection. It is also used in numerous other settings for a wide variety of purposes.
**Access Control.** By intercepting or wrapping calls to potentially dangerous Java methods, systems by Pandey and Hashii [102], Erlingsson and Schneider [47], and Chander et al. [23] can apply desired security policies to arbitrary tasks without requiring these policies to be built directly into the Java system code, as done with Java’s built-in security system.

**Resource Management and Accounting.** J-Kernel [68] and J-SEAL2 [13] both focus primarily on isolation of tasks. Bytecode rewriting is used to prevent tasks from interfering in each others’ operations. JRes [35] focuses more on resource accounting; bytecode rewriting is used to instrument memory allocation and object finalization sites.

**Optimization.** Cream [28] and BLOAT (Bytecode-Level Optimization and Analysis Tool) [100] are examples of systems which employ Java bytecode rewriting for the purpose of optimization. Cream uses side-effect analysis, and performs a number of standard optimizations, including dead code elimination and loop-invariant code motion. BLOAT uses Static Single Assignment form (SSA) [33] to implement these and several other optimizations.

**Profiling.** BIT (Bytecode Instrumenting Tool) [83] is a system which allows the user to build Java instrumenting tools. The instrumentation itself is done via bytecode rewriting. Other generic bytecode transformation frameworks, such as JOIE [30] and Soot [132], also have hooks to instrument Java code for profiling.

**Other Semantics.** Sakamoto et al. [120] describe a system for thread migration implemented using bytecode rewriting. Marquez et al. [89] describe a persistent system implemented in Java entirely using bytecode transformations at class load time. Notably, Marquez et al. also describe a framework for automatically applying bytecode transformations, although the status of this framework is unclear. Kava [138] is a reflective extension to Java. That is, it allows for run-time modification and dynamic execution of Java classes and methods.
2.2 Persistent Systems

Transactional rollback, presented in Chapter 5, deals with introducing database-style features into the language. Databases have always been capable of operating on data in a transactional manner, generally through queries. More recently, programming languages have supported orthogonal persistence. This is the notion that all data types, whether stored persistently or temporarily, are treated equivalently in the language. PS-algol [5] and Elle [1] were among the first programming languages to support orthogonal persistence. Napier88 [39] is a more recent example.

Persistent object systems have been around nearly this long. Persistent object systems provide orthogonal persistence in object-based environments. One of the earliest is POMS [29], the Persistent Object Management System. In this system, based on PS-algol, persistent and transient data are indistinguishable. POMS was followed up by CPOMS [21], a layer written in the C programming language that also provided persistence for PS-Algol.

Thor [86] is a more recent example of a persistent object system. It is an early example of systems that were not bound to a single programming language. Thor guarantees the integrity of the object store even when used by an unsafe language such as C++. It can provide this guarantee by only allowing access to persistent objects via their methods. The persistent objects themselves are implemented in Theta, a type-safe language.

More recently, Java has been the target of persistent systems research. PJama [106] is one persistent object system that has been developed for Java. PJama maintains an object store parallel with the system heap. Objects become persistent automatically by reachability from an object which is already persistent, or by being explicitly declared persistent. As objects become persistent, they are migrated to this object store, to eventually be written to disk.

Much of the current development in persistence-related techniques is taking place around PJama. One example is Daynès and Czajkowski’s lock state sharing mechanism [38]. Hosking et al. [72] discuss several techniques for optimizing read and write
barriers, using PJama as a basis. Finally, a number of novel ideas for implementing transaction management, explicitly transient state, and checkpointing are all based on PJama.

More recently, there has been research on implementing Java persistence outside of the JVM. The goal here is to allow portability across virtual machine implementations. Kutlu and Moss [80] describe a system that uses Java reflection. They do, however, make modifications to the JVM to support more extensive reflection. Marquez et al. [89] describe a persistent object system implemented for Java using bytecode transformations. This requires no modifications to the JVM.

2.3 Resource Accounting

Garbage collector memory accounting, presented in Chapter 4, is a mechanism for providing language-level support for per-task memory accounting through integration with the garbage collector. There is a fairly large field of related work both in the development of language-based resource accounting schemes and the advancement of the state of the art in garbage collection.

2.3.1 Operating System-Based Resource Accounting

Operating systems like UNIX have supported resource accounting and management almost since their inception. The top program and associated kernel facilities are a common interface for resource accounting, and the limit facility of the UNIX shell is the most common interface to UNIX resource management. Modern UNIX systems also include the getrlimit(2) and setrlimit(2) system calls for specifying per-process limits for a variety of system resources, including memory usage.

Several recent operating systems, including Angel [96], Opal [24] and Mungi [70], have been designed to support a single, large address space for all applications. Such systems, commonly designed for 64-bit architectures, can support data sharing semantics comparable to language-based systems, since all pointers are global. Regardless, single address space operating systems segregate memory into pages which are "owned" by and charged to specific processes, exactly as in traditional operating systems.
2.3.2 Language-Based Resource Accounting

Many of the earlier language-based security systems provided little or no support for resource accounting and management on the programs they ran. A number of projects have been developed to address this. A recent Scheme system called MrEd [57] supports thread termination and management of resources like open files. Some systems, such as PLAN [71], restrict the language to provide resource usage guarantee (termination, in this case).

Much of the recent research in this area has been focused on the Java programming language. Chander et al. [23] describe a system to target specific sorts of resource exhaustion attacks via bytecode instrumentation. The general technique they present is to replace calls to sensitive methods (for instance, for setting thread priority or creating a new window) with calls to customized methods that first verify that the operation is not harmful. Although this mechanism fails to address the problem of tracking resource usage, but it would be useful in conjunction with a resource accounting mechanism.

The multitasking virtual machine (MVM) [34] is a customization to the Java virtual machine implementing separation of tasks using a process abstraction. They assign a heap to each stack; however, data that has been live for long enough is moved to a shared heap by the garbage collector. The garbage collector is used to track how much data a task has live in the shared heap. Although this approach is similar to ours, the MVM does not allow tasks to share memory; as a result, they do not address the problem of memory accounting in the face of such sharing.

Another approach that uses the garbage collector to enforce resource limitations is rent collection [4]; objects on the heap are given a store of money that is debited by the garbage collector. Objects that run out of money are “evicted” and treated as garbage. If a task is not willing or able to pay for the memory usage of its allocated objects, those objects are collected. Rent collection has also been used to prevent side-channel attacks in a multi-level system [78]. Bertino et al. also tackle the problem of designing a garbage collector for a multi-level secure heap [12, 27]. These systems are primarily concerned with covert communication channels between tasks of different security levels that are allowed to share
objects references; by observing when objects are or are not garbage collected, information can be covertly passed. These systems are uninterested in measuring memory usage; likewise, we do not consider garbage collector covert channels in our work.

Wick et al. [141] also present a system for using a garbage collector to determine memory usage of tasks in Scheme based on reachability. One key difference between our work and Wick et al. is that they measure only a single usage value for each task, limiting the expressiveness of their security policies when many objects are shared among tasks. They likewise have no notion comparable to unaccountable references (see Section 4.1.3) to allow one task to explicitly accept the full charges for sharing an object with another task.

2.3.3 Garbage collection

Garbage collection has been around since at least the LISP programming language [92]. Wilson [142] provides an excellent overview of garbage collection techniques. Some more common techniques include mark-and-sweep [91], copying collectors [25], and generational garbage collectors [130]. We implemented memory accounting for copying and generational collectors.

2.4 Static Analysis and Security

Static analysis has seen a fair amount of success in the security community. Wagner et al. took a large step in applying static analysis to security when they utilized integer range analysis to detect buffer overflows, the most commonly exploited security hole [135]. Chess [26] and Evans et al. [48] performed more precise analysis and uncovered a wider class of bugs, including formatting bugs. CQUAL is another system for applying static analysis to annotated code to find security bugs [124, 76]. Similarly, ESC/Java [55] is an annotation-based static analysis tool specifically targeted at Java programs. Finally, Engler et. al applied static analysis to detect patterns in large code bases; instances that broke the pattern were considered bugs [46]. Using this technique, they uncovered null pointer dereferences, deadlock issues, and even points where the programmers’ unspecified invariants are violated.
A second class of security bug, race conditions, have also been addressed using static analysis. Flanagan and Freund use type theory with a great degree of success to find race conditions [61, 54]. Their approach requires program annotation, but includes a program to aid the annotation process. RacerX [45] also catches deadlocks and race conditions and does not require program annotations. Both projects require significant programmer energy to distinguish false positives. Finally, Porat applied static analysis to the problem of distinguishing mutable class fields from their stable counterpart [104]. Knowing that a field is never mutated guarantees that a use of that field will never be subject to a race condition.

Static analysis has also been used for secure information flow. The work was pioneered as early as 1975, by Denning [42], and proven sound by Valpano et al. [134]. JFlow [97] (later Jif [119]) is a system for labelling the flow of information in Java programs, and uses type theory to validate the labelling.

Finally, static analysis has been used in operating systems to ensure that operating system modules are safe. In SPIN, modules are written in Modula-3, a safe language, ensuring their safety [11]. Proof-carrying code allows for the same assurances to be made about assembly language programs [98]. Software-fault isolation is a general technique for isolating faults within individual modules without relying on hardware protection [136].

2.5 Module and Component Systems

A research topic that has been active in recent years is developing components and modules in a way that fit naturally with high-level languages. Most such systems were designed to simplify independent development and separate compilation. Numerous languages, including Java [67], provide some mechanism for creating packages, which can be used to indicate the boundaries of different modules. Java also supports on-demand run-time loading of code. Similarly, ML's functor system encodes code modules as structures, and functors, functions mapping a base module to an extended module, encode an extended module's dependence on a base module [87].

Units provide a more flexible mechanism for defining modules [56]. Unlike Java packages, there is no global namespace; units are first-order objects, and can be operated on
like other objects. Unlike ML's functors, units support dynamic loading and cyclical dependencies. The definition of a unit's code, the unit's dependencies and exports, and how one unit's exports resolve another unit's dependencies are all kept separate, allowing for maximum flexibility. In addition, units support mixins, a facility for extending classes by specifying the necessary modifications to the base class [58].

Units and mixins have been combined to form powerful component systems. MzScheme is a Scheme implementation that features units and mixins [53]. Jiazi [93] and ComponentJ [121] are component systems for Java based on units. Knit [111] is a system designed for supporting components in operating systems.
Chapter 3

Soft Termination

In this chapter, we introduce and discuss soft termination. Section 3.1 formalizes and describes what we mean by soft termination. Section 3.2 describes our Java-based implementation of soft termination, and mentions a number of Java-specific issues that we encountered. We present performance measurements in Section 3.3. We provide a summary of this chapter in Section 3.4.

3.1 System design

A large number of possible designs exist for supporting termination in language run-time systems. We first consider the naïve solutions and explain the hard problems raised by their failings. We then discuss how operating systems perform termination and finally, describe our own system.

3.1.1 Naïve Termination

One naïve solution to termination would be to identify undesired threads and simply remove them from the thread scheduler. This technique is used by Java’s deprecated Thread.destroy() operation.\(^1\) Unfortunately, there are numerous reasons this cannot work in practice, which led Sun to deprecate this method.

Critical sections.  A thread may be in a critical section of system code, holding a lock, and updating a system data structure. Descheduling the thread would either leave the system in a deadlock situation (if the lock is not released) or leave the system data

\(^1\)For more information, see [http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html](http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html)
structures in an undefined state, potentially breaking system invariants and destabilizing the entire system (if the lock is forcibly released).

**Boundary-crossing threads.** In an object-oriented system, a program wishing to inspect or manipulate an object invokes methods on that object. When memory sharing is unrestricted between the system and its tasks or among the tasks, these method invocations could allow a malicious task to hijack the thread from its caller and perhaps never release it. This is especially problematic if the thread in question is performing system functions, such as finalizing dead objects prior to garbage collection.

**Blocking calls.** Many language run-time systems have functions (especially I/O functions) which block. Blocking calls frequently involve a system call to the operating system. Descheduling a thread which is making a blocking call could cause problems if and when the corresponding system call returns, and should be avoided.

Another naïve solution is to force an asynchronous exception, as done by Java's deprecated `Thread.stop()` operation. Although this exception will wait for blocking calls to complete, it may still occur inside a critical section of system code. In addition, blocking calls could potentially never return, resulting in a nonterminable thread. Finally, a workaround is needed to prevent user-level code from catching and ignoring the exception.

### 3.1.2 Hard Termination

Operating systems such as UNIX support termination by carefully separating the kernel from the user program. When a process is executing in user space, the kernel is free to immediately deschedule all user threads and reclaim the resources in use. We call such a mechanism a *hard termination system* because once termination is signaled, user-level code may be terminated immediately with no harmful side effects. External resources, such as data files in the file system, may be left in an inconsistent state by this termination. However, in general, these inconsistencies do not threaten stability at the system level.

If the process is executing in the kernel, termination is normally delayed; a flag is checked when the kernel is about to return control to the user process. In cases where the
kernel may perform an operation that could potentially block forever (e.g., reading from the network), the kernel may implement additional logic to interrupt the system call. System calls that complete in a guaranteed finite time need not check whether their user process has been terminated, as the kernel will handle the termination signal on the way out.

3.1.3 Soft Termination

Unlike a traditional operating system, the boundary between user and system code in a language run-time system is harder to define. Although all code within the system is generally tagged with its protection domain, there is nothing analogous to a system call boundary where termination signals can be enforced. Furthermore, because system code might call back to user code, even a thread executing user code might be unsafe to terminate.

This section introduces a design we call soft termination and describes the properties we would find desirable. We present a formal model of soft termination based on code rewriting for a simplified language and prove that all programs will terminate when signaled to do so.

Key Ideas

Soft termination is based on the idea that a task may be statically instrumented to check for a termination condition during the normal course of its operation. Our goal is to perform these checks as infrequently as possible—only enough to ensure that a task may not execute an infinite loop when termination has been signalled. Furthermore, as with the UNIX kernel, we would like the termination of a task not to disturb any system code it may be using at the time it is terminated, so as to preserve system correctness.

The soft termination checks are analogous to safe points, which are used in language environments to insert checks for implementing stack overflow detection, preemptive multitasking, interprocess and intertask communication, barrier synchronization, garbage collection, and debugging functions. A good discussion on safe points is provided by Feeley [50]. In Feeley’s terminology, our design uses “minimal polling.”
\[ P = \Gamma M \]
\[ \Gamma = D \ldots D \]
\[ D = (\text{define } (f \ x) M) \]
\[ M = (f \ M) | (\text{if}_0 \ M \ M \ M) | (\text{let } (x \ M) \ M) | (\text{try } M \ M) | (\text{throw}) | V | x | (\text{CheckTermination}) \]
\[ V = c \in \mathbb{N} \]
\[ f, x = \text{identifiers} \]

Figure 3.1: Simple language used for our analysis.

**Formal Design**

For our analysis, we begin with a simple programming language having natural numbers, functions, conditional expressions, and simple exceptions (see Figure 3.1). In our language, a program is a collection of function definitions (\( \Gamma \)) followed by an expression to be evaluated (\( M \)). An expression can contain function applications as well as primitive operations, conditionals, and exceptions. For simplicity, functions are designed to each take a single natural number parameter (a number of schemes exist for representing multiple natural numbers using a single natural number). Section 3.2.2 discusses our handling of the richer control flow available in Java for our implementation of soft termination.

We write the semantics of our language using the same style as Felleisen and Hieb [51]. Figure 3.2 defines \( E \), the grammar of evaluation contexts for our language. An evaluation context is simply an expression with a subexpression replaced by a "hole" ([ ]). The hole acts as a placeholder in the context; \( E[M] \) represents the result of putting expression \( M \) into the hole of evaluation context \( E \). The hole is consistently located in such a way as to enforce a left-to-right order of evaluation. The reduction rules in Figure 3.3 are applied to these contexts, defining the behavior of the language.

The semantics for our language define three possible ending states: \( V \), error, and (throw). \( V \) represents a final state in which the program reduces to a value. The error state indicates that some error condition, such as a call to an undefined function, was reached in the evaluation of the program. The (throw) state occurs when the entire program reduces to (throw); it indicates that the program terminated as the result of an uncaught exception.
\[ E = \{ \setminus \} \mid (f \ E) \mid (i f_0 \ E \ M \ M) \mid (\text{let } (x \ E) \ M) \mid (\text{try } E \ M) \mid (\text{CheckTermination}) \mid (\text{throw}) \mid x \]

\text{final states} = V \mid \text{error} \mid (\text{throw})

Figure 3.2: Evaluation context for reduction of the simple language.

\[
E[(f \ V)] \rightarrow \begin{cases} 
E[V_0] & \text{if } \delta(f, V) = V_0 \\
E[[V / x] M]^{\dagger} & \text{if } (\text{define } (f \ x) \ M) \in \Gamma \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[
E[(\text{CheckTermination})] \rightarrow E[0] \text{ or } E[1] \quad \text{(depending on external conditions)}
\]

\[
E[(i f_0 \ 0 \ M_1 \ M_2)] \rightarrow E[M_1]
\]

\[
E[(i f_0 \ V \ M_1 \ M_2)] \rightarrow E[M_2] \quad \text{if } V \neq 0
\]

\[
E[(\text{let } (x \ V) \ M)] \rightarrow E[[V / x] M]^{\dagger}
\]

\[
E[(\text{try } V \ M)] \rightarrow E[V]
\]

\[
E[(\text{try } (\text{throw}) \ M)] \rightarrow E[M]
\]

\[
E[(f \ (\text{throw})\)) \rightarrow E[(\text{throw})]
\]

\[
E[(i f_0 \ (\text{throw}) \ M_1 \ M_2)] \rightarrow E[(\text{throw})]
\]

\[
E[(\text{let } (x \ (\text{throw}) \ M)] \rightarrow E[(\text{throw})]
\]

\[
E[x] \rightarrow \text{error} \quad \text{(unbound variables)}
\]

Figure 3.3: An operational semantics for our language. \(^\dagger\)The expansion \([V / x] M\) indicates that every instance of \(x\) in \(M\) should be replaced with the corresponding \(V\), according to the standard rules of lexical scope. This is defined for our language in Figure 3.4.

Although this is similar in nature to \text{error}, it is treated differently to indicate its different origin.

A (\text{CheckTermination}) expression reduces to a Boolean value (0 or 1), indicating whether termination for the current task has been externally (and asynchronously) requested. Since (\text{CheckTermination}) behaves as a termination signal, we model it by assuming that when evaluation begins (\text{CheckTermination}) evaluates to 0. Once (\text{CheckTermination}) becomes 1 (indicating that termination has been requested), it continues to be 1 until the task terminates.

The last five reductions require some explanation. The first four of these allow for the single-step reduction of (\text{throw}) expressions appearing as subexpressions in any other expression. When (\text{throw}) occurs as the body of a \text{let} or as the second or third parameter of an \text{if}, the default reduction rules already correctly reduce the expression, so no special case is needed.
\[
\begin{align*}
[V / x] (f \ M) & \mapsto (f [V / x] M) \\
[V / x] (\text{if}_0 \ M_1 M_2 M_3) & \mapsto (\text{if}_0 [V / x] M_1 [V / x] M_2 [V / x] M_3) \\
[V / x] (\text{let} (x \ M_1) M_2) & \mapsto (\text{let} (x [V / x] M_1) M_2) \\
& \quad \text{(the let expression overrides the term being replaced).} \\
[V / x] (\text{let} (t \ M_1) M_2) & \mapsto (\text{let} (t [V / x] M_1) [V / x] M_2) \\
& \quad \text{if } t \neq x \text{ (the let expression introduces a new term).} \\
[V / x] (\text{try} \ M_1 M_2) & \mapsto (\text{try} [V / x] M_1 [V / x] M_2) \\
[V / x] (\text{throw}) & \mapsto (\text{throw}) \\
[V / x] (\text{CheckTermination}) & \mapsto (\text{CheckTermination}) \\
[V / x] V' & \mapsto V' \\
[V / x] x & \mapsto V \\
[V / x] t & \mapsto t \quad \text{if } t \neq x
\end{align*}
\]

Figure 3.4: Lexical scoping rules for expanding \([V / x] M\) for any expression \(M\).

The last reduction rule reduces a variable, when put into the hole of the evaluation context, to \textbf{error}. Note that evaluation of \textbf{let} expressions and function applications replaces all bound variables with their values, according to the rules of lexical scope (see Figure 3.4). As a result, any attempt to evaluate a variable indicates that the variable is unbound and undefined, which is an error.

We include a delta (\(\delta\)) function which maps primitive names and valid arguments to values. Note that the delta function is simply an abstraction used to represent primitives in this language, and never actually appears in a program. We assume the syntactic property that no name collisions occur; that is, no function is defined more than once, and no primitive function is redefined.

As long as \(\text{CheckTermination}\) remains constant, this language is deterministic. That is, for any given program, when evaluated multiple times, the sequence of reductions applied will always be the same and the program will always terminate in the same final state. The language inherits this determinism from the property that for every syntactically valid program, there is exactly one reduction in the operational semantics which can be applied to that program, and the result is another syntactically valid program. A formal proof of this property is beyond the scope of this thesis.
Sample program:

\[
P = \begin{align*}
& \text{(define } (f_1 \ x) \ (\text{if}_0 \ x \ 1 \ \text{(throw)))} \\
& \text{(try } (f_1 \ (f_1 \ (f_1 \ 0))) \ (f_1 \ 0))
\end{align*}
\]

\[
\Gamma = \begin{align*}
& \text{(define } (f_1 \ x) \ (\text{if}_0 \ x \ 1 \ \text{(throw)))} \\
& \text{(try } (f_1 \ (f_1 \ (f_1 \ 0))) \ (f_1 \ 0))
\end{align*}
\]

\[
M = \begin{align*}
& \text{(define } (f_1 \ x) \ (\text{if}_0 \ x \ 1 \ \text{(throw)))} \\
& \text{(try } (f_1 \ (f_1 \ (f_1 \ 0))) \ (f_1 \ 0))
\end{align*}
\]

<table>
<thead>
<tr>
<th>Step</th>
<th>Expression</th>
<th>Reduction Rule</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(try ( f_1 \ (f_1 \ [f_1 \ 0]) )) ( f_1 \ 0) )</td>
<td>( E[(f \ V)] \leftrightarrow E[[V / x] \ M] ) if ( \text{(define } (f \ x) \ M) \in \Gamma )</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>2</td>
<td>(try ( f_1 \ (f_1 \ [\text{if}_0 \ 0 \ 1 \ \text{(throw)})] )) ( f_1 \ 0) )</td>
<td>( E[(\text{if}_0 \ 0 \ M_1 \ M_2)] \leftrightarrow E[M_1] )</td>
<td>Evaluation of ( \text{if}_0 ) conditional, where the test is 0.</td>
</tr>
<tr>
<td>3</td>
<td>(try ( f_1 \ [f_1 \ 1]) \ (f_1 \ 0) )</td>
<td>( E[(f \ V)] \leftrightarrow E[[V / x] \ M] )</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>4</td>
<td>(try ( f_1 \ [\text{if}_0 \ 1 \ M_1 \ M_2]) \ (f_1 \ 0) )</td>
<td>( E[(\text{if}_0 \ 1 \ M_1 \ M_2)] \leftrightarrow E[M_2] )</td>
<td>Evaluation of ( \text{if}_0 ) conditional, where the test is 1.</td>
</tr>
<tr>
<td>5</td>
<td>(try ( f_1 \ (\text{throw}) ) ( f_1 \ 0) )</td>
<td>( E[(f \ (\text{throw}))] \leftrightarrow E[(\text{throw})] )</td>
<td>Single-step reduction of ( \text{(throw) \ occurring} ) as a parameter to a function.</td>
</tr>
<tr>
<td>6</td>
<td>(try ( \text{throw} ) ( f_1 \ 0) ))</td>
<td>( E[(\text{try} \ (\text{throw}) \ M)] \leftrightarrow E[M] )</td>
<td>Evaluation of ( \text{try} ) expression catching a ( \text{(throw)}. )</td>
</tr>
<tr>
<td>7</td>
<td>( f_1 \ 0) )</td>
<td>( E[(f \ V)] \leftrightarrow E[[V / x] \ M] )</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>8</td>
<td>( \text{if}_0 \ 0 \ 1 \ \text{(throw)})]</td>
<td>( E[(\text{if}_0 \ 0 \ M_1 \ M_2)] \leftrightarrow E[M_1] )</td>
<td>Evaluation of ( \text{if}_0 ) conditional, where the test is 0.</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: The steps in evaluating the program \( P \). The left column shows the actual program in various stages of reduction. The subexpression that will be placed in the hole (and operated on by the reduction rules) is boxed. The middle column shows the reduction rule, as written in the operational semantics in Figure 3.3, that applies to the program at this stage. The right column describes the reduction.
Figure 3.6: The soft termination transformation.

Figure 3.5 describes a step-by-step evaluation of the sample program $P$. This program illustrates both the evaluation of functions, including the handling of function parameters, and the propagation and catching of exceptions.

Steps 1, 3, and 7 show the application of a user-defined function. Note in Steps 1 and 2, the parameter of the second application of function $f_1$ is reduced to a value before the application is reduced. Step 5 shows a (throw) expression single-stepping out of a function call. Step 6 shows the same (throw) expression being caught in a try expression. In Step 9, there are no more reduction rules to be applied, so the program returns the value 1.

**Proof of Result Preservation**

The soft termination transformation $\mathcal{X}$ is described in Figure 3.6. Rule 4 of this transformation inserts the check for termination before every function application. It also describes how a function's parameter is first evaluated separately, and is likewise recursively transformed by $\mathcal{X}$ to catch any nested function applications. The other transformation rules describe how the transformation continues recursively on expressions. We first prove that if termination has not been requested, the transformed program evaluates to the same final state as the original untransformed program.
Theorem 3.1. Assume the definitions in Figures 3.1 through 3.4 and 3.6. As long as \((\text{CheckTermination})\) is 0, evaluating any transformed expression \(\mathcal{X}[M]\) always results in the same final state as evaluating the corresponding expression \(M\).

Proof. We can show this by a structural induction on \(M\), the grammar of expressions in our language. In the base cases,

\[
M_{\text{base}} = (\text{throw}) \mid (\text{CheckTermination}) \mid V \mid x.
\]

In each of these cases, \(\mathcal{X}[M_{\text{base}}] = M_{\text{base}}\), so the behavior of the expression is preserved.

In the inductive cases,

\[
M_{\text{inductive}} = (\text{if}_0 M M M) \mid (\text{let} \ (x M) M) \mid (\text{try} \ M M) \mid (f M),
\]

where

\[
M = M_{\text{base}} \mid M_{\text{inductive}}.
\]

In the first case, \text{if}_0 expressions are transformed by the rule:

\[
\mathcal{X}[(\text{if}_0 M_1 M_2 M_3)] = (\text{if}_0 \mathcal{X}[M_1] \mathcal{X}[M_2] \mathcal{X}[M_3]).
\]

By the inductive hypothesis, evaluating each \(M_i\) has the same final state as evaluating the corresponding \(\mathcal{X}[M_i]\) as long as \((\text{CheckTermination})\) is 0. Because the result of evaluating the \text{if}_0 expression is solely dependent on these results, it is also unaffected by \(\mathcal{X}\). The cases of \text{let} and \text{try} expressions proceed similarly.

The final case, that of function applications, is not so straightforward. According to the definition of \(\mathcal{X}\):

\[
\mathcal{X}[(f M)] = (\text{let} \ (t \mathcal{X}[M]) (((\text{if}_0 \ (\text{CheckTermination}) \ (f t) \ (\text{throw})))).
\]

By the inductive hypothesis again, evaluating \(M\) has the same final state as evaluating \(\mathcal{X}[M]\). Any error or exception in \(M\) occurs in \(\mathcal{X}[M]\), causing the same final state. In the absence of errors or exceptions thrown, \(M\) and \(\mathcal{X}[M]\) evaluate to the same value \(V\). At this point, the original expression has reduced to \((f V)\), while the transformed expression has reduced to:

\[
(\text{let} \ (t V) (((\text{if}_0 \ (\text{CheckTermination}) \ (f t) \ (\text{throw}))))).
\]
which further reduces to:

$$(\text{if}_0 (\text{CheckTermination}) (f \ V) (\text{throw})).$$

Since (CheckTermination) is 0, this reduces to $(f \ V)$, the same as the original expression.

Because, for every expression, the $\mathcal{X}$ transformation does not affect the result of evaluating the expression as long as (CheckTermination) is 0, the behavior of the entire program is preserved.

Proof of Termination

Proving that a transformed program terminates in a finite number of reduction steps, given that (CheckTermination) is 1, is a straightforward exercise. We start with the grammar $M$ of all possible expressions from Figure 3.1. After transforming $M$ by $\mathcal{X}$, we call the resulting set of expressions $M_{\text{lock}}$; $M_{\text{lock}}$ is a subset of $M$. Assuming (CheckTermination) is 1, we wish to show the program enters a “locked” state where termination is guaranteed.

$$M_{\text{lock}} = (\text{if}_0 (\text{CheckTermination}) (f \ V) (\text{throw}) ) | (a)$$
$$\text{if}_0 1 (f \ V) (\text{throw}) ) | (b)$$
$$\text{CheckTermination} | (c)$$
$$\text{if}_0 M_{\text{lock}} M_{\text{lock}} M_{\text{lock}} | (d)$$
$$\text{let} (x M_{\text{lock}}) M_{\text{lock}} | (e)$$
$$\text{try} M_{\text{lock}} M_{\text{lock}} | (f)$$
$$\text{throw} | (g)$$
$$x | (h)$$
$$V | (i)$$

Expressions $c$ to $i$ are taken directly from the definition of $M$ in Figure 3.1. Expression $a$ represents the final case of $M$, function applications, as transformed by $\mathcal{X}$. Expression $b$ describes an intermediate step in reducing expression $a$ when (CheckTermination) is 1. Note in particular, that $\mathcal{X}[M] \subseteq M_{\text{lock}}$. To prove termination, we must show that, as long as (CheckTermination) is 1, then $M_{\text{lock}}$ is closed under program stepping, and that the syntactic length of the program will be strictly decreasing.

Lemma 3.1 (The Closure Property). Suppose (CheckTermination) is 1. For arbitrary expressions $M$ and $M'$, if $M \rightarrow M'$ and $M \in M_{\text{lock}}$, then $M' \in M_{\text{lock}}$. 
Proof. By inspection, for all possible expressions in $M_{lock}$, we observe that our semantics preserves closure.

Lemma 3.2 (The Syntactic Length Property). If (CheckTermination) is always 1, then for $M, M' \in M_{lock}$ such that $M \rightarrow M'$, we have that $|M'| < |M|$.

Proof. That is, as reduction rules are applied, a program in $M_{lock}$ gets shorter. To prove this, it suffices to observe that all reductions other than function application make the program smaller, and that for an expression in $M_{lock}$, all function applications are "fenced out" by a termination check.

Lemma 3.3. When (CheckTermination) becomes set, a program being evaluated enters $M_{lock}$ within at most two reduction step.

Proof. When a program is in $M_{lock}$ and (CheckTermination) is set, by Lemma 3.1, the program is guaranteed to stay in $M_{lock}$. If the program is not in $M_{lock}$ when (CheckTermination) is set, then it is either evaluating an application or a conditional which will result in an application (since these are the only two expressions where $M$ and $M_{lock}$ differ). Within one or two steps, then, it will be evaluating the body of the application, $\mathcal{X}[M]$. As noted above, this is defined to be in $M_{lock}$. In other words, a transformed program that is not in $M_{lock}$ is guaranteed to enter $M_{lock}$ within two steps.

Theorem 3.2. Any program in $M$, after being transformed by $\mathcal{X}$, is guaranteed to terminate in a finite number of steps, once the (CheckTermination) condition is raised.

Proof. By Lemma 3.3, within two steps of the (CheckTermination) condition being raised, the program is guaranteed to be in $M_{lock}$. By Lemma 3.1, the program is guaranteed to remain in $M_{lock}$. By Lemma 3.2, then, the program is guaranteed to get shorter with each reduction. Since program lengths are finite, a program transformed by $\mathcal{X}$ is guaranteed to reduce to some non-reducible expression (that is, to a single value or (throw)) in a finite number of steps, once the (CheckTermination) condition is raised.
\[ P = \Gamma \, M \]
\[ \Gamma = D \ldots D \]
\[ L = \text{system} | \text{task} \]
\[ D = (\text{define} \ L \ (f \ x) \ M) | (\text{define blocking} \ (f_{\text{blocking}} \ x) \ f_{\text{nonblocking}}) |
\]
\[ (\text{define primitive} \ (f \ x)) \]
\[ M = (f \ M) | (\text{if} \ _0 \ M \ M \ M) | (\text{let} \ (x \ M) \ M) | (\text{try} \ M \ M) | (\text{throw}) \ | V | x | \text{CheckTermination} \]
\[ V = c \in \mathbb{N} | \text{blocks} \]
\[ f, x = \text{identifiers} \]

Figure 3.7: An extended language for analysis, distinguishing codelets from system code, and identifying blocking functions.

3.1.4 Soft Termination with Task, System, and Blocking Code

Although this little language provides a significant subset of most programming languages, it likewise lacks many interesting features. Two in particular are blocking functions and a distinction between system and user code. Both of these features require extensions to the language. Figures 3.7 and 3.9 introduce the syntax and semantics of the extended language, and Figure 3.8 shows the evaluation context. Section 3.2.3 discusses our handling of blocking calls in our Java implementation of soft termination.

We first discuss the extensions made to this language. We then examine the soft termination transformation as applied to the extended language. We show that, once termination has been requested, a program in this language is guaranteed to terminate.

Language Extensions

The distinction between system and user functions is especially relevant as a feature of mobile code; it represents a distinction between trusted and untrusted code. This trust relationship is enforced by restricting the providers of untrusted tasks to declaring functions \textbf{task}. Only functions that are defined as part of the run-time system can be declared \textbf{system}; we assume that some preprocessor exists to remove any \textbf{system} declarations from user code. We use the term "system code" to refer to the collection of expressions declared in the bodies of all functions declared \textbf{system}, and the term "task" to refer to all other expressions in a program.
\[ E = [] | (f \ E) | (if_0 \ E \ M \ M) | (let \ (x \ E) \ M) | (try \ E \ M) | (CheckTermination) | (throw) \]

final states = \( V \) | error | (throw)

Figure 3.8: Evaluation context for reduction of this extended language.

\[
E[(f \ V)] \mapsto \begin{cases} 
E[V_0] & \text{if } (\text{define primitive } (f \ x)) \in \Gamma \\
& \text{and } \delta(f, V) = V_0 \\
E[[V / x] \ M]^\dagger & \text{if } (\text{define } L \ (f \ x) \ M) \in \Gamma \\
E[V_1] & \text{(that is, } f \text{ is declared blocking)} \\
& \text{and } \delta'(f, V) = V_1 \\
E[(f \ V)] & \text{if } (\text{define blocking } (f \ x) \ g) \in \Gamma \\
& \text{and } \delta'(f, V) \text{ is undefined} \\
E[V_2] & \text{(that is, } f \text{ is declared non-}
\text{blocking) and } \delta'(g, V) = V_2 \\
E[blocks] & \text{if } (\text{define blocking } (g \ x) \ f) \in \Gamma \\
& \text{and } \delta'(g, V) \text{ is undefined} \\
error & \text{otherwise}
\end{cases}
\]

The value of \( \delta'(f, V) \) and whether it is defined depend on external conditions.

\[ E[(\text{CheckTermination})] \mapsto E[1] \text{ or } E[0] \quad \text{(depending on external conditions)} \]
\[ E[(\text{if}_0 \ 0 \ M_1 \ M_2)] \mapsto E[M_1] \]
\[ E[(\text{if}_0 \ blocks \ M_1 \ M_2)] \mapsto error \quad \text{if } V \neq 0, \text{blocks} \]
\[ E[(\text{if}_0 \ V \ M_1 \ M_2)] \mapsto E[M_2] \]
\[ E[(\text{let} \ (x \ V) \ M)] \mapsto E[[V / x] \ M]^\dagger \]
\[ E[(\text{try} \ V \ M)] \mapsto E[V] \]
\[ E[(\text{try} \ (\text{throw}) \ M)] \mapsto E[M] \]
\[ E[(f \ (\text{throw}))] \mapsto E[(\text{throw})] \]
\[ E[(\text{if}_0 \ (\text{throw}) \ M_1 \ M_2)] \mapsto E[(\text{throw})] \]
\[ E[(\text{let} \ (x \ (\text{throw})) \ M)] \mapsto E[(\text{throw})] \]
\[ E[x] \mapsto error \quad \text{(unbound variables)} \]
\[ \delta(blocks?, \nu) = \begin{cases} 
1 & \text{if } \nu = blocks \\
0 & \text{otherwise}
\end{cases} \]

Figure 3.9: Operational semantics for the extended language described in Figure 3.7. This is largely a superset of the semantics of the original language, described in Figure 3.3. As in the original language, the expansion \([V / x] \ M\) indicates that every instance of \(x\) in \(M\) should be replaced with the corresponding \(V\), according to the standard rules of lexical scope. These are defined in Figure 3.4.
(1) $\mathcal{X}_g[\Gamma M] = \mathcal{X}_g[\Gamma] \mathcal{X}_g[M]

(2) $\mathcal{X}_g[D_1 \ldots D_n] = \mathcal{X}_g[D_1] \ldots \mathcal{X}_g[D_n]

(3a) $\mathcal{X}_g[\text{(define task} 
\begin{array}{l}
(f \ x) M \end{array})] = \text{(define task} (f \ x) \mathcal{X}_g[M])

(3b) $\mathcal{X}_g[\text{(define system} 
\begin{array}{l}
(f \ x) M \end{array})] = \text{(define system} (f_{\text{wrapper}} \ x) 
\begin{array}{l}
\text{(let} (t (f_{\text{nonblocking}} \ x)) (t \text{if0} \text{(blocks?} \ t) t 
\begin{array}{l}
\text{if0} \text{(CheckTermination} (f_{\text{wrapper}} x) \text{(throw})))))
\end{array})
\end{array})
\text{(define blocking} (f_{\text{blocking}} x) f_{\text{nonblocking}})
\text{Where the identifier} t \text{occurs nowhere else in the program.}

(3d) $\mathcal{X}_g[\text{(define primitive} 
\begin{array}{l}
(f \ x) \end{array})] = \text{(define primitive} (f \ x))$

\begin{align*}
(4) \mathcal{X}_g[f M] &= \begin{cases} 
\text{(let} (t \mathcal{X}_g[M]) 
\begin{array}{l}
\text{if} \text{(define task} 
\begin{array}{l}
(f \ x) M \end{array}) \in \Gamma 
\end{array}) 
\text{(X}_g[f] \mathcal{X}_g[M]) 
\text{Where identifier} t \text{occurs nowhere else in the program.} 
\text{Otherwise} 
\end{cases} \\
&= \text{f}_{\text{wrapper}} \\
&\text{if} \text{(define blocking} (f \ x) 
\begin{array}{l}
f_{\text{nonblocking}} \in \Gamma 
\end{array}) 
\text{(causing the definition of} f_{\text{wrapper}} \text{via rule 3c above)}
\text{Otherwise}
\end{align*}

(5) $\mathcal{X}_g[f] = \begin{cases} 
(\text{if0} \mathcal{X}_g[M_1] \mathcal{X}_g[M_2] \mathcal{X}_g[M_3]) 
\text{if} \text{(define task} 
\begin{array}{l}
(f \ x) M \end{array}) \in \Gamma 
\end{cases}
\text{Otherwise}

(6) $\mathcal{X}_g[(\text{try} \ M_1 M_2)] = (\text{try} \mathcal{X}_g[M_1] \mathcal{X}_g[M_2])

(7) $\mathcal{X}_g[(\text{let} (x M_1) M_2)] = (\text{let} (x \mathcal{X}_g[M_1]) \mathcal{X}_g[M_2])

(8) $\mathcal{X}_g[(\text{throw})] = (\text{throw})

(9) $\mathcal{X}_g[(\text{CheckTermination})] = (\text{CheckTermination})

(10) $\mathcal{X}_g[V] = V

(11) $\mathcal{X}_g[x] = x

Figure 3.10: The soft termination transformation for this extended language.
Because of the added number of other function types, all of which, including those behaving like primitive operations, are declared explicitly, we have also chosen to declare primitive operations explicitly using the **primitive** keyword. This is similar to how Java declares native methods.

The extended syntax greatly complicates how function applications are evaluated. If a function being applied has been declared **primitive**, the application reduces to the value of the $\delta$ function applied to that operation. If the function is declared **system** or **task**, the application reduces to the body of the function. Evaluation of **blocking** function applications is described below. If the function is not declared in the program, the application reduces to **error**.

**Blocking Functions.** Blocking functions are those functions that are known to block. That is, if a return value is not available, such functions may stall indefinitely waiting for the value to become available. Blocking functions generally provide some service of the underlying operating system, such as I/O operations. They are generally called through some interface provided by the operating system, and treated in a fashion similar to primitives in the language. As a result, we treat their definition and evaluation in a manner similar to how we treat primitive operations.

In particular, we define a function $\delta'$, similar to the $\delta$ function used to define primitives. $\delta'$ is a nondeterministic function indicating, based on external conditions, whether the application is defined and to what value the corresponding blocking function application should reduce. If the $\delta'$ function is undefined, this indicates that a return value is not available for the blocking function.

The **blocking** function declaration must name one additional function, which we call $f_{\text{nonblocking}}$. This function is a nonblocking version of the blocking function, which we call $f_{\text{blocking}}$. That is, when $\delta'$ is defined, applications of these two functions reduce to the same value. The behavior of these functions only differs when $\delta'$ is undefined. In this case, the expression $(f_{\text{nonblocking}} V)$ reduces to the value **blocks**, while the expression $(f_{\text{blocking}} V)$ reduces to itself, and will continue to reduce to itself as long as $\delta'$ is undefined. The third
and fourth reduction rules for function applications define the behavior for applications of $f_{\text{blocking}}$; the fifth and sixth rules define the behavior for applications of $f_{\text{nonblocking}}$.

**Description of $\mathcal{X}_2$ Transformation**

Because we have extended this language, we must now extend the soft termination transformation. We call this new transformation, described in Figure 3.10, $\mathcal{X}_2$, to distinguish it from the $\mathcal{X}$ transformation in Figure 3.6.

Rules 3a through 3d describe the transformation on function definitions. Rules 3a and 3b describe how the transformation continues recursively on **system** and **task** function definitions, and Rule 3d says primitive function declarations are unmodified. Rule 3c says a wrapper function is created for every **blocking** function. The wrapper uses the nonblocking function declared with each **blocking** function to simulate the effect of applying the blocking function.

This wrapper alternately polls the nonblocking function and checks to see if termination has been indicated. When the nonblocking function application reduces to a value other than **blocks**, the wrapper function application reduces to this value. If termination is ever indicated, the wrapper throws an exception. Otherwise, the wrapper function is recursively applied, and the process repeats. We use the identifier $f_{\text{wrapper}}$ to refer to such a wrapper function generated for some blocking function $f_{\text{blocking}}$. Note that different programming languages choose different abstractions for managing blocking I/O primitives. In Section 3.2.3, we show how this works in Java.

Rule 4 describes how function applications are handled for $\mathcal{X}_2$. If the function being applied is a **task** function, the termination check is added at the call site to the function, just as in the $\mathcal{X}$ transformation. If it is a **system** function, however, no termination checks are added. Adding termination checks on applications of **system** functions may cause an unexpected (throw) to be evaluated in a critical section of a **system** function, as described in Section 3.1.1. Recall from Section 3.1.2 that UNIX treats calls to system code similarly.

Note that where the original **system** function made an up-call to a task, the $\mathcal{X}_2$ transformation may cause an exception to be thrown at the call site. System code is responsible for catching this exception and proceeding appropriately. Because exceptions are already
a valid result of applying task functions, system code must already be prepared to handle this case. As a result, this adds no new constraints on system code.

Rule 5 describes how applications of blocking functions are replaced with applications of the corresponding nonblocking wrapper function. This guarantees that no blocking function is ever applied in the transformed program, simplifying the termination proof. The remainder of the rules describe how the transformation continues recursively on expressions.

Safety Property of System Code

For this extended language, we wish to prove that a program transformed by $\mathcal{X}_2$ terminates in a finite number of steps, given that (CheckTermination) is 1. This is complicated by the fact that system functions are not guaranteed to terminate, even after the program has been transformed by $\mathcal{X}_2$. As a result, we must assume some safety property of the system code in order to prove termination.

The intuition behind this safety property is that if the program diverges or reaches a fixed point, it is not the fault of system code. A program diverges when, as reduction rules are applied, the length of the program grows without bound. A program reaches a fixed point when, as reduction rules are applied, the result is the same program. The property is stated as follows:

**Definition 3.1 (Safety Property of System Code).**

Suppose for all declarations

\[(\text{define system} \ (f_{\text{system}} \ x) \ M) \in \Gamma,\]

for every application \((f_{\text{task}} \ V_0)\) and \((f_{\text{blocking}} \ V_1)\) in \(M\), where

\[(\text{define task} \ (f_{\text{task}} \ x) \ M_2) \in \Gamma \text{ and}\]

\[(\text{define blocking} \ (f_{\text{blocking}} \ x) \ f_{\text{nonblocking}}) \in \Gamma,\]

Even in system functions, blocking function applications are replaced with applications of the corresponding nonblocking wrapper. This can cause an unexpected (throw) to be evaluated in system code; to prevent this in our Java implementation of soft termination, blocking method calls from system code are not wrapped, as described in Section 3.2.3. Since the termination result we prove in Section 3.1.4 includes the assumption that blocking function applications return in finite time, this result is still valid.
there exists an integer $c > 0$ such that

\[
E[(f_{task} V_0)] \Rightarrow^c E[V_0'] | E[\text{(throw)}] \text{ and }
E[(f_{blocking} V_1)] \Rightarrow^c E[V_1'] | E[\text{(throw)}]
\]

for values $V_0', V_1'$. Then there is an integer $c_2 > 0$ such that

\[
E[(f_{system} V)] \Rightarrow^{c_2} E[V'] | E[\text{(throw)}]
\]

for every system function $f_{system}$, for some value $V'$.

The safety property states that, if every task and blocking function application in any system function reduces to a value or to (throw) in a finite number of steps, then every system function reduces to a value or to (throw) in a finite number of steps. This property applies to system functions in the original program, before transformation by $\mathcal{X}_2$. It does not come automatically, but we assume the authors of system functions guarantee it for their code. Recall from Section 3.1.4 that untrusted functions are never declared system.

By this property, system code will always terminate unless it applies a task or blocking function that either diverges or reaches a fixed point. As a result, system code acting alone will always return in finite time, and is never at fault if a system function fails to return.

**Proof of Termination**

In order to prove termination, we start with the grammar $M$ of all possible expressions from Figure 3.7. After transforming $M$ by $\mathcal{X}_2$, we call the resulting set of expressions $M_{lock_2}$ to distinguish from $M_{lock}$ defined in Section 3.1.3; like $M_{lock}$, $M_{lock_2}$ is a subset of
Assuming (CheckTermination) is 1, we wish to show the program enters a “locked” state, where termination is guaranteed.

\[
M_{\text{lock}2} = \begin{cases} 
(\text{if}_0 (\text{CheckTermination}) \ldots) & \text{if } (\text{define task } (f \ x) \ M) \in \Gamma \quad (a) \\
(\text{if}_0 1 (f \ V) \ (\text{throw}) \ldots) & \text{if } (\text{define task } (f \ x) \ M) \in \Gamma \quad (b) \\
(\text{if} \ (f \nonblocking \ V) \ldots) & \text{if } (\text{define blocking } (f \blocking x) \ldots) \nonblocking) \in \Gamma \quad (c) \\
(f \ V) & \text{if } (\text{define primitive } (f \ x)) \in \Gamma \quad (d) \\
(f \ V) & \text{if } (\text{define system } (f \ x) \ M) \in \Gamma \quad (e) \\
(\text{CheckTermination}) & \\
(\text{if}_0 M_{\text{lock}2} M_{\text{lock}3} M_{\text{lock}2}) & \\
(\text{let } (x \ M_{\text{lock}3} M_{\text{lock}2}) & \\
(\text{try } M_{\text{lock}2} M_{\text{lock}2}) & \\
(\text{throw}) & \\
x & \\
V & \\
\end{cases}
\]

Note in particular, that \(X_2[M] \subseteq M_{\text{lock}2}\). To prove termination, we must show that, as long as (CheckTermination) is 1, \(M_{\text{lock}2}\) is closed under program stepping and that the syntactic length of the program is decreasing. Closure is defined in detail in Section 3.1.3.

**Lemma 3.4 (The Closure Property).** Suppose (CheckTermination) is 1. For arbitrary expressions \(M \) and \(M'\), if \(M \rightarrow M'\) and \(M \in M_{\text{lock}2}\), then \(M' \in M_{\text{lock}2}\).

**Proof.** By inspection, for all possible expressions in \(M_{\text{lock}2}\), we observe that our semantics preserve closure. \(\square\)

**Lemma 3.5 (The Syntactic Length Property).** If (CheckTermination) is always 1, then for any \(M \in M_{\text{lock}2}\), there is some integer \(c > 0\) and \(M' \in M_{\text{lock}2}\) such that \(M \rightarrow^c M'\) and \(|M'| < |M|\).

**Proof.** That is, for any program in \(M_{\text{lock}2}\), after \(c\) reduction steps, it will have gotten smaller. Since programs are finite length, a program with these properties will terminate in a finite number of steps.

Based on the operational semantics, the reductions of expressions \(f\) through \(l\) all make a program strictly smaller. Likewise, applications of primitive operations and nonblocking functions (which is to say, expressions \(c\) and \(d\)) reduce to values in one step. Note that applications of blocking functions may never reduce to a value; however, the \(X_4\) transformation removes all such applications from the program.
Expressions $a$ and $b$ are applications of task functions fenced by (CheckTermination) checks. Since (CheckTermination) is 1, these expressions reduce to (throw). The final case is expression $e$, applications of system functions. This expression must be considered in two cases. The first case is if the function being applied is a blocking wrapper function, $f_{\text{wrapper}}$ (i.e., functions created by the $X_2$ transformation applied to a blocking function declaration). The resulting expression is in $M_{lock_2}$, and all applications of system functions from within $f_{\text{wrapper}}$ are fenced by a termination check. As a result, as long as (CheckTermination) is 1, applications of $f_{\text{wrapper}}$ reduce to a value in finitely many steps.

To address the second case, of any other functions declared system being applied, we recall the safety property in Section 3.1.4. This states that if, at every point where any system function applies a task or blocking function, that application reduces in finitely many steps, then the application of every system function reduces in finitely many steps.

By the $X_2$ transformation, wherever an application of the original system function reduces $(f_{\text{task}} V)$, for some task function $f_{\text{task}}$, the transformed function reduces $(\text{if}_0(\text{CheckTermination}) (f_{\text{task}} V) \text{ (throw)})$. Wherever the original system function reduces $(f_{\text{blocking}} V)$ for some blocking function $f_{\text{blocking}}$, the transformed function reduces $(f_{\text{wrapper}} V)$, where $f_{\text{wrapper}}$ is the wrapper function that corresponds to $f_{\text{blocking}}$.

We have already shown that, as long as (CheckTermination) is 1, for every wrapper function $f_{\text{wrapper}}$, for some $c > 0$,

$$E[(f_{\text{wrapper}} V)] \rightarrow^c E[V_0] \mid E[(\text{throw})]$$

for some value $V_0$. Likewise we can see that, as long as (CheckTermination) is 1,

$$E[(\text{if}_0 (\text{CheckTermination}) \ldots (\text{throw}))] \rightarrow^c E[(\text{throw})].$$

As a result, wherever the original system function applied either a task or a blocking function, the expression to which $X_2$ transforms this application reduces in a finite number of steps. By applying the safety property, we can conclude that an application of any system function reduces to a value in a finite number of steps, and all expressions in $e$ have the syntactic length property. □
Lemma 3.6. When (CheckTermination) becomes set, a program being evaluated enters $M_{\text{lock}2}$ within at most two reduction step.

Proof. When a program is in $M_{\text{lock}2}$ and (CheckTermination) is set, by Lemma 3.4, the program is guaranteed to stay in $M_{\text{lock}2}$. If the program is not in $M_{\text{lock}2}$ when (CheckTermination) is set, then it is either evaluating an application of a task function or a conditional which will result in such an application. Within one or two steps, then, it will be evaluating the body of the application, $\mathcal{X}_2[M]$. As noted above, this is defined to be in $M_{\text{lock}2}$. In other words, a transformed program that is not in $M_{\text{lock}2}$ is guaranteed to enter $M_{\text{lock}2}$ within two steps. \hfill {\Box}

Theorem 3.3. Any program in $M$, after being transformed by $\mathcal{X}_2$, is guaranteed to terminate in a finite number of steps, once the (CheckTermination) condition is raised.

Proof. By Lemma 3.6, within two steps of the (CheckTermination) condition being raised, the program is guaranteed to be in $M_{\text{lock}2}$. By Lemma 3.4, the program is guaranteed to remain in $M_{\text{lock}2}$. By Lemma 3.5, then, the program reduces to some smaller program in a finite number of steps. Since program lengths are finite, the program is guaranteed to reduce to some non-reducible expression (a value or (throw)) in a finite number of steps, once the (CheckTermination) condition is raised, given the safety constraint on system code stated in section 3.1.4. \hfill {\Box}

3.2 Java Implementation

In an effort to understand the practical issues involved with soft termination, we implemented it for Java as a transformation on Java bytecodes. Our implementation relies on a number of Java-specific features. We also address a number of Java-specific quirks that we would not expect to exist in other language systems. Examples showing how the transformations as implemented alter Java source language are shown in Figures 3.11 through 3.13. Although the transformations actually operate on Java bytecode, there is a direct mapping from the targeted bytecodes to the corresponding Java source representation, and using Java source language is clearer.
void foo() {
    if (termination_signal) {
        termination_handler();
    }
}

Figure 3.11: The soft termination transformation applied to any function definition. See Section 3.2.1 for a description of this transformation. This example takes into account the optimizations discussed in Section 3.2.5.

3.2.1 Termination Check Insertion

Java compilers normally output Java bytecode. Every Java source file is translated to one or more class files, later loaded dynamically by the JVM as the classes are referenced by a running program. JVMs know how to load class files directly from the disk or indirectly through “class loaders,” invoked as part of Java’s dynamic linking mechanism. A class loader, among other things, embodies Java’s notion of a name space. Every class is tagged with the class loader that installed it, such that a class with unresolved references is linked against other classes from the same source. A class loader provides an ideal location to rewrite Java bytecode, implementing the soft termination transformation. A task appears in Java as a set of classes loaded by the same class loader. System code is naturally loaded by a different class loader than tasks, allowing us to simplify the implementation by applying different transformations to tasks and system code. Our implementation uses the CFParse3 and JOIE [30] packages, which provide interfaces for parsing and manipulating Java class files.

The basic structure of our bytecode modification is exactly as described in Section 3.1.3. A static Boolean field, initially set to false, is added to every Java class. The CheckTermination operation, implemented inline, tests if this field is true, and if so, calls a handler method that decides whether to throw an exception. As an extension to the semantics of Figure 3.10, we allow threads and thread groups to be terminated as well as specific tasks,

void foo() {
    void foo() {
        if (termination_signal) {
            termination_handler();
        }
    }

    while (...) {
        while (...) {
            ...
            if (termination_signal) {
                termination_handler();
            }
        }
    }
}

Figure 3.12: The soft termination transformation applied to any loop. See Section 3.2.2 for a description of this transformation.

regardless of the running thread. The termination handler, when invoked, looks its caller and current thread up in a list of known termination targets. Note that, if the Boolean field is set to false, the run-time overhead is only the cost of loading and checking the value, and then branching forward to the remainder of the method body. Figure 3.11 shows how the soft termination transform would be applied to a Java method declaration.

3.2.2 Control Flow

Java has a much richer control flow than the little language introduced earlier. First and foremost, Java bytecode has a general-purpose branch instruction. We do nothing special for forward branches, but we treat backward branches as if they were method invocations and perform the appropriate code transformation. An additional special case we must handle is a branch instruction that targets itself. The effect of transforming a method with loops is shown in Figure 3.12.

Java bytecode also supports many constructions that have no equivalent Java source code representation. In particular, it is possible to arrange for the catch portion of an exception handler to be equal to the try portion. That means an exception handler can be defined to handle its own exceptions. Such a construction allows for infinite loops
void foo() {
    if (termination_signal) {
        termination_handler();
    }
    blocking.bar(...);
}

Bar blocking.wrapper.bar(...) {
    register.blocking(
        Thread.currentThread());
    // Uses stack inspection
    Bar tmp = blocking.bar();
    unregister.blocking(
        Thread.currentThread());
    return tmp;
}

Figure 3.13: The soft termination transformation applied to a blocking call. This figure includes an outline for the definition of the blocking call wrapper function. The signature of function blocking.wrapper.bar() is the same as that for function blocking.bar(). See Section 3.2.3 for a description of this transformation.

without any method invocation or explicit backward branching. Although such code is not allowed according to the JVM specification [85], the bytecode verifier we used treats such constructions as valid. We specifically check for and reject programs with overlapping try and catch blocks.

Lastly, Java bytecode supports a notion of subroutines within a Java method using the jsr and ret instructions. jsr pushes a return address on the stack, and ret consumes this address before returning. The Java bytecode verifier imposes a number of restrictions on how these instructions may be used. In particular, a return address is an opaque type that may be consumed at most once. The verifier’s intent is to ensure that these instructions may be used only to create subroutines, not general-purpose branching. As such, we instrument jsr instructions the same way we would instrument a method invocation and we do nothing for ret instructions.
3.2.3 Blocking Calls

To address blocking calls, we wish to follow the $X_e$ transformation outlined in Section 3.1.4. Luckily, all blocking method calls in the Java system libraries are native methods (implemented in C) and can be easily enumerated and studied by examining the source code of the Java class libraries.

Using a polling model for terminating blocking calls simplifies analysis, but it is not a very practical implementation. This is because of the processing time required for polling. However, Java provides a mechanism for interrupting blocking calls, `Thread.interrupt()`. If a thread has called a blocking function and is blocking, this method, when called on the thread, causes the blocking method to throw a `java.lang.InterruptedException` or `java.io.InterruptedIOException` exception.

As in the $X_e$ transformation described in Section 3.1.4, we still wrap blocking function calls; and like $X_e$, the wrapper returns if either a value is returned or termination is signaled. However, instead of polling a nonblocking function, the wrapper uses the interruption support already inside the JVM. When termination is requested for a task, if a corresponding thread is in a blocking call, that thread is interrupted with `Thread.interrupt()`.

To accomplish this, we must track which threads are currently blocking and the tasks on behalf of which they are blocking. The wrapper functions now get the current thread and save it in a global table for later reference. In order to learn the task on whose behalf we are about to block, we take advantage of the stack inspection primitives built into modern Java systems [137, 66].

Stack inspection provides two primitives that we use: `java.security.AccessController.doPrivileged()` and `getContext()`. `getContext()` returns an array of `ProtectionDomain` s that map one-to-one with tasks. The `ProtectionDomain` identities are then saved alongside the current thread before the blocking call is performed. When we wish to terminate a task, we look up whether it is currently in a blocking function call, and if so, we interrupt the corresponding thread.

Taking advantage of another property of Java stack inspection, we can distinguish between blocking calls being performed on behalf of system code and those being performed
indirectly by a task. We do not want to interrupt a blocking call if system code is depending on its result and system state could become corrupted if the call were interrupted. On the other hand, we have no problem interrupting a blocking call if only a task is depending on its result. Java system code already uses `getContext()` to get dynamic traces for making access control checks. The `doPrivileged()` mechanism is likewise already used in the Java API library to identify code to run with system privileges. This is used for performing security-sensitive operations that tasks themselves should not be authorized to perform. Thus we overload the semantics of these existing security primitives to include whether blocking calls should be interrupted.

In principle, any code where preserving system integrity upon termination is important should already be wrapped with `doPrivileged()`, and thereby executed with system privileges. However, verifying this would require an audit of the Java API library. The effect of this transformation on Java source is shown in Figure 3.13.

### 3.2.4 Invoking Termination

Our system supports three levels of granularity for termination: termination of individual threads, termination of thread groups, and termination of tasks. To terminate a thread or thread group, we must map the threads we wish to terminate to the set of tasks potentially running those threads and set the termination signal on all classes belonging to the target task. Furthermore, we must check if any of these threads are currently blocking and interrupt them (see Section 3.2.3). At this point, the thread requesting termination performs a `Thread.join()` on the target threads, waiting until they complete execution. Once all target threads have completed, the termination signals are cleared and execution returns to normal.

If multiple threads are executing concurrently over the same set of classes and only one is terminated, the termination handler will be invoked for threads not targeted, only to return shortly thereafter. These threads will experience degraded performance while the target thread is still running.

In the case where we wish to terminate a specific task, disabling all its classes forever, we simply set the termination signal on all classes in the task and immediately return. Any
code that invokes a method on a disabled class will receive an exception indicating the class has been terminated.

Once a task has been signaled to terminate, if a task’s thread is executing in a system class at the time, execution continues until the thread returns to a user class. If the task is currently making a blocking call, the call is interrupted and the thread resumes execution. Once the thread has resumed executing in the user’s class, it becomes subject to the soft termination system.

For all task threads that are executing within the task, if they try to call a method within the task, the method fails with an exception. If they try to perform a backward branch, the soft termination code will throw an exception. In all cases, each thread of control unwinds, preventing the task from performing any meaningful work. Finally, if any other task or the system makes a call into this task, it will fail immediately, preventing the task from hijacking the caller thread for the task’s own use. As proven in Section 3.1.4, the task is guaranteed to terminate.

Note that termination requests can be handled concurrently. A potential for deadlock occurs when a thread requests its own termination, or when a cycle of threads request each others’ termination. When a user is manually terminating threads or tasks, this would not be an issue. However, care should be taken to prevent untrusted tasks from invoking the termination operations. For this reason, these operations are protected using the same security mechanisms as other Java privileged calls.

## 3.2.5 Optimizations

If a Java method contains a large number of method invocations, the transformed method may be significantly larger than the original, potentially causing performance problems. To address this concern, we observe that we get similar semantics by moving the soft termination check from the call sites to the entry points of methods. Every function for which we performed a termination check before calling now instead begins with a termination check, so the resulting program will behave the same. Figure 3.11 shows the effect of using this optimization in transforming a Java function.
In addition, we implemented an optimization to statically determine if a method has no outgoing method calls (i.e., is a leaf method). For leaf methods, a termination check at the beginning of the method is unnecessary. If the method has loops, they will have their own termination checks. If not, the method is guaranteed to complete in a finite time. Regardless, removing the initial termination check from leaf methods preserves the semantics of soft termination and should offer a significant performance improvement, particularly for short methods such as "getter" and "setter" methods.

A more aggressive optimization, which we have not yet performed, would be an interprocedural analysis of statically terminating methods. A method that only calls other terminating methods and has no backward branches will always terminate. Likewise, we have not attempted to distinguish loops that can be statically determined to terminate in a finite time (i.e., loops that can be completely unrolled). Such analyses could offer significant performance benefits to a production implementation of soft termination.

3.2.6 Synchronization

A particularly tricky aspect of supporting soft termination in a Java system is supporting Java's synchronization primitives. The Java language and virtual machine specifications are not clear on how the system behaves when a deadlock is encountered [67, 85]. With Sun's JDK 1.2, the only way to recover from a deadlock is to terminate the JVM. Obviously, this is an unsatisfactory solution. Ideally, we would like to see a modification to the JVM where locking primitives such as the monitorenter bytecode are interruptible, like other blocking calls in Java. We could then apply standard deadlock detection techniques and choose the threads to interrupt.

It is also possible to construct Java classes where the the monitorenter and monitorexit bytecodes, which acquire and release system locks, respectively, are not properly balanced. Despite the fact that there exist no equivalent Java source programs, current JVM bytecode verifiers accept such programs. This makes it possible for a malicious program to acquire a series of locks and terminate without those locks being released until the JVM terminates. Our current system makes no attempt to address these issues.
3.2.7 Thread Scheduling

Our work fundamentally assumes the Java thread system is preemptive. This was not the case in many early Java implementations. Without a preemptive scheduler, a malicious task could enter an infinite loop and no other thread would have the opportunity to run and request the termination of the malicious thread. We can work around such an attack by inserting calls to `Thread.yield()` at the cost of some additional slowdown.

3.2.8 System Code Safety

Our work also assumes that all system methods that may be invoked by a task will either return in a finite time or will reach a blocking native method call which can be interrupted. This property of system code is stated and justified in Section 3.1.4. It may be possible to construct an input to system code that will cause the system code itself to have an infinite loop. Addressing this concern would require a lengthy audit of the system code to guarantee there exist no possible inputs to system functions that may cause infinite loops.

3.2.9 Memory Consistency Models

The Java language defines a relaxed consistency model where updates need not be propagated in the absence of locking primitives. In our current prototype, we use no synchronization primitives when accessing the termination flag. Since external updates to the termination signal could potentially be ignored by the running method, this could defeat the soft termination system.

Instead, we take advantage of Java's `volatile` modifier. This modifier is provided to guarantee that changes must be propagated immediately [67]. On the benchmark platform we used, the performance impact of using `volatile` versus not using it is negligible. However, on other platforms, especially multiprocessing systems, this may not be the case.

3.2.10 Defensive Security

Our prototype implementation makes no attempt to protect itself from bytecode designed specifically to attack the termination system (e.g., setting the termination flag to `false`, ei-
ther directly or through Java's reflection interface). Such protection could be added as a verification step before the bytecode rewriting.

3.3 Performance

We measured the performance of our soft termination system using Sun Microsystems Ultra 10 workstations (440 MHz UltraSPARC II CPUs with 128 MB memory, running Solaris 2.6), and Sun's Java 2, version 1.2.1 build 4, which includes a just-in-time compiler (JIT). A JIT compiles the Java bytecodes to native machine code at run-time, eliminating the overhead of interpreting Java code. Our benchmarks were compiled with the corresponding version of javac with optimization turned on.

We used two classes of benchmark programs: microbenchmarks that test the impact of soft termination on various Java language constructs (and also measuring worst-case performance), and some real-world applications. We measured the performance of these systems in three configurations: their original unmodified state, their state after being rewritten, and their state after being rewritten using the leaf method optimization discussed in Section 3.2.5. Generally, when we discuss results in this section, we refer to the optimized numbers because these better reflect the performance of a production soft termination system.

3.3.1 Microbenchmarks

We first measured a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, method and field accesses, exception handling, synchronization, and I/O. We used a microbenchmark package developed at the University of California, San
Figure 3.14: This graph illustrates the performance of rewritten microbenchmark class files relative to the performance of the corresponding original class files. The graph measures how much longer the benchmarked class took to run than the original class. The left bars show the overhead for the original transformed classes. The right bars show the overhead for the classes transformed using the leaf method optimization discussed in Section 3.2.5.

Diego, and modified at the University of Arizona for the Sumatra Project. The results are shown in Figure 3.14.

As one would expect, loops suffered the worst overheads, of between 100 and 170%. In the transformed version of the loop, the cost of the termination check is roughly the same as the cost of the looping construct itself, so it is sensible to see such a performance degradation. In addition, in some cases, the added termination checks might inhibit loop optimizations.

For other microbenchmarks, we saw much smaller overheads. The overhead of handling exceptions, performing synchronization, or doing I/O operations dominates the cost of checking for termination. The largest overhead of these was 13.7% for the synchronization microbenchmark. This overhead can be attributed to performing the termination check once for each iteration of the benchmark. For the I/O and exception-handling mi-
crobenchmarks, the performance figures are much better. Since I/O and exception handling are relatively costly operations, modifications do not have as significant an impact on performance.

We observe that the leaf method optimization generally has some performance benefit. The loop method invocation microbenchmark shows the most dramatic improvement; the optimized benchmark has nearly half the overhead of the unoptimized benchmark. In one case, the exception handling benchmark, the optimized program ran roughly 1% slower than the unoptimized program. Similar behavior occurred in the Linpack application benchmark. The optimized programs are genuinely performing fewer termination checks, but still have longer run-times. The culprit appears to be Sun's JIT compiler (sun-wjct). When the benchmarks are run with the JIT disabled, the optimized programs are strictly faster than the unoptimized programs. We have observed similar deviant behavior with Sun's HotSpot JIT running on SPARC/Solaris and x86/Linux. We have sent an appropriate bug report to Sun.

3.3.2 Application Benchmarks

We benchmarked the real-world applications JavaCup, Linpack, Jess, and OTP. These programs were chosen to provide sufficiently broad insight into our system's performance. JavaCup is a LALR parser-generator for Java, and Jess is an expert system shell. These programs were chosen to demonstrate how soft termination performed in tasks that are more dependent on I/O and computation than on iteration. Linpack is a loop-intensive floating-point benchmark. OTP is a one-time password generator which uses a cryptographic hash function. The results are shown in Figure 3.15.

For the JavaCup test, we generated a parser for the Java 1.1 grammar. When rewritten, it ran 6% slower. For the Jess test, we ran several of the sample problems included with Jess through the system, and calculated the cumulative run-times. This program, when

---

5http://www.cs.princeton.edu/~appel/modern/java/CUP/
6http://netlib2.cs.utk.edu/benchmark/linpackjava/
7http://herzberg.ca.sandia.gov/jess/
8http://www.cs.umd.edu/~harry/jotp/
Figure 3.15: This graph illustrates the performance of rewritten application class files relative to the performance of the corresponding original class files. The graph measures how much longer the benchmarked class took to run than the original class. The left bars show the overhead for the original transformed classes. The right bars show the overhead for the classes transformed using the leaf method optimization discussed in Section 3.2.5.

Rewritten, ran 3% slower. Both JavaCup and Jess represent applications that do not make extensive use of tight loops. Instead, these applications spend more of their time performing I/O and mathematical or symbolic computations. The performance results reflect this.

For the Java OTP generator, we generated a one-time password from a randomly chosen seed and password, using 200,000 iterations. There was a 18% increase in run-time. For the Linpack benchmark, there was a 25% increase in run-time. Linpack is a loop-intensive program, whereas OTP makes extensive use of method calls as well as loops. As a result, we would expect the overhead from soft termination checks to be somewhat higher than for the other two application benchmarks. Note in particular the benefit OTP got from the leaf method optimization.

3.3.3 Termination Check Overhead

To gauge the actual impact of our class file modifications, we counted the number of times we checked the termination flag for each benchmark. This gave us an idea of how much
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Termination Checks</th>
<th>Application Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unoptimized</td>
<td>Optimized</td>
</tr>
<tr>
<td>Empty Loop</td>
<td>10,000,043</td>
<td>10,000,002</td>
</tr>
<tr>
<td>Loop Field Operation</td>
<td>55,000,257</td>
<td>55,000,012</td>
</tr>
<tr>
<td>Loop Method Invocation</td>
<td>60,000,117</td>
<td>30,000,006</td>
</tr>
<tr>
<td>Exceptions</td>
<td>21,000,118</td>
<td>15,000,008</td>
</tr>
<tr>
<td>Synchronization</td>
<td>60,000,115</td>
<td>40,000,008</td>
</tr>
<tr>
<td>Input/Output</td>
<td>200,958</td>
<td>200,006</td>
</tr>
</tbody>
</table>

Table 3.1: Average total number of termination checks performed for each benchmark.

Extra work each rewritten program was actually doing. The results for all benchmarks are listed in Table 3.1.

The microbenchmarks essentially perform one simple operation repeatedly in a tight loop; as expected, all microbenchmarks performed one termination check per loop iteration, with few additional overhead checks. These results translate to roughly 40,000,000 checks performed for every second of run-time overhead for all but the input/output microbenchmark. This evaluates to around 10 CPU cycles of normal computation for each check performed.

For the input/output microbenchmark, however, only around 970,000 termination checks are performed per second of overhead. This can be almost entirely attributed to the additional overhead of the blocking call management code. It is also important to keep in mind that the cost of performing I/O far outweighs the cost of termination checks.

The application benchmarks reflect the results of the microbenchmarks. OTP, which suffered a much greater performance impact from the modifications than either Jess or JavaCup, performed over four times as many checks as JavaCup. Although Linpack performed fewer checks than either, it is a much shorter-running benchmark. For all of these benchmarks, the results translate to between 15,000,000 and 29,000,000 checks per second of overhead.
These performance figures seem to indicate that for real-world applications, the slowdown will be roughly proportional to how much the application's performance is dependent on tight loops. Applications that have tight loops may experience a slowdown of at worst a factor of two and more commonly 15 to 25%. Applications without tight loops can expect a more modest slowdown, most likely below 7%. The number of termination checks the system can perform per second seems not to be a limiting factor in system performance.

3.4 Summary

We have introduced a concept we call soft termination, along with a formal design and an implementation for Java, that allows for asynchronous and safe termination of misbehaving or malicious tasks. Soft termination can be implemented without making any changes to the underlying language or run-time system. Although our implementation is for Java, the basic design of soft termination does not depend on Java, and can be used with a variety of different languages.

Our Java implementation relies solely on class-file bytecode rewriting, making it portable across Java systems and easier to consider applying to non-Java systems. In real-world benchmarks, our system shows a slowdown of 3 to 25%. This could possibly be further reduced if we could leverage a safe point mechanism already implemented within the JVM.
Chapter 4

Garbage Collector-Based Memory Accounting

In the last chapter, I developed a language-based mechanism for terminating tasks. The main impetus behind this work was to allow for safe and effective termination without impeding a codelet's ability to share state. The next operating system service I address is resource management, memory management in particular.

In this chapter, I describe the design and implementation of our memory accounting system. I describe the design of this system in Section 4.1. Section 4.2 discusses our implementation of memory accounting and its performance impact. Finally, in Section 4.3 I discuss the sorts of policy semantics this system supports, and I summarize this work in Section 4.4.

4.1 Design

There are several different ways for a language-based system to track the memory usage of its individual tasks. We first discuss some proposed solutions, and describe the hard problems raised by their failings. We then discuss the design of our system.

Since we wish to support general-purpose programming languages, computing static bounds on memory usage is computationally infeasible. Instead, we need a mechanism that can efficiently measure memory usage, even in the presence of data sharing across tasks and with garbage collection managing the memory. Given such a measurement mechanism, we can then envision policies that detect when the system is running out of free memory. Using the measured data, these policies could identify misbehaving tasks and terminate them, reclaiming their resources, while allowing other tasks to continue running uninterrupted.
Figure 4.1: In language-based systems, memory may be shared among multiple tasks, so determining the memory usage of a single task is difficult.

### 4.1.1 Instrumented allocation

One common mechanism for determining the memory usage of tasks is to rewrite the task’s code at load time. Memory allocations are instrumented to charge tasks with memory usage when they allocate objects, granting rebates when those objects are finalized. This approach has the benefit that no modifications are required to the underlying language runtime system. JRes [35] and Beg and Dahlin [9] both instrument memory allocations as a way to account for memory usage by tasks in Java.

However, there are several problems to using this approach. First, only allocation that explicitly occurs in the task is charged to that task. Any implicit allocation or allocation performed by the system on behalf of the task is not charged to it. In addition, in both JRes and Beg and Dahlin’s system, accounting is performed on a per-thread basis. If a “system” thread or a another task’s thread calls into a task, it could potentially be “tricked” into allocating memory on behalf of the task, giving that memory away “for free.”

Furthermore, tasks can share memory with one another (see Figure 4.1). A task may allocate a block of memory, share that memory with another task, and later drop its pointer. In most language-based systems, however, memory is kept alive if any live pointers to it exist. As a result, another task could, out of necessity or malice, hold memory live; the task that initially allocated that memory would be forced to keep paying for it.
Figure 4.2: In traditional operating systems, each process is its own, self-contained box, so determining the memory usage of a process is equivalent to measuring the size of the box.

### 4.1.2 Process abstractions

Another common mechanism for accounting for memory usage is to use process abstractions. In some systems, each task is allocated its own heap, and the memory usage charged to that task is the size of that heap. KaffeOS [6, 7] is a system for Java that, in conjunction with an explicit process-like abstraction for Java tasks, provides a separate heap for each task. The multitasking virtual machine (MVM) [34] and systems by Bernadat et al. [10], and van Doorn [133] similarly use separate heaps or memory spaces to facilitate accounting for memory. Some systems [88, 127] even go so far as to run the JVMs in separate Unix processes on separate machines.

These systems accurately account for memory a task keeps live. However, inter-task communications and memory sharing are severely restricted, limiting the usefulness of the language. In addition, these systems are implemented with nontrivial customizations to the VM. Adapting these ideas to a new VM can require significant engineering resources.

In some systems, function calls and memory references are artificially restricted (either through some mechanism built into the run-time system or using code-to-code transformations). In this case, instrumenting memory allocations and object finalization yields an accurate accounting for the amount of memory used by a task. Examples include J-Kernel [68], J-SEAL2 [13], and Luna [69]. These systems are more accurate than strictly instrumented allocation. However, they still restrict inter-task communications and memory sharing among tasks.
4.1.3 Garbage collection-based accounting

Once we allow object references to be shared across tasks, the task that allocates an object in memory may not necessarily be the task that ends up using the object or keeping it live. Once a reference to an object has been given out, anybody could potentially hold that object reference. Clearly, we would like to only charge tasks for the memory they are keeping live, rather than the memory they allocate.

Under this rationale, live objects should be charged to those tasks from which they are reachable in the graph of heap objects. Conveniently, tracing garbage collectors already traverse this graph to find the reachable objects and free the space occupied by unreachable objects. By carefully managing the order in which the GC does its work and having the GC report back to us on its findings, we can use the GC as our tool for measuring each task's live memory footprint.

A typical garbage collector works by starting at a defined root set of references and doing a graph traversal to find all the memory reachable from those references. Memory not reached during this graph traversal is garbage and can be used for allocating new objects. In our system, we augment the collector to sequentially trace all the reachable memory from each task's root set.

The root set of a task is a set of roots in memory defined to be affiliated with that task; for example, our implementation defines it to be the static fields of all the task's classes plus the execution stacks of all its threads. For each task, the collector traces all reachable memory from its root set. As it does so, it computes the sum of the sizes of the objects it has seen. Once the traversal is complete, that sum is charged to the task currently being processed. Once the collector finishes iterating over all the tasks, it makes one final pass, starting with the set of all roots not affiliated with any particular task. Any objects which have not yet been reached after this completes are unreachable.

Handling shared memory

Because each object is only processed once in each garbage collection cycle, this method will find less and less shared memory as it goes from the start to the end of the list of tasks.
Figure 4.3: On a pass through the garbage collector, the first task to be scanned (in this case, task A) is charged for all memory reachable from it, while the last task scanned (in this case, task C) is charged for memory reachable only from it.

Figure 4.4: On subsequent invocations of the scan, the order of scanning is rotated; task A, the first task scanned in the previous example, is the last task scanned in this example. This process gives a range of memory usages for a task, including a maximum (all memory reachable from the task) and a minimum (memory reachable only from the task).

As indicated by Figure 4.3, the collector will find all the memory that the first processed task shares with others, and none of the memory shared by the last one processed. This asymmetry presents a problem: since the scanning mechanism treats each task’s shared memory differently, we get an inconsistent view of the memory usage picture.

One option would be to run the garbage collector separately for each task. This would return the total amount of memory being held live by each task, similar to the “precise” policy described by Wick et al. [141]. Since we would need to start garbage collection fresh for each task, such a system would impose additional time costs, processing each shared object once for every task that can reach it. Without additional computation (and
additional overhead), this system would also not report how much of the memory used by a given task is shared with other tasks.

We instead chose to address the asymmetry problem by rotating the order that tasks are processed on subsequent collections. The effect of this can be seen by comparing Figures 4.3 and 4.4; changing the processing order changes the memory charged to each task.

The first task processed yields a maximum value—an upper bound on memory reachable by that task and includes all memory it shares with other tasks. The last task processed yields a minimum value, indicating how much memory that task is responsible for that no other task has a reference to. Results for tasks in the middle give an intermediate value somewhere between these two extremes. Rotating tasks from the back of the processing list to the front means that the minimum and maximum values computed for each will be measured one collection apart from each other. This yields an imperfect snapshot of memory usage, but barring dramatic swings in memory being held live by a task in between collections, this rotation gives a valuable approximation to how much memory each task is both using on its own and is sharing with other tasks. The synthesis of this raw information into useful policies is discussed further in Section 4.3.

Unaccountable references

One concern of garbage collector-based memory accounting has been described as the "resource Trojan horse" problem [69]. In this case, task B might accept a reference to an object provided by task A. This object might in turn contain a reference to a very large block of memory. Task B will then be held responsible for that large block, even if it is unaware of the block's existence. Depending on the system's memory management policy, this could represent a denial of service attack on task B. Task B may want to accept a reference to an object controlled by an untrusted task without exposing itself to such an attack. Similarly, a system library providing access to a database may (generously) not want the client task to be charged for storage within the database. Finally, it might be the case that all tasks have pointers to and from a centralized manager system, and so there is a path in memory from each task to the memory of every other task. Our system as described so far would
naively follow these references and describe the whole system as one region being fully shared among all the tasks. This is clearly not the most insightful view of the picture; we want a way to support all these styles of references, yet still be able to separate tasks from one another for measurement of their memory usage.

We solve these issues by introducing *unaccountable references*. Analogous to a weak reference (which refers to some data without holding it live), this type of reference refers to data, holds it live, but prevents the referrer from having to pay for what’s on the other side. In our system, when the garbage collector encounters an unaccountable reference, it stops without proceeding to the object being referred to. After all tasks are processed, it starts again with all the unaccountable references in the system as roots. If the only path a task has to some memory is through an unaccountable reference, the memory will be guaranteed to be held live, but that task will not be charged for that memory.

A task must not be able to use unaccountable references to circumvent the memory accounting system. One solution would be to use language-level access control (e.g., stack inspection) to restrict the creation of unaccountable references to privileged code.

Our preferred approach is to permit any task to create an unaccountable reference, but to tag that reference with its creator’s name. When these references are processed, the accounting system charges the memory found to the reference’s creator. This technique, implemented as a small adjustment to the accounting system, nonetheless provides powerful semantics for memory sharing; a task can provide references to some service it’s providing, and make it explicit in the interface that clients will not be billed for the memory found on the other side of the reference.

**Generational GC**

Generational garbage collection presents some challenges to our system. In a generational system, not all objects are traced every time the GC system is invoked. Instead, objects are allocated into a “nursery” heap, and are tenured by frequent minor collections into a mature space, which is collected using some other algorithm when it fills. Memory in the nursery is transient: upon each collection, it is either tenured or reclaimed. Thus, we’re primarily interested in accounting for the memory that makes it to the mature space.
We can track a task's mature heap memory usage in two ways. When the mature space fills, we do a major collection and count mature heap memory that remains alive using the techniques described above in Section 4.1.3. When the nursery fills, a minor collection is performed. As objects are tenured, the size of each object is added to the total memory used by the task. At each major collection, this additive component is reset to zero. Thus, every tenured object will have been charged to one of the tasks that held it live while it was in the nursery. On subsequent major collections, the tasks holding the object live will share the cost of the object in the same fashion as they do for non-generational semispace collectors.

**Other memory management techniques**

We have implemented our system in a standard semispace collector and a generational collector, but we anticipate that it can be made to work with most precise, tracing collectors. In particular, we expect that it would map well to mark-and-sweep collectors, as this class of garbage collector also traces through memory finding live objects from a defined set of roots.

Our approach would not work if the memory management system used reference counting. Such a system does not do graph traversals over the space of objects in the heap, and so it would not discover the pattern of objects being held live by various roots, nor could it make any meaningful inferences about memory sharing.

A conservative garbage collector [18] would raise a number of difficult issues: tasks might be charged for memory discovered when the collector follows something that is not actually a reference. Unaccountable references would likely cause a significant performance hit, as each reference followed, unable to explicitly describe itself as an unaccountable reference, would have to be checked against a table of such references.

### 4.2 Implementation and Results

We implemented our design in Java using IBM's Jikes Research Virtual Machine (RVM) [2] version 2.1.0. We found the RVM to be extremely useful for our work: it is implemented in
Figure 4.5: Runtime overhead incurred by the accounting modifications on Boehm's artificial GC benchmark and on various real-world application benchmarks with the RVM "semispaces" and "copyGen" collectors.

Java, is largely self-hosting (e.g., the garbage collectors are, themselves, written in Java), and provides several different garbage collectors to choose from. We implemented our system as a set of changes to the RVM's simple copying collector (called "semispaces") and its two-generational collector ("copyGen"). GCTk\(^1\) is a flexible garbage collection toolkit for the RVM, but we chose to work with the default GC system that ships with the RVM as it satisfied our requirements.

The set of changes that we made to the RVM codebase is small; our changes can be expressed as a thousand-line patch against the original 64,000-line RVM codebase. Our modified RVM exposes additional functionality to allow the system to label which classes and threads are associated with which tasks, and to query the resource usage statistics of any given task. The resulting RVM is fully backwards-compatible with the original.

For the purposes of our prototype implementation, we defined a task to be a set of classes loaded by a particular ClassLoader instance, plus any threads that loaded those

\(^1\)See http://www.cs.umass.edu/~gctk/.
Figure 4.6: Runtime overhead of memory accounting on the multitasking microbenchmark with the RVM "semispaces" and "copyGen" collectors, varying the number of active tasks.

classes, plus those threads’ children. The root set of each task processed by the garbage collector consists of the static fields of all of its classes and the stacks of all of its threads.

We benchmarked our implementation on a 1 GHz AMD Athlon with 512MB of memory running version 2.4.18 of the Linux kernel. Different benchmarks allocate different amounts of memory, so we chose heap size appropriately in order to guarantee that the tasks would execute without allocation errors but still exercise the garbage collector sufficiently as to measure our modified system's performance.

4.2.1 Boehm microbenchmark

We wanted to ensure that our modifications to the garbage collector did not adversely impact its performance, so we benchmarked our implementation using Hans Boehm's artificial garbage collection benchmark\(^2\), which repeatedly builds up and throws away binary

<table>
<thead>
<tr>
<th>Garbage Collector</th>
<th>Load Time (sec)</th>
<th>GC Time (sec)</th>
<th>Exec Time (sec)</th>
<th>Total Time (sec)</th>
<th>Major Collects</th>
<th>Minor Collects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semispace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>0.50 ± 0.55</td>
<td>9.81 ± 4.25</td>
<td>5.50 ± 0.12</td>
<td>15.81 ± 4.81</td>
<td>7.58 ± 0.76</td>
<td>–</td>
</tr>
<tr>
<td>Modified</td>
<td>0.47 ± 0.46</td>
<td>9.92 ± 4.29</td>
<td>5.60 ± 0.12</td>
<td>15.99 ± 4.78</td>
<td>7.60 ± 0.76</td>
<td>–</td>
</tr>
<tr>
<td>Overhead</td>
<td>−6.14%</td>
<td>1.12%</td>
<td>1.72%</td>
<td>1.24%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>CopyGen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>0.57 ± 0.56</td>
<td>1.82 ± 1.08</td>
<td>7.19 ± 0.11</td>
<td>9.58 ± 1.43</td>
<td>0.03 ± 0.18†</td>
<td>13.63 ± 0</td>
</tr>
<tr>
<td>Modified</td>
<td>0.63 ± 0.76</td>
<td>1.83 ± 1.08</td>
<td>7.21 ± 0.24</td>
<td>9.66 ± 1.61</td>
<td>0.03 ± 0.18†</td>
<td>13.63 ± 0</td>
</tr>
<tr>
<td>Overhead</td>
<td>11.19%</td>
<td>0.54%</td>
<td>0.20%</td>
<td>0.83%</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.1: Mean run-time and standard deviation for the multitasking microbenchmark, across 58 runs of the benchmark, varying the number of concurrent tasks. The benchmark is run against two garbage collectors ("semispace" – a two-space copying collector, and "copyGen" – a generational collector) in two configurations ("original" – the unmodified RVM garbage collector and "modified" – adding our GC memory accounting patches). "Load time" includes the class loading and accounting system setup. "GC time" includes time spent in the GC itself, "Exec time" is the CPU time spent directly by the benchmark, and "Total time" is the sum of these components. "Major" and "Minor" are the average number of times the garbage collector was invoked during the benchmark runs. †Of the 58 benchmark runs, there was no major collection in 56 of the runs, and exactly one major collection in the remaining two.

trees of various sizes. Figure 4.5 shows the overhead of memory accounting for the two garbage collectors we used on this benchmark. The results indicate that the modified GC incurs a small percentage of overhead as a cost of doing its accounting.

4.2.2 Application benchmarks

We also benchmarked some real-world Java applications to get a sense of the overhead for programs not specifically designed to stress-test the memory subsystem. One limitation we suffered was that AWT is not yet implemented with the RVM, so we were somewhat limited in our choice of programs. We benchmarked the applications JavaCup,³ a LALR

parser-generator, Linpack, an implementation of a matrix multiply, and OTP, an S/Key-style one-time-password generator.

Figure 4.5 shows the overhead of our memory accounting system for these three applications for the two garbage collection systems. As with the Boehm microbenchmark, the slowdown is negligible. In the case of Linpack with the semispace collector, we actually saw a minuscule speedup. Linpack puts very little pressure on the GC system, allocating large arrays once, then processing with no further allocation. It's thus unsurprising that our changes to the GC system have minimal impact. A small speedup could result from fortuitous rearrangements of how code or data collides in the processor's caches, TLB, and so forth.

### 4.2.3 Multitasking microbenchmark

Since our system is designed to handle several tasks running concurrently, sharing memory amongst themselves, traditional single-tasking benchmarks are insufficient to exercise our system as it's intended to run. While we could have used a number of benchmarks from the database community, such as OO7 [22], these benchmarks are not primarily designed to place pressure on the garbage collector. To address this, we decided to write our own synthetic benchmark.

Our benchmark draws its inspiration from Boehm's, in that it also deals with binary trees, but in order to ensure a good degree of memory sharing, we used applicative binary trees. An applicative tree is a functional data structure, immutable once it is created. To perform an insert on an applicative tree, the node that would normally have been mutated is instead replaced with a newly allocated node, as are all of its ancestors. Each insertion thus allocates $O(\log n)$ new nodes. Throwing away the reference to the old tree likewise makes $O(\log n)$ old nodes dead, and thus eligible for garbage collection.

In our benchmark, a random applicative binary tree of tens of thousands of nodes is generated, and a reference to this tree is passed to each task. Each task then repeats two phases: adding new elements to its view of the tree, and randomly trading its view of

---

4[http://netlib2.cs.utk.edu/benchmark/linpackjava/].

5[http://www.cs.umd.edu/~harry/jotp/].
the tree with another task's view of the tree. Each task performs a number of insertions inversely proportional to the number of tasks present in the benchmark, such that the total number of insertions performed over the benchmark run remains constant regardless of the number of tasks participating in the test. The total running time of the benchmark thus stays relatively flat as we vary the number of concurrent tasks.

We ran this benchmark using a 64MB heap size; for the generational collector, we employed a 16MB nursery heap. We measured three time values: the setup time (the amount of time required to load all of the classes and assign them to their tasks), the time spent in the garbage collector, and the remaining runtime. Our results are presented in Figure 4.6 and Table 4.1. The graph in Figure 4.6 shows that performance overhead varies noticeably as we increase the number of tasks, but is generally quite small. In some cases, the modified system outperformed the original system, although in other cases, the original system outperformed the modified system. Such variations most likely result from fortuitous re-arrangements of how code or data collides in the processor's caches, TLB, and so forth. Table 4.1 shows averages over all the benchmark runs. On average, we observe that our system adds a negligible overhead to the benchmark's total running time (around 1%).

Another interesting observation is how often the garbage collector actually runs. The semispaces collector runs roughly once every two seconds. With the generational collector, however, major collections tend not to occur very often, if at all, most likely because our benchmark is keeping relatively little data live over long periods of time, forcing any memory accounting policy to rely on data collected during the minor collectors for its policy decisions. Section 4.3 discusses this in more detail.

### 4.3 Policy Semantics

The system described so far provides primitives for measuring memory being held live by various tasks in the runtime. These measurements, by themselves, do nothing; a policy engine that queries them is needed to place resource usage restrictions on tasks that are misbehaving. It's important that these policies be written with an awareness of what has actually been measured by the annotated garbage collector.
As discussed in Section 4.1.3, the usage values for a task measure two different statistics: a high-water mark, indicating how much memory that task is using, including memory it shares with other tasks, and a low-water mark that accounts for the memory used by that task alone. An intelligent policy would consider both of these values. It should be noted that these values may not reflect the most current state of how tasks are consuming memory. Tasks that have a large variation in the amount of memory allocated over time will be measured less accurately as a result. Policies might also look at intermediate measurements made when a task is neither the first nor the last to be processed by the garbage collector. These measurements may still be useful for adjusting the lower and upper-bounds on a task's memory usage.

We observe that in the presence of greater memory pressure (when it is likely to be more important to enforce limits on memory usage), the frequency of garbage collection will go up; accordingly, the frequency with which memory usage is measured also increases. This implies that the accuracy and timeliness of memory accounting will improve when it is most needed. Other factors, such as a choosing a smaller heap size, will also result in higher-resolution accounting data, although at the cost of some runtime efficiency.

When using a generational collector, the same upper and lower bounds exist. Exact measurement of memory usage is only available after a major collection. However, a generational collector is explicitly designed to minimize such collections of the mature space. For an example of the scarcity of major collections, see Table 4.1. However, information is available from minor collections that we can use to refine the measurements from the last major collection. The amount of memory tenured on behalf of each task during minor collections can be added to the upper bound of that task's memory usage. A similar approach is taken by MVM [34]. On the next major collection, this incremental adjustment becomes obsolete, and is discarded. As a result, the high and low boundaries measured during major collections have the same properties as the measurements made in non-generational semispace collectors.

Regardless, the quality of these measurements is not as good as the measurements available in a semispace collector. If a task is consuming an excessive amount of memory, it may take a long time to be discovered using this methodology alone. Instead, these measure-
ments should only be used as a way of indicating likely culprits. If the system is running low on available memory, it's unclear which task or tasks is responsible, although the larger ones observed thus far would seem likely candidates for further analysis. It's possible to explicitly force a major collection at any time, so it would be sensible in low-memory conditions to perform extra garbage collections to arrive at a culprit. Such extra analysis would make sense to avoid falsely terminating the wrong task.

This leads to a general observation: we can always trade additional CPU overhead for additional accounting accuracy. As an example, we could run a copying collector twice in succession in order to get a precise picture of some task's low and high usage statistics. The measurements that are provided by default are generated in the course of the garbage collector's normal business, but asking the garbage collector to run more often in order to improve the quality of data gathered might be a reasonable choice for some policy engines.

Other policies can be feasibly implemented beyond those based on the amount of memory being held live by each task. For example, a policy may wish to charge tasks for the time spent copying memory on their behalf by the garbage collector. With a trivial change to our annotated garbage collectors, we could push a previously hidden cost (time spent in the garbage collector) back onto the tasks that incurred it, perhaps suitably modifying the tasks' thread priorities.

Finally, we observe that accurate accounting of memory usage is predicated on there being discernible boundaries between tasks. If all tasks have references to and from some sort of central switchboard, such that every task has a path to all the memory of every other task, then the measured numbers will be out of synch with reality. Unaccountable references should be used in such systems in order to provide segmentation. For pre-existing software applications not yet adapted to use memory accounting, this may represent a non-trivial engineering effort.

4.4 Summary

Although Java and other general-purpose language-based systems have good support for memory protection, authorization, and access controls among mutually distrustful parties,
there is little or no support for monitoring or controlling the resource usage of the individual parties. Such mechanisms allow boundaries to be established on memory usage, preventing denial of service attacks and generally increasing a system’s reliability. Existing mechanisms either limit communications and memory sharing among tasks, can be fooled into charging the wrong task for memory usage, or don’t gracefully support handing objects off from one task to another.

Knowing which task allocated an object is not as important as knowing which task is holding a live reference to that object. Our memory accounting system allows tasks to be charged for any memory they reference. The system is integrated into the garbage collector, piggy-backing on the periodic memory scans normally performed to maintain the memory heap. As a result, our system derives an accurate measure of memory usage while having almost insignificant performance overhead (typically less than 3% on a variety of benchmarks). Additionally, the accuracy of our measurements improves when there is increased memory pressure on the system, again at no additional performance cost. These measurements, combined with the task termination system discussed in Chapter 3, allow for the graceful implementation of a variety of memory usage policies in language-based systems.
Chapter 5

Transactional Rollback

In the last two chapters, we developed language-based mechanisms for terminating tasks and accounting for task memory usage. Both of these systems assume unfettered sharing among tasks. However, this shared state can still become corrupted when a task is asynchronously terminated; restarting the task can have unpredictable results. Similarly, shared state can lead to race conditions among tasks.

In this chapter, we introduce transactional rollback, a solution to the problem of restarting terminated tasks. Section 5.1 discusses our design goals and presents and justifies our design. In Section 5.2, we discuss a Java-based implementation of our transactional rollback design, and address a number of implementation-specific issues. Section 5.3 presents performance results. We summarize transactional rollback in Section 5.4.

5.1 System Design

The ability to terminate tasks, assumed to be executing transiently in an otherwise long-running system, is a necessary property of this long-running system. We have addressed this problem with soft termination, discussed in the previous chapter. However, as a result of a task's untimely termination, shared state might become inconsistent. To address this, we need a mechanism to return the system to a known-consistent state.

A number of possible designs exist for rolling back state in a language run-time system. We begin by explaining how transactions are an appropriate mechanism for rollback. We then explore the range of possible designs, and explain how we chose to approach the problem. We also address issues in the design of rollback.
5.1.1 Rollback and Transactions

The most straightforward solution for designing language-based rollback is to simply keep a record of changes made by a task, and roll the changes back if the task is terminated. However, due to the concurrent nature of multi-threaded language systems, where multiple tasks may read and write to shared data in parallel, termination could still result in inconsistent state. Two general types of data conflicts complicate our design, read-after-write conflicts and write-after-write conflicts.

Write-after-write conflicts occur when two tasks write to the same variable. If one of the tasks is terminated, it becomes unclear how to roll back that transaction’s writes to the variable. Read-after-write conflicts occur when one task reads a variable another task has previously written to. If the writing task is terminated, the value read by the reading task becomes invalid, and the reading task must now be terminated, or at least restarted, as well.

Write-after-write and read-after-write conflicts are well known in the domain of databases. They are generally addressed by ensuring that all state is operated on strictly by ACID (atomic, consistent, isolated, durable) transactions. In particular, we can prevent these conflicts by constraining the order of operations within transactions to prevent the offending cases. The resulting order is said to have the properties of serializability and strictness. Conveniently, a well known locking protocol, strict two-phase locking, guarantees these properties. Any textbook on databases, such as Silberschatz et al. [126], provides a more in-depth coverage of this material.

For our system, then, each task runs within a transaction, utilizing the system memory as a database. In the rest of this section, we show how this can be integrated into a language system. We also discuss some issues that arise in adding database functionality to a language run-time system for supporting transactional rollback.

5.1.2 Architecture of Transactional Rollback

Transactional rollback has a number of similarities in design to language persistence. Persistence is the notion that the state of a program is maintained even in cases as extreme as the computer rebooting. To accomplish this, the system must keep track of which data
a program accesses. At prescribed points in the code, the system must save the state to disk, making it persistent. It must also deal with system failures before the data has been written to disk. Similarly, transactional rollback must keep track of which state a task has modified. If the task is terminated, the changes the task has made must be rolled back to stored, stable values. The system might track this meta-state at many different levels.

Likewise, there are numerous parallels between transactional rollback and language security. In language security, the basic goal is to protect system-level invariants, such as task separation and resource limitations. Transactional rollback is concerned with protecting unspecified invariants at the user level. Like language persistence and language security, we can design transactional rollback to run below the language run-time system, inside the run-time system, or above the run-time system.

**Below the Run-Time System**

At the lowest level, transactional rollback can be designed to operate below the language run-time system, generally as a service of the operating system. Since operating system mechanisms are simple and well-understood, greater assurance can be provided by adding transactional behavior at this level. However, the operating system only sees pages and words in memory, and does not understand the semantics of the data it is operating on. As a result it cannot take advantage of these semantics to prevent unnecessary contention for system resources.

Numerous persistent object systems run at the operating system level. The operating system can use the page access patterns of running tasks to determine which pages of memory need to be made persistent. This is the approach taken by such persistent systems as Grasshopper [41], KeyKOS [60], and others. Similar approaches are taken by software distributed shared memory systems [107] to propagate changes. Finally, Howell [74] describes an implementation of Java persistence which operates above the operating system but below the language run-time system. A more complete discussion of operating system support for persistence and transactional systems is provided by Dearle and Hulse [40].

The operating system can also be used to enforce language security invariants. For language security, the operating system already has built-in protections for cross-domain ac-
cess of state. However, the operating system uses processes to separate protection domains, so individual tasks have to be run in separate JVM processes. At a more extreme level, one can take advantage of the separation afforded by running the tasks on different machines entirely. Several systems use these mechanisms to provide language security [88, 127].

**Inside the Run-Time System**

We can also implement transactional rollback as a customization to the language run-time system. With the language run-time system's semantic understanding of the language's data structures, we can provide transactional rollback at the granularity of these data structures. This more precise granularity can result in fewer cases of false sharing contention (that is, cases where two tasks are accessing memory that is within the same memory page, but is actually used for separate and unrelated objects). This granularity has also been shown to reduce contention in distributed shared memory systems [75].

The language run-time system implementation itself does not necessarily suffer from the performance or design constraints imposed on tasks running above the language run-time system. However, this approach suffers from a lack of portability; to provide transactional rollback in different implementations of the same programming language, the design must be re-implemented for each language run-time system.

Language persistence is commonly integrated into the language run-time system. This approach is the earliest in persistent systems. Persistent systems grew up around such early persistent programming languages as PS-algol [5] and Elle [1], as well as the later Napier88 [39], in which persistence support existed as a part of the language run-time system. The earliest persistent object system was POMS [29]; more recent examples include Thor [86] and Mneme [95]. Java has been a specific target of modifications to support persistence, with such systems as PJama [106].

Finally, the language run-time system can provide enforcement of language security. The language run-time system understands the data structures a program is using, and can use this information to provide more precise protection. In fact, many systems enforcing language security exist as part of the language run-time systems because these aspects of language security are actually integrated into the design of the language. A number of
Java resource management systems also rely on customizations to the JVM [6, 7, 10, 133]. Some, such as PERC [99], even go so far as to modify the language, in the case of PERC adding primitives to ensure atomic execution of blocks of code and imposing run-time restrictions on these blocks. Vino [122] directly tracks the resources used by its tasks, allowing for a limited form of transactional rollback.

**Above the Run-Time System**

Designing transactional rollback to run on top of the language run-time system solves the issue of portability. Since the transactional rollback system is designed in terms of the language itself, any implementation of the language run-time system can use the transactional rollback system unmodified. What's more, code-to-code transformations are well-understood in language theory, and a high-level design based on code-to-code transformations could be more easily adapted to work with many different programming languages. However, these code-to-code transformation systems suffer performance and flexibility penalties because they are unable to modify all aspects of the underlying language run-time system. In designing soft termination, discussed in Chapter 3, facing these same tradeoffs, we chose to implement our system as a code-to-code transformation, and the performance penalties were still quite reasonable (worst case benchmarks, doing numerical processing, experienced an 18 to 25% overhead, where other benchmarks experienced only a 3 to 6% overhead).

This approach has been used in designing language persistence. In such systems, the task is transformed at run-time to provide the persistent system, also running in the language run-time system, with access information for that task. Marquez et al. [89] describe a persistent system implemented in Java entirely using bytecode transformations at class load time.

Some aspects of language security can also be enforced on top of the language run-time system. In these systems, the task is transformed as it is loaded by the run-time system to make run-time checks enforcing language invariants. These run-time checks likewise run entirely on top of the language run-time system. In addition to soft termination, such checks
have also been used for access control [47, 102, 137] and resource management [23, 35, 68, 81].

5.1.3 Design Discussion

Because we want a portable system, we have chosen to implement transactional rollback above the language run-time system using code-to-code transformations, as described in section 5.1.2 above. We begin by describing the transactional rollback transformation. We also describe some complications inherent in designing transactional rollback.

The Transactional Rollback Transformation

The code-to-code transformation for transactional rollback is relatively straightforward. First, each subroutine of the task is duplicated, and a parameter is added to the duplicate subroutines. This parameter represents the current transaction, and is used to identify the current task. In addition, all subroutine applications from duplicate subroutines are rewritten to instead call the duplicate-equivalent subroutines, passing the current transaction parameter along. This effectively duplicates the call graph of the entire system, forming separate transactional and non-transactional contexts for each task.

Each object instance is mapped one-to-one to a lock. The duplicate subroutines are rewritten such that before a task can access a data structure, the transaction which corresponds to the task must first acquire the lock corresponding to the data structure. As mentioned in Section 5.1.1 above, lock acquisition follows the strict two-phase locking protocol to ensure the system's consistency. If a deadlock is detected, the task is terminated, and its modifications rolled back. The original subroutines remain unmodified, allowing for minimal overhead in the event that a non-transactional context is desired; the cost of this flexibility is a 2× overhead in the code size of the program.

When a write lock is granted, the corresponding data structure must additionally be backed up. A reference is thus maintained from every data structure to its shadow backup. If the current transaction is aborted, any modifications will be rolled back, which depending on implementation may additionally require the maintenance of references from the
Figure 5.1: Before a transaction has begun operating on the object, its backup is empty and it points to an empty lock. The original object consisted only of the fields $c_1$, $c_2$, and $c_3$. The backup and lock fields were added in transforming the object for transactional rollback.

Figure 5.2: Transaction 1 has read from the object. It has a read lock on the object, but the object has not been written to, so no backup has been created. Grayed-out fields are those which are not modified in the operation.

Figure 5.3: Transaction 1 has written to the object. It has a write lock on the object, and the backup has been allocated and filled with the old values of each field. Grayed-out fields are those which are not modified in the operation.
backups to the original data structure. When the write lock is granted, a shallow copy of
the data structure is made into the backup. Deep copies are unnecessary, as any data struc-
tures referred to by the current data structure are themselves backed up as needed. Note
that a data structure is only backed up once per transaction, the first time the write lock is
acquired. Figures 5.1 through 5.3 illustrate a transaction reading from and writing to an
object.

If a transaction commits, any backups being maintained for that transaction’s write
locks are thrown away. If the transaction aborts, its backups are restored. Aborts occur
whenever the corresponding task abnormally terminates, either because of a bug, in re-
sponse to a termination request by the system, or to break a deadlock. Locks are released
when the transaction finishes, either by committing or aborting.

Deadlocks

Introducing transactions to a language, as in transactional rollback, can cause a previously
deadlock-free program to deadlock. These deadlocks must be dealt with, either by prevent-
ing them beforehand or by detecting them after the fact. Deadlock prevention algorithms
involve acquiring locks before they are needed and in a well-defined order. Unfortunately,
tasks are not written to use any particular lock access discipline (and are not, in general,
designed around the thought that they might be running in a transactional environment),
and attempting to statically add such a discipline to a task would be infeasible without
grossly overestimating the lock usage of a task. This makes it unreasonable to attempt an
implementation of deadlock prevention.

Deadlock detection simply involves maintaining a directed graph of lock dependen-
cies. If this graph ever develops a cycle, there is a deadlock. This can be determined by
performing a simple graph traversal when a lock is requested. If a cycle is detected, the
newest transaction in the cycle is terminated. This guarantees that at least one transac-
tion in the system (the oldest) will never be terminated due to deadlock, and therefore that
deadlock will never prevent the system from making forward progress. These concepts are
well-known in the field of databases [126].
Note that locks available to a language are not directly used in transactional rollback, although such language-based locks, if available, may be used short-term within critical sections of the lock manager. As a result, our deadlock-detection scheme does not address deadlocks which are preexisting in the task. This also introduces the possibility that tasks may deadlock on a mix of normal language-based locks and transactional rollback locks.

If the lock-state of language-based locks is accessible to our deadlock-detection algorithm, we can incorporate this information to detect such "mixed" deadlocks. Otherwise, we could implement a timeout for transactional rollback locks. When some application-specific heuristic function determines that no forward progress is occurring (for instance, by detecting that no lock activity has occurred for one minute), we can begin aborting transactions until the system resumes making forward progress.

5.2 Java Implementation

We chose to implement transactional rollback for the Java programming language, giving us experience that directly applies to a popular language and which can be easily applied to other object-oriented languages. We also favored Java for the presence of preexisting tools to help implement and debug the code-to-code transformations.

In this section, we discuss how to transform Java code to support transactional rollback. We then discuss a number of implementation issues and how we addressed them. Some are Java-specific, and would not pose a problem for other language systems, while others are universal.

Our implementation of transactional rollback utilizes Java bytecode rewriting; we use IBM's JikesBT\(^1\) bytecode toolkit. Examples of how Java source code is modified to support transactional rollback can be found in Figures 5.4 through 5.7. Note that while the transformer actually operates on Java bytecode, there is a straightforward mapping from the targeted bytecodes to the corresponding Java source. We present Java source code for the purpose of clarity.

\(^1\)http://www.alphaworks.ibm.com/tech/jikesbt
void foo(...) {
    ...
    bar(...);
    ...
}

void foo(...) {
    ...
    bar(...);
    ...
}

void foo.transact(..., Transaction t) {
    ...
    bar.transact(..., t);
    ...
}

Figure 5.4: Transactional rollback transformation applied to method definitions and method calls in Java. \( t \) represents a \texttt{Transaction} object which is passed from method to method along the call stack.

5.2.1 Locking Code Insertion

The systems we have in mind in our design of transactional rollback include those which run potentially untrusted code. As a result, we make no assumptions about the code as it is input into the system, and the system itself must ensure that the code has been transformed to support transactional rollback. We can ensure this by performing the transactional rollback transformations in the class loader, as the class is being loaded into the JVM.

A compiled Java program is represented as a set of \texttt{.class} files, each of which contains the bytecode for a single Java class. These classes are loaded, typically on-demand, by a \texttt{class loader}. Class loaders are built into most JVMs and can be extended to support other functionality, such as rewriting code as it is loaded. By performing code transformations in the class loader, the point through which all code is loaded into the system, we can guarantee that all code in the system will be consistently transformed.

The transformer implements the design described in Section 5.1.3 for transactional rollback. All methods are duplicated, and a \texttt{Transaction} parameter is added to the end of the parameter list of the duplicate methods. All method calls within these duplicate methods are then rewritten to call the duplicates of the original targets, passing along the \texttt{Transaction} argument from the caller to the callee. Finally, all state accesses in these
void bar(...) {
    ...  
o1.i1 = o2.i2;
    ...
}

void bar(...) {
    ...
    o1.i1 = o2.i2;
    ...
}

void bar.transact(..., Transaction t) {
    ...
    o2.readLock(t);
    o1.writeLock(t);
    o1.i1 = o2.i2;
    ...
}

Figure 5.5: Transactional rollback transformation applied to object field accesses (read and write). o1 and o2 represent arbitrary Java objects with fields i1 and i2, respectively.

duplicate methods are preceded by calls to the lock manager. These transformations are illustrated in Figures 5.4 and 5.5.

A number of instance fields need to be added to every class. First, every instance must have a reference to a lock object. Since we employ lock state sharing (see Section 5.2.2), any state specific to the object (such as other transactions waiting for the object to become free) need also be stored in the object. Finally, we need a way to store backup data for when a task must be rolled back. To address this, two new Java classes are created. The instance backup class stores backups for instances variables; it is instantiated on demand by the backup routine. The static backup class stores backups for static class variables; it is instantiated when the class is initialized. This transformation is illustrated in Figure 5.6.

The backup operation, when called on an object, saves the fields of that object into the respective backup object, instantiating the backup object if necessary. The fields are copied using Java’s assignment operator. If the field is a primitive, then rollback will restore the appropriate value. If the field is an object, then any modifications of that object require write locks on that object. If the backup operation is working on an array, as opposed to an object, similar backups are made, but some special handling is necessary. See Section 5.2.3 for details.
public class Foo {
    Object o;
    static float f;
}

public class Foo implements TransObject {
    Object o;
    static float f;
    Foo$Backup foo.backup;
    static Foo$StaticBackup foo.static;
    Lock lock;

    public void readlock(Transaction t) {
        ...
    }

    public void writelock(Transaction t) {
        ...
        foo.backup = new Backup(o);
        ...
    }

    public void commit(Transaction t) {
        ...
        foo.backup = null;
        ...
    }

    public void abort(Transaction t) {
        ...
        o = foo.backup.o;
        ...
    }

    public class Backup {
        Object o;

        Backup(Object _o) {
            o = _o;
        }
    }

    public class StaticBackup
        implements TransObject {
        float f;
        Lock lock;

        Backup(float _f) {
            f = _f;
        }
    }
}

Figure 5.6: Transactional rollback transformation applied to an arbitrary class, causing the creation of static and dynamic backup classes. We have also shown the lock management functions created for managing the instance data of the class. The equivalent functions for the static data are put in the StaticBackup class; these are not shown.
The static backup class services an additional purpose. When a class has been rewritten, it is also declared to implement the `TransObject` interface. This interface allows the transactional system to operate on transaction-enabled objects in a uniform manner. As static classes cannot be passed as parameters and cannot be operated on through an interface, we instead use the static backup class. It is created as an implementation of `TransObject`, wrapping the static class for the transaction system.

### 5.2.2 Lock State Sharing

We implement two-phase locking at the granularity of object instances to guarantee consistency. We used object instance granularity as a matter of convenience. In particular, it allows us to use Java interfaces as the primary interface between the lock manager and the units of locking. We can also store some lock state in objects, eliminating the overhead of separately allocating additional objects.

Note that the lock manager must maintain lock state for each object in the system. There will tend to be many more objects in a system than active transactions operating on these objects, and multiple objects will tend to have the same lock state. Traditionally, a distinct lock object would need to be allocated, instantiated, and maintained for each object that was created. To take advantage of this pattern, we use NLSS [38], a form of lock state sharing. In such a system, objects with the same lock state (that is, objects locked by the same transactions and in the same modes) will have references to the same lock object.

NLSS also speeds the process of unlocking objects when a transaction completes. Because multiple objects may share the same lock state, removing the completed transaction from the lock state of each locked object in turn may not be necessary. Instead, we iterate over the table of unique lock states, removing the finished transaction from each. This eliminates the need to maintain mappings from each transaction to the objects that the transaction has locked for reading. We still must maintain a mapping from each transaction to the objects it has write-locked, for the purpose of rollback. We expect this set to be much smaller, however.
5.2.3 Arrays

Arrays are treated specially in Java. On the one hand, they are instances of `java.lang.Object`. They cannot be treated as primitives, because they are composite data structures. In addition, two separate objects could maintain references to the same array. As a result, it is necessary to maintain lock state for each array independently. On the other hand, it is not possible to add or change methods or fields of arrays, so they cannot be treated like other objects.

Since we must maintain the lock state for each individual array and we cannot modify the array implementation, we must maintain an external mapping from arrays to their lock state. A hash table is used to store this mapping. To lower the cost of hash table accesses, we memoize hash table queries. In the common case, when the lock state we are looking for is in the memo table, the additional work performed when locking an array versus a regular object is a method call to get the array’s `hashCode()` and a method call to retrieve the lock from the memo table. In practice, tasks which rely heavily on array manipulation experience significant slowdown in our system. Fixing this would require access to the Java implementation for arrays to directly add a reference to the array’s lock in the array object’s header.

One workaround is to replace each array with a wrapper (called a `box`) that contains the array as well as the lock state. This requires maintaining the class inheritance relationships between boxes to mimic the relationships between arrays. In addition, all array accesses, even those in non-transactional methods, need to be through boxes to maintain the consistency of state.

5.2.4 Constructors

Constructor methods also need to be dealt with specially in Java. Arbitrary work can be done in constructors, so they must be rewritten just like any other Java method. Java constructors are also responsible for initializing variables to appropriate default values. This includes the new fields which are introduced as part of the transactional rollback (e.g., the lock field, the backup fields).
We observe an opportunity for optimization here, as noted in Daynès and Czajkowski [38]. When a task running inside a transaction instantiates a new object, it is only visible to that task until either the transaction aborts, and the object becomes unreachable; or the transaction commits. Furthermore, the very act of instantiation writes to the object.

Therefore, when an object is instantiated, we initialize the lock pointer of the object to the single write owner for the transaction. The single write owner is a lock structure which identifies the transaction as having an exclusive write lock. Since lock states are shared, we only need to create the single write owner once for each transaction. This optimization eliminates the need for newly instantiated objects to be explicitly (and inevitably) locked, reducing overhead, particularly if we implement a fast-path, as discussed in Section 5.2.7 below.

5.2.5 Native Methods

Java programs can invoke native methods (methods not written in Java), which cannot be transformed. In particular, this means that we cannot pass Transaction objects to the native methods. This is problematic because the native code might then call back to Java code, which then needs some way of getting the current transaction. Otherwise, any field accesses made in the Java method are not protected by the transactional system's concurrency control.

To solve this problem, we rewrite Java classes to store the current transaction with the current thread before calling a native method. If the native method then calls back to the task, we can restore the current transaction. Our system only adds the transaction restoration logic at upcall points that we know occur in our benchmark tests, such as when threads are started.

However, native code can make upcalls to any Java method; all Java methods would need to be converted to include the transaction restoration logic, similar to the analysis performed in SAFKASI [137]. SAFKASI showed performance overheads of 15 to 30% for passing its security context argument to all methods in the system. We would expect a similar overhead here.
5.2.6 Open Files

As noted by Howell [74] and others, open files and network connections pose a problem because they can be used to store state where the transactional system cannot roll it back. They also themselves represent state (such as file offsets) that is managed natively by the language run-time system or operating system. Should the task terminate prematurely, there is no way for the transactional system to roll back any writes to files or onto the network, or otherwise restore the state of the object.

We observe, however, that multiple instances of a task will not tend to share the same open file descriptor or network socket. As a result, if a task terminates prematurely, all of its open file descriptors and network sockets, which were created while the task was running, become inaccessible. If multiple instances of a task share an open file in an environment where tasks can fail asynchronously, the system is equally at risk of inconsistent data in files with or without transactional rollback; we do not exacerbate the problem.

5.2.7 Optimizations

We implemented a number of optimizations on the code, and can foresee a number of others that may speed the code up considerably. The optimizations we did implement were inspired by the results of profiling runs of the code. This is discussed in Section 5.3.3. The simplest optimization we use is a fast path. As part of the thread locking transformation, we insert code inline to check whether the locking operation is necessary. Only if we determine that a lock needs to be acquired do we actually attempt to acquire it. This will generally save at least one method call per field access, and could potentially save many more.

As a more ambitious optimization, we observed that the most commonly accessed object in a method is this, the object upon which instance methods generally operate. In almost all cases, a method will access its this object. As a result, if we can statically determine that a method accesses the this object at least once, we can acquire a single lock for that object at the beginning of the method. If any accesses are writes, we acquire a write lock. Otherwise, we acquire a read lock. All subsequent locks of this are omitted.
We would like to extend the above to apply to all objects in a method. If we could statically determine that multiple locks are acquired for the same object, all but the first could be removed. Similarly, if a read lock request and a write lock request were made on the same object from the same method, the read lock request would be redundant.

To address this problem, we could use a variation of lazy code motion [79, 43], a well known compiler optimization. Using this technique, we could eliminate more of the redundant lock acquisitions within a method. Since this method has been shown to be optimal, any redundant lock requests that can be detected will be detected with this method. Lazy code motion also guarantees that no code paths are extended, meaning a lock request is only made if it is necessary. Hosking et al. [72] discuss similar methods for optimizing such a system. Note that with any of these methods, there still needs to be one lock acquisition per object per method. Interprocedural analysis may help further eliminate these lock operations.

5.2.8 Integration into a Real-World System

As mentioned in Section 5.1, we assume that this system is already running in a pseudo-transactional environment. That is, there is a long-running system with a number of transient tasks running concurrently in separate threads and providing some services to the system. In this case, starting a transaction coincides with starting a task thread: the method Transaction.doTransaction() is called with a java.lang Runnable object as a parameter. The run() method of this Runnable object, as rewritten by the transactional rollback transformer, is the entry point to the transaction. This mechanism is shown in Figure 5.7. When the method returns, either normally or abnormally, the transaction is completed, either having committed or aborted, respectively. If the transaction was aborted, an exception is thrown. The system hosting the tasks might choose to implement a loop which automatically restarts tasks if they are aborted.
public class Foo {
    void foo() {
        Transaction.doTransaction(
            new Runnable {
                void run() {
                    bar();
                }
            });
    }
    void bar() {
        ...
    }
}

public class Foo implements TransObject {
    void foo() {
        Transaction.doTransaction(
            new Runnable {
                void run() {
                    bar();
                }
            }
        );
        void run.transact(Transaction t) {
            bar.transact(t);
        }
    }
    void bar() {
        ...
    }
    void bar.transact(Transaction t) {
        ...
    }
}

Figure 5.7: Transactional rollback transformation applied to a code block which starts a transaction. The run() method of the Runnable object is duplicated and the transaction parameter is added; this method is the entry point for the transaction. Note the duplicate Foo.foo.transact() is not shown.
5.3 Performance

We measured the performance of our system using an AMD Athlon workstation (1.2 GHz CPU with 512 MB of RAM, running Linux version 2.4.16) and version 1.3 of Sun's Java 2 JVM, which includes the HotSpot JIT. We used two sorts of benchmarks, microbenchmarks that measure the performance of specific Java language constructs, and some real-world programs. We performed three different measurements for each benchmark. The first was the unmodified benchmark. In the second, the class files were rewritten, but were not executed in transactional contexts. This case was designed to strictly show the performance overhead of the larger class files and additional per-object data storage (with nothing stored there, but taking up more space). Finally, we ran each benchmark in a transaction to measure the total overhead of the transaction system. For the application benchmarks, we further measured the performance with an "array cheat" which we will describe below.

5.3.1 Microbenchmarks

We first ran a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, method and field accesses, exception handling, synchronization, and I/O. We used a microbenchmark package developed at University of California, San Diego, and modified at University of Arizona for the Sumatra Project\(^2\). The results are shown in Figure 5.8.

The microbenchmark results were entirely unsurprising. In the benchmark which focuses on field and array accesses, the overhead of the transactional system was significant (over 10\(\times\) overhead). All other test benchmarks had significantly lower overhead. The only other microbenchmark with an overhead of more than 10% was the loop test, at 1.5\(\times\). However, note that this reflects an actual margin of 0.008 seconds, and the overhead shows up for the rewritten classes in both transactional and non-transactional contexts.

\(^2\)The original web site is http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html. The source we used was distributed from http://www.cs.arizona.edu/sumatra/ftp/benchmarks/Benchmark.java
Figure 5.8: This graph illustrates the performance of the rewritten classes run both inside and outside of a transactional context, relative to the performance of the corresponding original class files. The graph measures how many times longer the benchmarked class took to run than the original class. The left bars show the overhead when running the rewritten classes using only the non-transactional code. The right bars show the overhead when running the rewritten classes using the transactional code and in a transactional context.

5.3.2 Application Benchmarks

We benchmarked the real-world applications JavaCup\(^3\), Linpack\(^4\), Jess\(^5\), and OTP\(^6\), to get a feel for the performance impact of our transformations on real-world code. JavaCup is a LALR parser-generator for Java. Jess is an expert system shell. Linpack is a loop-intensive floating-point benchmark. OTP is a one-time password generator which uses a cryptographic hash function. The results are shown in Figure 5.9.

The first trend we noticed was in the overhead of the transformed code not running in a transactional context. Jess and JavaCup demonstrated relatively large overheads, compared

\(^3\)http://www.cs.princeton.edu/~appel/modern/java/CUP/
\(^4\)http://netlib2.cs.utk.edu/benchmark/linpackjava/
\(^5\)http://herzberg.ca.sandia.gov/jess/
\(^6\)http://www.cs.umd.edu/~harry/jotp/
Figure 5.9: This graph illustrates the performance of the rewritten classes run both inside and outside of a transactional context, as well as inside with the array cheat enabled, relative to the performance of the corresponding original class files. The graph measures how many times longer the benchmarked class took to run than the original class. The left bars show the overhead when running the rewritten classes using only the non-transactional code. The middle bars show the overhead when running the rewritten classes using the transactional code in a transactional context. The right bars show the overhead when running the rewritten classes in a transactional context with a cheat enabled to eliminate the hash table lookup on array accesses.

The overhead to OTP and Linpack. These overheads are roughly proportional to the sizes of the respective packages. The transformation added four instance fields and one static field to each class. In addition to two extra classes that need to be loaded per original class, the original class file is itself grown to over three times its original size. This code and memory bloat can be blamed for some of the overhead.

When transactions were enabled, the overheads were impressively large — performance overhead ranged from a factor of $6 \times$ to $23 \times$. Clearly, these numbers indicate our system is not yet suitable for deployment in practice. We suspected that a major component of this overhead was related to our handling of arrays (see Section 5.2.3). To study this further, we implemented an "array cheat." While no longer semantically sound, this cheat eliminates the hash table lookups to find the per-array locks (one global lock is used).
and it no longer performs backups of arrays. All other lock operations were performed unchanged. This cheat represents an upper bound on the performance benefit that might be achieved with a hypothetical extensible array implementation, giving us a slot per array to store the locks. Our measurements show significant gains relative to the original transactional system (reducing the overhead to between a factor of $2 \times \text{ and } 15 \times$). Excluding Linpack, which is a fundamentally array-driven benchmark, the overhead experienced on the other application benchmarks with the array cheat was at most $5 \times$. The relative speedup on OTP was quite impressive (from $23 \times$ to $4 \times$). OTP allocates a large number of small arrays which it uses for temporary storage and which then quickly become garbage. Clearly, such a program introduces a large overhead when external references must be kept to each temporary array.

An interesting question is how these performance numbers compare to other systems that attempt to solve similar problems. When reading the literature on persistent object systems, we have not found many papers willing to compare their performance to the original, non-persistent system. Marquez et al. [89] present a code-to-code transformation that implements orthogonal persistence and compare its performance to JDK 1.2 with no support for persistence. They indicate their system, with a warm disk cache, has a roughly $9 \times$ performance cost relative to JDK 1.2 when running their test benchmark. This confirms that other persistent systems experience overhead similar to what we observed. However, further optimization is still necessary before this system is viable in production.

5.3.3 Optimization Profiling

In order to gain insight into the best methods for optimizing the system, we instrumented the transformed classes to provide an accounting of when locks were acquired. This includes attempts to acquire locks when the fast path determined the acquisition is not necessary. Our goal was to develop a strategy to allow the transformer to statically determine that a lock acquisition is not necessary and omit it.

In analyzing these results, an interesting pattern emerged: a large portion, and for some benchmarks, the vast majority, of redundant lock requests were on the this object. This result is because a method will access fields of its own object far more frequently than
fields of other objects. Note that it is never safe to remove all lock acquisitions for an object from a single function without performing inter-procedural analysis. However, there were still many redundant lock requests of this even without counting a required first lock acquisition.

Our solution, and the only optimization we performed for our system, was, whenever we could statically detect that a method would be accessing the this object, we acquire the appropriate lock to this at the beginning of the method and omit any lock requests for this anywhere else in the method. This reduces many redundant lock requests while adding limited deadlock pressure. The result is best illustrated in the OTP benchmark.

OTP uses the MD5 hash function to generate one-time passwords. The MD5 implementation keeps mathematical state in object fields, and performs long sequences of mathematical operations on them. Before this optimization, each mathematical operation needed to be prefixed by a lock acquisition. Afterwards, a lock acquisition was only needed at the beginning of each function. The result was a substantial drop in overhead for this benchmark. Smaller drops were observed in other benchmarks.

Even with this optimization, however, there is still considerable room for improvement. There are still a large number of subsequent lock acquisitions to the same object, read lock acquisitions followed closely by write lock acquisitions, and other redundant lock acquisitions. As discussed in section 5.2.7, data flow and control flow analysis could be used to identify redundant lock requests and reduce this overhead.

5.4 Summary

Termination is a crucial capability for providing resource control in language-based systems. In the face of data sharing, the ability to guarantee that data remains consistent on task termination is equally important. Transactional rollback provides one language-based, portable solution to this problem. Our design is independent of any particular language run-time system, allowing implementations which do not depend on a single language. Unfortunately, our Java implementation shows a worst-case overhead of 23×, with overheads of 6 to 7× in the absence of extensive array usage.
While these overheads are quite large relative to the original system's performance, they represent a starting point for semantics that are otherwise unavailable to the designer of systems that must reliably execute untrusted or buggy tasks. A number of opportunities exist to further optimize our system, especially if we allow modifications to the language run-time system. However, the resulting semantics are also fairly restrictive, limiting data access to one task at a time. In the next chapter, we explore an alternative approach to providing task isolation.
Chapter 6

Preserving Task Isolation

In the previous chapter, I discussed transactional rollback, a language-based, portable solution to maintaining the consistency of data shared among multiple tasks. Unfortunately, this system had very high overhead and restrictive semantics that would limit its general applicability. In this chapter, I present an alternative approach to separating tasks. This approach is far more flexible, allowing some data sharing and inter-task communication while providing a means to specify restrictions.

We propose to use static analysis to achieve semantics similar to operating system processes. By taking advantage of the type safety and access protections provided by language-based systems, we can achieve separation of tasks running within the language-based system. In addition, we can allow sharing among tasks, providing for separation policies far more flexible than those available with process separation. This is similar to language component systems, which seek to simplify independent component development by separating them with clear interfaces. Since this system is based on static analysis and is run before the code executes, there is no run-time overhead.

In Section 6.1, we provide a brief overview of the kinds of inter-task bugs that might occur in multi-tasking systems. Section 6.2 describes the design of our language-based solution to separation. We discuss a prototype implementation of this system for the Java programming language in Section 6.3, and describe the results of running this prototype over several real-world programs in Section 6.4.

6.1 Taxonomy of Inter-Task Bugs

Before discussing the design of our system, we present an overview of the types of bugs language-based task separation is meant to address. We consider bugs in terms of program-
mer intent. That is, a bug is any case where the actual behavior of the program diverges from the intended behavior of the program. In some cases, this intent can be derived directly from the syntax of the code (e.g., by declaring the length of an array, the programmer implicitly specifies that any reference outside that bound is an error). In other cases, especially those involving interaction among tasks, it may not be clear without more information whether a given operation was intended.

We discuss three general classes of bugs. In pointer leakage bugs, a task’s private data is shared. In time of check to time of use (TOCTOU) bugs, a property treated as invariant by a task is mutated. Finally, in thread capture bugs, a thread critical to a task’s execution exits the task and never returns.

6.1.1 Pointer Leakage

A pointer leakage bug occurs when a programmer intends for data to remain private, but accidentally leaves a way for another task to read or modify that data. This may include exposing an object directly or exposing methods which leak a pointer to that object. Similarly, it may include either direct leakage or indirect leakage (e.g., through a shared system data structure). By exposing too much of a task’s internal state, a programmer may open the door for other bugs.

The specific problem we seek to solve is related to the secure information flow problem [42, 134, 119]. These systems focus on the flow of arbitrary bits of information, while our primary focus is on objects. Once an object has been leaked, it is possible to invoke methods on that object, possibly causing additional security violations. More recently, pointer-aliasing analysis has been used to find instances of pointer leakage, as well as other security bugs [140].

6.1.2 Time of Check to Time of Use Bugs

A time of check to time of use bug (TOCTOU) can occur in any concurrent system when one task assumes that some property remains invariant over a critical section, but another task can change the property during this critical section. A well-known example of a TOC-
TOU bug that can be exploited with devastating security implications is a temporary file race condition [31].

TOCTOU bugs have been well understood with operating system processes sharing file system state, but can just as easily occur in multithreaded language run-time systems sharing object state. Whenever communications or data sharing is allowed across task boundaries, TOCTOU bugs become feasible. This is true even if the communicating tasks are restricted to communicating through a remote procedure call (RPC) mechanism; whenever one task can directly cause the execution of code within another, there is the possibility that a TOCTOU bug may exist.

However, inter-task communication is an important part of many complex systems. Furthermore, allowing the direct sharing of data might make the system easier to implement, and allowing direct communication without the need for RPCs provides a boost in efficiency and performance. Needless to say, this complicates any attempt to statically enforce separation because two domains sharing a pointer to a data structure does not necessarily signal a TOCTOU bug. What makes the concurrent access a TOCTOU bug is that the programmer does not intend for concurrent access to occur at a particular time. It is not possible to automatically detect TOCTOU bugs without some expression of the programmer's intent (e.g., the use and verification of locking primitives). However, any mechanism that can audit how sharing occurs in a system can be used to help a programmer audit a system for TOCTOU bugs.

Numerous researchers have focused on detecting race conditions in code [61, 54, 45]. Note, however, that not all TOCTOU bugs are race conditions. In fact, if a thread calls into another task between a data check and the subsequent use of that data, there could be a TOCTOU bug with only one thread running.

### 6.1.3 Denial of Service via Thread Capture

Thread capture bugs occur when a task invokes a subroutine in another task with the expectation that the call will return, and it never does. In many cases, it is not possible to determine whether an invocation crosses a boundary from one task to another; for example, if a task passes an object into another task, methods invoked on that object could result
in thread capture bugs. Such bugs result in denial-of-service to the task whose thread is captured. Shapiro [125] discusses similar threats arising from inter-task communication among operating system components.

Unfortunately, in many cases, inter-task function calls are vital to the operation of a program; dynamic web server plug-ins, for example, need access to state shared with the server. In normal operation, the web server needs to call into the plug-in to pass the web requests and needs to wait for a response from the plug-in to transmit the response back to the client. Similarly, the server may need to call some initialization routines in the plug-in. Failures in one plug-in (whether through bugs or malicious code) should not cause catastrophic failure of the server.

6.2 Design of Task Separation

A number of design decisions we made in defining our task-separation system are documented below. We discuss how we provide task separation, separately addressing isolation of data and task-critical threads of control. We then motivate these decisions and explain why we made these decisions. In describing our design, we assume an object-oriented environment with classes, methods, and fields.

6.2.1 Protecting Data with Soft Boundaries

The core idea behind our system is for the programmer to specify groups of classes that exist within a single boundary, and to specify a minimal interface that is exposed outside of the boundary. In many respects, this is comparable to prior work on module systems for ML [15, 16] as well as the MrED system for Scheme [57], although we use static analysis not only to hide internal details but to verify that there is no way for some internal state to be inadvertently exposed to external manipulation. Likewise, we wish our system to operate, like lint [77], as a separate pass from the usual compilation process, allowing the programmer to decide to ignore the warnings we produce. We refer to our design as "soft boundaries," because of the flexibility that we enable.
Specifying Data-Sharing Semantics

To declare the desired data-sharing semantics, the programmer first partitions the program into protection domains. Each domain encapsulates a group of classes operating together and trusted (or not) as a unit. We expect the programmer to place each task or plug-in into a separate domain.

The protection domain declaration includes an identifier to name the domain, the location in the local filesystem of the classes that make up the domain, and an optional application programming interface (API) restriction. The API restriction allows one domain to act as the sole gateway into another domain, similar to the ring-style permissions that can be enforced with confinement type systems [144]. The outer domain might consist of abstract classes calling into or interfaces implemented by the inner domain. By hand inspection of the outer domains, a programmer can easily verify the extent to which an API is limited, and the static analysis can ensure that no code in the system bypasses the API.

Outer domains may include code as well as abstract interfaces and can be allowed to access state of the inner domain using the standard type rules of the programming language. This added flexibility can make it easier to accommodate legacy code, but it places a higher burden on the wrapper to be implemented correctly.

There are two special domains. The first is for classes for which no domain is explicitly specified. The second is for classes provided by the language run-time system’s API, which we call system classes. Classes without a specified domain are treated as if they are not wrapped by any API. System classes are special in two ways. First, every domain implicitly includes the domain of system classes as a soft boundaries API. This encodes the policy that system classes are implicitly trusted. Second, for the purpose of API analysis, instances of system classes instantiated in a non-system domain are considered part of the declaring domain rather than part of the system domain.

Critical Data

Note that, just as programmers might make mistakes in the access declaration of a variable, they might make mistakes in the definition of the soft boundaries API. Furthermore, soft
boundaries by design offer weaker protection than hard boundaries and require sufficient familiarity with the system that mistakes may occur. For instance, a protection domain may allow access to private state through a shared object, not realizing that the object is shared with another domain. Similarly, an API method may offer direct access to private data. Any separation scheme that allows sharing must account for the possibility that programmers might mistakenly share data that they intended to keep private.

Our solution is to allow the programmer a second chance at protecting a domain’s private data by declaring some data as \textit{critical} and then use static analysis to study how that data may be reached. In particular, we verify that critical data are never accidentally mutated by publicly accessible methods, pointers to critical data are never returned by publicly accessible methods, and critical data are never directly accessed by other domains. This critical data check serves two purposes: it acts as a check on the soft boundaries API that was specified, and it allows the programmer to fine-tune the enforcement of soft boundaries.

Note that maintaining a variable’s privacy is a weaker claim than asserting that no information flow occurs [134, 97, 119]. In particular, we make no claims as to the confinement of state declared critical. It is possible that enough information might leak out to allow another domain to infer the value of the critical data. In addition, declaring an object as critical has no direct effect on objects the critical object may point to or otherwise influence. If these objects are critical, they should be explicitly declared so. Critical data declarations can be thought of as a technique for “point sampling” the effectiveness of other soft boundary declarations.

\subsection*{6.2.2 Protecting Task Threads with Soft Boundaries}

In addition to protecting a task’s state from outside interference, it may be necessary to protect critical threads being used by a task from being captured as described in Section 6.1.3. In this case, we are concerned not with the API exported by a domain, but with method invocations out of the domain that may not necessarily return to the caller, causing a denial-of-service attack on the caller.
Determining in general, statically or at run-time, whether a method invocation will return is undecidable (it is equivalent to the halting problem). One of the most common solutions to such problems is to use timeouts. If an inter-task method call has not returned after a given amount of time, the thread is interrupted and the caller sees an exception or some kind of error as a result. This approach runs the risk of interrupting useful computation while at the same time allows an actual denial of service to persist for some time before it is remedied. Comparable semantics might be achieved by converting direct method calls to use message passing. The caller’s thread will never reach the callee, avoiding any need to interrupt the callee, but introducing the significant run-time overhead of message passing.

The approach we choose is for the caller to have “sacrificial” threads with which to call across a domain boundary, and label critical threads such that static analysis can prove that only theses sacrificial threads will ever cross the boundary. While some message passing may be required between critical and sacrificial threads, this will vary from application to application. This is also consistent with the common “thread pooling” paradigm, where a program tries to reuse a fixed number of threads, and may grow the pool as the need arises.

As a matter of security, threads are assumed to be critical unless they are explicitly declared otherwise. Such declaration is made at the point where the sacrificial thread begins execution. This includes any entry points from other tasks. It also includes classes within the task where a new thread is created. Finally, it includes any points where the program begins running (for instance, a main() method).

Static analysis can then assist the programmer in guaranteeing that a critical thread never invokes a method in another domain. If a sacrificial thread is captured, Rudys et al. offer a run-time mechanism for safely terminating the capturing task [117] without disrupting the remainder of the system. One special case in this analysis is system classes. Because system classes are assumed to be safe—they are assumed to return—even non-sacrificial threads are allowed to call into them.
6.2.3 Design Motivations

We now describe some of our basic design decisions, including why we believe static analysis is preferable to run-time mechanisms, and why we believe we need an explicit high-level specification of boundaries between tasks.

Static Analysis versus Run-Time Enforcement

Static analysis and run-time enforcement are suited to different classes of problem, which make them better-suited to different sorts of problems. Static analysis can detect bugs ahead of time, and can thus improve the stability of a program without requiring any performance overhead. On the other hand, static analysis does not have as much information as may be available at run-time, and thus must be necessarily conservative. In practice, this means that static analysis tools may report false positives, through which a programmer must sort, whereas run-time mechanisms may be more accurate. On the other hand, static analysis may be able to prove that an undesirable state will never occur, whereas a run-time system may not have enough information to predict much about the future of the computation.

For example, run-time mechanisms can efficiently measure memory usage, while statically proving that a program meets specific resource bounds is undecidable for general-purpose programming languages. On the other hand, static analysis can determine the extent to which objects of any given type might propagate through a system, which can be easily verified against a static policy. Making such checks at run-time would be feasible, but could add significant overhead, possibly requiring permission checks every time an object pointer is copied. The focus of our work in the abstract is determining whether a program ever enters an unsafe state, which implies a static analysis solution.

Boundary Definitions

Our goal with soft boundaries is to help the programmer distinguish unintended sharing from intended sharing, both to find errors in a program and to improve the program's design. That is, the challenge is how to help programmers specify their intent. Anything we can learn without requiring programmer annotations will increase the scalability and usabil-
ity of our tool. However, explicit annotations can be more precise and may be necessary in some cases.

**Implicit deduction of intent.** Security analysis can sometimes deduce the programmer’s intent from unannotated code. Consider the problem of buffer overflows. A common way to trigger a buffer overflow is to cause a program to write past the last index of an array. However, the intent of the programmer, and the security policy, is implicit in the array declaration: an array declared of length $n$ is intended to have $n$ elements; a write past the $n^{th}$ location is a violation of that intent. In this case, the programmer’s intent is simple to deduce because it is directly derivable from the program. Accordingly, many programming languages already prevent this violation by limiting the program’s ability to write past the end of an array.

In those cases where intent is ambiguous Engler et al. [46] observe that patterns can be an effective guide in determining the programmer’s intent. However, patterns are less applicable when applied to task separation; if the programmer did not necessarily write their program with proper separation in mind, tasks may interact without any consistent strategy. Without an explicit declaration of the boundaries and the rules for interacting across them, an implicit system will not likely be able to make a meaningful analysis.

Programmer intent is most difficult to deduce in cases where there is no discernible difference between a line of code that is safe and one that is a security hazard. For example, using `getter` and `setter` methods is common in object-oriented programming, and public `getter` and `setter` methods appear in countless classes for manipulating private variables. However, only a program’s designer knows whether a private variable should or should not be externally readable or writable. Unfortunately, the programmer’s intent cannot be determined from the code.

We would like to prevent bugs due to unintended component interaction, even if that interaction occurs commonly in the code. As a result, automatically inferring the correct placement and semantics of soft boundaries is impractical. Soft boundaries require some simple and concise mechanism for the programmer to specify her intent.
Explicit expression of intent. If intent cannot be deduced implicitly from the code, a mechanism for a programmer to express her intent explicitly must be designed. If the mechanism is overly complicated, then the programmer is likely to misstate her intent. Types are an example of a well-studied expression of programmer intent and have been widely leveraged for security purposes (see, e.g., [49, 134, 69, 124]). For example, the restriction on casting integers to memory pointers is required to provide memory protection and type safety for a language.

Declared types provide very fine-grained semantics; each occurrence of a variable must be declared as a particular type, even if the type could be inferred from its use. This ensures that the programmer is not consistently mistaken as to the type of a variable. Employing such a fine-grained approach in establishing soft boundaries means labeling every sharable object. Such labeling could be difficult to apply to a legacy program, requiring the programmer to make extensive annotations, and may necessitate restructuring the whole program to accommodate an extended type system. Algorithms such as Hindley-Milner, based on algorithm \( \mathcal{W} \) developed by Milner [94], can be used to infer types automatically. However, we run into the same problem as we had with implicitly inferring intent, namely that misuse could be consistent.

Instead, we will investigate a coarse-grained approach. Rather than specifying a sharing policy for every possible object, we instead focus on method invocation. The programmer enumerates what methods may be used to access a task and what threads are permitted to exit the task; any other access to a protected task's data is considered a bug. This allows the programmer to make a concise specification of the interface between tasks, but still requires the programmer to ensure that the interface, as specified, is safe.

6.3 Java Implementation

We implemented soft boundaries in Java because of the range of packages available for analyzing and operating on Java programs. The tools we implemented to help enforce soft boundaries use static analysis to accomplish their job. In particular, these tools require a call graph, a class hierarchy graph, and pointer aliasing information. Rather than implementing
Figure 6.1: The format for a task declaration consists of three fields, the task name, the task's location, and the task's API, delimited by "::"s. The domain's location refers to the location of the domain's classes on the host computer's filesystem. The keyword restrict is used to associate a task with its wrapper API.

such analyses, this work leverages the control and data flow analysis in the Soot framework [62, 131, 109]. Soot is a powerful byte code optimization framework that runs over Java class files. Soot translates Java class files into one of several intermediate representations and performs analysis and optimizations over the code. Our analysis is performed over Jimple, a three-address-form intermediate representation very similar in appearance to assembly.

Soot was developed as a research tool to investigate Java optimizations and has been successful as such [128]. SPARK, the pointer alias analysis package provided with Soot, was initially designed to compare various techniques for disambiguating pointer aliases [84]. Empirically, the resulting pointer alias and call graph information is sufficiently accurate for our work; in some cases, Soot concludes that a variable could point to every object or nearly every object, but most conclusions it draws seem reasonable.

6.3.1 Specifying Boundary Semantics

In our implementation of soft boundaries, domain declarations take the form

\[
\text{domain name::filesystem location[::restrict=domain's API]}.
\]

The filesystem location specifies a class file or directory. If the location specified is a directory, all classes contained in that directory and all subdirectories are included in the domain. As an exception, if a class file would be included in two domains, it is included in the domain with the most specific specifier. Class files are used as the unit of domain construction because this is the level at which executable code is incorporated into Java, and the executable code is the entity from which domains need protection. The only exception
is Java standard API classes; instances of Java standard API classes instantiated in a non-system domain are considered part of the declaring domain. The restrict keyword is used to specify that access to the domain is restricted to another, specified protection domain, which we call the API. The only way to access a protection domain that specifies a restriction is via the specified API, either as a wrapper or by virtue of an internal class implementing a Java interface defined in the API.

The domain declaration for the Miniature Java Web Server, described in Section 6.4.1, is shown in Figure 6.1. The first three declarations place the two servlets and the main server each in its own protection domain, called CGI, FILE, and SERVER respectively; these are the tasks running in the program. The fourth domain, PROTECTION, specifies the Java Servlet API. The restrict keyword is used to specify that the fourth domain should be used as the API for each of the other three domains; any inter-task communication must pass through this API. If no interface restrictions are specified for a domain, then none are enforced.

Critical data is specified by passing an object instance to an empty method called Util.critical() (class Util is a utility class created in the soft boundaries codebase solely to house the critical() method). While this method does nothing at run-time, our static analysis tool looks for the call, and any pointer passed in is considered critical with respect to the calling protection domain. Alternative ways of specifying critical data include introducing a critical keyword to be used in declaring critical variables, or specifying critical data outside the code entirely (e.g. by specifying the line number and variable name of a critical variable). To specify control semantics, we created a dummy interface called Sacrificial; declaring that a class implements this interface is equivalent to declaring that any thread entering or being created in this class is sacrificial. Again, a number of alternate methods could have been employed.

6.3.2 Implementation of API Checker

We built the API checker to verify that a program follows the specified API (see Section 6.2.1 for a description of soft boundary APIs). The checker returns two classes of diagnostics. Warnings are returned when the API abstraction is broken. This occurs if a
Figure 6.2: A code example demonstrating how the API checker would treat different cross-domain behavior. (a) shows the domain separation of the classes. (b) shows the classes themselves. Classes Internal and External have been declared in different domains, with Internal wrapped by the API API. The comments describe the errors and warnings occurring in each line of code.
class which implements a public API also implements non-API public methods. It also occurs if an external class invokes an API method directly on the class instance. In these situations, although the API has not actually been violated, the underlying implementation is exposed to an API violation.

The second class of diagnostics, errors, are returned when an API has actually been violated. These occur when a non-API public method has been invoked from outside the protection domain or instance fields have been accessed or manipulated. In these situations, some outside class is completely circumventing the API and accessing the domain in an unintended manner.

For complete protection domain separation, both errors and warnings should be fixed. Any method intended to be invoked from outside the protection domain should be in the API, and internal methods should not be exposed to other protection domains. Figure 6.2 gives an example of errors and warnings in two interacting Java classes. The External class uses the Internal class which is wrapped by interface API.

Errors

Errors involving invocations of methods not in the API are found by examining the call graph for all calls into a protection domain. It is then a simple process to both compare those calls to the methods in the API and have Soot return a line number for each offending invocation. The labeling of any direct data accesses within a domain is accomplished in a similar way: every field access in the program is scanned, and any accesses that violate an API are flagged.

Warnings

Finding public methods not in the API involves comparing all public methods in the domain against methods declared in the API; any public methods not appearing in the API are flagged. Finally, finding methods invoked directly instead of through an interface requires checking the declared type of the receiver at the time of invocation. Again, although this does not violate the API, it does break the abstraction established by the API.
Interaction with the API

In some cases, it is desirable to treat an API as a separate protection domain for the purpose of analysis. One case where we chose to do this was for the Java Servlet API. In this case, the servlet container uses the Servlet API to communicate with the servlets, and the servlets use the API to communicate with the server; it is not appropriate to attach the Servlet API to any particular domain.

By contrast, in the case of the FreePastry system (see Section 6.4.1 for more details), the API is only used to wrap the Pastry core classes. The API also implements some code that interacts with the Pastry core. In this case, it is appropriate to treat the API and the Pastry core domain as one large domain; otherwise, the amount of inter-domain interaction would be artificially high. As a result, we provide an option for specifying whether or not a domain’s API wrapper should be treated as part of the domain.

6.3.3 Implementation of Critical Data Checker

We built the critical data checker to verify that a protection domain’s private data stays private. We wish to provide an additional line of defense, preventing access to private data even in the event that an API method allows such access. Section 6.2.1 explains why such a mechanism is useful. Our critical data analysis works by using Soot’s SPARK pointer alias analysis engine. An object is declared critical by passing the object to a Util.critical() method (again, class Util is a utility class created in the soft boundaries codebase solely to house the critical() method) shortly after instantiation. Although this method does nothing, the critical data analysis determines which object was passed to the method, and uses pointer-aliasing analysis to find other uses of the reference. Since pointer aliasing is a conservative analysis, all cases of critical data misuse are found, although there may be some false-positives.

Critical data analysis checks for two different classes of usage of the object. First, it checks that the critical object is never dereferenced outside the protection domain that owns the object. This includes checking that critical data is never passed out of a protection domain via a method parameter or a method return. Similarly, it includes checking that
the object’s fields are never modified from outside the object’s protection domain. These guarantee that code in another protection domain never gets direct access to a critical data reference. Critical data analysis also checks which methods entering a protection domain ever lead to a modification of a critical object. This prevents another protection domain from using API methods to modify protected data without the programmer’s awareness.

6.3.4 Implementation of Thread Tracker

Finally, we built a thread tracker to check which threads cross a boundary into another protection domain. The thread tracker must find all threads that directly or indirectly pass control flow across a boundary. Thread tracking is motivated in Section 6.2.2. Identifying what threads may cross a boundary is accomplished by first building the set of methods that pass control flow across a boundary, either directly or indirectly.

To construct this set, the thread tracker first determines, from a simple call graph examination, which method calls directly call into another protection domain. Then, the set of all methods calling into these methods is produced, again by examination of the call graph. This process iterates until it reaches either a method which starts a thread (in Java, this includes method `run()` in implementations of the `java.lang.Runnable` interface as
well as method run() in subclasses of java.lang.Thread and methods which enter
the protection domain. These collectively are the methods where a thread begins execution
within a protection domain. Figure 6.3 gives a graphic illustration of the construction of
this set.

The thread tracker examines this list of thread starting points, checking which threads
started at these points are sacrificial (i.e. which are permitted to exit the protection domain).
Any non-sacrificial thread that could eventually exit the protection domain constitutes a
violation of the soft boundary for that protection domain, and is flagged. At this point, the
programmer can either fix the problem (perhaps by creating a new child thread to make the
cross-boundary call) or declare the thread as sacrificial (if it is).

There are two exceptions to the analysis. First, as noted in Section 6.3.2, there are some
cases where one task is divided into multiple protection domains (for instance, if only a
subset of the task should be considered an API into the rest of the task). In these cases,
threads are permitted to freely cross from one protection domain to the other, since the two
halves trust one another. The other exception is system classes: all system API classes are
considered implicitly safe.

6.3.5 Implementation Issues

Closed world assumption. Java's success and growth has historically been closely linked
to the Internet; a program written in Java can be run on any platform, a pragmatic paradigm
for writing Internet applications. Indeed, Java's security mechanisms are targeted towards
safely running untrusted mobile code downloaded from the Internet without risking the
security of the system [137]. The mobile code paradigm describes an open world, one in
which new classes can be loaded into the virtual machine during run-time.

A down-side with the current implementation of soft boundaries is that it is directed at
working under a closed world assumption. The analyses performed as part of soft bound-
aries rely on the assumption that all classes are available when the analysis is performed.
The addition of new classes can create new control and data flow paths, paths which the
initial analysis did not consider. Thus, when securing an open-world program, the analysis
would have to anticipate control and data flow paths that do not yet exist, whereas securing a closed-world program only requires checking paths that actually exist in the application.

However, since many Java programs operate in a closed world, with all classes immediately available, we believe that focusing on a closed world is reasonable, even though it is not always desirable. The closed world assumption in particular enables a developer to more easily implement soft boundaries over legacy code. Note that some of our analysis can be applied to secure programs operating in an open world; some instances are mentioned in the results in Section 6.4.

A related problem is that Java programs can use reflection. Java reflection allows Java programs to access fields, methods, and classes dynamically by name. The analysis tool we use cannot determine the precise meaning of an arbitrary method invocation or field access performed through reflection. There are some limited cases where we can, such as on invocation of `java.lang.Class.newInstance()` where the receiver is known. However, these are the exception rather than the rule.

**Ownership.** A second issue that comes up is object ownership, as it determines how objects are treated and where violations of policy occur. The protection domain or task owning an object has access to all data and functionality of the object. As with the location of boundaries, ownership can be determined in several ways. For example, when trying to bound the memory used by a component, shared ownership works (see Chapter 4). Under shared ownership, all tasks using an object can be treated equally and charged for the object. This makes sense in the case of memory accounting because all tasks pointing to an object are helping to keep that object alive in memory.

In soft boundaries, however, data within an object should not necessarily be accessible in other protection domains. In Java, because there is no practical way to prevent a method in a class from accessing private data in instances of the class, every instance of the object should be in the same domain as the code. Thus, for the sake of soft boundaries, object ownership is established by the location of the class’s code, even if an instance is created in another domain. The one exception to this rule is system classes; each instance of a system
class belongs to the domain where it was instantiated. This enables container classes used by a domain to receive the same protection as the domain itself.

One consequence of this decision is that there is no way for multiple instances of a class to exist in different protection domains. However, since the code executing for each instance of a class is the same, it should be equally trusted. If a data structure is used extensively by multiple protection domains, it can be placed in its own protection domain, and treated as a service provided to other tasks (as we did with the Java servlet API used by the Miniature Java Web Server, described in Section 6.4.1).

6.4 Results

We built prototypes of the tools described in Section 6.3 to assist in establishing soft boundaries over four real-world systems, the Miniature Java Web Server, FreePastry, Sandstorm, and the Java Analysis Studio (JAS). Table 6.1 gives an overview of some relevant statistics of the complexity of these programs. The table shows the programs to be large enough that we can expect them to exhibit interesting data-sharing behavior. We then ran our three prototype tools, the API checker, the critical data checker, and the thread tracker, over the programs. The results are described in the following Sections, grouped by the tool that produced them.

6.4.1 Four real world programs

We implemented our soft boundaries mechanism around for real-world applications, the Java Miniature Web Server, FreePastry, Sandstorm, and the Java Analysis Studio. All four are multi-task frameworks in which tasks from a variety of different parties might be concurrently run. All four are written in Java, the language-based system most used in deploying multi-task programs such as these. Java has a number of protection mechanisms already built in to it, and is therefore a natural fit for designing and implementing a separation mechanism. These programs characterize the sorts of environments in which soft boundaries might be needed.
<table>
<thead>
<tr>
<th>Classes</th>
<th>Lines of Code</th>
<th>Public Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Analyzed</td>
</tr>
<tr>
<td>Web Server</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>FreePastry</td>
<td>493</td>
<td>240</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>334</td>
<td>190</td>
</tr>
<tr>
<td>JAS</td>
<td>582</td>
<td>263</td>
</tr>
</tbody>
</table>

Table 6.1: This table holds the number of classes and lines of code for each of the applications. The total numbers show the statistics for the entire application package. The analyzed numbers show the statistics for the classes that were actually analyzed by Soot. Soot only analyzed classes that the application used.

The Miniature Java Web Server

The Miniature Java Web Server is a minimal web server written in the Java language\(^1\). It is written as a servlet container, which can load servlets written using the Java Servlet API. It includes two standard servlets, one to serve static files and one to run external dynamic content plug-ins using the Common Gateway Interface (CGI). It can also be extended with any other servlet written using the Servlet API.

Servlets have a well-defined API that limits their interaction with the server to prevent the servlets from disrupting the server. When the server receives a request for a servlet, it packages the request into an object and passes this request object, along with a corresponding response object to the servlet. The request object should contain all the information the servlet needs to respond, and the response object contains the methods for sending the response to the requestor. However, the server cannot necessarily trust that the servlet will always perform its task and return. Further complicating the situation, servlets cannot necessarily trust each other, either.

We performed our analysis on version 1.42 of the Java Miniature Web Server, consisting of 5,392 lines of code (The current version is 1.7a). For the purpose of our soft boundary code, we established a soft boundary separating the servlets from each other and from the

---

servlet container. The servlet container's small size means that detailed analysis of the code was much easier.

**FreePastry**

FreePastry\(^2\) is an implementation of the Pastry peer-to-peer system [113]. Pastry is a peer-to-peer substrate, meaning it only provides the basic infrastructure of a peer-to-peer system. This infrastructure includes membership management and message routing. Tasks can be run on top of this substrate, using the functions in the substrate to provide some service to the peer-to-peer system. For instance, PAST is a distributed file storage service that runs over Pastry [114].

FreePastry, as well as other peer-to-peer routing infrastructures, provide a common key-based routing (KBR) API [36] to applications. Although peer-to-peer applications are commonly built for one particular peer-to-peer implementation, if the KBR API is enforced, then porting the application to other peer-to-peer implementations becomes a lot easier. Similarly, an application should not be able to use this interface or bypass it to modify data internal to the routing infrastructure, possibly interfering with the node's connectivity and even disrupting other nodes on the peer-to-peer network.

We implemented separation for version 1.32 of FreePastry, totalling 59,405 lines of code (the current version is 1.4). An implementation of the PAST distributed file storage service is shipped with FreePastry. To test FreePastry, we established a soft boundary between the core Pastry code and the code implementing PAST.

**Sandstorm**

Sandstorm\(^3\) is an implementation of SEDA, the staged event-driven architecture [139]. For high demand Internet services such as Amazon.com or Yahoo, service latency is critical. The server should be robust enough to gracefully recover from bugs and continue servicing requests. The SEDA architecture is intended to perform well with parallelized Internet

---

\(^2\)See [http://freepastry.rice.edu/](http://freepastry.rice.edu/).

\(^3\)See [http://sourceforge.net/projects/seda/](http://sourceforge.net/projects/seda/).
applications that may cause other architectures to sputter. SEDA provides an infrastructure for implementing such servers.

We implemented separation for version 3.0 of Sandstorm, consisting of 39,948 lines of code. We ran the Haboob web server as a SEDA task on top of Sandstorm. We specify a soft boundary between the Sandstorm infrastructure and the Haboob web server. However, note that the Sandstorm-Haboob relationship is somewhat different from the program-task relationships described above because state in Sandstorm is wholly owned by Haboob. The primary purpose of the soft boundary, therefore, is to shield Haboob from changes to the internals of Sandstorm across different versions. In addition, due to the stressful environment in which Sandstorm operates, it is imperative that bugs in a SEDA task cannot escalate to disrupt the Sandstorm infrastructure itself.

**Java Analysis Studio**

The Java Analysis Studio (JAS)\(^4\) is a Java graphical data analysis and visualization framework. It is highly modular, with most of its functionality implemented as plug-ins to the main package. It is built around the FreeHEP\(^5\) library, which was originally built to simplify data analysis for high-energy physics applications.

One problem we ran into was that, because of JAS loads plug-ins using Java reflection (in particular, it uses the `java.lang.Class.newInstance()` to instantiate plug-ins), our analysis failed to follow the control flow edges into the plug-ins; this problem is described in more detail in Section 6.3. This may deflate the number of bugs found in this example. To address this, we would have to replace the `java.lang.Class.newInstance()` calls with explicit instantiation of the plug-ins and invocations of their constructors.

We implemented separation for release 0.7.7 of JAS version 3, which consists of 41,950 lines of code. In addition, JAS makes heavy use of the FreeHEP library, which consists of over 100,000 lines of code. We placed the JAS code in one protection domain and the FreeHEP library in another. Each plug-in was likewise placed in its own protection domain.

---

\(^4\)See http://jas.freehep.org/jas3/.
\(^5\)See http://java.freehep.org/.
<table>
<thead>
<tr>
<th></th>
<th>Errors</th>
<th></th>
<th>Warnings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data accesses</td>
<td>Non-API Invokes</td>
<td>Non-API methods</td>
<td>Direct invokes</td>
</tr>
<tr>
<td>Web Server</td>
<td>0</td>
<td>8</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>FreePastry</td>
<td>0</td>
<td>0</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>5</td>
<td>61</td>
<td>152</td>
<td>4</td>
</tr>
<tr>
<td>JAS</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: This table shows the number of API errors and warnings generated by the API checker the Miniature Java Web Server, FreePastry, Sandstorm, and the Java Analysis Studio. *Data access* are direct accesses to the fields of a protection domain from outside the domain. *Non-API invokes* are invocations of non-API methods from outside the protection domain. Non-API methods are declarations in an exposed class of public methods not declared in the API. Direct invokes are invocations of API methods where the declared receiver type is a class in the protected domain rather than the API.

JAS also uses the JDOM XML library\(^6\) and the OpenIDE library for registering and looking up components.\(^7\) These were placed in their own protection domains, as well.

### 6.4.2 Results from API checking

We ran our API checker, described in Section 6.3.2, over our four test programs to evaluate how closely they follow their API. Table 6.2 reports errors and warnings in each program. The errors report violations of the API that need to be fixed. Warnings should also be examined to achieve a more robust partitioning, but doing so is unnecessary. The results support the feasibility of establishing soft boundaries in real world software.

**Miniature Java Web Server**

There were eight API errors in the servlet container protection domain of the Miniature Java Web Server, all of which involved servlets invoking non-API methods of the servlet container. The apparent culprit is that there are a number of classes in the servlet container

---


\(^7\)See [http://openide.netbeans.org/lookup/](http://openide.netbeans.org/lookup/).
protection domain that are clearly intended to be used as utility classes for use by the servlets or for interaction between servlets and the servlet container, even though they do not implement the Java Servlet API. Upon closer examination, all eight method invocations appear to be safe to include in the API.

The web server had so few errors because the code base is small and Java provides the Servlet API which addresses most server-servlet interaction needs. This is in contrast to a large number of shared objects, showing that although the servlet container and servlets have a high level of interaction, most of it follows the servlet API. The specification provides a natural boundary to even poorly-designed code. However, as the critical data analysis results in Section 6.4.3 will illustrate, security hazards can arise, even in the presence of an API, if a programmer does not realize the danger in allowing access to private data or accidentally allows such access.

FreePastry

We defined FreePastry’s API as the commonapi directory in the FreePastry distribution. FreePastry had no API violations, but there were numerous API warnings. FreePastry has 150 public methods not in the API. Of these public methods, most are harmless, but there are some that could cause problems if invoked, an issue explored further in the critical data analysis results in Section 6.4.3. It may seem troubling that some of these public methods are a potential security hazard, and the API checker only raised warnings, but the results reflect the closed-world assumption discussed in Section 6.3. If any of these methods were actually invoked by the application, the invocation would have triggered an API violation, however, since errors returned by the API checker represent actual security hazards rather than potential security hazards, none of the methods were marked. As long as any added classes have the API checker run over them, the public methods are safe and invocations of methods not in the API can be caught when they occur.
public void handleEvent(QueueElementIF item) {
    ...
    BufferElement payload = resp.getPayload();
    // payload is a pointer to an object in another protection domain
    byte paydata[] = payload.data;
    ...
}

Figure 6.4: Haboob violating the SEDA API.

Sandstorm

Sandstorm’s API was taken to be the api directory included in the Sandstorm distribution. Sandstorm, an implementation of the SEDA architecture, had thirteen invocations of methods not in the API and four direct data accesses. Haboob was the only application to directly access data. Figure 6.4 shows an example of one such access. In the code, method handleEvent is violating the Sandstorm API by choosing to bypass the existing public getter for the data field; the BufferElement class is in the Sandstorm protection domain. The accesses turn out to not be security concerns, but are still poor programming practice. Later versions Sandstorm may alter the treatment of data introducing subtle bugs that could have been avoided had Haboob been written to access Sandstorm through the proper channels.

Java Analysis Studio

The Java Analysis Studio’s API was taken to be the FreeHEP library, where the API for interaction between components is specified. No violations of the API were found, and only three public methods were defined that didn’t appear in the API. Since these methods are not invoked anywhere, they don’t pose an immediate danger. A brief inspection reveals no danger posed by invoking these methods.
API Checking in an Open World

Although these results assume that API checking is applied in a closed world, the results are also helpful in an open world. In an open world, however, we have to worry about the cases where an API violation could occur. That is, the warnings become as important as the errors. It may be necessary to restructure the program to eliminate these warning cases and to ensure that only methods in the API are exposed to other protection domains. Unfortunately, this process is more involved than its equivalent in a closed world.

6.4.3 Critical Data Analysis

We then ran the critical data checker, described in Section 6.3.3, over the Miniature Java Web Server and FreePastry. In both cases, objects that should not be modified outside the core classes were marked critical and, in both cases, the results yielded potential security concerns. Sandstorm is a code library to save the programmer from having to worry about low-level event details. Because any state held in Sandstorm inherently belongs to the client application using Sandstorm, it is safe for the application to directly or indirectly influence this state. This makes critical data difficult to identify if it even exists. As a result, we did not perform critical data analysis over Sandstorm. Similarly, because we are not sufficiently familiar with the relatively large size of the Java Analysis Studio codebase, we could not confidently label any state as critical.

Miniature Java Web Server

Examining the code for the web server revealed that much of the critical data was stored as fields in the servlet container’s primary implementation class, Serve. Because of the container-servlet abstraction provided by the servlet API, the servlet has no particular need to dereference instances of class Serve. We marked the instantiation of class Serve as critical, and critical data analysis revealed that this object was being passed to the servlets as the servlet context.

Figure 6.5 shows the offending line of code: the instance of class Serve (object this in the code snippet) is being cast as the ServletContext and passed in to the instantiate
public class Serve implements ServletContext, RequestDispatcher
...
private void addServlet(...)
{
...
servlet.init( new ServeConfig( (ServletContext) this,
    initParams, urlPat ) );
registry.put( urlPat, servlet );
...
}
...

Figure 6.5: The Serve class is the servlet container's primary implementation class. Notice that it implements ServletContext. The method shown here is the method used by the server to add new servers. ServeConfig implements ServletConfig. As a result, every servlet can get a reference to the servlet container's base object through the Java Servlet API by requesting its context.

ServeConfig (which extends ServletConfig), which allows instantiated servlets to get a reference to the Serve object (by calling ServletContext); this is all according to the API, however. The problem is in which classes were chosen to implement the API, a program design issue. The programmer assumed it was safe to pass the server to the servlets, which is only true if the servlets are completely trusted and free of bugs.

To investigate the impact of using class Serve as the servlet context, we additionally marked the mappingtable and the realmstable critical. The former maps paths to servlets and the latter maps users to passwords, when passwords are required. Neither of these structures should be accessible to a malicious servlet, but critical data analysis reveals seven modifications to them through four entry methods. Of these four entry points, two are setters for these fields, and one is the destroyServlets() command. The remaining method is the serve() method, which starts the server. All four methods are security hazards, but since none are in the API, the API is correct. To fix this, the server object should not be used as the servlet's context.
FreePastry

In FreePastry, the Node class holds data critical to the functioning of the Pastry network. This data includes the node’s ID and the routing table. If an application has the ability to change its node’s ID then a malicious application could rapidly change the ID, causing surrounding nodes to interpret each change as a new node appearing on the network. Each surrounding node would consequently delete a subset of their keys and each change would propagate down the network.

The node’s ID was declared critical and the checker was run over the code. Although the application does not explicitly change the node’s ID, three of the eight public functions that the tool identified as dangerous allow the application to change the ID in a dangerous manner: setdigit(...), xor(...), and setbit(...). Even though the information critical to the node ID is declared private, any of these functions could be used to perpetrate the attack just mentioned, however, none of these methods are in the API so the API is correct.

Critical Data Checking in an Open World

As discussed in Section 6.2.1, the primary purpose motivating the critical data checker was to ensure that the API does not give access to critical data. In a closed world, this is all that is needed, since the API checker will verify that only API methods are called. In an open world, however, this is no longer the case. In discussing the results of the API checker (in Section 6.4.2 above), we note that public methods not in the API should either be made non-public or added to the API; the critical data checker provides a quick check to reject methods from the API. In FreePastry, for example, critical analysis reveals that setdigit(...), xor(...), and setbit(...) are dangerous and should not be added to the API. Other public methods might be safe.

Using the critical analysis to determine which methods can safely be added to the API is not without risk. Critical analysis uses a fine-grained specification, and any state not marked critical is assumed to be non-critical. It is therefore imperative that some care and
<table>
<thead>
<tr>
<th>Edges leaving domain</th>
<th>Unique starting classes</th>
<th>Unique entry points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server</td>
<td>79</td>
<td>5</td>
</tr>
<tr>
<td>FreePastry</td>
<td>149</td>
<td>5</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>55</td>
<td>10</td>
</tr>
<tr>
<td>JAS</td>
<td>1101</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 6.3: Results of applying the thread tracker. *Edges leaving domain* is the total number of edges exiting a protection domain. *Unique starting classes* is the number of thread starting points leading to an edge leaving the protection domain. *Unique entry points* is the number of edges entering a protection domain which lead to edges leaving the protection domain.

Common sense be applied in adding methods to the API based on the results of critical analysis.

### 6.4.4 Thread tracker results

Finally, we ran the thread tracker, described in Section 6.3.4, over our four test programs, the Miniature Java Web Server, FreePastry, Sandstorm, and the Java Analysis Studio. The results of this analysis are shown in Table 6.4.4. They show that the effort required to check thread starting points within a protection domain is far less than the effort required to check all edges exiting the domain.

**Miniature Java Web Server**

When run over the Miniature Java Web Server, the thread tracker returned that nearly all boundary edges trace back to a class called ServeConnection, which starts a new thread for each servlet. In addition, a few boundary edges traced back to anonymous inner classes. The only issue of concern is illustrated in 6.6. Additional servlets, programmed by the user, are all instantiated using the same thread; if one servlet’s constructor does not return, it would prevent subsequent servlets from loading. But such an error should be caught when the server is first started and, consequently, cause minimal damage. If the
public class Serve implements...

private void readServlets(File servFile)
...

while (se.hasMoreElements()){
    servletname = (String)se.nextElement();
    // addServlet() instantiates a new servlet.
    addServlet(servletname, ...);
}
...

Figure 6.6: The Serve class uses a new thread to call readServlets which determines
which adds all servlets besides the default two. If one servlet’s constructor does not return
then the loop can add no more servlets.

server is extended to dynamically load servlets, however, each servlet should be constructed
in its own thread.

**FreePastry**

Figure 6.7 displays the code for three dangerous boundary-crossing function calls that occur
in the FreePastry code base. In (a), FreePastry has received a message and is inquiring with
the application whether to forward it. In (b), FreePastry is informing the application that the
node’s leaf set has changed. And in (c), FreePastry has received a message for the current
application and is delivering it. These functions are defined in `rice.P2P.commonAPI.
Application` as the functions through which the FreePastry node communicates with an
API so the invocations are necessary; the problem is that they can occur in the Pastry node’s
primary thread of control. One solution to this problem is for each node to keep a thread
pool and use a separate thread for each communication between node and application.

**Sandstorm**

The thread tracker collapses 55 crossings into 10 starting methods and 8 entry methods.
In the worst case, to separate Sandstorm from Haboob requires examining only 18 classes.
However, as mentioned in Section 6.4.1, because Sandstorm is running on behalf of Ha-
public class PastryEndpoint extends ...{
    ...
    public final boolean enrouteMessage(...){
        ...
        return application.forward((RouteMessage) msg);
        ...
    }
}

(a)

public void leafSetChange(...){
    ...
    application.update(nh, wasAdded);
    ...
}

(b)

public void receiveMessage(...){
    ...
    application.deliver(rm.getTarget(), pMsg.getMessage());
    ...
}

(c)

Figure 6.7: The three examples each show a lines in the Pastry code base where Pastry calls into the application. The methods forward, update, and deliver are functions through which Pastry talks with the application task. Under the current implementation, however, the calls are made using the main Pastry thread.

... boob, there is no risk of a thread exiting Sandstorm never to return. In fact, starting any unnecessary threads to try and increase separation would be counterproductive, slowing an efficiency critical system.

Java Analysis Studio

The thread tracker again significantly reduced the amount of analysis necessary to determine cross-boundary calls. Despite there being 1101 separate cross-boundary method calls, they were restricted to 68 different entry methods, and only 17 unique thread starting points within the codebase. A brief inspection revealed no significant threats.
Thread Tracking in an Open World

To determine if a protection domain has a thread escape bug, the thread tracker only needs to consider the code in that protection domain. As a result, the same analysis we perform ahead of time can just as well be performed as tasks are being loaded. In addition, even in an open world, the core system – the one we are most concerned about protecting – is generally loaded from the start.

6.5 Summary

When tasks from different sources are run together, terminating tasks that share data and control flow can be dangerous. In this chapter, we introduced soft boundaries. Unlike the transaction-based boundaries discussing in Chapter 5, soft boundaries are a far lower-overhead way of achieving task separation. Since they are based on static analysis, soft boundaries have essentially no run-time overhead. Soft boundaries also allow for data sharing among tasks.

Soft boundaries are similar in construction to component-based systems. Component-based systems, like soft boundaries, seek to establish and enforce clear interfaces among tasks. In the case of component systems, the goal is to simplify the development and maintenance of the code. When code is modularized and closely follows a specified interface, it is easier to understand the interactions among the various components. As this chapter shows, it also makes the code amenable to static analysis.

We applied soft boundaries to four different Java plugin-based systems, the Java Miniature Web Server, FreePastry, Sandstorm, and the Java Analysis Studio. In the Java Miniature Web Server, soft boundaries revealed cases where the Java Servlet API was being bypassed, some cases where critical data was leaked, and a potential thread-capture bug. In FreePastry, soft boundaries revealed a number of functions unwittingly exported which gave access to sensitive system state, as well as several places where thread capture could occur. No definitive bugs were found in Sandstorm or the Java Analysis Studio, although there were some warning flags raised.
Chapter 7

Issues with Modifying Language Semantics

In the previous chapters, I discussed several language-based mechanisms which add operating system-style semantics to language-based systems. These systems use a variety of techniques to actually implement the additional semantics. Semantics were either be added below the language level through alterations to the language run-time system, at the language level through systematic alterations to code being run on the system, or above the language level through static analysis of code run in the language run-time system.

In this chapter, I compare our experiences implementing systems using these various alternatives. Section 7.1 discusses code-to-code transformation, including some techniques for optimizing the transformed code. In Section 7.2, I discuss modifying the language run-time system to support new semantics. Finally, I discuss how static analysis can be used to modify a language in Section 7.3.

In this chapter, modifications are discussed in terms of the Java programming language and the Java Virtual Machine. Although the ideas explored in this thesis are applicable to any language-based system, all proof-of-concept implementation was done in Java. In this chapter, since I discuss practical issues with modifying a language, it is only natural to place the discussion in the context of the language that was actually modified.

7.1 Code-to-Code Transformation Issues

Code-to-code transformation is one technique for adding semantics to a language-based system. It is commonly implemented by rewriting the intermediate bytecode executed by the language run-time system. This can be done either by statically rewriting the intermediate object files on disk or by dynamically rewriting the intermediate code as it is loaded.
Figure 7.1: How a Java bytecode transformation changes the process of loading a class. If an already loaded class, `Foo`, uses an as yet undefined class `Bar` (either accesses a static member or creates an instance) (1), the JVM traps the undefined reference to `Bar` (2), and sends a request for the class loader to load the class (3). The class loader fetches the class file (`Bar.class`) from the filesystem (4). In standard Java, the input class is then loaded into the JVM (5). In a bytecode rewriting system, the bytecode transformer is first invoked to transform the class (4a). In either case, the class is now loaded in the JVM (6).

This process is illustrated in Figure 7.1; although described in terms of Java, the same idea applies to other language-based systems.

7.1.1 Implementations

Two of the systems I discuss in this thesis, soft termination and transactional rollback, were implemented in Java with bytecode rewriting. I present a brief overview of each system here, but a more complete discussion can be found in the chapters devoted to each system. I then discuss some implementation issues and limitations, as well as some optimization techniques we used.
Soft Termination

Soft termination is a technique for safely terminating tasks. The basis for soft termination is that the task doesn’t need to terminate immediately, as long as it is guaranteed to eventually terminate. We implemented soft termination by first adding a termination flag field to each class. We then instrument each method, preceding all backward branches with termination checks to prevent infinite loops and beginning all methods with termination checks to prevent infinite recursion. Detailed discussion of soft termination, including performance results, can be found in Chapter 3.

Transactional Rollback

Transactional rollback was designed to complement soft termination in a resource management system. Where soft termination seeks to prevent system state from being left inconsistent by a task’s untimely termination, transactional rollback seeks to preserve the consistency of the task’s persistent state. We do this by keeping track of all changes made by a task, and if the task is terminated, the changes are undone. Each task is run in the context of a transaction to avoid data conflicts. We implement transactional rollback by first duplicating each method in all classes (including system classes). The duplicate methods take an additional argument, the current transaction, and in the duplicate methods, all field and array accesses are preceded by respective lock requests. Detailed discussion of transactional rollback, including performance results, can be found in Chapter 5.

7.1.2 Implementation Issues

The flexibility of Java bytecode is both its benefit and its curse. A bytecode rewriting system must deal with every feature in the bytecode. We first deal with “local” features, which is to say, aspects of the Java bytecode where an individual class or method can be rewritten without needing any knowledge of how other classes in the JVM might interoperate with it.
In many cases, particularly when we consider changes that affect the method signatures advertised by a class, we must consider how our changes interact with other Java classes, with native code, and even with classes that may not have been loaded into the system yet.

**Constructors**

Constructors have two properties which set them apart from normal methods. First, no operations, whether initialization or otherwise, can be performed until either the super-class constructor or a same-class constructor has been called, although arguments to a super-class constructor may be legally computed. This presented a problem in the soft termination system.

Soft termination is designed to check the value of the termination flag at the beginning of every method, including constructors, but inserting this check into a constructor results in a verifier error. Instead, the best we can do is to check for termination immediately following the call to the super-class constructor. As a result, we may not be able to terminate a loop constructed with constructors that recursively invoke other constructors before calling their super-class constructor.

Transactional rollback requires every object to be locked before it can be modified. Normally, this is accomplished with the addition of a specialized lock instance to every object instance. However, while an object is being constructed, this lock instance will be null. To address this, we needed to add checks everywhere an object is accessed to check whether its lock instance is null, slowing down all locking code to handle this special case.

**Exceptions**

Exceptions (and errors) can introduce implicit or asynchronous control flows into a Java method. Asynchronous exceptions, induced by calls to `Thread.stop()`, can occur literally anywhere, and at any time. For this reason, among other reasons, Sun has deprecated the method. Implicit exceptions typically occur as a side-effect of a program error, such as a `NullPointerException` or an arithmetic errors. Here, the control flow edge from
the potentially faulty operation to the construction and throwing of an exception does not appear explicitly in the Java bytecode.

This is a problem if the semantics being added involve an additional parameter being passed to all function calls (as in the case of transactional rollback). The constructor of the generated exception needs to receive the added parameter. With transactional rollback, this was not an issue because the constructors for these implicit exceptions never use the added parameter, so it can be ignored. If the parameter is needed, one alternative is to use thread-local storage. However, thread-local storage is expensive, requiring a hash table operation on every store, so should be avoided.

We also observe that, in Java bytecode, an exception handling block (that is, a try-catch block) is represented as offsets delimiting the try block and the offset for the start of the catch block. The JVM specification requires that the catch block start strictly after the try block ends. That is, an exception-handler cannot handle its own exceptions. However, Sun's JVM does not enforce this restriction.

This can be used by a malicious task to evade a soft termination signal. The exception handler acts as a backward-branch; when an exception is thrown in the exception handler, control passes to an earlier point in the text of the program. If this exception always occurs, like the termination signal of soft termination, the result will be an infinite loop. We solved the problem by having the soft termination system detect and reject such exception-handling blocks.

**Threads**

Threads can be thought of as a special case of native methods which make up-calls to Java code (in this case, during thread initialization, the up-call happens to be on a new thread while control returns on the original thread). In transactional rollback, we needed state computed in the parent thread to be sent to the child thread. This was performed by modifying java.lang.Thread to have an additional field where the parent thread can store context information in the child thread. This context is consulted when the child's run() method is invoked by the thread run-time system.
Another important issue with threads is controlling them when they block. Blocking can be caused by synchronization primitives (see below) or by native I/O methods, which might not return right away. Luckily, all of the JVM’s methods which might block will respond to calls to Thread.interrupt(), causing the formerly blocked thread to immediately throw an exception and canceling the operation that was previously under-way. We used this mechanism with soft termination to signal blocking threads that we wished to kill.

However, in implementing soft termination, we found that some mechanism was still needed to determine which thread is blocking, and whether it was blocking on behalf of system code (which should not be interrupted) or on behalf of a task (which we want to interrupt). We chose to wrap blocking methods with code to register the current thread with the soft termination system as blocking before the call, and unregister it afterward. We use Java’s stack inspection mechanism to determine the context of the blocking call (system versus task). The soft termination system could now interrupt blocking threads as necessary.

**Verification**

We have seen several cases where our own code had to effectively extend the Java bytecode verifier in order to guarantee correctness of our system. We saw these issues with Java’s synchronization and exception features. We also saw cases where Java’s verifier got in the way of perfectly sound program transformations, particularly with regard to the restrictions on how Java’s constructors invoke their super-class constructors.

Ideally, the Java bytecode verifier should be disentangled from the current class loading system to stand on its own. This would simplify the addition of new checks to the verifier, such as checks for undesirable exception and synchronization behavior, and it would make it easier to remove checks that, at least in our context, are unnecessary, such as the super-class constructor checks. Furthermore, a modular Java bytecode verifier would be quite useful to our systems as a mechanism for checking our input before we rewrite it, allowing us to make stronger assumptions about the quality of our input and reducing the opportunity
for carefully crafted tasks to trick a code rewriting system into mistakenly outputting a rewritten task with more privilege than it should have been given.

“Special” Methods and Classes

Every JVM has certain methods and classes which are special in some way. Sun’s JVM, for example, doesn’t allow the rewriter to add fields to `java.lang.Class` or `java.lang.Object`. Another mechanism that some Java virtual machines use is to special case the invocation of some method to instead execute some internal machinery. In this case, any changes made to that method in Java are ignored, because the method is never actually executed. We use this technique when implementing garbage collection-based memory accounting (see Section 7.2.2 for details).

If a global transformation could be applied to all Java classes in a consistent way, such as with transactional rollback, then the resulting system would remain self-consistent. However, with special methods and classes, it becomes more difficult to have this consistency. Now, we must keep a list of special classes and methods and treat calls to them as special cases.

For transactional rollback, this means that the transaction state cannot be passed as an additional argument to all methods. Instead, we store the transaction in thread-local storage, where it can be recovered if needed. If the special method returns back to the caller, then no special action is needed. If the special method calls back into Java code, then the transaction can be loaded from thread-local storage, and execution continues. Because transactional rollback duplicates all methods and supports two modes of execution, with transactions and without transactions, this can be done fairly smoothly. If we were rewriting all methods instead of duplicating them, however, we would need a wrapper method to support up-calls from native code, because the native code making the up-call is not aware of the transformation.
Inheritance

Java's class inheritance is also a complicating factor in global transformations. When a subclass overrides a special method, as described above, it inherits the "specialness" of the method. For example, `java.lang.Object.hashCode()` is a native method. Any other class can override this, providing a Java (or native) implementation. At an arbitrary call site, where `java.lang.Object.hashCode()` is invoked, there may not be enough information to determine a more specific type for the callee. Thus, the caller must assume the worst case: the callee is special. This requires saving the security context, and then making a call to the special method. Of course, if the concrete type was something other than `java.lang.Object`, and it had, in fact, overridden `hashCode()`, then control would enter a wrapper method which would recover the security context and call back into the world of rewritten code.

Open World Java

Java is an open world system. That is, classes can be loaded and the class hierarchy modified at run-time. The only restrictions that Java makes are that for a class to be loaded, all of its super-classes must already be loaded. Even with this restriction, the open world assumption complicates code analysis.

In both soft termination and transactional rollback, the transformations are applied fairly locally to each class. As a result they can be applied in an open world. However, assuming an open world prevents us from optimizing our transformation in ways that rely on knowledge of code in other classes. For example, in Section 3.2.5, we propose eliminating termination checks from a method if we can statically prove that this method terminates. A method is guaranteed to terminate if it has no backward branches and calls no methods or if all called methods terminate.

In an open world, however, every Java class to be inserted into the system can potentially invalidate a judgment made by the optimizer. As a result, the optimizer would need to back out the optimizations that now reside in code previously loaded into the system. As
we discussed in Section 7.1.3, such functionality has become available in the JDK 1.4 debugging architecture, and might make such optimizations possible, even in an open world.

**Bootstrapping**

Bootstrapping presents a unique problem to bytecode rewriting systems. When the JVM is launched, it normally proceeds through the initialization of its core classes before loading any applications. These core classes are, by necessity, carefully designed to avoid circular dependencies in their static initializers. Circular dependencies can be particularly hazardous in JVMs implemented themselves in Java, where the very first classes to be initialized are all "special" to the system in some way, and are very fragile with respect to changes.

In the implementation of soft termination, we largely did not need to worry about bootstrapping because we only had to transform tasks, not the entire system. For transactional rollback it was necessary to transform everything (more precisely, everything that was not special, in some fashion or another). Fortunately, we have the benefit that we support two modes of operation: with and without transactions. As a result, the JVM can initialize itself normally, and we only transition to the transactional world when we are about to start a task.

If we were rewriting all methods instead of duplicating them, however, the bootstrapping process becomes more difficult. We basically have to support a mode of execution that still works without all the core Java classes loaded. The particulars obviously vary depending on the nature. In a system like transactional rollback, we could simply pass null as the transaction state parameter. We'd obviously have to ensure that transformed methods can handle the case where the transaction is null.

### 7.1.3 Limitations

The primary limitation of employing source-to-source transformations comes from restrictions in the interface exported by the language run-time system. In the case of Java, a number of internal JVM structures and processes are not exposed to the language, even
at the bytecode level. As a result, certain features need to be integrated into the language run-time system itself.

Class Reloading

Once classes are loaded into the JVM, they can neither be unloaded nor reloaded. Likewise, one cannot control when a class’s static initializer will be called or when dead objects’ finalizers will be invoked. Java version 1.4 includes a feature in its debugger architecture called HotSwap, which allows classes to be reloaded\(^1\). Existing activation records continue to run with code from the original version of the class. Also, the new class’s static initializer is not run again, and new fields added to existing classes will be null or zero. This new feature was not available when we were building our systems, and would have been a welcome addition, despite its limitations.

Memory Management

One of the features on our “wishlist” is the addition of hooks into the JVM’s memory system. Had we been able to exploit the garbage collector’s safe point traps, our soft termination system could have performed periodic checks with more flexibility and lower overhead than trapping backward branches. Also, by modifying the garbage collector, we might have been able to enforce memory usage policies that are more robust than those used in JRes and J-SEAL2, which assume that whoever allocated a block of memory should be responsible for “paying” for that memory. This is exactly how we implemented garbage collector-based memory accounting (see Chapter 4). New pluggable garbage collection systems for Java, such as GCTk [14], may allow us to implement such features without requiring changes to the underlying JVM. In addition, the Real-time Specification for Java\(^2\) allows programs to create individual memory regions treated differently by the garbage collector and select from which region memory is allocated.

\(^1\)See http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html for more information.

\(^2\)See http://www.rtj.org/ for more information.
Thread Scheduling

Thread scheduling is another black box subsystem of the Java virtual machine. No mechanism is provided for either replacing the thread scheduler or fine-tuning how threads are scheduled; the only interface provided is the Thread.setPriority() method. The Real-time Specification for Java does allow for replacing the thread scheduler, but was not available to us when this was implemented.

Native Methods

Native methods, while not strictly part of the JVM, are also treated as black boxes. We cannot control where a native method might go, and how that native method might behave. Native methods might perform arbitrary computations and are not necessarily guaranteed to return in a timely fashion (e.g., I/O routines might block indefinitely while they wait for data to arrive).

Furthermore, we have no control over up-calls from native methods back to the Java classes which we do control. In particular, we have no access to the Java Native Interface (JNI) calls used by native methods to interface with Java. If we could intercept these calls, then we could transform native methods to see a view of the Java classes consistent with what the Java classes themselves see after being transformed. Since that is not an option with current JVMs, we have adopted a number of strategies to cope with native methods, described in Section 7.1.2.

Note that certain JVMs implemented in Java, such as BulletTrain and IBM's RVM [2], use fewer native methods than JVMs implemented in machine code. The tradeoff is that they have more methods which are not native but are treated specially by the JVM. In our bytecode transformations, these methods need to be treated the same as standard native methods. See Section 7.1.2 for details on such special methods and how we handled them.

Arrays

Transactional rollback needs to be able to save backup copies of all objects before they are written. For most classes, we can create "backup" fields for every original field in the class.
Assigning the backup to refer to the original object is sufficient to preserve the original value of the backup. However, for arrays, this no longer works; there is no place in the array to store a reference to the array's backup. Our solution is to maintain a global hash table that maps arrays to their backups. For each array, the backup array must be the same size as the original. Creating this backup requires copying the whole array. For tasks that make extensive use of arrays, whether large or small, this creates a significant overhead. Our preferred solution would be for Java to have a mechanism to let us add our own field to all arrays. Java's arrays already track their length; we want one more reference that we can use transparently to the Java application.

**Synchronization**

The semantics of Java’s `monitorenter` and `monitorexit` bytecode instructions and synchronized method modifier cannot be changed through bytecode rewriting. When a deadlock occurs in Sun’s JDK, the only way to recover is by restarting the JVM. The JDK 1.4 debugging architecture provides a mechanism to address this (a debugger is allowed to forcibly pop activation records from a thread's stack), which might be useful to clean up after deadlock. This can similarly be used to terminate threads that are in a deadlock situation.

Another issue which soft termination had to deal with was the exact semantics of the `monitorenter` and `monitorexit` bytecodes, which acquire and release system locks, respectively. If these calls are not properly balanced, it becomes possible to lock a monitor in such a way that terminating the task will not cause the monitor to be released. Despite the fact that neither the JVM nor the Java language specifications allow such construction, current JVM bytecode verifiers accept such programs. Our soft termination system did not attempt to deal with this problem.

### 7.1.4 Optimizations

In the development of our systems, we used profiling and microbenchmarks to determine where optimization would be most effective. Microbenchmarks in particular could tell
us where our code is slowing down the most. Based on that, we would try to eliminate
transformations in these locations. By minimizing the amount of code we added through
transformations, we could improve the overall performance of the system.

Similarly, profiling told us where we might be doing more work than necessary. This
allowed us to rapidly focus our attention on the optimizations that might matter for our
systems. Of course, running our programs with such profiling slowed them down, but these
profiling checks are only actually included during code rewriting when we wish to gather
profiling data.

**Soft Termination**

The unoptimized design of soft termination required a termination check to be inserted
before every call site. For methods with multiple call sites, there would be one termination
check added per call site. We measured, through profiling, that on average there was more
than one termination check per method invocation. This led us to "push" the termination
checks from the call site to the head of the callee, thus reducing the number of termination
checks.

Next, we could easily determine which methods, having no outward method invoca-
tions, are guaranteed to return in a finite time. For these methods, the termination check at
the beginning of the method is unnecessary and can be omitted. In some cases, this more
than halved the overhead of soft termination relative to the overhead of the unoptimized
soft termination system. These optimizations are also described in Section 3.2.5.

**Transactional Rollback**

In our transactional rollback system, we learned a number of seemingly obvious facts
through profiling. We observed that, for most lock acquire operations, the transaction ac-
quiring the lock was the same transaction already holding the lock. We also observed that,
by far, the most common object to be locked in a given method is the this object.

These measurements led us to some simple optimizations with profound effects on
system performance. By checking if the lock to be acquired is already held by the current
transaction, we generally saved at least one method call, sometimes more. Likewise, by checking if a method contains multiple locking operations on this and consolidating them to a single operation at the head of the method, we were able to remove a significant number of lock operations. These two optimizations alone bought us a 25% speed improvement. These optimizations are also described in Section 5.2.7.

7.2 Language Run-Time System Design Issues

For some language modifications, however, code-to-code transformation is not sufficient. For many mechanisms internal to the underlying language run-time system, no interface is exposed to the language. One example of this is memory management. Although most language run-time systems employ sophisticated garbage collection systems to manage allocated memory, the only interface provided in the language is the memory allocation call (new in Java), and a method to forcibly run a garbage collect (System.gc()).

We modified the language run-time system to implement one of the systems discussed in this thesis, the garbage collector memory-management system. As discussed above, the JVM provides no interface to the memory management system in the language, save for memory allocation. I give a brief overview of the system here, and discuss some implementation issues we encountered with this technique.

7.2.1 Garbage Collector-Based Memory Accounting

Garbage collector-based memory accounting is a system for accurately determining the memory usage of a task running in a language-based system. Since memory can be shared among tasks, and an object may remain live even when the task which created the object is finished with it, simply charging memory to the task which allocated it is not a workable solution. Instead, memory is charged to a task which keeps that block of memory live.

Because nearly every language-based system has a garbage collector whose sole job is to determine which memory is live (for the purpose of reclaiming dead memory), this was the logical place to integrate the accounting system. And since Java, for which we implemented our system, provides no way to add hooks to the garbage collector, we were forced
to modify the JVM itself, customizing the garbage collector to count memory. A detailed discussion of the design and implementation of this system can be found in Chapter 4.

7.2.2 Implementation Issues

Modifying the garbage collector to support memory accounting was very straightforward. The only complication was in providing an interface for the language to access memory accounting data. Rather than create a native method to provide the interface, we took advantage of the fact that we could modify the virtual machine. An empty method was created as an API to the memory management system; the virtual machine detected invocation of this method, causing some internal code to run in the virtual machine. This technique is described in greater detail in Section 7.1.2.

7.2.3 Limitations

The main limitation of language run-time system modification is compatibility. Once you make a modification to a language run-time system implementation, it becomes incompatible with every other run-time system implementation for that language. Although from a prototyping standpoint, this is fine, if you want to deploy the system in the real world, this can be a problem.

7.3 Static Analysis of Code

Another limitation of both the code-to-code transformation as well as the run-time system modification approaches to adding semantics is that these systems can only act based on execution history. As a result, such systems are necessarily reactive. There is no way to detect a violation and remove it using these systems, unless the bug is serious enough to crash your entire system. Finally, systems that operate at run-time have some run-time overhead.

We used static analysis to implement soft boundaries. Unlike memory management, which deals with an explicitly run-time property, and termination, which requires some run-time mechanism, code boundaries are static properties of code. As a result we chose to
implement them via static analysis, to avoid the run-time overhead that would be associated with run-time enforcement. I give a brief overview of the system here, and discuss some implementation issues we encountered with this technique.

7.3.1 Soft Boundaries Implementation

Soft boundaries are implemented as a set of static analysis tools, which verify that the correct separation properties are maintained among tasks (i.e. protection domains). The tools do this by checking three general properties of the code being run. The first and most basic is that an API specified between two protection domains is never violated. The second is that a thread critical to a task never exits the task. Finally, the tools check that any data that is specified as critical to a task is never accessed or modified from outside the task.

Unlike memory accounting, which is necessarily a run-time capability, and termination, which is provably not statically analyzable, the soft boundary checks can be done statically. In addition, all of these situations indicate bugs in the code, and are not simply run-time events to check. As a result, an offline solution is preferable in this situation. This gives developers a chance to check and fix their code before it is deployed. A detailed discussion of the design and implementation of these analyses can be found in Chapter 6.

7.3.2 Implementation Issues and Limitations

The major limitation of static analysis is that it doesn't actually enforce anything, it merely advises that a violation of some desired property is possible. In the context of soft boundaries, this might be a notice that, for instance, the API separating two protection domains is violated at a particular call site. If the notice is ignored, then a violation is still possible. On the other hand, this allows the possible violation to be corrected before the system is deployed.

The static analysis tool primarily uses pointer aliasing analysis to determine to which objects a given variable might point. This static analysis is done conservatively, which means that some reported violations may be false positives. Unfortunately, whoever uses
our tool must distinguish between those reports that are actual bugs and those that are false-positives. This is a tradeoff for having the opportunity to fix bugs ahead of time.

7.4 Summary

Code-to-code transformations, run-time system modification, and static analysis are all powerful techniques for modifying the semantics of a language. Each technique has its uses and its limitations. Static analysis is the best option when the property being checked can be statically checked. If the language provides a mechanism for your desired semantics, then code-to-code transformation is the way to go. Otherwise, run-time system modification is your only choice.
Chapter 8

Future Work

There are a number of interesting problems related to this work that remain to be solved. In Section 8.1, I discuss some issues with deploying a complete system of language-based protections. Some interesting analyses that could be explored for understanding soft boundaries are covered in Section 8.2. Finally, Section 8.3 presents some places where the techniques we employed might be advanced.

8.1 Building a Complete System

This thesis discusses a number of research prototype systems for implementing language-based protections. Deploying language-based protections in the real world would require solving some additional interesting research problems. First, the problem of resource accounting must be solved for more than just memory. In addition, there are real-world implementation issues that present a challenge.

8.1.1 Complete Resource Accounting

In Chapter 4, we described a language-based system to account for memory usage by individual tasks using the language's garbage collection facility. Similar such systems could be designed to account for consumption of other system resources, such as network bandwidth and CPU time. As with memory accounting, there are two problems to be solved in either case. First, a mechanism is needed to do the actual counting. Once we have this counting mechanism, we need some consistent rule for charging a particular task for consumption of the resource.

In CPU accounting, the first problem is rather easy to solve. Since many language-based systems are multi-threaded and perform their own thread scheduling, a CPU ac-
counting system could simply count how much CPU time is scheduled for each thread. In
the event the language-based system implements threads using the underlying operating
system threads, the language-based system could query the operating system to find how
much CPU time was consumed by each thread.

Choosing which task to charge for a thread’s computation time is a somewhat harder
problem. Each thread must keep track of the task whose code it is executing at every instant.
In addition, if the thread is executing system code, the accounting system must decided
whether the computation is being done on behalf of a particular task, and if so, which task.
Finally, some scheme must be devised to prevent an attacking task from making a large
number of calls into a victim task, causing the victim to get charged for a large amount of
CPU time.

With network bandwidth accounting, actually figuring out how much network band-
width is used by a particular call is not easy. This is because there are likely a number
of system API functions that involve network traffic. In addition, remote procedure calls
frequently involve network communications. Finally, depending on how files are stored,
filesystem calls might require network transfers.

By and large, however, system code doesn’t need to make network calls, so all network
traffic can be charged to the task making the connection. The one exception to this is remote
procedure calls. If a remote procedure call involves a transfer of control from one task to
another, the accounting system should support policies that charge either or both tasks for
the network traffic involved.

8.1.2 Real-World Implementation Issues

This thesis considers the language-based protection mechanisms independently. In a real-
world implementation, the mechanisms would have to operate concurrently. This is not
simply a question of implementation, however. The designs would have to be studied to
ensure that there are no problematic interactions among the pieces.

One particular issue that would need to be evaluated has to do with running the soft
boundaries static analysis tools. Since the soft termination system rewrites the code that
is executed, should the static analysis be applied before or after the code is rewritten?
The arguments for running analysis after the transformation is that this is what is actually executed on the system. The arguments for running the analysis first are that the soft termination transformation might introduce false positives into the analysis results. It would also have to be shown that the soft termination transformation preserves soft boundary semantics.

8.1.3 Module System Integration

A related implementation issue is the context in which this work is deployed. Current research is being done in the language theory community on building high-level language-friendly module systems [56]. Such systems are primarily intended to simplify the development process. They allow for the development of independent, reusable software components (modules). Each component can specify abstract dependencies. The components can be combined, with a dynamic linker automatically resolving dependencies.

Our work on language-based protections could be integrated into such a module system. The exported interfaces and dependencies could be tied in with the soft boundaries checker at development time. Soft termination could be built on top of this, allowing for the safe termination of modules. Garbage collector memory accounting could be used to account for memory usage on a per-module basis. Because both works deal with the interfaces between modules, it is a very natural fit.

8.2 Analysis of Soft Boundaries

The static analyses performed by soft boundaries provide good heuristics for measuring whether the boundaries are maintained. Developing an theoretical model would go a long way towards proving some soundness properties of our static analysis, and may even help to minimize the number of false positives generated. Two approaches we might take include analyzing the operational semantics of soft boundaries and modelling soft boundaries in terms of type theory.

An operation semantics modelling of soft boundaries would proceed much like the analysis we performed for soft termination in Chapter 3. Some language would be defined
with a fairly simple operational semantics and the desired features. We would then define
the analyses performed by soft boundaries in terms of this operational semantics. At this
point, we could prove properties about the language as relates to our soft boundaries design.
Ideally, we would prove that soft boundaries correctly enforce data isolation and sharing
limitations.

Another class of analyses we might apply to soft boundaries is using type theory. Such
analysis has been proposed for numerous security problems, including information flow,
pointer confinement, and data race prevention. A comparative study between these systems
and soft boundaries would be a first step. We could further consider whether soft boundaries
could be expressed as a type system.

8.3 Techniques of Implementation

Another possible avenue for extending this work lies in the techniques we employed. We
employed three different techniques to add our protection semantics, as discussed in Chap-
ter 7. In all three cases, there were places where improvements to the techniques would
lead to an improved system.

The first technique employed was code-to-code transformations, as exemplified by soft
termination and transactional rollback, discussed in Chapters 3 and 5, respectively. Both
systems are somewhat limited by the capabilities of the underlying language; it would
be worthwhile to study how modifications to the underlying language might improve the
systems. This might also lead to improvements in the security semantics of the systems as
well as improved performance.

The second technique is language run-time system modification, as exemplified in the
garbage collector memory accounting system discussed in Chapter 4. Although we de-
signed the system with arbitrary garbage collection in mind, some modifications might be
needed to support future memory management schemes. A useful extension to this work
might involve designing an appropriate language-level interface to the garbage collection
system. Of course, it would be important that this interface allow for any sort of memory
management.
The final technique is static analysis, as exemplified in the soft boundaries work discussed in Chapter 6. In particular, this work relies heavily on the results of pointer-aliasing analysis. A further exploration of pointer-aliasing analysis techniques as applied to this work would therefore be in order. Similarly, it may be interesting to explore whether this work necessarily depends on pointer-aliasing analysis, and maybe some other analysis techniques could give the same results.
Chapter 9

Conclusions

Language-based systems offer several advantages over operating systems running traditionally compiled programs. Like operating systems, these language-based systems routinely host potentially buggy or malicious code from external sources that are not necessarily trusted. Operating systems have mechanisms that let us safely run such programs without risking the well-being of the overall system. Java and other general-purpose language-based systems have good support for memory protection, authorization, and access controls. However, a poorly-designed multi-task system can still be vulnerable to security-sensitive bugs. What's more, even in a well-designed system, malicious code can still commit a denial-of-service attack through misuse of system resources.

In this thesis, I proposed a language-based model for providing security and separation semantics for language-based multi-task systems. Static analysis and type safety as well as other language-based mechanisms are used to provide semantics for components similar to the semantics provided by an operating system to its processes. Since inter-task communication is increasingly becoming a necessary part of multi-task systems, we allow inter-task communication with no overhead. We have developed a suite of tools based on this model to implement language-based security.

We have developed a mechanism to help solve the problem of task termination. This mechanism, soft termination, allows us to safely and asynchronously terminate tasks. Unlike many similar systems, soft termination places no limits on state sharing among tasks. We present a formal proof of the effectiveness and safety of soft termination. We also discuss a Java implementation based on bytecode rewriting. In real-world benchmarks, our system shows a slowdown of 3 to 25%. We discussed soft termination in Chapter 3.

We have also developed a language-based mechanism for memory accounting. When multiple tasks are sharing a block of memory, a memory accounting system needs to make
the determination of which task to charge for the memory. In our mechanism, garbage
collector-based memory accounting, we choose to charge a block of memory to all tasks
which are keeping the memory alive. We use the garbage collector, which all language-
based systems have, to determine which blocks are alive, and which tasks are keeping each
block alive. Since the system is a simple addition to the garbage collector, it adds almost
no overhead to garbage collection, and in some cases improved the overall performance of
benchmarks. Garbage collector-based memory accounting was described in Chapter 4.

In the presence of state sharing, the ability to roll the system back to a safe state and
to restart programs safely can be as important as the ability to terminate a task in the first
place. Transactional rollback enables us safely to restart terminated tasks. We discuss
the design of transactional rollback as well as a Java implementation based on bytecode
rewriting. Our implementation shows a worst-case overhead of $23 \times$, with overheads of
6 to $7 \times$ in the absence of extensive array usage. We discussed transactional rollback in
Chapter 5.

Transactional rollback has high overhead and restrictive semantics that would limit its
general applicability. As a result, we also introduced an alternate approach to separating
tasks, which we called soft boundaries. Soft boundaries have far more flexible seman-
tics, allowing some data sharing and inter-task communication while providing a means
to specify restrictions. We used static analysis to achieve data-sharing the sorts of flexible
inter-task sharing semantics we desired. Soft boundaries were described in Chapter 6.

We used a variety of techniques to implement these systems. Soft termination and trans-
actional rollback were implemented in Java using bytecode rewriting. Garbage collector-
based memory accounting was implemented as modifications to the Java virtual machine.
Finally, soft boundaries were implemented through static analysis of the code. All of these
techniques are capable of implementing similar sets of semantics, with varying degrees of
efficacy. In Chapter 7, I discussed implementation issues surrounding these techniques, as
well as their benefits and drawbacks.

These have given us a language-based framework for ensuring separation and safety
among mutually distrustful tasks in a language-based system. We can effectively termi-
nate tasks that are misbehaving and safely restart them. We can track memory usage of
individual tasks, even when those tasks share memory, with soft termination being used to enforce limits on the amount of memory these tasks use. Finally, we can specify limits on inter-task communications and data sharing, with static analysis to tell us if these limits are being followed. This provides the same degree of protection for tasks that operating systems provide to processes, using language-based techniques.
Bibliography


[64] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.


