Anomaly Detection in Dynamic Execution Environments

by

Hajime Inoue

B.S., University of Michigan, 1997

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2005

©2005，Hajime Inoue

Dedication

To the memory of my brother, Kenji.

Anomaly Detection in Dynamic Execution Environments

by

Hajime Inoue

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2005

Anomaly Detection in Dynamic Execution Environments

by

Hajime Inoue

B.S., University of Michigan, 1997

Ph.D., Computer Science, University of New Mexico, 2005

Abstract

In the past few years, languages which run on virtual machines, like Java and C#, have become popular. These are platforms as well as languages, and they are characterized by being verifiable and garbage collected, and include Just-In-Time compilers, large standard libraries, and runtime profilers. I call platforms with these features dynamic execution environments (DEEs).

The runtime infrastructure of DEEs allows access to features of execution that were previously difficult to observe. My research consists of a series of case studies in which I build systems to classify behavior of a particular feature into normal and abnormal and then use that classification for either security or optimization purposes. These systems are anomaly detectors.

I build anomaly detection systems for method invocations, permissions, and method invocation sequences. I call them *dynamic sandboxes*, and they are they are used to detect

intrusions or system faults. I also show that an anomaly detector can be used to predict object lifetimes resulting in an improved garbage collector.

Contents

List of Figures xiii

List of Tables xviii

1 Introduction 1

1.1 Program Behavior is Regular . . . 2

1.2 Dynamic Execution Environments . . . 4

1.3 Observable Features . . . 8

1.4 Overview . . . 11

2 Background 13

2.1 Related Work . . . 14

2.1.1 Fault Tolerance . . . 15

2.1.2 Intrusion Detection . . . 16

2.1.3 Anomaly Detection . . . 18

2.1.4 Sandboxing . . . 19

2.1.5 Policy . . . 21
2.1.6 Java Security . . . 23
2.2 Data: JVMs, Benchmarks and Exploits . . . 27

3 The Simplest Feature: Method Invocation 30

3.1 Motivating Application: an Anomaly IDS by Method Invocation Observation 31
3.2 Dynamic Sandboxing . . . 34
3.3 Experimental Results . . . 38
3.3.1 Effectiveness . . . 39
3.3.2 Efficiency . . . 40
3.3.3 False Positives . . . 42
3.4 Discussion . . . 44
3.5 Summary . . . 46

4 Using Permissions to Infer Standard Security Policy 47

4.1 Motivating Application: Anomaly Intrusion Detection . . . 48
4.2 The Java Security Infrastructure . . . 49
4.3 Policy Inference Implementation . . . 51
4.4 Experiments . . . 53
4.4.1 False Positives and Generalization . . . 54
4.4.2 Performance . . . 57

4.4.3 Comparison to the Chapter 3 Dynamic Sandbox 59
4.5 Future Extensions 60
4.6 Summary 61

5 Object Lifetime Prediction 62

5.1 Motivating Application: Better Garbage Collection 63
5.2 Object Lifetime Prediction 64
5.3 Self Prediction 73
5.3.1 Fully Precise Self Prediction 73
5.3.2 Logarithmic Granularity 75
5.3.3 Variations 75
5.4 True Prediction 76
5.5 Zero-Lifetime Objects 77
5.6 Prediction and Object Types 78
5.7 Exploiting Predictability: Towards an Ideal Collector 79
5.7.1 A Limit Study 79
5.8 Related Work 84
5.9 Discussion and Conclusions 87

6 Methods, Method Sequences, and other Variants 92

6.1 Motivating Application: Fault Detection 93

6.2 Benchmark Behavior 94
6.3 The Metron UAV simulation 97
6.4 Scenarios 98
6.5 Dynamic Sandboxing 101
6.6 Implementation 102
6.7 Experiments and Discussion 104
6.7.1 Sandboxing with SSP=1 105
6.7.2 Sandboxing with SSP=2 109
6.7.3 Per-Thread Sandboxing 113
6.8 Intrusion Detection 115
6.9 Possible Extensions and Conclusions 116

7 Other Features and a Complete System Design 118

7.1 Other Features 118
7.1.1 Types 119
7.1.2 Method Arguments 121
7.1.3 Method Frequency 122
7.1.4 Per-thread Custom Sandboxes 126
7.1.5 Feature Fusion 127
7.2 The Complete System 128
7.2.1 Stack Introspection 128

7.2.2 Windowing 129

7.2.3 Response 130

7.2.4 Memory Prediction 132

7.3 Summary 137

8 Conclusion **140**

8.1 Contributions 140

8.2 DEEs and Anomaly Detection 143

Appendices **145**

A Classification of Java Exploits from the CVE Dictionary **146**

References **160**

List of Figures

1.1 Consider the large rectangle as the space of all possible behavior for a program for some feature of execution. My thesis is that the empirical behavior of the program is much less than the possible behavior and that behavior is predictable. 3

1.2 Organization of a dynamic execution environment. 7

2.1 Intrusion detection systems and anomaly detection systems are specific types of fault detection systems. 15

2.2 The execution stack of an applet as it tries to execute a privileged operation. Because the system domain includes all Permissions, the `checkPermission` call will succeed if the applet's domain includes the permission corresponding to the privileged operation. 24

2.3 The Java standard libraries are growing with each release. Each resource within the library must be properly protected or a malicious program may exploit it. 26

3.1 Method invocation frequency in HelloWorld and LimeWire. 32

3.2 Interface to the JIT compiler. The first if statement allows compilation of the method or forces a profile check. The second if statement adds the method to the profile. 37

4.1 A portion of the standard Java policy file. The file is actually 48 lines long including comments. It grants Java extensions privilege to do anything. It grants a smaller set of Permissions to all other code. The comments (not shown) advise that allowing `stopThread` is inherently dangerous. 50

4.2 Pseudocode for the `checkPermission()` method of `SecurityManager`. If the check call is initiated within this method, return. Otherwise, if training is activated, then add the policy to each protection domain, rewrite the policy file, and refresh the policy. Then check the Permission. 52

4.3 The performance of the SPEC JVM98 and DaCapo benchmarks under no SecurityManager (No Security), while generating a policy (Training), and running with that policy (Testing). 57

5.1 A single predictor entry: The SSP describes the execution path of the program. Each integer encodes the method and position within the method of the next method call. The entire string denotes the allocation site. All byte arrays (JVM type [B) allocated with this stack string prefix had a lifetime of 64 bytes. 68

5.2 The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (including singletons). 69

5.3 The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (excluding singletons). 70

5.4 The effect of stack prefix length on predictor size and coverage for the Java Olden benchmark *perimeter* (excluding singletons). 70

5.5 Death-Ordered Collector: The graph shows the fractional object volume of the different heaps in the simulated benchmarks. ZLS is the Zero-Lifetime Space. SS is the Semispace heap. KLS is the Known-Lifetimes Space. 81

6.1 A fault tolerant system relying on anomaly detection. The anomaly detection system observes behavior and flags anomalies. Anomalous behavior is then analyzed and a signature for a specific fault is determined. That signature is used to flag future anomalies as known faults which incorporate specific responses. The system described here incorporates only the shaded box: the anomaly detector. 93

6.2 How the SSP prefix works. Stack frames are pushed on the top when a method is invoked. They are popped off when they exit. The stack string prefix of length 2 is the signature of the top 2 frames concatenated together: Method3-Method2. 95

6.3 A screenshot of the Metron UAV simulation with the graphical interface. The surveillance sim is on the left and the search sim is on the right. . . 98

6.4 Reducing false positives. An anomaly is triggered only if two events are correlated in time. In this example, an anomaly is defined as 2 events that occur within a 3 method invocation span of time. 101

6.5 Filtering method invocations. No method invocations are logged until a method from the trigger list is invoked (Uav.calculateBestPath in this example). Config.getConfig() is on this ignore list and is ignored. The other methods are recorded because they were invoked within a method on the trigger list and are not on the ignore list. Thus the stack visible to the analysis engine is, from top to bottom: *ValueMap.isValid(), MovementDecider.determineBestPath(), Uav.CalculateBestPath()*. All methods invoked above SearchSim.newTimePeriod will continue to be recorded until the stack falls below that level. This example is not taken from the simulation code. 103

6.6 Growth in profile size over repeated training. The profiles grow very slowly. Sandbox size, however, is only loosely related to the number of events produced during a simulation using the sandbox. 111

7.1 Dynamic sandboxing using types. The sandbox interposes itself between the mutator (the application) and the allocator. In normal operation, the allocator is given a type and returns a reference to a region of memory to place an instance of that type. A sandbox, implemented as an allocator proxy, would allow only allocation of types allowed by its profile. 119

7.2 Unified dynamic sandbox. A central repository analyzes behavior from interface proxies. Anomalous behavior is analyzed and then the proper response is initiated through the appropriate proxy. 130

7.3 The 3 SSPs that need to be considered for an allocation at execution point 10 with their associated predictions. Although the SSP length is 6, the allocator need consider at most 2 positions for linear comparisons, and only 1 for random access. 133

7.4 Linear allocator generated for the execution point 10 shown in Figure 7.3. The allocator needs to consider only two positions of the SSP to uniquely distinguish them. 135

7.5 A visual representation of the death-ordered collector. The ovals within the top and bottom rectangles represent objects within the normal heap and the known lifetime space (KLS), respectively. Objects in the KLS are arranged in order of predicted death. Arrows indicate references from one object to another. Arrows that overlap the remembered sets rectangle must be considered as roots when a collection occurs (though the sets do not have to be unified in any particular implementation). The dotted line indicates the current time. A DOC starts a collection with objects at the far left, the ones with the earliest predicted time-of-death, up to the current time. Remembered sets are necessary because objects not considered for collection, those in the normal heap and those not predicted to be dead, are assumed to be alive and thus any objects pointed to by them must also be considered alive. 138

7.6 Differences between threads in the Metron UAV simulation. Each row corresponds to a thread and each column corresponds to a method. Pixel (i, j) is white if method i is invoked at least once during the run of thread j. There are approximately 250 threads and 2600 methods shown. 139

List of Tables

2.1 Description of the programs I used in my research. They consist of three sets: the Java Olden benchmarks, used primarily in garbage collection research, the SPEC JVM98 and JBB benchmarks, and the DaCapo benchmarks. I developed the *HelloWorld*s and *HttpTrojan* myself. LimeWire [72] was acquired from LimeWire LLC and the UAV simulation was acquired from the Metron Corporation. 29

3.1 The total number of classes and methods within the standard and associated helper libraries bundled with various versions of Sun's standard Java Runtime Environment. 33

3.2 The number of different application and library methods invoked during standard runs of the Jolden benchmark on JDK version 1.4.2. The benchmarks each consist of 2 to 10 classes. 34

3.3 Efficiency of sandbox generation and protection on the Olden benchmarks. 41

3.4 Efficiency of sandbox generation and protection on the synthetic benchmark. 41

5.1 Trace statistics. For each trace, the number of objects allocated (Column 3) and the total size of all allocated objects (Column 4) are given. Column 5 shows the number of allocation contexts; each site is counted only once, even if executed more than once, and sites that are not executed in these particular runs are not counted. The top section of the table lists the traces used for the self prediction study (Section 5.3). The bottom part of the table lists the training traces used in the true prediction study (Section 5.4); traces from the top section are reused for testing true prediction. 72

5.2 True prediction. Coverage and accuracy using predictors generated from a benchmark run using a smaller set of input for both fully-precise and logarithmic granularities against a separate, larger benchmark run. Coverage is the percentage of objects for which the system makes predictions, and accuracy is percentage of those objects for which my predicted lifetime was correct. 77

5.3 Fully precise zero lifetime self prediction: Column one lists the benchmark program; column 2 shows the fraction of zero-lifetime objects out of all dynamically allocated objects for that benchmark; column 3 shows the percentage of zero-lifetime objects predicted (coverage); and column 4 shows the prediction accuracy. SSP lengths are as described in Table 5.6. 78

5.4 The ratio of bytes copied in the DOC system to the bytes copied in the semispace collector for heap sizes of 1.1, 2, and 4 times the minimum semispace heap size required by the DOC system. Smaller numbers are preferable. 83

5.5 The mark/cons ratios for various heap sizes of the DOC and semispace collector. 83

5.6 Self prediction results. The first two columns of fully precise and logarithmic granularity give results using predictors including singletons using an SSP of length 20, with two exceptions: *jess* and *javac*, for which I used the larger SSP value reported in the 5th column. 89

5.7 Self prediction for three categories of objects according to object type. For each of the three categories of types (virtual machine, library, application), the percentage of total allocated objects that fall in the category is given, together with the percentage of objects in the category that are predicted. The rightmost column is the overall percentage of objects predicted (corresponding to the first column of Table 5.6). 90

5.8 The bytes allocated to the different heaps and their maximum sizes, and with μ, ε, and factor of improvement based on a 50MB heap. 91

6.1 The number of unique stack string prefixes (SSPs) for various prefix lengths for the SPEC JVM98 benchmarks. An SSP length of 1 denotes methods without context, as in Chapter 3. 96

6.2 Data from SSP=1 (non-stack) based detection of anomalies using a window and threshold of 1. 106

6.3 Data from SSP=1 (non-stack) based detection of anomalies using a window and threshold of 2. 107

6.4 Data from SSP=2 detection of anomalies using a window and threshold of 1. 108

6.5 Data from SSP=2 based detection of anomalies using a window and threshold of 2. 109

6.6 Sensitivity of variants using combined training data. A checkmark indicates that the system reliably identified the scenario as anomalous. . . . 110

6.7 Per-thread sandboxing for a window and threshold of 1. Events reflect the average number of events in the maximum thread of each scenario. . 114

6.8 Per-thread sandboxing for a window and threshold of 2. Events reflect the average number of events in the maximum thread of each scenario. . 115

7.1 Number of application and library classes used during a run of the SPEC JVM98 benchmarks. 120

7.2 Average frequencies for α (method invocations that do not appear in the sandbox profile) and their ratio over nop. 123

7.3 Average frequencies for β (methods that do appear in the sandbox profile but not in the trace) and their ratio over nop. 124

7.4 Presents the ratio of each scenario with nop and the specified *spread* (1, 2, 4, 8 , 16) with a γ of 1. 125

7.5 The average number of hash table queries that are required to identify an allocation context for a given ssp and benchmark. The two sets of data shown are for the SSP lengths used in Table 5.6. For each set, I calculate the average depth using hash tables constructed using the SSP with linear access and with random access. 136

Chapter 1

Introduction

The last decade has seen a dramatic shift from natively compiled languages to those hosted on virtual machines. The popularity of Java and C# is testament to such a shift. The growing use of so called "scripting" languages like PERL, Ruby, and Python is further evidence. All of these languages run or will run on virtual machines [56, 77, 104].

Virtual machines (VMs) are simulators for abstract computer architectures. They provide different characteristics and capabilities than the real machine that they run upon. The runtime infrastructure required for VM-hosted applications is much greater than that for applications running natively. A VM can examine and manipulate the behavior of applications in ways that operating systems cannot. The differences in architecture that lead to the greater ease of observation are why I call these new VMs *dynamic execution environments*.

In previous research, members of my group have examined the behavior of the network or operating system with respect to anomalous behavior [37, 111, 50]. Developers working with DEEs have a much greater ability to examine behavior, but until now have used it only for a limited amount of performance optimization. This dissertation expands that research by examining the behavior of several features of DEEs with the goal of exploiting them in novel applications.

1.1 Program Behavior is Regular

> *The behavior of applications in DEEs is regular, and that regularity can be exploited.*

Regarded generally, this thesis is not controversial. Indeed, it has been observed that applications behave predictably time and again. Modern hardware is designed with many assumptions about the regularity of program behavior. Modern architectures rely on cache and branch prediction to optimize performance. Caches exploit the fact that accesses to memory are often local in terms of time and space. Likewise, branch prediction takes advantage of the fact that programs follow the same execution paths again and again. The field of software engineering is less interested in examining the regularities of program execution, although there are some examples. There is a rule of thumb that 90% of execution is spent in 10% of code [65]. Caches are also used by the operating system to great effect.

However, most potential regularities within software have not been exploited for two reasons: the features have no standard representation, and they are unobservable without performance penalties. Frequently, the only method of observing program behavior has been through the use of debuggers, and even then, gaining significant knowledge of program internals is often impossible in a mechanical way. The knowledge of the developer is usually required.[1]

My hypothesis is that for the many features of execution that are observable (and standardized) within DEEs, regularities in behavior can be discerned, mapped, and used to predict and improve future behavior. Figure 1.1 graphically depicts this. Although the possible behavior of most features is vast, the observed behavior is much smaller and that behavior is predictable. If that is true, it could be used in several applications. The systems implementing the applications, described in following chapters, are called anomaly

[1] The one exception is the automatic exploitation of the 90-10 rule by compilers. Compilers can examine execution profiles to determine better optimization strategies.

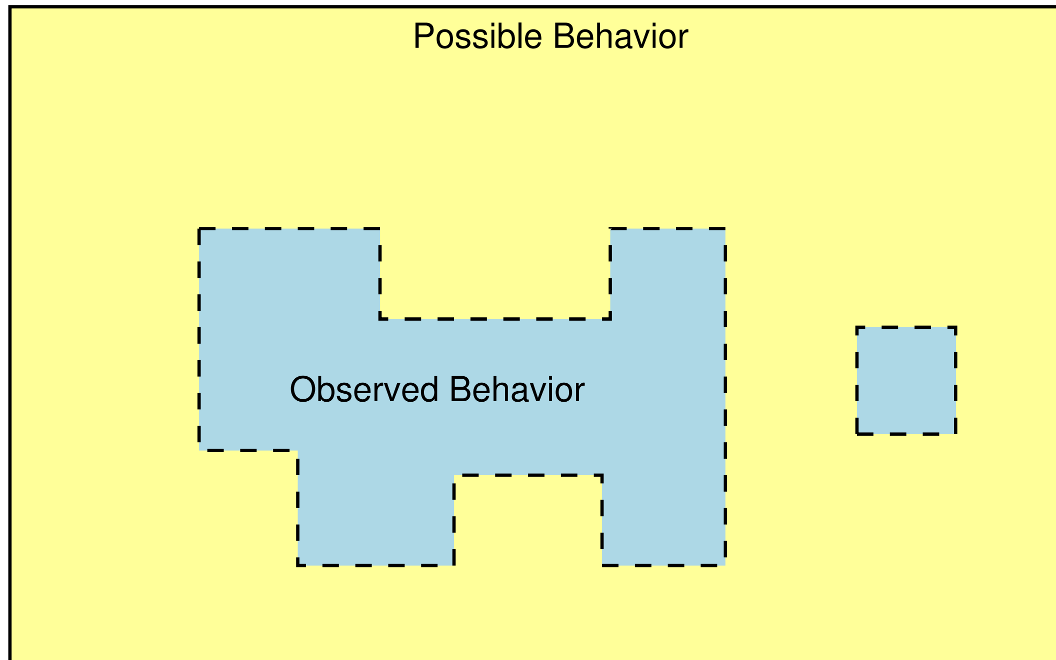

Figure 1.1: Consider the large rectangle as the space of all possible behavior for a program for some feature of execution. My thesis is that the empirical behavior of the program is much less than the possible behavior and that behavior is predictable.

detection systems (described in more detail in Chapter 2), although not all are security related.

I concentrate on both security and optimization. Using profiling information is an old idea, and it is the reason that features are observable within DEEs. My optimization scheme is a novel one, however, and the idea of leveraging optimization information for security is also new. Security demands that detection and response be realtime and online. Likewise, optimization must balance the effort to analyze the behavior with the gain in performance. The representations and calculations I use to analyze the behavior of these features must therefore be efficient.

Where the systems add to the standard security features of DEEs, I call them *dynamic sandboxes*. Unlike standard sandboxes which rely on static descriptions of policy, these

anomaly detection systems learn policy through observation.

Obviously a motivation for my work is to build better systems. There is another one, however. Most studies of program behavior are driven with specific applications in mind. I believe that program behavior is interesting even when it does not directly lead to improvements. Computer systems have become so complex that we do not understand them. This complexity results in erratic, unpredictable behavior. Dynamic execution environments add to this complexity. Instead of an application running on top of an operating system, an application interacts with a complex simulator which itself interacts with the OS.

Program behavior must be studied empirically. The vast possible space of behavior coupled with theoretical barriers like the halting problem make most static inference of behavior futile. DEEs increase complexity, but they also allow us to observe and study program behavior. This characteristic, coupled with my belief in the eventual dominance of these systems, motivates my research.

In the next section, I describe the characteristics of dynamic execution environments. Then, I list the features of execution that a system might monitor and describe my reasons for choosing or excluding them. Finally, I present an overview of the remaining chapters of this dissertation.

1.2 Dynamic Execution Environments

Beyond the popularity of Java and C#, the convergence of several trends is resulting in the ubiquity of dynamic execution envrionments: security, dynamicism, and portability. Security is now grudgingly recognized as an important feature of systems. The sandboxing technology built into Java and C# is a primary reason for their popularity. Another example supporting the growing importance of security is the rise of virtual machines like Microsoft's Virtual PC [52] or those made by VMWare [107]. They sandbox applications

by simulating multiple VMs on each true machine. Each application is convinced it is the only one running. The dynamic character of these systems is also a feature. Applications written for Java's Enterprise Edition platform can be configured without shutting down the VM, and can often be upgraded as well [75]. Portability is another factor. Java, C#, and the scripting languages will run identically on any native platform that supplies a VM. Similarly, VMWare "images" of machines can be moved or duplicated between them. These features are so compelling that it is clear that DEEs will continue in their popularity.[2]

All virtual machines are not DEEs. Although Virtual PC and VMWare's VMs provide many of the benefits of DEEs, they do not provide all of them.[3] Those VMs emulate traditional architectures, and therefore do not include many of the advantages provided by true DEEs. Those advantages are conveyed by five characteristics of dynamic execution environments:

Typed Instruction Set The reasons these environments use typed instruction sets are described in the Section on Java security 2.1.6. These are not specific to Java. In other contexts typed instruction sets are referred to as "typed assembly languages", or TALs [78]. The same group has noted that typed instruction sets are a type of proof carrying code. In short, typed instructions leads to a straightforward verification mechanism, ensuring that instructions execute with fidelity to the standard.

Just-in-time (JIT) Compilation Typed instruction sets are rarely the native instruction sets of the new platform. These platforms could be implemented as interpreters, but are now usually compiled to native code. The reason for this is performance—interpreters effectively translate DEE instructions to native instructions every time a given piece of code is executed. With compilation, code is translated once, and then runs at native speed on every subsequent execution. Additionally, applications are not compiled in their entirety

[2]Language popularity is often difficult to gauge. Studies of google searches [112] and book sales [81] show the obvious strength of Java, C#, and PERL.

[3]Virtual PC and VMWare are different in that VMWare simulates devices but not the processor while Virtual PC emulates everything. For my purposes this distinction does not matter.

when the application is loaded. Rather, code is compiled as needed; that is, "just-in-time". This has two benefits. First, initial responsiveness is good, since usually only a small portion of the application is needed to start. Second, many applications execute only a small portion of the entirety of their codebase during typical execution. JIT compilation can thus reduce pause times and the total amount of required computation.

Garbage Collection (GC) Like the use of typed-instruction-sets, garbage collection is used to eliminate many of the programming errors and security vulnerabilities found in code written for non-DEE platforms. These include familiar errors like memory leaks or segmentation faults that are indicative of vulnerabilities.

Garbage collection is a technique that allows for the implicit deallocation of memory. These algorithms do not compute when allocated memory is no longer going to be used. That is, in general, an uncomputable problem. Instead, they try to determine when memory becomes unreachable from a set of roots. These roots are typically the execution stack and a set of global variables. Because DEE platforms do not allow pointer manipulation, a portion of memory that is unreachable cannot be used again and is thus safe to be reused. Jones's *Garbage Collection* textbook is a complete reference on the subject [60].

Standard Library DEEs also have large standard libraries. Large libraries are almost a prerequisite to popularity in execution platforms, so their presence is not unique to DEEs. DEEs are slighty different from other environments in that the library is not an add-on. The library is a required part of the platform standard. These libraries always increase in size as the standard matures; this is clear in Java's case, as Figure 2.3 demonstrates. Much of an application's behavior derives from the content of these libraries, including security vulnerabilties and performance characteristics.

Runtime Profiling Finally, DEEs monitor the execution of their applications. This is not unique to DEEs, most compilers allow feedback-directed optimizations when combined

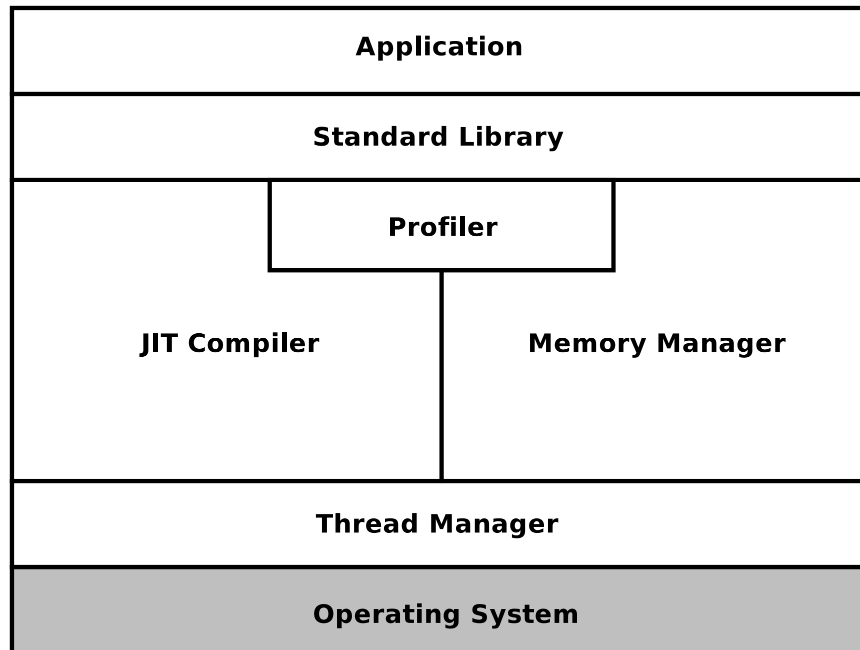

Figure 1.2: Organization of a dynamic execution environment.

with profilers [94].[4] Unlike traditional profilers, which slow applications while running, DEEs profile so that the JIT compiler and garbage collector can improve performance while the application is running.

These characteristics standardize and make observable many aspects of program execution. Object allocation and layout, function calling conventions, and access to resources are identical throughout every application that runs on the DEE, and that data is available in the DEE's internal structures as it runs.

Figure 1.2 shows DEE organization. The application interacts with standard libraries. Its code is compiled by the JIT compiler, and memory is allocated and freed by the allocator and garbage collector. If it is multithreaded, another manager switches and schedules those. Ordinarily, an application never interacts directly with the OS. The DEE provides

[4]The GNU Compiler Collection (GCC) is the notable exception.

those services.

Java's popularity makes it the best example of a DEE, but it is not the only one. Older languages like Lisp or Smalltalk qualify. Newer platforms like Microsoft's .NET and Perl6 with its Parrot VM are also dynamic execution environments. Microsoft is using .NET for major systems, such as the file system and the graphical user interface in the next iteration of Windows (Longhorn).

I used Java for every system described herein. For most of time during which I conducted my research, Java was the only popular DEE. Sun released a standard version which allowed me to conduct experiments. There have also been several open source versions available, allowing me to investigate and modify their internal structure.

These JVMs allowed access to several features during execution. In the next section I list several potential features to study and explain for each the reasons I chose to examine or ignore them.

1.3 Observable Features

Applications that run directly on top of the operating system are mostly opaque to the OS. Without a debugger, a process is a collection of memory and some data about the devices it is using. One can organize the memory into code and data, but it is difficult to determine any greater structure. Applications control their own internal environment. There are only two standards: the instruction set and the system call interface.[5] System calls are the mechanism that allows applications to interact with the OS.

DEEs, because they provide more services to applications, have standardized many more features than operating systems. The various features that can be observed in a

[5] It is primarily for this reason that host-based intrusion detection systems like Somayaji's pH analyze system calls.[97].

DEE can be divided into two main types: execution and memory. Desirable features have two characteristics: they can be observed and analyzed in realtime, and they should give meaningful information about the state of the application. Thus, although regularities are desirable, uniform regularity in different programs is not.

The following list describes several of the features I considered. Features that I developed into anomaly detection systems are listed in bold.

Execution

Instructions: instructions are the fundamental unit of execution. In Java, however, instructions are difficult to monitor individually. Furthermore, it is not clear that they can be analyzed to provide meaningful regularities.

Basic Blocks: Basic blocks are lists of instructions that have one entrance and one exit. If one thinks of instructions as words, basic blocks are sentences. Like instructions, though, basic blocks are not a convenient unit to study. The interface to the JIT compiler and the rest of the JVM is built to manipulate methods, not basic blocks.

Methods: Methods are perhaps the most analogous unit to system calls in DEEs. Because the number of callable methods is much larger than the number of system calls, the first system I describe observes methods rather than method sequences. Methods are observable through a debugging interface, through code instrumentation, or through the JIT interface.

Method Arguments: The large number of possible arguments to methods increases the possible space to an extraordinary degree. I did not discover a generalization that would enable me to accomodate such a space, nor did I find an efficient algorithm to analyze it. I speculate briefly about a possible approach in Section 7.1.2.

Method Frequencies: Method frequencies are used extensively for optimization in DEEs

and are observed in the same way as individual methods. I therefore tried to build a system that could leverage this information for security purposes. However, this system was not stable enough to work reliably, as I recount in Section 7.1.3.

Method Sequences: Given the success of pH, building an analogous system using method sequences was inevitable. I explore various permutations of a system observing method sequences.

Permissions: Permissions are used to protect resources in Java. I was successful in building a tool to automatically learn the permissions required to run an application and specify it as a standard security policy.

Program Paths: Larus showed that programs spend most of their times in what he called "hot paths" [69]. Traces of execution through a series of methods, hot paths can reveal a large amount about the global behavior of an application. Hot paths are useful for optimization, but given their similarity to method frequencies, seemed difficult to build an anomaly system around.

Memory

Types—Observing the types allocated during an application's run seems analogous to observing which methods are run, and indeed they are. Types are so analogous to methods that such a system was redundant. I briefly describe the tradeoffs between methods and types in Section 7.1.1.

Memory Management—The standard statistic of garbage collection, object lifetime, generates more irregularity than regularity. The system I designed, more a regularity than anomaly detection system, explores an optimization for those objects that do have predictable lifetimes.

Memory Structure—Objects are connected through references, forming a graph structure

of memory. This graph is available through the garbage collection system. I did not investigate this feature because I did not intuit an algorithm for discovering regularities, nor did I discover an efficient representation of normality.

A thorough and systematic investigation of each observable feature would require a team of researchers. Instead, I explored the behavior of the selected features of execution through a series of case studies. In each case study I build a system to analyze the regularities of the selected feature and identify anomalous from regular behavior. I am able to demonstrate the truth of my thesis statement for each selected feature, and then make a prediction as to how the behavior could be exploited in a practical system.

The different systems are motivated by two applications: security and memory optimization. I call the security systems *dynamic sandboxes*.

1.4 Overview

The chapters are arranged as follows:

- Chapter 2 describes the tools and related research that are common to multiple chapters. Where related work is specific to a single case study, it is included in that chapter.

- Chapter 3 examines the regularities in method invocation behavior and describes the behavior and performance of the simple dynamic sandboxing system to prevent unwanted intrusions of computer systems. This chapter is an expansion of a paper presented at the New Security Paradigms Workshop 2002 [53].

- Chapter 4 moves away from methods and examines Permissions. Permissions are used to protect resources under the standard Java security infrastructure. This chap-

ter includes work presented at the International Conference on Programming Languages and Compilers 2005 [54].

- Chapter 5 examines the predictability of object lifetimes. I present results of a prediction technique for several benchmarks and then analyze the performance of a possible implementation. The chapter is adapted from a paper submitted to IEEE's *Transactions on Computers* [55].

- Chapter 6 expands on Chapter 3 by introducing variants on the simple dynamic sandbox. These variants include correlating anomalies in time, as well as observing sequences of method invocations. This chapter is an expansion of a demonstration for the DARPA Taskable Agents Software Kits (TASK) project.

- Chapter 7 briefly recounts explorations of several features that did not evolve into systems, and then describes how a complete system. It integrates the systems used to explore the various features of the case studies, could be built.

- Chapter 8 summarizes the results and contributions of my work.

Chapter 2

Background

The applications of my research fall into two areas: security and garbage collection. The unifying concept is the goal of classifying regularity in program behavior by creating anomaly (or regularity) detectors. The design is therefore dependent on specific applications and its interaction with several subsystems of the platform on which is resides: the security infrastructure, compilers, memory manager, runtimes, and program behavior. For the most part, I only leverage the already existing behavior of those subsystems. They are described in Section 1.2.

Only one of the systems has improved performance as its goal. Its related work is included in that chapter. The rest have security applications and are called *dynamic sandboxes*. Their related work is described in the first section of this chapter. Although my work is meant to target all dynamic execution environments, my research is implemented using Java, and therefore I present an overview of security work with respect to Java.

All the systems must be tested by running applications on top of the modified or simulated DEE. A description of the tools, benchmarks, and exploits, forms the second section of this chapter.

2.1 Related Work

In the large, computer security aims to protect the resources of computing systems from unauthorized access. More specifically, according to Anderson's seminal work on computer security, secure systems prevent unauthorized information release, unauthorized modification of information, and unauthorized denial of services [4].

Somayaji divides the operation of these systems into three categories [97]:

- **Preventative**—Systems whose resources cannot be used without proper authorization. These are systems that simply "work" and are obviously the ideal. The sandboxing infrastructure of DEEs is an example.

- **Detection**—Systems which detect attempts at unauthorized use and prevent them. These two halves, detection and response, are often implemented separately. Research in this area concentrates on detection, since preventing unauthorized use is often an easy operation such as returning an error message in lieu of completing the operation.

- **Recovery**—These systems are similar to the second category except for their timing. The goal is to identify unauthorized use of resources after it has occured, and to recover from it.

Dynamic sandboxing is primarily an exploration of technologies of the second type. In conventional terms, dynamic sandboxing incorporates aspects of conventional sandboxing, fault detection, anomaly and intrusion detection. It is built on top of the standard sandboxing tools of the underlying DEE.

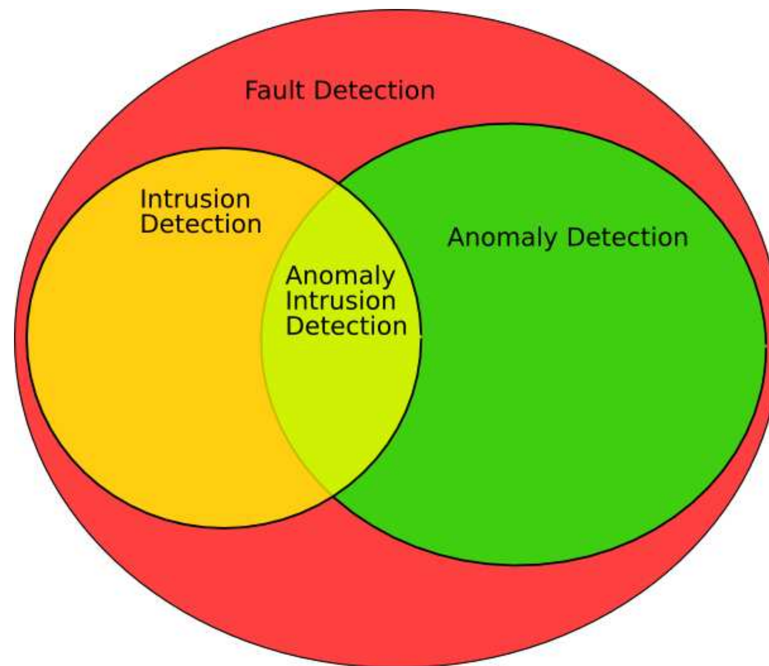

Figure 2.1: Intrusion detection systems and anomaly detection systems are specific types of fault detection systems.

2.1.1 Fault Tolerance

Fault tolerance is a feature of systems that use many of the mechanisms developed for anomaly and intrusion detection. Fault tolerant systems detect faults and then respond to correct them. Fault tolerant systems behave correctly, even when they are composed of imperfect hardware and software. Most of this field is concerned with fault tolerance in hardware, which is largely irrelevant to my work. Software fault tolerance, however, can be seen as a superset of much of my own work, as well as that of other intrusion and anomaly detection systems. Figure 2.1 depicts the relationships between the various detection systems.

Software fault tolerant systems can be divided into two categories: design diversity and fault detection and recovery. Systems based on design diversity replicate function

among several different implementations of the software. In *n*-version programming, *n* different implementations run concurrently, with the system executing whatever actions constitute a majority among the *n* systems [7]. Faults are observed when actions differ among implementations. No special recovery is needed as the systems continue to take the actions of the others.

Most fault tolerant systems, however, rely on special code to detect and respond to errors. *Exceptions* are a standard method of response for fault detection integrated into many high level languages and DEE runtimes. Exceptions are tests of program state. When that state corresponds to a specific error, an exception corresponding to that state is *thrown*. The runtime then searches for a block of code (the recovery block) corresponding to the specific exception. When that code is executed, the exception is said to have been *caught*. Exception handling is the standard recovery idiom for non-diversity fault tolerant systems. Furthermore, it is my proposed mechanism for recovery in dynamic sandboxing (see Section 7.2.3.)

2.1.2 Intrusion Detection

Much of my work seeks to detect intrusions, known as intrusion detection systems (IDSs). James Anderson introduced the notion of monitoring systems for intrusions in 1980 with his paper, "Computer Security Threat Modeling and Surveillance" [5]. Dorothy Denning described requirements for the implementation of intrusion detection systems in 1987 [30]. Since then, a large number of these systems have been introduced. These systems either look at individual systems (host-based), or network traffic. Axelsson [8] presents a more complete description. From a computer security viewpoint, dynamic sandboxing can be viewed as a host-based anomaly intrusion detection system.

Most host-based systems do not rely on anomaly detection. Instead, they are provided with databases of signatures that denote unacceptable behavior. These are similar to virus

detection monitors with the difference that they examine other execution features rather than binary files. Logins, program execution requests, and system calls are some of the features that they monitor. Examples are the commercial ISS RealSecure OS Sensor [57], and IEQ's SystemAnalyzer [80], and free tools like LogSentry [95].

A few intrusion detection systems build up behavior models based on static analysis. This is not anomaly detection, because the resulting model incorporates all possible behavior based on the application's code. Wagner and Dean's system is a good example [108]. Their system builds a model of all possible system call orderings and then insures that only those sequences are allowed, preventing naïve code injection attacks. However, many attacks, such as trojan horses, do not change possible behavior, merely exploit unknown possible ones.

There are few intrusion detection systems for Java. Virus scanners examine Java bytecode for known exploits. Perhaps the best example is the STAT IDS modified to instrument Java [96]. Java does not work well with these systems because they rely on observing distinct event streams from the applications they monitor. Java confuses these systems because most Java VMs rely on user threads instead of system threads. System threads are scheduled by the OS and therefore OS-based systems can easily distinguish between events from each thread. User-threaded systems schedule several internal threads on a smaller set of system threads. Thus the event order in each user-thread can confuse systems observing only system threads.

Systems that rely on databases of known behavior are known as signature-based. They have the advantage that they can monitor behavior as soon as they are deployed. Their disadvantage is that they cannot recognize unknown attacks—the signature database must be updated with every new attack. Furthermore, signatures developed within one particular laboratory can exhibit false positives when deployed in production environments. Because of these disadvantages, others have developed systems that learn normal behavior and detect intrusions and other anomalous behavior by measuring deviations from that normal.

These are anomaly detection systems.

2.1.3 Anomaly Detection

In anomaly detection, intrusions are simply anomalous behavior. Because dynamic sandboxes are anomaly detection systems, they are both fault detectors and intrusion detectors. Anomaly detectors build a profile of normal behavior by first observing the application (the training phase). Once a profile of normal behavior has been constructed, novel behavior is flagged as anomalous (the testing phase).

Dynamic sandboxing is not intrusion detection, however. It is *anomaly detection*, in which intrusions are simply anomalous behavior. Dynamic sandboxing is a logical step from the previous anomaly detection systems developed within my research group. These systems were developed using biological systems as an ideal: advanced vertebrates' immune systems solve many problems analogous to those found in computer systems: distinguishing intrusions from normal operations, remembering old instrusions, identifying new intrusions, and preventing and stopping intrusions.

There are a variety of host-based anomaly detection systems. An early system, Haystack, combined signatures with anomaly detection [93]. Recent commercial systems are modelled closely on the pH approach. Sana Security's Primary Response and Attack Shield use system call sequences to detect intrusions [51]. CylantSecure's method also involves in-kernel monitoring, but Cylant has not published their feature set.

Other anomaly detection systems look at more than just OS-based features. Several examine aspects of execution behavior not visible to the kernel. Almost all of these are signature-based, but their instrumentation techniques are mostly independent of the detection apparatus. There are two ways to accomplish this: modifiying source and automated instrumentation. SRI released an API to couple an application to a detection engine [2]. Kuperman's library interposition system automatically generates detectors by inserting its

own proxy for library calls that update the IDS state [66].

DIDUCE is a similar example for Java [43]. A debugging tool, DIDUCE tries to learn variable invariants during execution and then informs the developer when those invariants are violated. Beyond DIDUCE, there have been very few intrusion or anomaly detection systems aimed at Java applications. Java was designed to eliminate most of the common vulnerabilities and there have been relatively few Java exploits. Java does not have perfect security, however, as I discuss in Section 2.1.6.

Some applications of anomaly detection look beyond features of execution. Dawson Engler's Stanford group works on improving program robustness. His work on anomaly detection in source code has also spawned a company named Coverity. These systems learn conventions by studying the source code of large applications. For example, Engler's system can learn when locks are required to manipulate objects or the order in which locks must be acquired [35]. When a portion of source code varies from these conventions, it is flagged as a potential bug.

Previous systems developed by others in my group have endeavored to solve these problems. Somayaji's pH has particularly influenced my work [97]. pH builds an internal model of normal behavior for each process using system call sequences. Although pH is conceptually similar to my own research, it differs in the level at which it monitors behavior, and in its response. pH is implemented at the kernel level. It cannot monitor features available within applications. My own research is an exploration of features available at this higher level of execution. My own research does not focus on sophisticated response, although I discuss some approaches in Section 7.2.3.

2.1.4 Sandboxing

Sandboxing is a preventative technology. The concept of sandboxing was introduced by Wahbe and colleagues in the application of software fault isolation [110]. Wahbe described

a technique in which an application is modified so that it cannot execute or access portions of its address space. That is, it is confined to a “sandbox”. The Java platform is currently the best example of this kind of sandboxing.

In a more general sense, however, many techniques implement sandboxing. Traditional Unix permissions are a form of sandboxing. The kernel monitors and restricts resources like files, the network, and devices. The PERL tainting system is another example. It restricts code from accessing data (resources) that are untrusted by the interpreter.

Another sandboxing technique uses virtual machines. Instead of restricting the behavior of individual applications, VMs create separate platforms within a physical machine. The IBM mainframe OS is an early but good example. Their VM/370 allowed one mainframe to appear as tens or hundreds of separate machines [42]. Resources, such as execution time or disk, can be assigned separately to specific virtual machines. This is becoming more common on a smaller scale with commercial products for commodity machines like those from Microsoft, and VM Ware.

Recently there has been an increase in VM-based sandboxing systems, possibly due to Java’s popularity. Systems such as RISE [9], program shepherding [63], and Valgrind [90] all combine VMs and sandboxing technology. Each of the systems sandboxes individual processes, unlike the previous examples. RISE and program shepherding aim at preventing code injection. In RISE, code residing on the host’s disk can be executed. All other code becomes randomized, making successful code injection unlikely. In program shepherding, all branches are examined and monitored as the application runs. Branches are only allowed if they conform to a given policy; for example, jumps into the stack may be disallowed.

Valgrind is different than the previous two examples. It is a debugging and performance monitoring tool. It monitors execution similarly to Wahbe’s description, but does not restrict behavior by default. It can easily be modified, however, to provide that behav-

ior through the use of plugins.

These three systems take a similar approach to dynamic sandboxing. They use a virtual machine environment to monitor and alter the behavior of applications. There are two main differences. First, my approach determines legal behavior by training—DEEs are already sandboxed so dynamic sandboxing is used to implement the second (Detection) behavior on Somayaji's list. Second, the semantics of execution are much easier to discern in DEEs like Java then in the native code examined by systems like RISE and Valgrind.

2.1.5 Policy

Sandboxes cannot operate without policy. The sandbox's security policy is its shape (see Figure 1.1)—the resources that are "in" and "out" of the sandbox. Sometimes this policy is implicit in the design of the sandbox. Often, however, the sandbox can be configured.

One of the central ideas of software engineering is the separation of mechanisms from policy. In security systems, protection mechanisms implement a set of tools that can be used to specify a range of security policies, providing greater flexibility. Anomaly intrusion detection systems provide ways to monitor, if not restrict, access to system resources, and can therefore be viewed as policy inference algorithms. The security policy is the set of all non-anomalous actions.

The Principle of Least Privilege (or Least Authority) was first described by Saltzer and Schroeder in "The Protection of Information in Computer Systems" [85] and continues to be a prominent principle of computer security. It states that "programs should operate using the least set of privileges necessary to complete the job." This is useful because it limits the amount of damage that can occur due to malicious subversion or through simple bugs. The goal of training in anomaly detection systems is to automatically infer the policy of least privilege. In the analogy of Figure 1.1 again, one wants to define the smallest possible area.

There are other tools that learn policies. Configuring firewalls is in many ways analogous to configuring Java security policy and there are several tools available. A good example is a system by Burns [17]. There is at least one other tool, SPECTRE, for dynamically inferring security policies and specifying that policy in a standard language [87]. Its inference policies are specific to web services, however. Lam and Chiueh describe a system called *Paid* which limits application behavior to statically calculated system call sequences [24]. They refer to this as "Automatic Extraction" of sandboxing policy. Its approach is more similar to other host-based anomaly detection systems than to our approach. Their system protects a new resource, system call-order, and then infers a policy. It is very similar to pH with the difference that pH learns policy dynamically.

Although it doesn't learn policy, Naumovich's work on consistency in J2EE security policies [79] relates to my own research. His system statically analyzes the methods invoked by different roles (groups of users) to suggest inconsistencies in policy. J2EE policies are in a different policy language (XML) and are converted to Java policies automatically. His approach addresses a different level of security and is based on source analysis rather than on observed behavior.

Most policies, adhering to common engineering principles, are explicit. They can be written down and understood apart from the mechanism. Until recently, most research in computer security has been about mechanisms, about how to guarantee control of resources, rather than what resources should be protected. This makes sense, of course, because the policy is almost always application dependent. The necessary flexibility in design policies is why separation of policies and mechanisms is such a well known design pattern.

Recently, research in policy languages has come to the fore. There are several general policy languages that have been proposed and several that are in use. Some examples are KAoS [15], Ponder [29], Rei [62], and Condell's SPSL [26]. In a paper comparing some of these languages, Tonti and his coauthors describe five features of policy languages:

expressiveness, simplicity, enforceability, scalability, and analyzability [103]. I do not examine these languages in detail for two reasons. First, these languages are meant to be general and are expressive enough that one can reason about policies based on the policy alone. For example, KAoS can detect and resolve conflicts in policies automatically.

The policy's representations in dynamic sandboxing are not meant, with one exception, to be meaningful to users. The one exception, described in Chapter 4 deals with writing standard Java policies. The Java policy language is not a general language, and thus succeeds in expressiveness, simplicity, enforceability, and scalability, as the policy language is tailored directly to the Java platform. I describe the mechanisms that the Java policy language configures next.

2.1.6 Java Security

Java is a unusual language in that its security infrastructure was designed along with the language, instead of bolted on later, or omitted completely. It is designed to prevent the range of security faults commonly found in code written in C: buffer overflows and type subversion. Or, as the virtual machine specification points out, "...it omits many of the features that make C and C++ complex, confusing, and unsafe" [71].

At its base, Java's security rests on its instruction set: a typed assembly language [71] that eliminates pointer manipulation. This ability to reason about types allows the JVM to *verify* that all instructions operate on properly typed data. Not allowing pointer manipulation eradicates many common errors as well. Combined with runtime bounds checking for arrays and type checking during casts, the JVM prevents common security faults. This approach is similar to that used previously by ML and other functional languages that rely on strong, static type checking to prevent runtime errors.

Relying on the integrity of the JVM, the Java platform implements a flexible sandbox-

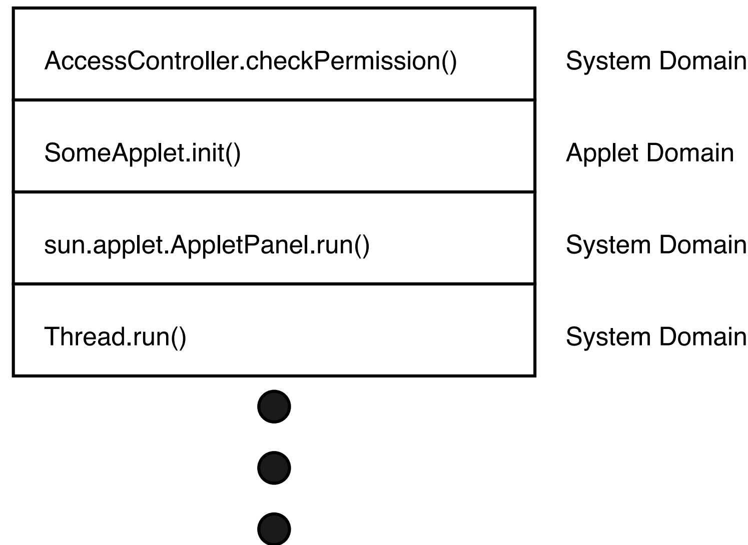

Figure 2.2: The execution stack of an applet as it tries to execute a privileged operation. Because the system domain includes all Permissions, the `checkPermission` call will succeed if the applet's domain includes the permission corresponding to the privileged operation.

ing mechanism on top of it.[1] Each Java sandbox is referred to as a *Protection Domain*.[2] A domain consists of a set of code and a set of permissions attributed to that code.

A JVM can include a large number of Protection Domains. A simple application consists of two domains: all standard library code is one domain and the application another. The standard library has all permissions. An application can execute a privileged operation if all protection domains on its execution stack have that permission. Thus, the set of permissions a method can execute is the intersection of the permissions of all protection domains it is executing in. Figure 2.2 depicts the execution stack of a Java Applet as it

[1] My research was conducted using several versions of the Java platform, but is applicable to all versions later than 1.2 (1.5 is the latest released version at the time of this research. I do not describe much of the early work in Java security (prior to 1.2), such as Wallach's work in stack introspection, because it has been incorporated into the Java platform.

[2] I ignore the security infrastructure for Java Enterprise Edition (J2EE). J2EE security give authority to actors (authorized people or programs), instead of blocks of code. J2EE policies are specified in XML documents that are then translated into standard Java policies.

executes a privileged operation.

Two classes control Permission checks, the SecurityManager and the AccessController. The existence of the two is a historical accident. The SecurityManager came first, and it was initially intended that policy configuration would be handled by reimplementing it. In Java 1.2, the AccessController was introduced to allow policy configuration files written in a human readable language.

Permissions and privileged operations are inherent to the language, not the JVM. New permissions, and consequently, new privileged operations, can be created by developers. This robust and flexible security infrastructure has given the Java platform a deserved reputation of security. Like most large systems, however, Java does have security vulnerabilities.

Java Vulnerabilities

There have been a number of security vulnerabilities reported for the different implementations of Java over the past several years. Those vulnerabilities can be categorized into three types:

- **VM Bugs**—Early Java VMs had several verifier errors, which allowed bad casts, leading to potential subversion of the VM. This class of bugs has declined in recent VMs because Java bytecode verification is now well understood and the implementations are mature.

- **Errors in the Standard Libraries**—Sensitive operations need to be protected by checking Permissions. If these checks are omitted, then applications can potentially access resources that are denied by their policy. These are a growing problem because the size of the Java libraries is increasing quickly (See Figure 2.3).

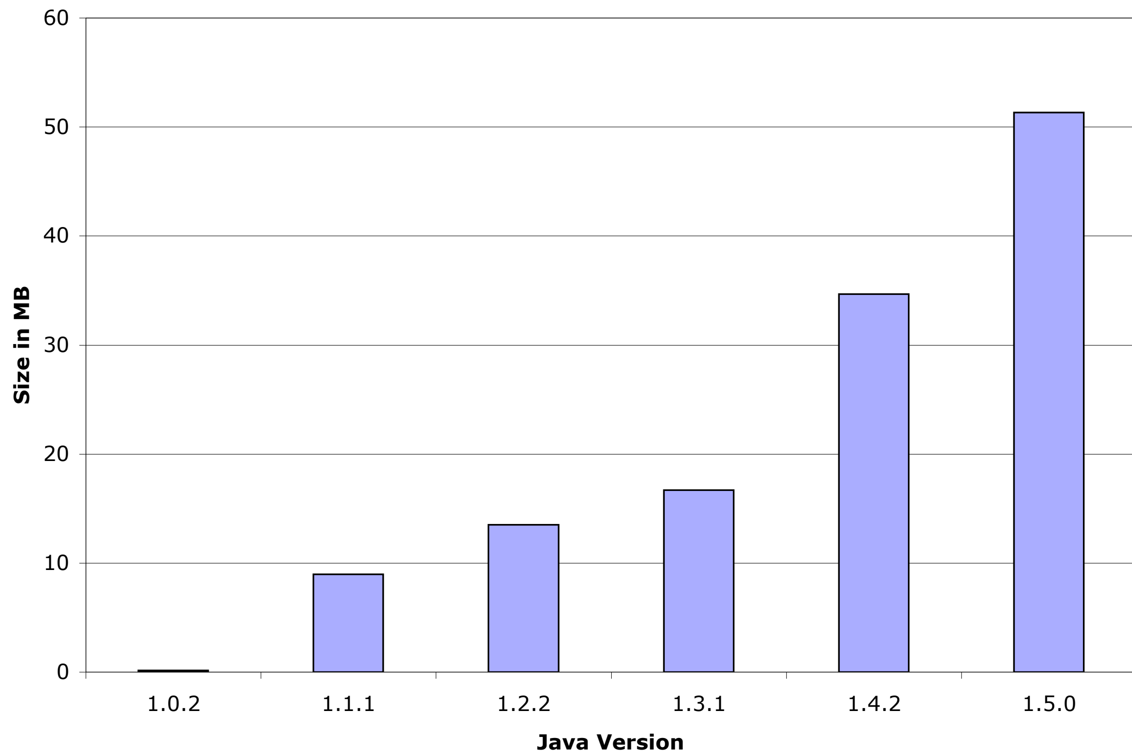

Figure 2.3: The Java standard libraries are growing with each release. Each resource within the library must be properly protected or a malicious program may exploit it.

- **Policy Misconfiguration**—Reports of incorrect policies are infrequent due to their site specific nature. Security policies are necessarily tailored to the individual application and host. However, there have been some reports of sandbox misconfiguration for applets [74]. Policy misconfiguration is likely to be a continuing problem as long as they are configured manually. Like configuring network firewalls, Java security policy specification requires the ability to integrate knowledge of the host, network, and application.

Appendix A shows all public Java platform vulnerabilities reported since 1999 to the Common Vulnerabilities and Exposures dictionary maintained by the MITRE corporation. I have classified them in terms of the three types. Of the 51 exploits listed, 6 are VM, 2 are

policy, and 43 are library vulnerabilities. It should be noted that two out of the first three exploits reported are VM bugs, and the first policy bug was not reported until 2002.

2.2 Data: JVMs, Benchmarks and Exploits

Practical applications of this research are derived from examining the execution behavior of programs running within Java virtual machines. The experiments described in this work were conducted using three different JVMs: Intel's Open Runtime Platform [67], IBM's JikesRVM [83], and Sun's reference implemention of Java. Different versions were used throughout the course of research and are listed for each experiment.

JVMs are just the platform, however. The behavior of *applications* on these platforms is the focus of my research. I have used benchmarks recognized within the field as the best available. Over the course of research, experiments have used three sets of benchmarks: the Java Olden benchmarks [84], the SPEC JVM98 [98] and JBB benchmarks [99], the DaCapo benchmarks [12], as well as some others that were useful for this research but are not recognized as benchmarks.[3] Table 2.1 provides descriptions of the programs that comprise the benchmark sets.

These are the best benchmark suites available. Most Java programs are not widely distributed, however. They are used as "middleware": applications that serve as intermediaries between other applications such as web servers and system software like databases.[4] Their security is critical because they interact with computers connected to the internet. Because they are specific to individual companies, they are not distributed and any security vulnerabilities go unreported.

There are relatively few publicly known exploits for any given JVM. There are even

[3] The version of the DaCapo benchmarks used for this research was *dacapo040924.jar*.

[4] There are many companies producing middleware applications and few producing commercial end user ones.

fewer for those VMs that are open source. Publicly reported vulnerabilities are reported and classified in Appendix A. Very few of these were discovered through wild exploits, and unfortunately, most are tied to specific JVMs, OSs, and web browsers. Additionally, I was only able to acquire one of them. I was forced to test the security applications of dynamic sandboxing against a set of synthetic exploits. These are the exploits I used:

- **StrangeBrew** is the only exploit used in its wild state [58]. StrangeBrew was the first Java virus found in the wild. When an infected file is run, it examines the current directory for Java `.class` files, and infects the constructors of those files if they are not already infected. It determines whether a class is infected by examining file length: a class is infected if the length of its `.class` file is a multiple of 101.

  Unfortunately, the StrangeBrew wildtype does not work in JVMs later than Java 1.0. I developed an exploit which manipulates class files using the same rules as the wild type to simulate infection using more recent Java versions.

- **BeanHive** was the second virus found that targeted Java. It is an interesting virus in that most of its code is not stored in the infected `.class` files. Instead, only enough code is added to download the helper classes that enable the virus to search through the current directory and infect any class files currently uninfected. It adds the infection stub to the end of all constructors. Symantec could never get BeanHive to function as intended [101]. I wrote a reliable version using the Apache BCEL libraries [6].

- **HttpTrojan** is an exploit I developed. It is a simple webserver with a backdoor which allows the user to execute arbitrary commands on the remote machine.

- **Port25** is a short class file that tests whether it can connect to an outside server [106]. In effect, it probes the current policy to determine if it is constrained by the applet sandbox. Its most likely incarnation is as part of a trojan because it performs no function other than the probe.

Olden	
bh	N-body problem solved using hierarchical methods
bisort	Bitonic sort both forwards and backwards
em3d	Simulates electromagnetic waves propagation in 3d
health	Simulates the Colombian health care system
mst	Computes minimum spanning tree of a graph
perimeter	Computes perimeter of quad-tree encoded image
power	Solves the Power-System-Optimization problem
treeadd	Sums a tree by recursively walking it
tsp	The traveling salesman problem
voronoi	Voronoi triangle decomposition
SPEC JVM98/JBB	
_201_compress	Uses Lempel-Ziv to compress and decompress some strings
_202_jess	Expert systems shell based on NASA's CLIPS system
_209_db	Emulates database operations on a memory resident database
_213_javac	The Java compiler from JDK 1.0.2
_222_mpegaudio	Decodes MPEG layer3 (mp3) files
_227_mtrt	Multi-threaded raytracer draws a scene with a dinosaur
_228_jack	Java parser generator is a lex and yacc equivalent
pseudojbb	Java Business Benchmark altered for GC research
DaCapo	
antlr	Produces a parser and lexer for a set of grammar files
batik	Renders several SVG files
bloat	Analyzes and optimizes bytecode in several Java class files
chart	Constructs several charts using JFreeChart
fop	Translates a XSL-FO file into PDF
hsqldb	Simulates a typical JDBC application
jython	Interprets several python applications
pmd	A lint style program for Java
ps	Interprets a set of postscript files
xalan	Compiles XML files to HTML
Miscellaneous	
HelloWorld	A simple program that prints "Hello World"
HelloWorld 2	Like HelloWorld, but also calls its constructor
LimeWire	A peer-to-peer filesharing program
HttpTrojan	Simple web server which includes a backdoor
Metron UAV Sim	Multi-threaded simulation of unmanned aeronautical vehicles (UAVs)

Table 2.1: Description of the programs I used in my research. They consist of three sets: the Java Olden benchmarks, used primarily in garbage collection research, the SPEC JVM98 and JBB benchmarks, and the DaCapo benchmarks. I developed the *HelloWorld*s and *HttpTrojan* myself. LimeWire [72] was acquired from LimeWire LLC and the UAV simulation was acquired from the Metron Corporation.

Chapter 3

The Simplest Feature: Method Invocation

This chapter begins a series of case studies, with one per chapter. In this chapter, I show how patterns of method invocations can be used for anomaly intrusion detection. Method invocation is first because in many ways it is the simplest feature of execution in the Java Virtual Machine. It is also the feature most analogous to the system call. Although many would consider instructions or basic blocks the fundamental feature of execution, this is an inconvenient unit in Java Virtual Machines. JVMs are built to manipulate methods. Types or classes might also be considered a fundamental unit, but in this chapter I consider execution rather than memory. For a short analysis of classes, see Section 7.1.1. I chose anomaly intrusion detection as the application because most intrusions should be easy to detect by examining the method invocation behavior of the application. I start with a review of anomaly intrusion detection, followed by a description of characteristics of method invocations within the Java Virtual machine. I then describe the implementation and efficacy of an anomaly intrusion detection system that uses method invocations as its feature set. I finish with a discussion of the results.

3.1 Motivating Application: an Anomaly IDS by Method Invocation Observation

Anomaly intrusion detection is the use of anomaly detection to detect violations of security policy. I discussed a body of related work in Section 2.1. To summarize, anomaly detection assumes the future is like the past—I observe normal behavior by observing it for a period of time, and afterwards flag all behavior that deviates from the original observations. In anomaly intrusion detection behavior, I assume that these anomalies are intrusions and then respond accordingly. Unlike most intrusion detection systems, the dynamic sandbox focuses on behavior, not structure. Intrusion code injected into the system, if it does not run, is not detected.

Method invocation is the atomic unit of execution in Java. Methods are functions attached to classes. All applications begin by invoking the method `main` and end by returning normally or by exiting through an exception. During execution, an application's `main` computes progress by invoking (or calling) other methods. Those methods invoke other methods, and so on. Methods are bundled into classes. All execution occurs within methods attached to classes.

It is well known that there are predictable characteristics of program execution. A familiar example is the 90/10 rule, the rule of thumb that 90% of a program's execution is spent in 10% of the code. This relates directly to methods. In fact, one finds extremely regular behavior when examining method invocation frequency. Consider the canonical HelloWorld program, written in Java, shown in Figure 3.1a. In Java, HelloWorld consists of a one-line main routine which calls `System.out.println()`.The graphs plot methods against their frequency and are sorted by frequency. Plotted using log-log scales, they exhibit power-law behavior (in languages this is known as Zipf's law). This behavior confirms the 90/10 rule and suggests that most application behavior is easily observed. The heavy tail of a power-law, however, implies that some behavior is unlikely to be observed

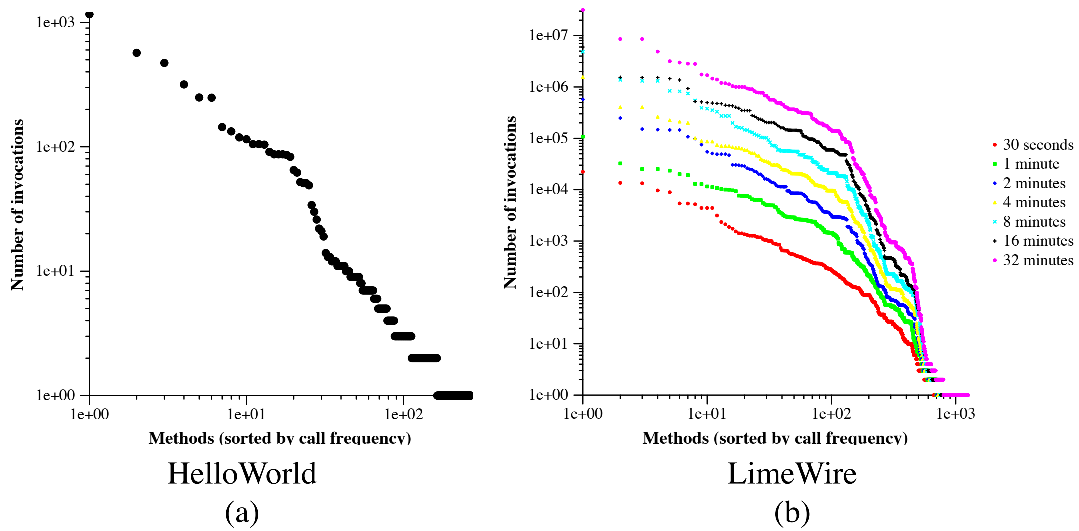

Figure 3.1: Method invocation frequency in HelloWorld and LimeWire.

during training, leading to false positives. LimeWire's behavior, shown in Figure 3.1b, is more complicated. The LimeWire graph shows the evolution of method invocation frequency behavior over time. It starts off somewhat linearly but then decays into more complicated multiple power-law domains.

This strengthens confidence in an anomaly IDS because the overall decay is faster than power-law. HelloWorld primarily shows the bootstrapping behavior of the VM. LimeWire is a real, commercial application. The faster than power-law decay shown in the longer runs of the program indicates that the rate of novel behavior drops over time, allowing proportionally shorter training times.

A Java application running without an explicit security policy (the default mode), may execute any method belonging to its own classes as well as all methods within the standard Java library. The number of library methods is usually much larger than those of the application. Library methods are important because they provide the interface through which an application can omanipulate the system—it communicates with other processes, the network, or reads and writes files by invoking them. An application that cannot access

Java Version:	Total Classes	Total Methods
1.0.2	642	5562
1.1.1	1559	12626
1.2.4	4818	38079
1.4.2	10240	85969

Table 3.1: The total number of classes and methods within the standard and associated helper libraries bundled with various versions of Sun's standard Java Runtime Environment.

the standard library is a brain without a body—it is powerless. The library classes and methods available to applications are dependent on the Java platform on which they are running.

An anomaly IDS relies on differences between the profiles of anomalies and normal behavior. If applications exercise the entirety of the space of possible behavior during normal execution, the anomaly IDS will not be able to identify intrusions. Table 3.1 shows the total classes and methods available to applications on several versions of Sun's Java platform. Table 3.2 shows the number of different library and application methods invoked by the Java Olden and SPEC JVM98 benchmarks. The tables show that these benchmarks exercise only a tiny proportion of the standard library, and thus intrusions might show different behavior.

Although anomaly IDSs assume that instrusions will exhibit different behavior than normal execution, it is not clear that they must. A common exercise in misuse detection systems is to try to identify runs of different applications [68]. An examination of the intersection of a pairwise comparison of the set of benchmarks (Table 3.2 shows a geometric average difference of about 68 (data not shown). The comparison of *bisort* and *treeadd* generated the minimum score of 17. They executed different application methods but used the same set of 204 library methods. Thus, behavior "profiles" differ by about a third in the average case and by a minimum of 8%, providing a basis for implementing an

anomaly IDS based on method invocation. In the rest of this chapter, I describe the first of the dynamic sandboxes.

The Olden benchmarks are small applications that are similar in many ways. Larger, more realistic applications are likely to generate more idiosyncratic behavior because they include more application specific code and they invoke a larger and different set of library methods. Because of this, and to keep the initial exploration simple, my initial implementation monitors only method invocations.

Benchmark	Application Methods	Library Methods	Total
bh	54	215	269
bisort	12	204	216
em3d	18	201	219
health	26	211	237
mst	23	212	235
perimeter	42	196	238
power	29	147	176
treeadd	5	204	209
tsp	13	210	223
voronoi	44	209	253

Table 3.2: The number of different application and library methods invoked during standard runs of the Jolden benchmark on JDK version 1.4.2. The benchmarks each consist of 2 to 10 classes.

3.2 Dynamic Sandboxing

Dynamic sandboxing for a given program consists of two activities: sandbox generation and sandbox execution. In the first, a sandbox profile is constructed by running the program with an instrumented JVM. During this training session, profiling information is recorded to the sandbox profile. The sandbox is initially empty and grows during the training run by accumulating records for each unique behavior. Because nothing is added

that is not observed, each sandbox is customized to a given program and context in which it is executed. During sandbox execution, behavior that is not in the profile is considered anomalous.

Dynamic sandboxing is meant to complement, not replace, the standard sandbox. Java's standard security model creates fixed boundaries within which a program must execute while dynamic sandboxing detects execution anomalies. Although I have not addressed the question of response in the current implementation, a dynamic sandbox response is potentially very flexible, and I discuss possible responses in Section 7.2.3. Actions could range from logging the anomaly to terminating the application, as I do here. In the standard Java security model, there is only one response, which is to disallow the attempted behavior by the throwing of an exception.

Dynamic sandboxing's efficacy is directly related to the stability of the instrumented program's behavior. If the underlying program continually executes large amounts of novel code, it will generate a high number of false positives. As I have argued in the previous section, however, the overall faster than power-law decay of applications over time suggests that the system can derive a profile that avoids most false positives.

During the training session, each time a new method is invoked, its signature is written to a log. This log constitutes the sandbox profile. The profile for the HelloWorld program, for example, contains 268 methods. Every method, including those in libraries and natives (methods written in C rather than Java), was included in the profile. During execution, the log is checked for the method signature before the method is compiled.

I modified Intel's Open Runtime Platform (ORP) to perform dynamic sandboxing [67]. From a user's perspective, I made two changes to the ORP interface. First, I added two flags `-profile <filename>` and `-sandbox<filename>`, which write a profile to a file and read in the profile to be used as a sandbox, respectively. The two flags may be used simultaneously.

The mechanisms of generating and using the sandbox profile rely on ORP's Just-in-time (JIT) compiler implementation. When ORP loads a class it does not compile the Java byte code to native code. Rather, it follows a lazy strategy, delaying native compilation of each method until it is invoked. In the place of the native code, ORP inserts small stubs called trampolines, which operate as follows:

- Call the JIT compiler for the specified method.
- Patch the jump table, the list of pointers to native code, to call the newly compiled native code.
- Jump to the newly compiled code.

Dynamic sandboxing requires a slight modification of this strategy. I modified ORP to append the class and signature to the end of the sandbox profile. This takes place in the profiling phase and before the JIT compiler is called. When in sandboxing mode, the modified ORP checks the method against the sandbox profile. Only if the profile contains the method does ORP continue with JIT compilation. If the method does not appear in the profile, ORP informs the user of a security violation and exits.

The lookup of methods in the sandbox profile is currently implemented as a linear search through the file. The sandbox profile file format is simply a list of class and method signature pairs prefixed by their combined string length.

A summary of the algorithm is presented in pseudocode in Figure 3.2. This code is executed only if the method has not been previously invoked. If the JIT is enabled, then the system is in the training stage. If the JIT is not enabled, then it calls `profile_method_load` with the current sandbox. After that, if it is in profiling mode, the method is added to the profile. Note that there are two profiles active here, the profile the system is constructing, called *profile*, and the sandbox profile, called *sandbox*. The system `profile_method_load` checks to see if the method signature is in the profile

```
int result;
if ( jit_status == ON )
{
    compile_method(method);
}
else
{
    profile_method_load(sandbox, method);
}

if ( profiling == ON )
    add_method_to_profile(profile, method);
```

Figure 3.2: Interface to the JIT compiler. The first if statement allows compilation of the method or forces a profile check. The second if statement adds the method to the profile.

and then loads and compiles the method. Ideally, `profile_method_load` would load the native code directly from the profile. To accomplish this, however, I would have had to write the equivalent of a linker. This is future work and is discussed in the context of a complete system in Section 7.2.

This implementation of dynamic sandboxing was designed to be efficient. First, I am able to use a JIT compiler, so programs execute efficiently. Second, instead of performing a check on every method invocation, I only perform the check the first time a method is invoked. There is one remaining implementation inefficiency—the linear search of the sandbox profile. Although its worst case is linear in the number of unique methods invoked (corresponding to a linear search through the file), in practice it is much closer to constant because the order of methods in the profile reflects the execution path from a previous run. The execution paths of later runs are usually similar, so that the profile file pointer is usually pointing to the method description queried by ORP.

Dynamic sandboxing in ORP is only one of several implementations I have developed.

An earlier implementation relied on Kaffe's interpreter [102], which ran very slowly. I have also developed implementations by instrumenting libraries and adding a wrapper around the Sun JIT compiler interface. This allows programs to run with the standard Sun libraries. However, there is a large performance cost to this last approach, because there is overhead on every method invocation instead of only on the first.[1] The benefit, however, is that I can examine the behavior of all applications instead of the subset that run under the free software classpath implementation [39]. Also, simulation has allowed me to investigate other features without the complexity of modifying the virtual machine itself. I describe those results in subsequent chapters.

3.3 Experimental Results

In order to be useful, the implementation should have the following properties: It should be effective at stopping attacks, be efficient, and experience few false positives. As I discussed above, there is a paucity of documented attacks in the wild on production Java programs, which makes it difficult to perform large-scale systematic testing. I tested the system against two exploits, StrangeBrew, the first Java virus found in the wild, and HttpTrojan, which I developed myself. I also discuss the system's theoretical performance against the other exploits described in Section 2.2. To assess the performance of the system, I tested against a synthetic benchmark and the Olden benchmarks.

[1] There is a small difference in behavior between these systems. In the Sun-based system I prevent methods with the same implementation from running if they are associated with different types. In ORP, once a method is compiled it can be invoked even if the specific instance's type is not in the profile.

3.3.1 Effectiveness

I tested dynamic sandboxing against a Java virus and a simple HTTP server with a backdoor. The Java virus, named StrangeBrew, is the first reported virus targeting Java programs [58]. When invoked, the virus searches the current directory for uninfected class files. For each uninfected class, it adds a copy of itself to the class and modifies the constructor to call itself. It then pads the file to a multiple of 101 so it can determine a class's infection status without opening it. I was able to use the wild version for this research.

The StrangeBrew virus readily infects any application, including the canonical simple program HelloWorld. When the infected program is run with a profile gathered against the uninfected class, no security violations are found. That is because `main()` never calls the infected constructor—no HelloWorld instance is created. It is not until the anomalous code attempts to execute that the sandbox can detect a problem. Depending on one's point of view, this is either a "feature" or a "bug." In my view, the foreign code is not dangerous until it executes, and in this way intrusion detection can devote its resources to code that is about to cause damage. Thus, dynamic sandboxing focuses on *behavior*, not structure. With this in mind, a second line was added to HelloWorld which calls the constructor. The new class proved infectious. With sandboxing enabled, however, the first call into the virus violated the profile, causing ORP to exit, and the attack was prevented. This result would be seen for any program, not just HelloWorld. Because it is the foreign virus code that is identified, the dynamic sandbox is able to prevent the virus from infecting any dynamically sandboxed program.

The second exploit I tested is a backdoor in a simple HTTP server (HttpTrojan). The backdoor was implemented as a special command which allows a remote client to execute arbitrary commands on server. A sandbox profile was constructed by running the HTTP server and exercising it by downloading pages. I then activated sandboxing with the custom profile. When I attempted to the backdoor, the sandbox trapped the first call in the

a call to `System.exec`, without disrupting other legal uses the server. Again, dynamic sandboxing is effective here because sandbox is a reflection of behavior, not structure. The exploit does not insert foreign code into the application, although I believe sandboxing would protect against that as well (based on the example described earlier). In this case, the malicious code is in the application itself. Because the backdoor consists of "dead application code" (code that is never executed), the sandbox effectively eliminates it.

This work was completed before BeanHive and Port25 were reported, and neither exploit would run under the implementation's incomplete Java libraries. Although I cannot directly test these exploits, assuming they would run with proper library support, the system would likely detect both. The system would easily detect BeanHive due to the extensive library support necessary to copy itself. Port25-based trojans would also be detected if the hosted program did not initiate network connections. This is a weaker result, because many applications that benefit from intrusion detection are network based.[2] To summarize, the system detects the two exploits tested, and would likely detect the two I was unable to test on ORP.

3.3.2 Efficiency

Dynamic sandboxing should be efficient as well as effective if it is going to be useful. The implementation should perform efficiently for applications in which methods are executed repeatedly. For many interesting programs, notably server applications such as LimeWire, this is true. Indeed, this is almost universally true, because the number of total invocations is much larger than the number of methods available in most applications.

To confirm that the implementation is efficient on realistic programs, I tested dynamic

[2] Method sequences or the arguments to the network methods may be more effective for these trojans. I investigate these features in Chapters 4 and 6.

sandboxing against the Olden benchmarks. I ran the benchmarks 15 times each with no flags, then the `-profile` flag, and finally the`-sandbox` flag, for a total of 45 runs.[3]

ORP Parameters	Mean user + system time in seconds (Std. Dev.)
Default (no sandboxing)	304.59 (0.26)
Logging the sandbox profile (-profile)	308.93 (0.35)
Dynamic sandboxing enabled (-sandbox)	308.03 (0.19)

Table 3.3: Efficiency of sandbox generation and protection on the Olden benchmarks.

The results appear in Table 3.3. Profiling and sandboxing each were about 4 seconds slower than the unobserved benchmark (an overhead of less than 2%), confirming the expectation that the implementation would be efficient.

Thus, the Olden benchmarks show that the average case is efficient. How efficient is the worst case? I wrote a synthetic benchmark to test the efficiency of the implementation under conditions in which the overhead of lookup cannot be amortized over multiple invocations. Because overhead is isolated to the first invocation of each method, the benchmark consists of a class with 1000 empty static methods, each invoked once. After gathering the sandbox profile, I modified the benchmark to invoke the methods in the exact reverse order of the profile—the pathological case. The benchmark was run 100 times under with no arguments, `-profile`, and `-sandbox`. Two sets of sandbox runs were made: once with a generated profile and once with a profile modified to produce worst-case behavior. See Table 3.4 for the resulting data.

ORP parameters	Mean user + system time in seconds (Std. Dev.)
Default (no sandboxing)	0.26 (0.01)
Logging the sandbox profile (-profile)	0.28 (0.01)
Dynamic sandboxing enabled (-sandbox)	0.28 (0.01)
Dynamic sandboxing on pathological benchmark	5.83 (.21)

Table 3.4: Efficiency of sandbox generation and protection on the synthetic benchmark.

[3]I used the standard benchmark parameters with three exceptions: 2048 instead of 4096 on Barnes-Hut, 512 instead of 1024 on Minimum Spanning Tree, and 10 instead of 20 on the TreeAdd (tree traversal) benchmarks. The original parameters exceeded the unconfigurable ORP heap size.

Sandbox generation and the synthetic benchmark had modest performance decreases of 6%. The pathological case had an enormous slowdown of 2216%. The slowdowns shown for profiling and dynamic sandboxing exaggerate the true effect they have on applications and are included to only show worst case behavior. First, overhead is only incurred on the first invocation of a method. It is small when amortized even over a modest number of invocations, as seen in the Olden benchmarks. Second, the performance results do not reflect the cost of invoking a JIT for non-empty methods. In real applications, any overhead in profiling or sandboxing is quickly overwhelmed by the cost of JIT compilation. Finally, the pathological case shows that the lookup scheme is not efficient in all cases. It is optimized for method invocations to occur in the same order they were encountered during profile generation—the pathological case is ordered exactly opposite. This is unlikely in most programs, so moving to a scheme like hashing would make the system slower in the normal case. The typical case would be much more expensive, although including hashing in addition to the current scheme might improve performance at the expense of space. It is also possible the benchmarks' "typical" behavior is too varied to be captured optimally in one data structure—a slowdown in the search speed might be indicative of anomalous behavior.

3.3.3 False Positives

The exploits and efficiency tests give evidence that dynamic sandboxing can be effective and efficient at stopping exploits. The last requirement for an IDS is a low false positive rate. For the experiments described above, I found zero false positives. This is certainly encouraging, but not conclusive given the limited nature of the tests. For real applications, I would expect to see at least some false positives.

The false-positive rate is related to the problem of "perpetual novelty" in interesting applications. An anomaly-detection system can never be sure that it has observed the entire

range of normal behavior. One approach to this is that of generalizing over the space of observed patterns, with the hope that the generalization will include most new legitimate behavior as in [70, 49, 73].

A second approach is to look at the distribution of novel patterns in the data and use that to make predictions about the distribution of novel patterns in the data. A rank/frequency plot is often used to study such distributions, such as the plots shown in Figure 3.1. When plotted on a log-log scale, the slope of the rank/frequency curve reveals the proportion of frequently seen behavior to potential false positives.

For example, consider the LimeWire trace recorded for 32 minutes discussed in Section 3.1. We can see that the distribution falls off more quickly than a power-law. This tells us that the frequency of rare events decreases faster than a polynomial and slower than exponential. Only 162 methods are required to obtain 0.01 false positive rate per method invocation and 444, 35% of total number of methods executed during the trace, to push that rate to 0.0001.[4] As the program is run longer, the relative frequency of these rare events will decrease (this can be seen by examining the different trajectories plotted in Figure 3.1.

Finally, the size of the entire profile is small enough that we can record the long tail of the power-law in the profile. I emphasize again that I am not looking at paths or sequences, which increase the size of the space combinatorially. LimeWire, a widely used commercial application, invokes fewer than 1300 methods. In fact, the entire possible space is only on the order of tens of thousands of methods, depending on the specific Java distribution used. As Somayaji noted, smaller spaces have advantages in the space requirements, generalization, and profile generation time [97].

[4] I would have liked to have used LimeWire to test performance, but it currently does not run under ORP. I instrumented the LimeWire bytecode in order to get method data, but as a result the program was too sluggish for human interaction.

3.4 Discussion

Dynamic sandboxing, like other anomaly IDS, infers policy from behavior. This is predicated on the idea that normal execution can reveal and document complex policies more reliably and efficiently than users or developers can. As we know from familiar environments like UNIX, user devised policies are often flawed. As with any empirically derived model of normal behavior, dynamic sandboxing comes with the risk of imperfect detection, that is, false positives and false negatives. In practice, the number of false positives is expected to be low, but due to the lack of exploits in the wild, there is limited experimental evidence to justify this prediction.

This differs from specification based approaches, like that of Wagner and Dean [108], that do not allow the possibility of false positives. Although they looked at system call sequences, an analogous system could be devised for Java. Although they prefer static linking, a realistic Java IDS would build the specification from the bytecode class files at load time. The IDS would then limit execution to a larger sandbox, the entire possible call-graph of Java methods determined from the bytecode. This provides little protection over the traditional sandbox, and is simply a more complex verification mechanism. Once an attack has circumvented the traditional security mechanisms, it can execute anything, because those code paths would have been examined statically at class load time. Considering my experiments, both the viruses, trojan HTTP server, and Port25 exploit would run successfully. This is because the code, which is the basis for the specification in the hypothesized Wagner and Dean type IDS, carries the exploit with it. The ORP implementation of dynamic sandboxing prevents the security fault because it focuses on behavior, not structure.

Specification-based approaches often have significant performance penalties. The Wagner and Dean system experienced slowdowns on the order of seconds per transaction [108]. I believe, however, that IDSs will not be used unless they impose minimal performance

penalties. Also, I believe IDSs should prevent intrusions from gaining control. Our systems must therefore be computationally efficient and online. Dynamic sandboxing meets these goals. The system incurs a small amount of overhead on the compilation of each method, but no penalty thereafter. For real applications, this is a very small difference.

Given that I studied only two attack classes, it is interesting to consider how dynamic sandboxing would fare against a wider range of attack types. Because dynamic sandboxing stops novel code from executing, the sandbox would prevent it from using any methods outside the traditional Java sandbox. A successful exploit would need both to disable the traditional security mechanisms and use only previously invoked methods (perhaps with different arguments). Against all three classes of exploits (verification bugs, security manager bugs, and policy bugs) dynamic sandboxing would likely perform well against naïve attackers. In each case, the initial intrusion might succeed, but new code could not execute if it used any methods not already in the profile. One possibility is that the exploit could jump to native code, which dynamic sandboxing could not stop, because it acts only within the Java domain. An intelligent adversary would need to embed the payload within methods already in the profile, in what is essentially a mimicry attack.

For the most part, I have declined to investigate or implement response mechanisms. Collectively, this research is an investigation of regularities of program behavior. Even though each chapter is targetted toward a specific application, the aim is to describe how these regularities may be exploited in a future system, not to produce one. Response is important, however, and I discuss possible reactions to anomalies here and their collective implementation in Chapter 7.

3.5 Summary

I described a strategy, called dynamic sandboxing, which is an anomaly intrusion-detection system for applications running in Java-like environments. To show its efficacy, I implemented a prototype system using a limited amount of profiling information and tested it against two exploits. I presented additional arguments that the system satisfies three criteria of a successful IDS: high true-positive rate, low false-positive rate (zero for the tests reported here), and low performance penalties.

Chapter 4

Using Permissions to Infer Standard Security Policy

In the previous chapter I experimented with a system that examined method invocation, a very general feature of execution in dynamic execution environments. In this chapter I examine a feature tied directly to the Java language standard and libraries—permissions.

Permissions are used in Java very similarly to how they are used in real life. In Java, a portion of code "checks" permission to use a resource. The code continues execution with that resource without restraint if it is allowed. If permission is not granted, the code requesting the resource stops abruptly and must do without it.

If permissions are viewed as just another feature of execution, as we viewed method invocation, the set of permissions required to execute a program can yield a profile of normal behavior. Furthermore, this profile should reflect the security policy of the application—indeed, it should *be* the security policy. Therefore, we would expect this profile to be stable, as a security policy should be.

A security policy in Java is the set of permissions given to each module of code. Tra-

ditionally, a security conscious developer writes this policy explicitly. In many ways it is part of the application itself. As I argued in Section 2.1.6, this leaves applications open to vulnerabilties, both from bugs in the standard library, which the previous chapter focused on, but also on policy bugs, which are less likely to be reported because they are specific to applications' configurations. A system to infer the security policy that confers the fewest permissions, in accordance with the principle of least privilege (see Section 2.1.5), would be desirable.

4.1 Motivating Application: Anomaly Intrusion Detection

Like the previous chapter, the goal of this chapter is to present the design and test the performance of a prototype system to detect anomalies. The profile used to determine whether a permission request is anomalous or not is the standard security policy. The profiles used in the previous chapter were also security policies, but the profile used here is different. First, it is explicitly recognized as such by the standard. Second, the policy is constructed to be edited by hand. This is possible in the earlier system, but the profiles are larger and less likely to be understood without significant knowledge of the Java runtime and specific application.[1]

Automatically generating security policies is not useful unless generating the security policy is difficult. If our permissions are very general, such as allowing to access the network or permission to access the disk, this task is easy enough that no automation is required. If we wish to restrict an application to the minimum system resources required, however, we want fine granularity; we want to restrict an application to specific network hosts and to individual files on the disk. There is a tradeoff, however, between the ease of stating a policy and its granularity. A binary policy (all-or-nothing access) is easy to state,

[1] In the Java Enterprise Edition the underlying security policy is automatically generated from a different security language that assigns Permissions to roles, usually users, instead of code modules. Like the regular Java security policy, however, the JEE policy is generated by hand.

but it is not a good representation of the security requirements of most applications.

The trend in systems is to increase the degree of granularity in specifying access to resources and Java is no exception. A short introduction to Java security was presented in Section 2.1.6. Here, a review of Java security and a more complete description of Java's policy infrastructure will be useful to understand the specifics of the prototype.

4.2 The Java Security Infrastructure

Java's original security model divided programs into applets and applications. Applets had a highly restrictive policy, while applications had no restrictions at all. Later versions of Java introduced finer grained security models to allow a closer fit between requirements and privileges. Predictably, the expressiveness of the current policy language makes it difficult to understand the exact privileges an application requires.

Indeed, Java's current security mechanism supports highly precise policies [40]. The sandbox supports almost any imaginable policy if one reimplements the security classes, while the policy language is very expressive in terms of the granularity of resources it can represent, although it does not support arbitrary algorithms. The policy language is essentially a mapping of resources to code. A sandbox is configured by granting specific Permissions to code. The ability to execute a protected operation depends on the set of granted Permissions and other details such as the origin of the code and the identity, if any, of its digital signers.[2] A collection of classes, signed or not, constitutes a *protection domain*, the basic unit of Java security.

The protection domain itself can be viewed as a sandbox since each domain has an attached set of Permissions, which constitutes its policy. Domains can interact, but pro-

[2] "Permission" is capitalized when I refer to subclasses of the Java class java.security.Permission.

```
grant codebase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};

grant {
    permission java.lang.RuntimePermission "stopThread";
    permission java.net.SocketPermission "localhost:1024-", "listen";
    permission java.util.PropertyPermission "java.version", "read";
    ...
};
```

Figure 4.1: A portion of the standard Java policy file. The file is actually 48 lines long including comments. It grants Java extensions privilege to do anything. It grants a smaller set of Permissions to all other code. The comments (not shown) advise that allowing `stopThread` is inherently dangerous.

tected actions are allowed if all domains include the relevant Permission. A common case of this was presented in Figure 2.2. It occurs when application code calls standard library code, for example, when manipulating files. In this case, there are two domains: the system domain, which includes all the standard libraries, and the application domain. The application domain can open a file if it is given that permission—the system domain is given all permissions. Thus Java platform security can be viewed as a set of interacting sandboxes.

Two classes control security: the AccessController and the SecurityManager. In the early days of Java, the SecurityManager was responsible for implementing the applet sandbox. The SecurityManager is the class that determines whether permission is granted. The AccessController was added to allow for fined grained access control and a human readable policy. Previously, security policy was embodied in the SecurityManager. The SecurityManager determined the result of a Permission check. Now, the SecurityManager usually delegates to the AccessController.

There are several different subclasses of Permission. Each can be constructed with a number of names denoting a specific operation. For some the name indicates the permis-

sion, others are annotated with a set of allowed actions. For example. The permission specified by

```
permission java.io.FilePermission "/etc/passwd", "read";
```

denotes the ability to read the password file. Clearly, a protection domain's policy can be fine-grained. The default Java security policy file is 48 lines long including comments. An excerpt of the policy file is displayed in Figure 4.1.

Specifying a policy in accordance with the principle of least privilege is difficult for even moderately complicated Java programs and will become more difficult as additional Permissions types are added. Today, even specifying a reasonably common policy seems too complicated. For example, Sun's VMs by default do not initialize the SecurityManager [40].

In the remainder of the chapter I describe a method for learning minimal policies and evaluate its efficacy and performance. I then discuss the results, comparing them with the method invocation-based sandbox presented earlier.

4.3 Policy Inference Implementation

The goal is to automatically infer (learn) the minimum required permissions for each required domain in a given application program. I do this empirically during a training phase in which I run the program and record all Permissions that are not implied by the current policy. I start with a policy that grants no permissions. I then add those Permissions to the current policy in the policy language provided by Java.[3] The policy file at the end of the run is the record of all Permissions required for that run. In subsequent runs, the policy file is used to enforce the inferred sandbox.

[3] I ignore the differences between signed and unsigned code for these experiments. For signed code, my dynamic sandbox could simply recognize the signatories and output them with the policy.

```
public void checkPermission(Permission p)
{
    if (recursive()) return;
    if (training)
        foreach(ProtectionDomain d)
            if (!Policy.getPolicy().implies(d, p))
            {
                writePolicy(d, p);
                Policy.getPolicy.refresh();
            }
    super.checkPermission(perm);
}
```

Figure 4.2: Pseudocode for the `checkPermission()` method of `SecurityManager`. If the check call is initiated within this method, return. Otherwise, if training is activated, then add the policy to each protection domain, rewrite the policy file, and refresh the policy. Then check the Permission.

I implemented the dynamic sandbox using a custom SecurityManager to log all calls to `checkPermission()`. Within `checkPermission()`, a private method determines if the required Permission is implied by the current policy. The individual `check` methods that do not take Permission arguments construct Permissions internally with the appropriate arguments and then call `checkPermission()` directly. One performance decreasing detail is that methods, called from the custom SecurityManager to check whether Permissions need to be added, themselves call `checkPermission()`. These are recognized by walking the execution stack and and suppressed since these checks are not needed during regular execution. Calls to `checkPermission()` are added to a specific execution context if it is provided. If it is not provided, I add the Permission to each Protection Domain in the current execution context, rewrite the policy file, and refresh the policy. Figure 4.2 presents the pseudocode for `checkPermission()`.

The implementation required two new classes for training: the custom SecurityManager and an application launcher to install the SecurityManager. Subsequent runs require

no special code. Training runs are initiated with the command:

```
java -Djava.security.policy=<application>.policy DSLauncher
<application>
```

and runs with the inferred policy are invoked the usual way:

```
java -Djava.security.manager
-Djava.security.policy=<application>.policy <application>
```

where `<application>` is the name of the class to be executed. Arguments to the class are appended to the end in the usual way. The policy file need not exist because all necessary Permissions are inferred automatically and added to the policy file. If the policy file exists, the system policy is initialized with that policy. This is helpful in tuning existing policies.

4.4 Experiments

To be practical, the implementation needs to be effective at stopping attacks, experience few false positives, and be efficient to run. First, I explore its effectiveness at stopping attacks, describing experiments against four exploits. Next I discuss false positives, and finally report timing runs against the SPEC Java benchmark suite to assess efficiency.

I tested the dynamic sandbox against the four familiar exploits: **StrangeBrew** [58, 100], **BeanHive** [101], **Port25** [106], and **HttpTrojan**. Section 2.2 provides an overview of the exploits.

The first three exploits were tested with a small host program that either reads or writes to a specified file. A sample workload was generated that reads and writes several files

in the current directory, reads /etc/passwd, and reads and writes some files to /tmp. I generated a policy file for the workload and confirmed that no policy violations occur when running identical workloads. Then an infected version of the host program was run. This tested the sandbox's ability to catch true positives, not false ones. All experiments used Sun's Java 1.4.2.

The dynamic sandbox detected and stopped policy violations of three out of the four exploits. StrangeBrew was the only failure, and it failed because StrangeBrew's behavior is so unambitious—it runs and modifies code only in the current directory. If the virus were more comprehensive (e.g., by ensuring that its code is called in other functions or by recursively searching the directory structure for Java code) then the dynamic sandbox would detect a policy violation.

In BeanHive, the dynamic sandbox identified the creation of the URLClassLoader as a policy violation. Because it could not download its infection code, BeanHive failed. Similarly, Port25 failed because the inferred policy does not include the permission to resolve the `www.netscape.com` address.

HttpTrojan is perhaps the most interesting of the exploits, because it is a fully functional server application that includes a large amount of network and file manipulation. I trained it by browsing a series of pages from the author's web sites, making sure that it hit all usual error states (pages not found, etc.). When tested, the backdoor code failed to execute because the inferred policy denied all files permission to execute.

4.4.1 False Positives and Generalization

As we will see in the Chapter 6, insuffcent training and unstable behavior can result in large numbers of false positives. A false positive in this context would correspond to a legitimate behavior of the program that violates the minimal security policy. In particular, I was concerned about false positives that arise from insufficient training examples or from

configuration changes that necessitate revising the policy. The host application used for the first three test exploits required no tuning (zero false positives) because it only manipulated files in a small number of directories.

The more realistic exploit, HttpTrojan, did require tuning for false positives. In its current implementation, the dynamic sandbox grants file permissions by file name and action. Thus, the training runs must browse all files that will be available during testing. For web sites containing many files this could become cumbersome, so the policy file was manually edited to permit access to any file or directory within a specified base directory. Because the sandbox is expressed in the Java policy language, it is straightforward to make these changes. Similarly, the sandbox assigns network permissions by IP address and port. Permissions were relaxed to include all ports above 1024 and all IP addresses within our university. I foresee that similar tuning would likely be required for many large applications.

Tuning the policy to remove false positives is often a matter of generalization. In the HttpTrojan, Permissions to most of the files used the same base directory so I generalized the permission to allow reads to all files in that directory. Similarly ports and IP addresses were also generalized. Although I did this manually for the experiments, it was easy to automate the process.

I added simple heuristics were added to generalize network and file system permissions as a response to the false positives found while training HTTP Trojan. File Permissions state the name of the file and the action the permission allows: read, write, execute, or delete. A very simple heuristic was chosen. If greater than some arbitrary number of files in a directory are allowed an action, I enable that action for all files in the directory.

I added a generalization heuristic for the Socket class as well. If more than a given threshold number of permissions allows a specific IP address with different ports the same action (accept, resolve, listen, or connect), the heuristic allows all port numbers between

the minimum and maximum in the ungeneralized set of permissions.

The addition of these heuristics allowed HttpTrojan to execute without most false positives. The few that remained involved IP addresses. IP addresses are more sensitive than ports, and I decided a heuristic involving IP addresses might create vulnerabilities.

Beyond generalization of just two Permissions types, simple heuristics for each type could be defined to determine if sets of Permissions should be coalesced into more general ones. In addition to reducing false positives, generalization is used to minimize the size of policy files. A network application that explicitly listed every port and incoming IP address would incur significant performance penalties. Thus, generalization can be used to prevent the size of the policy file from exploding during training.

Unfortunately, it is difficult to generalize the generalization. Heuristics must be tailored for each Permission type and even then there is no guarantee that the heuristic is sound. The policy languages discussed in Section 2.1.5 were designed so that the policy could be mechanically analyzed, but Java's policy language does not allow that. The advantage of writing a human-readable policy comes from the necessity of occasionally manually editing policy code.

False positives arise in part from the Principle of Least Privilege and are not specific to our system (overly restrictive firewalls are a common example). Anytime a policy creates restrictions, especially one that is learned empirically, there is a risk of false positives. They will be rare when the policy is much less restrictive than the needs of the application. The convenience and protection of this dynamic sandbox is a reasonable tradeoff for false positives that can be repaired through generalization or other heuristics.

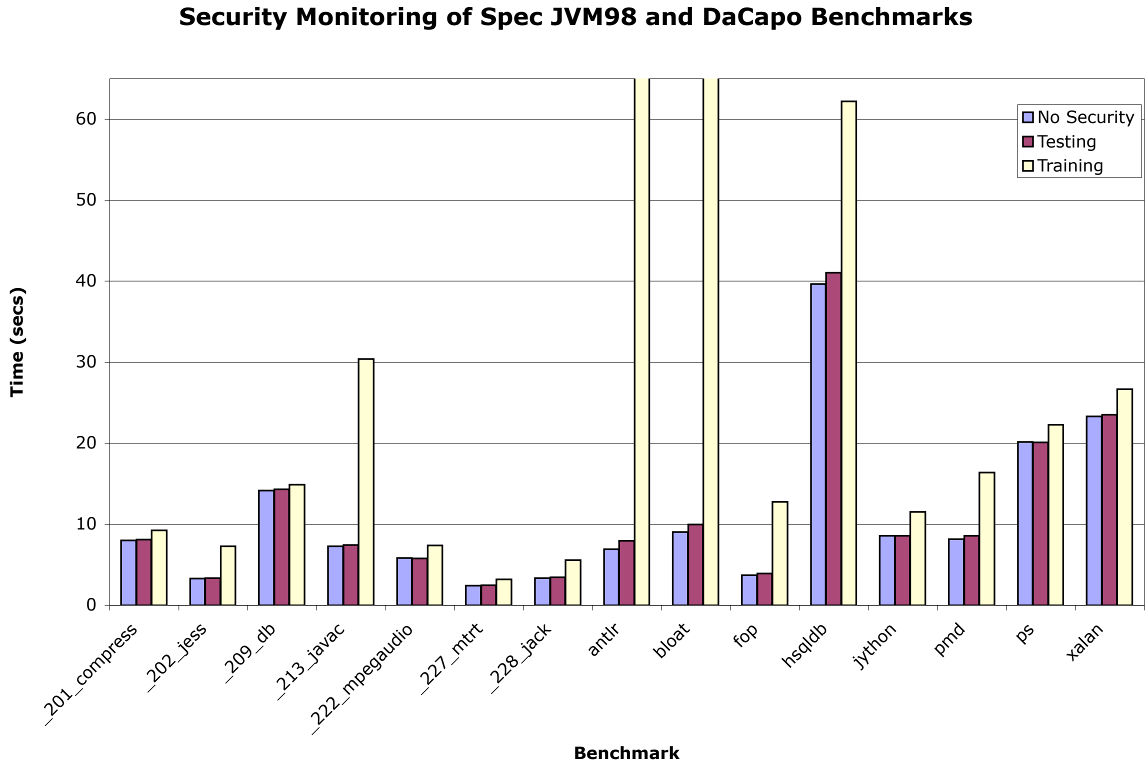

Figure 4.3: The performance of the SPEC JVM98 and DaCapo benchmarks under no SecurityManager (No Security), while generating a policy (Training), and running with that policy (Testing).

4.4.2 Performance

I ran experiments using the SPEC JVM98 and the DaCapo benchmark suites [98], [12] and report three sets of numbers for reach benchmark in the suite: *no security*, *training*, and *testing*. *No security* times the application running without a SecurityManager (the Java default). *Training* measures the total application time when runs are inferring a policy (without generalization heuristics). *Testing* measures the total application time during runs in which the policy is enforced.[4]

[4]Experiments were conducted on a PC running Linux 2.4.21 with a 1.7Ghz Xeon processor and 1 GB of memory in single user mode. The time reported for each benchmark is an average of ten

Figure 4.3 shows the running times of each benchmark under the thee different conditions. The performance penalty for running with the sandbox in place is modest, averaging about 2%. This is notable because inferred policies are typically much longer than the standard Java policy.[5]

Although this is a one-time cost, the penalty for training is greater and differs significantly between the SPEC and DaCapo benchmarks. For SPEC, it ranges from 4% for *db* to 267% for *javac* and averages 34% (this shrinks to 24% if *javac* is excluded.) Five of the eight DaCapo benchmarks show performance degradations for training of greater than 50%: *antlr*, *bloat*, *fop*, *hsqldb*, *pmd*. In particular, *antlr* and *bloat* run more than 16 and 170 times slower than without security.

The training time arises from the computation required to keep the profile parsimonious and by writing and then reading the file whenever a Permission is added. More efficient training regimes may be possible, although writing out the policy as it changes may benefit the administrator; he or she could decide what tuning or generalization is necessary as the application is running.

The performance on the benchmarks shows that performance deficiencies are common to a particular type of program: compilers. The worst performers all parse or modify code. All the rest run within twice of their normal running times. Compilers are a worst case because they interact so heavily with the file system. They serially open many different files, and consequently need permissions for all of them. Once analyzed, they do not use those Permissions again.

Compilers are not usually run in a sandboxed environment. They are not a security concern because they are not long running nor do they communicate with external resources. Thus, their poor performance is less of a concern.

---

runs. Generalization was not used during training.

[5] The sizes range from tens to thousands of lines.

Performance is not problem for programs that are not compilers. Training security policies for applications can be viewed as part of the installation and configuration process. Running with security enabled produces few performance penalties at runtime. The training cost, while significant, would likely be reasonable for most of the applications I envision, which are often I/O bound.

4.4.3 Comparison to the Chapter 3 Dynamic Sandbox

In testing the systems against the exploits, I found the Permissions sandbox does not prevent StrangeBrew from running while the method invocation sandbox does. The difference can be attributed to virus behavior—modification and insertion of code directly intersects with the feature of method invocation. The Permissions sandbox must look at file modification behavior to prevent virus activity. This works at preventing infection in other code, but is not as effective when write permissions are required by the bytecode itself, as the StrangeBrew case demonstrates.

There are more fundamental differences. First, the previous system only sandboxed one resource, methods, while this system includes all the resources that are explicitly Permission checked. Second, the previous system required VM modifications; it could not be implemented in pure Java without significant performance penalties. Those modifications provided benefits, however. This system is vulnerable to VM bugs and some library bugs. Dynamic sandboxing by methods is orthogonal to the standard security apparatus and prevents many of those faults.[6] Finally, the current system produces short, human readable profiles of normal behavior. The older system used large files of method signatures, location in the profile had a large impact on performance.

The two systems are complementary. Each is able to prevent security faults the other

[6]Dynamic sandboxing by methods ensures that methods not listed in the normal profile are never compiled and thus can never be executed. This is “policy as mechanism” [9]. A set of mechanisms that allows a flexible policy, like Java provides, is more likely to be subverted.

would tolerate. Together, they address the entire space of security faults: VM bugs, library bugs, and policy bugs.

4.5 Future Extensions

The system described in this chapter is a prototype. It is not meant to have the performance or features of a production system. The performance of the custom SecurityManager could be improved signficantly at the cost of portability. The SecurityManager would still be implemented in Java, but it would rely on library implementation details that are idiosyncratic to the library itself and not specified by the library standard. Generalizations could be extended as a plugin system instead of being tightly integrated into the SecurityManager core.

Beyond these incremental improvements, I see two interesting additions. First, I am interested in adding the ability to explicitly state Permissions that cannot be granted. This could be stated in an *anti*-policy that uses the usual policy syntax. The SecurityManager would refuse to add these Permissions during training and inform the operator of a fault. Sun has contemplated adding such a mechanism but has thus far refused due to its greater complexity [40]. Second, I would like to extend this work beyond Java. Microsoft's .NET virtual machine includes a security infrastructure similar to Java's. Should .NET become ubiquitous, as is likely with the release of Longhorn, a similar system to the one developed here would be useful.

I discuss how an integrated production system including sandboxing of several features, including Permissions, could be implemented in Chapter 8.

4.6 Summary

In this chapter I described the first practical system to infer minimum security policies for Java applications. I showed that it is both effective and efficient at deriving and enforcing policies. These policies, while not optimal, form a useful basis for hand-tuning. Editing policies is familar to administrators because the policy format is the Java standard. The implementation of dynamic sandboxing uses Permissions as the feature from which to construct the sandbox.

Chapter 5

Object Lifetime Prediction

The system described in this chapter is very different than those presented in Chapters 3 and 4. The goal of those case studies was to build systems that could identify unusual behavior in different features of execution. Although the methodology of this chapter is similar—I build a system to predict regularities of another feature—the objective is quite different. Instead of identifying irregularities for special treatment, I treat the *regularities* as special.

This chapter examines the behavior of the memory management system in the Jikes RVM, a Java virtual machine produced by IBM [3].[1] Because the purpose of this case study is so dissimilar to the previous ones, this chapter is more self-contained. I had originally aimed to build an anomaly detection system similar to those in previous chapters that observed features from the memory management system, but the behavior of the memory system was not conducive to such a system. I discuss some of that work in Section 7.1.1. Instead, I expound upon the regularities I discovered in the behavior of one aspect of the memory management system and present a potential system to exploit them. In Chapter 1 I briefly described garbage collection, the automatic memory management system used in

[1] This JVM was previously named Jalapeño.

DEEs. Before I present the results of my own work, I motivate it by making the case for improved garbage collection systems.

5.1 Motivating Application: Better Garbage Collection

Two popular languages today, C# and Java, are almost synonymous with the DEEs that they run upon. Both are garbage collected. Garbage collection (GC) improves developers' productivity by removing the need for explicit memory reclamation, thereby eliminating a significant source of programming error. However, garbage-collected languages incur increased overhead, and consequently, improvement in their performance is essential to the continuing success of these languages. Many algorithms have been proposed over the several decades since GC was invented, but their performance has been heavily application dependent. For example, Fitzgerald and Tarditi showed that a garbage collector must be tuned to fit a program [36]. Another approach relies on larger heap sizes and simply runs the collection algorithms less frequently. However, this does not always result in better performance [16]. GC algorithms typically make certain assumptions about the lifetimes of the application's objects and tailor the collection algorithm to these assumptions. If the assumptions are not borne out, poor performance is the outcome. What is needed is the ability to make accurate and precise predictions about object lifetimes and to incorporate these predictions into a general GC algorithm that works well for a wide range of applications.

The overhead of GC, compared to explicit deallocation, arises from the cost of identifying which objects are still active (*live*) and which are no longer needed (*dead*). GC algorithms, therefore, go to some lengths to collect regions of memory that are mostly dead. The ideal garbage collector would collect regions where all the objects died recently, so that heap space is not wasted by dead objects, and living objects are not processed unnecessarily. To do this, the allocator would need to know the exact death time of an object

at the time it was allocated, and then it could allocate it to a region occupied by objects with the same death time. To date, this has been accomplished only in a limited way by a process called "pretenuring." Pretenuring algorithms make coarse predictions of object lifetimes, predicting which allocations will result in long-lived objects and then allocating them to regions that are not frequently collected. For example, in Blackburn's pretenuring scheme [14], objects are allocated into short-lived, long-lived, and eternal regions. As this chapter will show, the inability to predict lifetimes precisely is an obstacle to the ideal garbage collector. I also show how allocation-site information available to the VM can be leveraged to improve object lifetime prediction and how that ability might be exploited by the JIT compiler and collection system.

The organization of this chapter is as follows. First I demonstrate that there is a significant correlation between the state of the stack at an allocation point and the allocated object's lifetime. Next, I describe how this information can be used to predict object lifetimes at the time they are allocated. I then show that a significant proportion of objects have zero lifetime. Next, I analyze the behavior of a hypothetical hybrid GC (the death-ordered collector) that uses my prediction method, examining its implementation overheads and describing its best-case behavior. Finally, I compare my results with related work and discuss other potential applications.

5.2 Object Lifetime Prediction

My approach is inspired by Barrett and Zorn's work on object lifetime prediction in C applications [11]. In particular, both methods use similar information, the predictors are constructed similarly using run-time profiling, and I have adopted their "self prediction" test. In addition, I have made several extensions. First, I am using a garbage collected language, Java, in which deallocation is implicit. Second, I have introduced fully precise prediction; Barrett and Zorn used only two bins—short and long-lived. Finally, I have

conducted a more detailed analysis, the contents of which form the bulk of this chapter.

As mentioned earlier, one goal of object lifetime prediction is to improve performance by providing run-time advice to the memory allocation subsystem about an object's likely lifetime at the time it is allocated. Similar to Barrett and Zorn, I accomplish this by constructing an object lifetime *predictor*, which bases its predictions on information available at allocation time. This includes the context of the allocation request, namely the dynamic sequence of method calls that led to the request, and the actual type of the object being allocated. I refer to this information as the *allocation context*; if the observed lifetimes of all objects with the same allocation context are identical, then the predictor should predict that value at run-time for all objects allocated at the site.

The system described here, like most of the other systems described in my research, relies on a small system implemented within the VM to retrieve the data, and external simulations which analyze the data. Because I have not yet integrated my predictor into the memory allocation subsystem, my testing is trace-driven and not performed at run-time. If fully implemented, my system would operate similarly to other profile-driven optimization systems. First, a small training run would be used to generate the predictor, instead of logging the trace. Subsequent runs would then use the predictor for the various optimizations discussed below. A description of the implmentation of this type of system, integrated into the VM with the other systems described in my research, can be found in Chapter 8.

I consider two circumstances for prediction: self prediction and true prediction. *Self prediction* [11] uses the same program trace for training (predictor construction) as for testing (predictor use). Self prediction provides an initial test of the hypothesis that allocation contexts are good predictors of object lifetimes. Although self prediction is not predicting anything new, it allows us to study the extent to which the state of the stack is correlated with the lifetime of the object allocated at that point. This provides evidence that true prediction is possible. *True prediction* is the more realistic case, in which one small training

trace is used to construct the predictor, and a different larger trace (generated from the same program but using different inputs) is used for testing. If self prediction performance is poor, then true prediction is unlikely to succeed. But, accurate self prediction does not necessarily imply successful true prediction. Although I have not analyzed it in detail, I expect that this latter case is most likely to occur in heavily data-driven programs.

The load on the memory-management subsystem is determined by heap allocation and death events, and it is independent of other computational effects of the program. Therefore, the lifetime of an object in GC studies is determined by the number of bytes of new memory that are allocated between its birth and its death. More specifically, object lifetime is defined as the sum of the sizes of other objects allocated between the given object's allocation and death, and it is expressed in bytes or words.

Predictor performance is evaluated using four quantitative measures: precision, coverage, accuracy, and size:

- *Precision* is the granularity of the prediction in bytes. A fully precise predictor has precision of one byte; e.g., it might predict that a certain allocation site always yields objects with a lifetime of 10304 bytes. A less precise predictor might predict from a set of geometrically proportioned bins, such as 8192–16383 (I refer to these as *granulated predictors*). Or, as I mentioned before, from a small set of bins such as short-lived, long-lived, and eternal. My aim is to achieve high precision (narrow ranges). In practice, the ideal precision will depend on how the memory allocation subsystem exploits the predictions.

- *Coverage* is the percentage of objects for which the predictor makes a prediction. I construct the predictors so that they make predictions only for allocation contexts that can be predicted with high confidence. Thus, in some cases the predictor will make no prediction, rather than one that is unlikely to be correct, and the memory allocation subsystem will need a fallback allocation strategy for these cases. Although

the decision to predict is made per allocation site, the natural measure of coverage is the percentage of actual object allocation events that are predicted (a dynamic count) rather than the percentage of sites at which a prediction can be made (a static count). Ideally, coverage should be as high as possible.

- *Accuracy* is the percentage of predicted objects which are predicted correctly. That is, among all the objects allocated at run time for which a prediction is made, some will have a true lifetime that falls in the same range as the predicted lifetime; the range is defined by the precision parameter. Accuracy should be as high as possible.

- *Size* is the number of entries in the predictor, where an entry consists of a descriptor of an allocation site and the prediction for that site. Because the predictor incurs space and time overhead at run-time, smaller sizes are better.

There are interesting tradeoffs among precision, coverage, accuracy, and size. For example, a predictor must choose between coverage and precision. Increasing the predictor size (adding more entries) allows either greater coverage (by specifying predictions for objects not previously covered) or greater precision (by specifying different predictions for those objects). There is also the obvious tradeoff between coverage and accuracy.

I construct predictors in two stages. First, I collect an execution trace for each program and then construct the predictor itself. The trace includes accurate records of each object allocation and death. For each allocation event, the system records the object identifier, its type, and its execution context. The execution context represents the state of the entire stack at the time of allocation, consisting of the identifiers of the methods and bytecode offsets, stored as integers. I refer to this information as the *stack string*. In most cases the amount of information is reduced by recording only the top few entries of the stack (the *stack string prefix* (SSPs) that I also examine in Chapter 6, and study the effects of varying the length of the prefix.[2] Each death event is recorded to a precision of one byte.

[2]The stack string prefix is not to be confused with the supervisor stack pointer which has the

This full precision is unique in object lifetime traces of garbage-collected languages. Object lifetimes reported in the literature almost always have coarse granularity for garbage-collected languages [14]. This is because object death events can only be detected at a collection point, and collections are relatively infrequent. I used an implementation of the Merlin generation trace algorithm [47] within the JikesRVM to collect fully precise object-lifetime data. Merlin makes absolute precision practical by not enforcing frequent garbage collections. Instead, it imprints a timestamp on objects at memory roots when an allocation occurs.[3] During a collection, an object's death time can be computed by determining the last time stamp in the transitive closure of a group of objects.

Stack String Prefix	Type	Lifetime
2724:1563:1858:1490:3984:3030	**[B**	**64**

Figure 5.1: A single predictor entry: The SSP describes the execution path of the program. Each integer encodes the method and position within the method of the next method call. The entire string denotes the allocation site. All byte arrays (JVM type [B) allocated with this stack string prefix had a lifetime of 64 bytes.

A predictor consists of a set of predictor entries. A predictor entry is a three-tuple <SSP, type, lifetime>, as shown in Figure 5.1. The trace is used to construct a predictor for the corresponding program. An entry is added to the predictor if and only if all lifetimes during training corresponding to the SSP and type are identical. This implies that if any two objects allocated at the same allocation site have different lifetimes, or *collide*, the predictor will not make a prediction for that allocation site.

This type of predictor is computationally efficient and tunable. For example, *singletons* typically dominate the predictor and increase its size significantly. Singletons are entries

same acronym.

[3]Memory roots are the references to objects that a program can manipulate directly. Examples are registers, and particularly in Java, the references in the program stack.

for which only one object was allocated during training, so no collisions could have eliminated them. These entries might be removed to form a smaller predictor without greatly reducing coverage (because each entry is used so infrequently), as shown in Figure 5.4.[4] More sophisticated strategies could also be devised to optimally balance coverage and size.

I consider three aspects of lifetime prediction:

- *Fully Precise Lifetime Prediction:* Object lifetimes are predicted to the exact byte.
- *Granulated Lifetime Prediction:* A lower precision approach in which the predictor bins lifetimes according to the scheme $bin = \log_2(lifetime + 1)$. Because most objects die young, the effective precision of this method is still high.

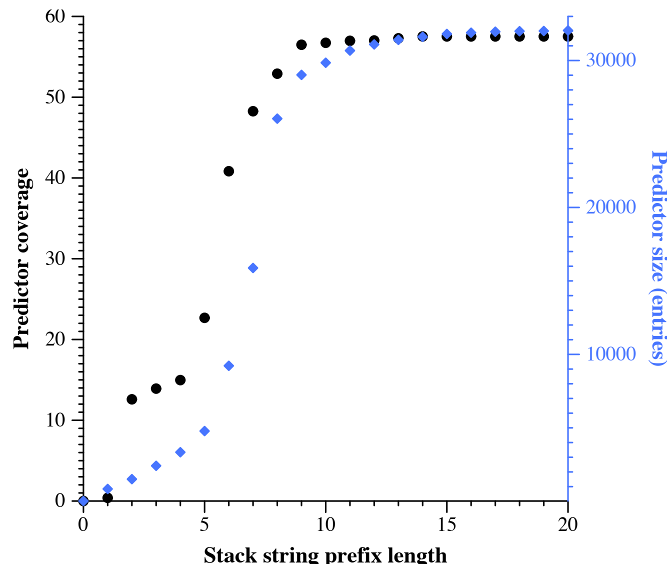

Figure 5.2: The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (including singletons).

- *Zero Lifetime Prediction:* The predictor predicts only zero lifetime objects (those that die before the next object allocation). I discovered that some benchmarks gen-

[4]The benchmark *perimeter* from the Java Olden suite [20, 84], is used here because it displays the behavioral archetype.

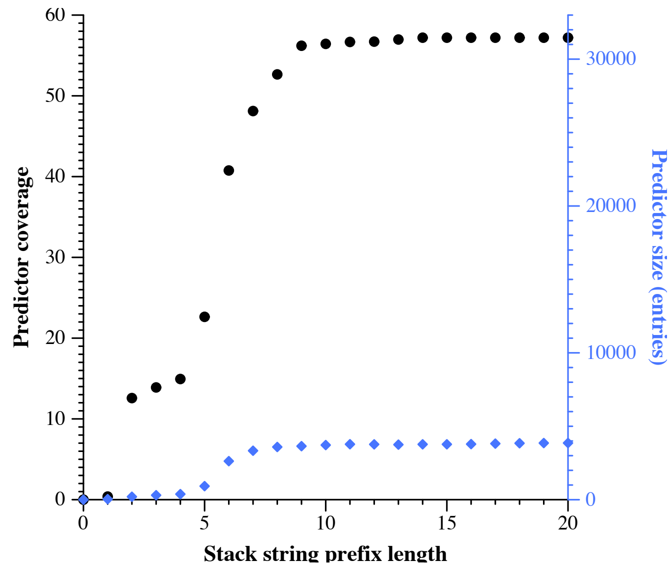

Figure 5.3: The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (excluding singletons).

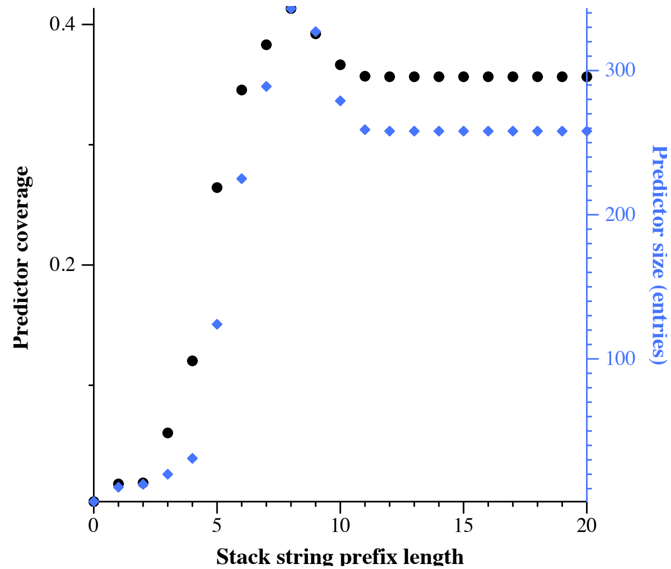

Figure 5.4: The effect of stack prefix length on predictor size and coverage for the Java Olden benchmark *perimeter* (excluding singletons).

erate a large number of zero-lifetime objects. Predicting zero-lifetime objects is an interesting subproblem of fully precise prediction.

I illustrate these concepts and tradeoffs on the example benchmark *pseudojbb*.[5] Figure 5.2 shows how predictor coverage and size depend on the SSP length as it is varied from 0 to 20; I used fully precise lifetime prediction, and singletons were retained in the predictor. Figure 5.2 plots the SSP length along the horizontal axis. For each plotted SSP value, I synthesized a predictor from the training trace. The predictor's coverage, i.e., the percentage of object allocations in the trace for which the predictor makes a prediction, is plotted along the vertical axis. Predictor coverage improves with increasing SSP length, as more information is available to disambiguate allocation contexts. However, this effect plateaus at an SSP of about length 10, suggesting that 10 is a sufficient SSP length. Figure 5.2 also shows the growth of predictor size, i.e., the number of entries, with increasing SSP length.

Figure 5.3 allows us to see the effect of removing singletons from the predictor. Notably, predictor coverage is almost unchanged, but predictor size is dramatically reduced. Excluding singletons reveals interesting dependencies among SSP length, predictor size, and coverage. There is a tradeoff between collisions and singletons—if the SSP is too short, too many objects collide; if it is too long, the SSP converts the entries into singletons and bloats predictor size. This effect is illustrated by the *perimeter* benchmark, shown in Figure 5.4, which has a maximum at around SSP length 8, and then decays to a plateau. The maximum divides the regime of collisions, on the left, and the regime of singletons, to the right.

I report results in object counts, rather than bytes, because I was interested in how well the predictor performs. Object counts are the natural unit for this consideration. However, bytes are often used in the garbage collector literature, and I collected these data as well

[5] Section 2.2 describes the benchmarks used in my study.

with similar results. In other words, object size is not significantly correlated with the ability to predict object lifetimes.

Testing and self prediction				
Benchmark	Command line	Objects allocated	Bytes allocated	Static Allocation sites
compress	-s100	24206	111807704	707
jess	-s100	5971861	203732580	1162
db	-s100	3234616	79433536	719
mpegaudio	-s100	39763	3579980	1867
mtrt	-s100	6538354	147199132	897
javac	-s100	7287024	228438896	1816
raytrace	-s100	6399963	142288572	992
jack	-s100	7150752	288865804	1184
pseudojbb	70000 transactions	8729665	259005968	1634
Training				
Benchmark	Command line	Objects allocated	Bytes allocated	Static sites
jess	-s1	125955	6978524	1143
javac	-s1	20457	2641064	1188
mtrt	-s1	328758	11300528	989
jack	-s1	512401	21911316	1183
pseudojbb	10000 transactions	2778857	103683020	1634

Table 5.1: Trace statistics. For each trace, the number of objects allocated (Column 3) and the total size of all allocated objects (Column 4) are given. Column 5 shows the number of allocation contexts; each site is counted only once, even if executed more than once, and sites that are not executed in these particular runs are not counted. The top section of the table lists the traces used for the self prediction study (Section 5.3). The bottom part of the table lists the training traces used in the true prediction study (Section 5.4); traces from the top section are reused for testing true prediction.

I report data on the SPECjvm98 and SPECjbb2000 benchmarks. I also collected data on the Java Olden benchmarks [18] (data not shown), but their smaller, synthetic nature produces outlier behavior. The SPEC benchmarks consist of useful “real world” programs (with the exception of *db*, a synthetic benchmark constructed to simulate database queries) and are intended to be representative of real applications run on Java virtual machines. They are written in a variety of programming styles and exercise the heap differently. For a detailed study of the individual benchmarks memory behavior, see Dieckmann and Holzle [33]. Table 2.1 describes the individual benchmarks, and Table 5.1 gives some

general run-time characteristics of the benchmarks.

I used the JikesRVM Java Virtual Machine version 2.0.3 from IBM. The specific configuration was OptBaseSemispace with the Merlin extensions [47]. This means that the optimizing compiler was used to compile the VM, and the baseline compiler compiled the benchmarks. The GC was the default semispace collector (though note that the generated trace is independent of the collector used).

5.3 Self Prediction

Self prediction tests the predictor using the same trace from which it was constructed. Thus, prediction accuracy is not an issue—if the predictor makes a prediction at all, it will be correct. Of interest are the tradeoffs among precision, coverage, and size. I report results for full-precision and logarithmic granularities, and consider the effects of including or excluding singletons from the predictors. Table 5.6 shows the results. For each benchmark, a set of preliminary runs was conducted to determine the optimal SSP value (shown in the columns labeled “SSP”). The optimal SSP value was determined separately for fully precise and for logarithmic cases.

5.3.1 Fully Precise Self Prediction

The results for fully precise self prediction (exact granularity) are shown in the left half of Table 5.6, for predictors with and without singletons.

Predictors including singletons

All of the benchmarks show some level of coverage, notably on the synthetic benchmark *db*. Greater than 50% coverage is achieved on *compress*, *mpegaudio*, *mtrt*, *pseudojbb*, and

jack, and more than 20% on the remaining two, *jess* and *javac*. The predictor achieved greater than 90% coverage only on *db*. From my experience with *db* and a set of smaller benchmarks not reported here, I believe that very high coverage numbers are not typical of realistic applications. These results suggest that fully precise prediction can achieve reasonable coverage on some but not all applications. As we will see, however, even moderate coverage may be beneficial (Section 5.7).

Predictors excluding singletons

When singleton entries are removed from the predictors (Columns 4-5 in Table 5.6), I expect coverage to drop, but I would like to know by how much.

Two benchmarks, *compress*, and *mpegaudio*, have predictors that include large numbers of singletons. The coverage for each drops roughly by half when singletons are excluded. For the rest, coverage drops by less than a percent, while the number of predictor entries shrinks dramatically: the smallest decrease is a 30% drop by *javac* while *db* is less than 1% its previous size. The average decrease in size is greater than 76%. Again, *db* could be anomalous because it is synthetic.

Although other studies of lifetimes, such as Harris [44], based prediction only on current method and type, I found no benchmarks for which type alone was sufficient to generate predictors with significant coverage. Benchmarks *jess* and *mtrt* needed at least the method containing the allocation site (an SSP of length one) to have significant coverage, and the rest needed more. Perhaps type is not required at all, but it disambiguates allocations in the rare case that different types are allocated at the same allocation point. Note that using a flow-sensitive notion of the stack, recording both the method and bytecode offset of each call.

In summary, the fully precise predictors cover a significant fraction of all benchmarks. With singletons excluded, the predictors still have significant coverage while decreasing

the number of entries by an average of 76%.

5.3.2 Logarithmic Granularity

As one might expect, the performance of the granulated predictors, also shown in Table 5.6, is better than the fully precise predictors, because it is an easier problem. Improvement in coverage ranges from *db*'s less than 1% to *javac*'s greater than 60%. The average is 7% improvement. The behavior of the granulated predictors when removing singletons is similar to the fully precise case. The only dramatic change is in *compress*, which behaves like the rest of the benchmarks; *mpegaudio* remains the outlier.

Because of their logarithmic bin size, the predictors are highly precise for the large number of short-lived objects. Granulated prediction has more immediate application than the exact case, because the training phase could in principle be performed more quickly and without relying on the Merlin algorithm.[6] I concentrate on the exact case, however, because I expect that information about exact behaviors will reveal new avenues for optimization.

5.3.3 Variations

I also studied a broader definition of predictor—one in which the predictor handled lifetimes that varied over each allocation as an arithmetic progression. For example, consider a loop that allocates an object of a particular type to a linked-list. When the loop exits, some computation is performed on the list and it is then collected. Each object in the loop has a lifetime that is less than its predecessor by a constant that is the size of the object.

[6]This could be accomplished using a generational collector in which collection for a generation is forced at time multiples of its previous generation. During training, for example, the second generation would be collected once for every two collections of the nursery for a logarithmic granularity with the base the size of the nursery. This allows for pretenuring type optimizations, but falls short of the "ideal" garbage collector.

To handle this, I need to predict using differences. More formally, the predictor entries become four-tuples <SSP, type, lifetime, increment>. The lifetime is not set during training unless the increment is found to be zero. It is updated every time an object is allocated by adding the increment. In this example, a predictor entry would be created if each object's lifetime differed by a constant increment in the order in which it was allocated. The predictor's lifetime would be initialized by the first object allocated during testing. Subsequent predictions would be made by adding the increment (which would be negative), to the lifetime. Note that absolute lifetime predictions cannot be made online until after the first object matching an entry has died.

Interestingly, this new predictor does not perform well for any of the benchmarks. I found only one benchmark for which it performs well, *em3d* from the Java Olden suite, in which non-constant allocations account for 36.39% of all predictions. This is not a good indicator of predictive strength, however, because almost all of these objects are singletons. For the other benchmarks, the increase in predictive ability averaged 1.1%.

Another variation is to use the optimizing compiler both for run-time and the benchmarks (the OptOptSemispace configuration). I tested this variation and the results are qualitatively similar to the earlier experiments (data not shown).

5.4 True Prediction

Barrett and Zorn found that true prediction accuracy is high for those benchmarks that have high coverage in self prediction and are not data-driven. I tested true prediction against a subset of the benchmarks to see if this correlation holds with higher levels of precision. I used *jess*, *javac*, *mtrt*, *jack*, and *pseudojbb*. I used the SSP lengths specified in Table 5.6 and included singletons. Although this was not an exhaustive study, it demonstrated that true prediction performs well, with results comparable to the Barrett and Zorn study, even

with the more stringent requirement of full precision.

Benchmark	Full Precision		Logarithmic	
	Coverage	Accuracy	Coverage	Accuracy
jess	1.22	99.88	1.34	99.95
javac	0.48	81.79	0.54	85.95
mtrt	0.18	99.28	0.24	99.77
jack	61.45	99.87	67.17	99.69
pseudojbb	57.09	99.99	63.20	99.85

Table 5.2: True prediction. Coverage and accuracy using predictors generated from a benchmark run using a smaller set of input for both fully-precise and logarithmic granularities against a separate, larger benchmark run. Coverage is the percentage of objects for which the system makes predictions, and accuracy is percentage of those objects for which my predicted lifetime was correct.

Results for the five examples are shown in Table 5.2. For both fully precise and logarithmic granularity, all of the predictors are highly accurate. For three of the benchmarks, however, the high accuracy comes at the price of coverage. Coverage is insignificant for *jess*, *javac*, and *mtrt*. The other benchmark predictors show considerable coverage. The difference in coverage is probably due to the degree the program is data-driven. For example, the training run of *jess* is quite different from its test run. In *pseudojbb*, the only difference is the length of the run. Although not exhaustive, these examples give evidence that highly precise, true prediction is possible for some applications and that when precise prediction is possible it is highly accurate.

5.5 Zero-Lifetime Objects

A zero-lifetime object is allocated and then dies before the next object is allocated. The ability to study object lifetimes with full precision allows the study of the behavior of zero-lifetime objects.

Table 5.3 shows the fraction of zero-lifetime objects generated by each benchmark and

the fraction of those that were able to predict using self prediction. Interestingly, many of the benchmarks allocate large numbers of zero-lifetime objects.

All of the SPEC benchmarks generate a large percentage of zero-lifetime objects, with *javac* allocating the least at 13%. I explore the potential consequences of this result in Section 5.7.

Benchmark	% of all objects	% predicted	% predicted of possible
compress	21.72	20.86	96.03
jess	39.63	19.96	50.36
db	45.06	45.01	99.97
mpegaudio	25.98	25.29	97.34
mtrt	40.01	33.37	83.39
javac	12.95	10.48	80.93
raytrace	41.30	29.57	71.60
jack	43.44	0.22	0.49
pseudojbb	20.82	18.75	90.04

Table 5.3: Fully precise zero lifetime self prediction: Column one lists the benchmark program; column 2 shows the fraction of zero-lifetime objects out of all dynamically allocated objects for that benchmark; column 3 shows the percentage of zero-lifetime objects predicted (coverage); and column 4 shows the prediction accuracy. SSP lengths are as described in Table 5.6.

5.6 Prediction and Object Types

In order to study how prediction results are affected by an object's type, I developed a simple classification of allocated objects according to their type. I used the following categories: application types, library types, and virtual machine types (since the virtual machine I use is written in Java itself). Library types are those classes belonging to the `java` hierarchy. VM classes are easily identified by their `VM` prefix. Application classes are all others.

As Table 5.7 shows, global coverage (defined as greater than 90%) was usually associ-

ated with high coverage of application types. This makes sense because application types dominate for most benchmarks. The exception, *db*, allocates many library types, which also have high coverage. A predictor's coverage depends on its ability to predict types resulting from application behavior, rather than the underlying mechanisms of the compiler or VM.

5.7 Exploiting Predictability: Towards an Ideal Collector

In the previous sections I demonstrated that for some programs we can accurately, and with full precision, predict the lifetimes of a large percentage of objects. In this section, I discuss a possible application of this technique: an improved memory management system.

I begin with an analysis of the maximum performance improvement that could be expected. To do this I make best-case assumptions; for example, assuming perfect accuracy. I finish with an analysis relaxing this assumption, allowing the collector to handle mispredicted lifetimes. Throughout, I ignore training times. Training is considered to be part of the development or installation procedure rather than part of normal execution.

5.7.1 A Limit Study

In the Introduction to this chapter I discussed the ideal garbage collector. The core idea behind my simulated allocator is to segment the heap into a nearly ideal collector for those objects whose lifetimes are predictable, and to use the rest of the heap in the traditional manner. The combined memory system is a hybrid of a standard collector and my nearly ideal collector. I refer to the combined system as the *death-ordered collector* (DOC). The nearly ideal heap is composed of two subspaces: the *Known-Lifetimes Space* (KLS) and the *Zero-Lifetimes Space* (ZLS). I assume that the heaps are of fixed size and compare

against a semispace collector to simplify the analysis.

The ZLS is simply a section of memory large enough to hold the largest object allocated there during a program execution. No accounting overhead is necessary because these objects have zero lifetime. They die before the next allocation, assuring that it is safe to overwrite them. One might assume that these are stack-allocated.

The KLS is more complicated. It is logically arranged as a series of buckets. Each bucket is stamped with its time-to-die, and sorted in order of the stamp, from earliest to latest. It is for that reason I refer to this heap as the death-ordered collector. Upon allocation into this heap, the time-to-death is calculated from the predicted lifetime and current time, a bucket is created for the allocation, inserted into the list, and then newly allocated memory is returned to the application. Collections are easier: the collector simply scans the list, returning buckets to the free-list, until it finds a bucket with a time-to-die greater than the current time. The efficiency of the death-ordered-heap is very high under my current assumptions—only allocation is slower due to the prediction during allocation.

Whether the hybrid arrangement is efficient depends on the sizes of the heaps and the amount of allocation within each. The sizes of the two heaps depend on their maximum occupancies, which we can measure. Likewise, we know the amount of allocation that would occur in each of the heaps.

The performance of this arrangement thus depends on the allocation characteristics of the application. To study how this would work in practice, I used self prediction to simulate a best-case scenario for several benchmarks that showed a significant (>50%) degree of self prediction. I set the sizes of the ZLS and KLS to the maximum values observed during training. Table 5.8 provides the absolute numbers of bytes allocated to the three spaces and Figure 5.5 shows the relative allocations.

Garbage collector performance can roughly be captured by two metrics: (1) the overall time overhead and (2) the distribution of pause times for collections and time spent in the

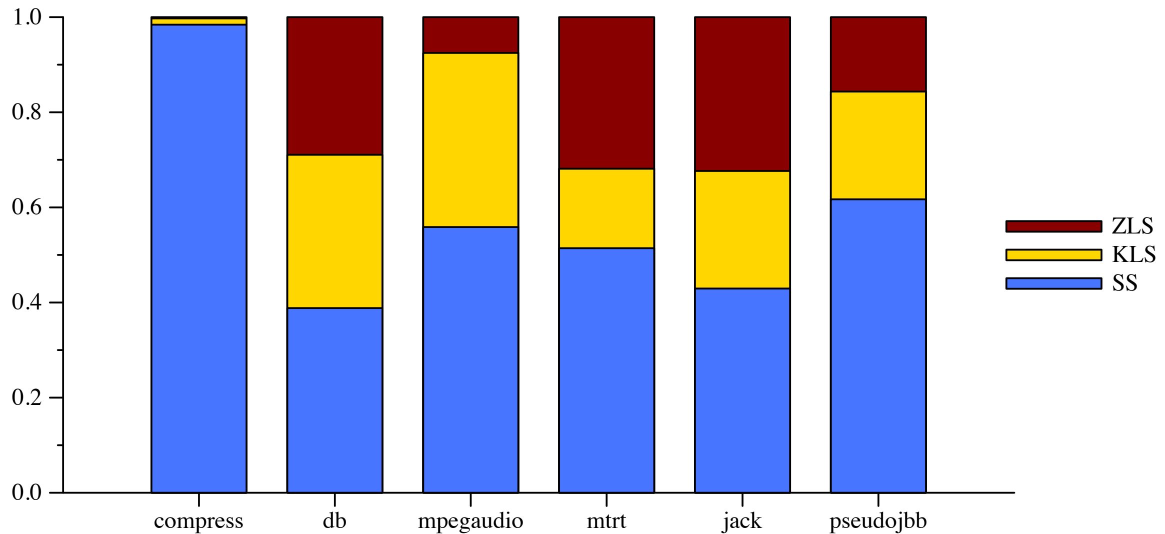

Figure 5.5: Death-Ordered Collector: The graph shows the fractional object volume of the different heaps in the simulated benchmarks. ZLS is the Zero-Lifetime Space. SS is the Semispace heap. KLS is the Known-Lifetimes Space.

application between collections. For my DOC system, the time between full collections is the number of bytes allocated before the semispace heap requires collection, because the traditional collector dominates ZLS and KLS maintenance. The time between full collections of the SS collector is increased by allocation to the ZLS and KLS heaps but is reduced due to its smaller size, as the total heap size, the combined size of ZLS, KLS, and SS, remains fixed.

I now quantify the performance of the death-ordered collector. Assume h is the total heap size, μ is the fraction of the total heap devoted to the KLS and ZLS, ε is the fraction of bytes allocated into the known and zero lifetimes heaps, and o the heap occupancy after a GC (the survival rate of the heap). The time between collections is simply the number of bytes that are free in a semispace heap after a collection. This is restated as the number

of bytes that can be allocated before another collection:

$$T_{DOC} = \frac{(1-\mu)(\frac{h}{2} - \frac{ho}{2})}{1-\varepsilon}$$

The division by two comes from the implementation of the semispace collector—a collection occurs when it is half full. The standard case of the single semispace heap occurs when μ and ε are 0:

$$T_{ss} = \frac{h}{2} - \frac{ho}{2}$$

The ratio of the DOC and the semispace equations is the factor of improvement over a single heap:

$$\frac{T_{DOC}}{T_{ss}} = \frac{1-\mu}{1-\varepsilon}$$

Therefore, the improvement in time between full collections is simply dependent on ε, which can be calculated during the simulation, and μ, which depends upon the chosen heap size. My results for a heap size of 50MB, Table 5.8, show improvement for all benchmarks.[7]

I now consider the total time overhead. In copying collectors, like semispace, a good first-order metric is the mark/cons ratio. This is the number of bytes copied by the collector divided by the total number of bytes allocated.

Table 5.4 shows bytes copied by the DOC heap divided by bytes copied by the single heap. I simulate the heap using sizes of 1.1, 2, and 4 times the minimum size necessary for the hybrid's semispace heap.[8] Here too, I show improvement for all benchmarks

[7] *compress* shows little improvement, but it is an outlier in terms of memory behavior. It tends to allocate large chunks of memory on startup and only free them on exit.

[8] Because the semispace heap reserves half its space at any time, it actually requires twice this amount of memory.

Benchmark	1.1	2	4
compress	0.84	0.7	0.85
db	0.64	0.35	0.09
mpegaudio	0.51	0.47	0.94
mtrt	0.56	0.49	0.43
jack	0.59	0.43	0.43
pseudojbb	0.63	0.69	0.64

Table 5.4: The ratio of bytes copied in the DOC system to the bytes copied in the semispace collector for heap sizes of 1.1, 2, and 4 times the minimum semispace heap size required by the DOC system. Smaller numbers are preferable.

Benchmark	1.1		2		4	
	DOC	SS	DOC	SS	DOC	SS
compress	0.49	0.59	0.41	0.59	0.09	0.1
db	2.49	3.92	0.25	0.71	0.02	0.25
mpegaudio	0.88	1.72	0.46	0.97	0.46	0.49
mtrt	1.8	3.24	0.29	0.6	0.11	0.25
jack	1.86	3.16	0.33	0.78	0.13	0.29
pseudojbb	3.84	6.1	0.56	0.81	0.18	0.29

Table 5.5: The mark/cons ratios for various heap sizes of the DOC and semispace collector.

(smaller is better), especially when heap sizes are small. Not only could the DOC achieve a significant reduction in copying cost (40% or more), but it would do so across a wide range of heap sizes, and for programs in which the baseline overhead of collection is high (mark/cons ratios as high as six, as shown in the "SS" columns of Table 5.5).

In summary, the DOC heap would both increase the time between allocations and decrease the total pause time. If it could be implemented efficiently, an idea I discuss in Section 7.2, the DOC heap has the potential to greatly increase garbage collection performance.

5.8 Related Work

Although most of the background and related work for my research was presented in Chapter 2, much of the related research for this chapter is specific only to it. Now that I have presented my experiments, results, and an analysis of the performance of a potential system, I present that work in this section.

There has been little study of Java's memory behavior outside the context of GC algorithms. The focus has been on studying collectors rather than how Java applications use memory. And, if we restrict ourselves to object lifetime prediction, there has been only a small amount of work for any language.

In one of the few studies of Java's allocation behavior, Dieckmann and Hölzle studied in detail the memory behavior of the SPECjvm98 benchmarks using a heap simulator [32, 33]. They found that more than 50% of the heap was used by non-references (primitives), and that alignment and extra header words expanded heap size significantly, since objects tended to be small. They confirmed the weak generational hypothesis for Java, though not as firmly as in other languages (up to 21% of all objects were still alive after 1 MB of allocation). This is the most in-depth study of the benchmarks' allocation and lifetime behavior, although a study of access behavior was reported by Shuf [92]. Focusing on garbage collectors, Fitzgerald and Tarditi [36] demonstrated that memory allocation behavior differs dramatically over a variety of Java benchmarks. They pointed out that performance would have improved by at least 15% if they had chosen the appropriate collector for the appropriate benchmark. They report that the most important choice is whether or not to use a generational collector and pay the associated penalty for the write barrier.

Lifetime prediction has almost always been studied in the context of pretenuring or similar schemes. These schemes rely on a training or profiling stage to learn lifetimes before they can be exploited. Static heuristics have not been used in published work to this

point, although Jump and Hardekopf found that objects that escape their thread are usually long-lived [61].

Cheng et al. [21] describe pretenuring in an ML compiler using a simple algorithm that associates allocation contexts with lifetime during profiling—sites that produce objects that survive one minor collection with 80% probability are pretenured.

As discussed earlier, my work has many similarities to Barrett and Zorn's "Using Lifetime Predictors to Improve Memory Allocation Performance" [11], which used a similar method to construct predictors. However, Barrett and Zorn used C applications so lifetimes were explicit. Their predictor also was binary; it predicted that objects were either short-lived or long-lived.

Cohn and Singh [25] revisited the results of Barrett and Zorn using decision trees based on the top n words of the stack, which includes function arguments, to classify short-lived and long-lived objects. They improved on Barrett and Zorn's results, but at the cost of computational expense because they used all the stack information. By contrast, my algorithm uses only the method identifier and bytecode offset.

Blackburn et al. [14] used coarse-grained prediction with three categories in Java, using the allocation site and lifetime as features to construct pretenuring advice for garbage collectors. They found they were able to reduce garbage collection times for several types of garbage collection algorithms.

Shuf et al. [91] decided that segregating objects by type rather than age, as in generational collection, was more promising. They found that object types that are allocated most frequently have short lifetimes. They then used the type as a prediction of short lifetime, dividing the heap into "prolific" (or short lifetime) and regular regions.

Seidl and Zorn [88, 89], sought to predict objects according to four categories: Highly referenced, short-lived, low referenced, and other. Their goal was to improve virtual memory behavior rather than cache performance as in [11]. Their prediction scheme was again

based on the stack; they emphasized that during profiling, it was important to choose the right depth of the stack predictor: too shallow is not predictive enough and too deep results in over-specialization.

Harris [44] studied pretenuring using only the current method signature and bytecode offset. He considered using the SSP, but decided it provided little information unless recursion is removed. He speculated that using the class hierarchy might be an easier and less expensive way to predict lifetimes, as related types usually have the same lifetime characteristics. His conclusion about the usefulness of SSP may have been a result of his methodology (he considered a maximum SSP length of 5), and the large granularity (short and long-lived objects).

Some studies used more information than just the stack and allocation site. These typically do not do pretenuring, which concentrates on where to put an allocation, and therefore needs a lifetime prediction at birth. Rather, these other methods focused on finding an efficient time to collect, and thus made relative predictions about deaths.

Cannarozzi et al. [19] used a single-threaded program model and kept track of the last stack frame that referenced an object. They observed that when the last reference is popped, objects in that frame are likely to be garbage.

For example, Hayes [45, 46], using simulation, examined which objects were entry or "key" objects into clusters of objects that die when the keyed object dies. For automatically choosing what objects are keyed, he suggested random selection, monitoring the stack for when pointers are popped, creating key objects, and doing processing during promotion in generational garbage collection. In effect, the keyed objects are used to sample the clusters.

Similarly, Hirzel, Diwan, and Hind [48] looked at connectivity in the heap to discover correlations among object lifetimes. They found that objects accessible from the stack have short lifetimes, objects accessible from globals are very long-lived, and objects con-

nected via pointers usually die at about the same time.

My own work resembles much of the work described here in its use of the allocation site and stack for constructing the predictor and in its reliance on a training (profiling) phase. My work extends this earlier work by increasing the precision of lifetime prediction, specifically the ability to make fully precise predictions. In addition, many of the earlier methods do not make specific predictions; indeed, some do not make predictions at all.

One most obvious application of my method is as a hinting system, which would identify objects that might be allocated on the stack instead of the heap. Stack allocation is cheaper than heap allocation and has no garbage collector overhead. Currently, the principle method of identifying such objects is through escape analysis. Determining which objects escape the stack is in general a difficult, costly analysis.[9] Although this can be performed statically, in Java it occurs at run time because the class file format has no way to encode this information. Therefore, it would be helpful to speed up the analysis by identifying objects that are likely to not escape the stack.

5.9 Discussion and Conclusions

Most GC algorithms are effective when their assumptions about lifetimes match the actual behavior of the applications, but beyond coarse-grained predictions such as pretenuring, they do little to "tune" themselves to applications. The ideal garbage collector would know the lifetime of every object at its birth. In this chapter, I have taken a step toward this goal by showing that for some applications it is feasible to predict object lifetimes to the byte (referred to as *fully precise* prediction). In addition, I showed how a memory system could

[9] A description of the performance costs of escape analysis can be found in Deutsch's *On the Complexity of Escape Analysis* [31] and a Java specific implementation in Choi's *Escape Analysis for Java* [22].

exploit this information to improve its performance.

It is remarkable that fully precise prediction works at all. Previous attempts at prediction used a much larger granularity, in the thousands of bytes. In particular, Barrett and Zorn used a two-class predictor with a division at the age of 32KB. It is not surprising that the predictor they described worked well, given that 75% of all objects lived to less than that age. Cohn and Singh's decision trees [25] worked very well at the cost of much greater computational complexity. Blackburn's pretenuring scheme [14], used a coarse granularity. The method described here is the first to attempt both high precision and efficient lifetime prediction, and it does so using a surprisingly simple approach. An area of future investigation is to consider other prediction heuristics and to test them on fully precise prediction. Because my accuracy is already so high, the goal here would be to increase coverage.

My results show that a significant percentage of all objects live for zero bytes, a result that required the use of exact traces. Because my predictors are able to cover zero-lifetime allocation contexts, the zero-lifetime results have clear applications in code optimization. Zero-lifetime object prediction could be used to guide stack escape analysis so that some objects are allocated on the stack instead of on the heap.

Object lifetime prediction could also be used as a hinting system, both for *where* an allocator should place an object and *when* the garbage collector should try to collect it. This would be a more general procedure than pretenuring, and it would support more sophisticated garbage collection algorithms, such as multiple-generation collectors and the Beltway collector [13].

Benchmark	Fully Precise					Logarithmic				
	incl. singletons		excluding singletons			incl. singletons		excluding singletons		
	size	coverage	size	coverage	SSP	size	coverage	size	coverage	SSP
compress	9133	67.78	792	33.33	10	9692	85.66	1381	84.84	10
jess	24999	23.40	3326	23.03	24	25873	34.87	4197	34.51	26
db	8847	90.36	73	89.91	3	9474	90.56	348	90.09	4
raytrace	14761	41.61	325	41.27	4	15509	42.36	514	41.97	4
javac	109357	28.63	75571	28.17	32	144844	45.72	111058	45.25	32
mpegaudio	17550	78.42	1931	39.67	8	18260	89.68	2704	51.12	9
mtrt	12490	50.11	1566	49.79	3	13167	50.92	306	50.53	3
jack	29542	61.25	14126	61.04	20	31659	66.30	14600	65.90	17
pseudojbb	32044	57.52	3861	57.20	14	33158	63.65	4861	63.23	14

Table 5.6: Self prediction results. The first two columns of fully precise and logarithmic granularity give results using predictors including singletons using an SSP of length 20, with two exceptions: *jess* and *javac*, for which I used the larger SSP value reported in the 5th column.

Benchmark	VM		Library		Application		% Predicted
	% Alloc.	% Pred.	% Alloc.	% Pred.	% Alloc.	% Total Pred.	
compress	37.97	63.63	11.89	53.41	50.14	74.33	67.78
jess	0.57	78.90	19.69	99.01	79.74	5.25	23.40
db	0.29	61.47	94.83	94.78	4.88	94.78	90.36
mpegaudio	42.48	70.51	10.16	66.57	47.37	88.05	78.42
mtrt	0.19	68.40	1.94	67.01	97.87	49.74	50.11
javac	15.91	5.86	26.54	32.21	57.54	33.28	28.63
raytrace	0.22	67.97	1.03	66.68	98.76	41.29	41.61
jack	3.22	96.94	48.26	42.99	48.52	77.05	61.25
pseudojbb	0.53	73.35	33.19	33.81	66.27	88.43	57.52

Table 5.7: Self prediction for three categories of objects according to object type. For each of the three categories of types (virtual machine, library, application), the percentage of total allocated objects that fall in the category is given, together with the percentage of objects in the category that are predicted. The rightmost column is the overall percentage of objects predicted (corresponding to the first column of Table 5.6).

Benchmark	Bytes Allocated			Minimum Size			DOC Allocator Results		
	ZLS	KLS	Semispace	ZLS	KLS	Semispace	$\mu \times 10^3$	ε	Improvement
compress	295868	1510568	111916176	4108	524036	8353592	10.32	0.016	1.01
db	23547452	26227396	31590624	4108	1238758	9239776	24.28	0.61	2.51
mpegaudio	525388	2557324	3902044	4108	524452	3213560	10.33	0.44	1.77
mtrt	47714408	25006584	76978460	4108	523500	8547516	10.31	0.49	1.92
jack	94581356	72275544	125578356	4108	524412	3476036	10.32	0.57	2.30
pseudojbb	44643628	64643588	176059172	4108	545820	28401300	10.74	0.38	1.60

Table 5.8: The bytes allocated to the different heaps and their maximum sizes, and with μ, ε, and factor of improvement based on a 50MB heap.

Chapter 6

Methods, Method Sequences, and other Variants

In Chapter 3, I examined the simplest feature of execution in the application of anomaly intrusion detection. This chapter extends that work in several directions. First, I increase the granularity of the feature by looking at the execution context of methods as they are invoked. Another way to state this is that in this chapter I describe a dynamic sandbox that observes method sequences. Next, a method for correlating anomalies is introduced, decreasing the number of false positives. Following that, I describe a scheme to filter selected method invocations from observation. This is important because for some applications the behavior I am interested in observing is a just a subset of the entire application. Finally I describe a way to sandbox individual threads instead of the application as a whole.

All of these variants of the Chapter 3 sandbox are tested in the context of fault detection (see Chapter 2). Fault detection requires a faulty program. Unlike the simple programs discussed earlier, a large, imperfect program is required to test my methods. The majority of this chapter examines both method invocation and method sequence behavior in the context of a large simulation of unmanned aerial vehicles (UAVs). Before the specific

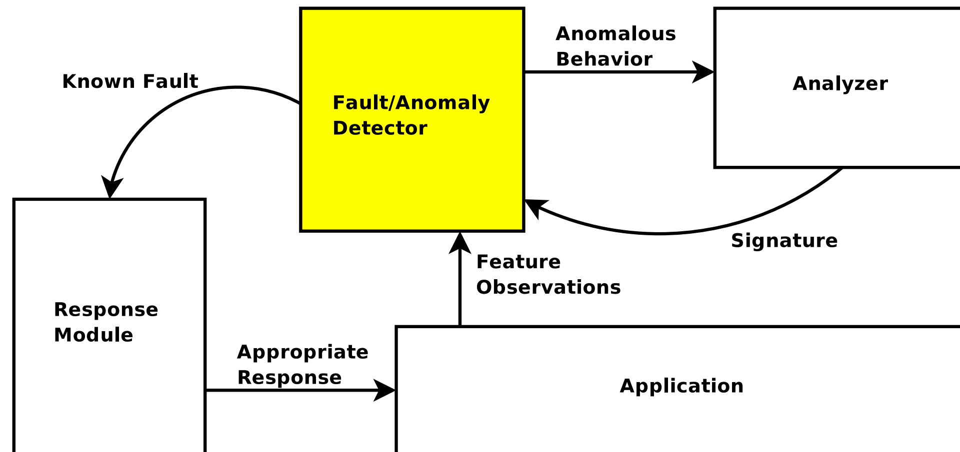

Figure 6.1: A fault tolerant system relying on anomaly detection. The anomaly detection system observes behavior and flags anomalies. Anomalous behavior is then analyzed and a signature for a specific fault is determined. That signature is used to flag future anomalies as known faults which incorporate specific responses. The system described here incorporates only the shaded box: the anomaly detector.

case study, however, I start the chapter by summarizing the application and examining the behavior of method invocation.

6.1 Motivating Application: Fault Detection

Software fault tolerance consists of fault detection and response. In this chapter I discuss an anomaly detector that is designed to detect faults. It is an expansion of the domain of dynamic sandboxing, from just the intersection of anomaly and intrusion detection to a general purpose system for fault detection (see Figure 2.1). The assumption is that behavior not observed during training indicates a fault state.

The system described below is not a complete fault tolerant system. Figure 6.1 depicts a complete system for fault tolerance. The system flags anomalies, but does not then determine if the system is in an error state, and if it is, what the proper response should be.

Method sequences were included as a feature for the fault tolerance application because simply looking for the presence of method invocations, as in Chapter 3, is unlikely to be sufficient. Faults, unlike intrusions, emerge from internal interactions. No code is injected so behavior is unlikely to dramatically change similarly to that of a compromised system. Thus, some context is required, and that context is provided by method invocation order. If a fault occurs, affecting program behavior, the change in program behavior should be evident in the order in which methods are called, even if the application calls no new methods.

A real application is required to test this hypothesis. I used the Metron UAV simulation. Before the case study, however, I present some data concerning benchmarks to illustrate typical behavior of method sequence profiles.

6.2 Benchmark Behavior

The SPEC JVM98 benchmarks are larger than the Olden benchmarks presented in the previous chapter. Around 1000 unique methods are invoked during a typical SPEC benchmark run, compared to about 225 for an Olden.

To provide context, I examine the sequence order of method inovocation. There are two ways to do this. First, one could look at the stream of method invocations as they are invoked. This is simplest to implement and fastest during execution if the analysis is not done within the same process as the VM. Unfortunately, this is unsuitable for multi-threaded applications, and therefore, I do not explore it in this case study. The second way is to examine the execution stack.

The top of the execution stack, or the stack string prefix (SSP), allows the capture of context.[1] The stack string prefix is a fixed length list of methods from the top of the stack

[1] In Somayaji's pH, this corresponds to the window size.

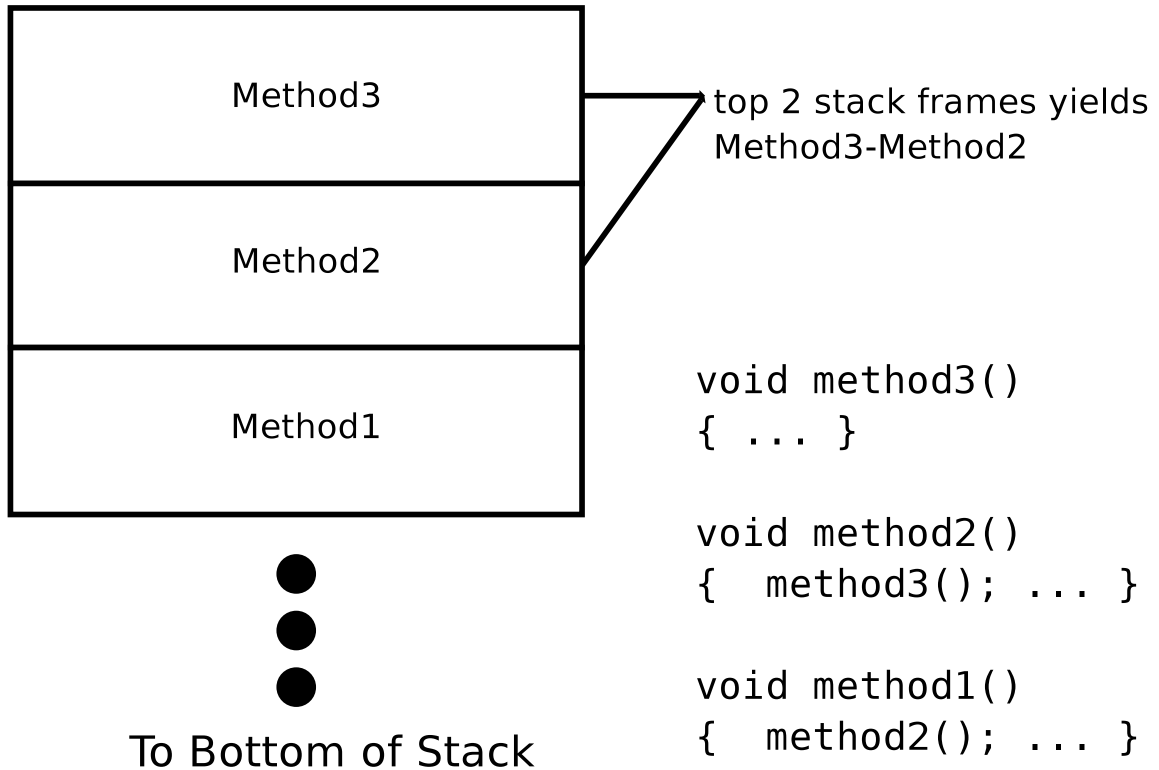

Figure 6.2: How the SSP prefix works. Stack frames are pushed on the top when a method is invoked. They are popped off when they exit. The stack string prefix of length 2 is the signature of the top 2 frames concatenated together: Method3-Method2.

(where the most recently invoked method resides) through the active method execution frames. For example, if Method1 invokes Method2 which invokes Method3, the SSP of length 2 during the execution of Method3 is Method3-Method2. See Figure 6.2 for a visual respresentation.

In Table 6.1, I present the number of unique SSPs for the SPEC JVM98 benchmarks during a standard run. The first column is equivalent to the numbers provided in the previous chapter. Profile sizes can potentially increase dramatically as the SSP length increases. The possible profile size is m^l, where m is the number of possible methods and l is the SSP length. Under the default security policy (in which all library methods

Benchmark	SSP Length					
	1	2	3	4	8	16
_201_compress	889	1644	2265	2854	4490	8892
_202_jess	1323	2743	4323	6022	11048	23464
_209_db	905	1731	2484	3247	5625	10898
_213_javac	1687	4810	11381	24517	167231	594712
_222_mtrt	1056	1960	3420	3420	5428	11910
_227_mpegaudio	1038	2089	3911	3911	6350	12468
_228_jack	1135	2398	5330	5330	16558	47592

Table 6.1: The number of unique stack string prefixes (SSPs) for various prefix lengths for the SPEC JVM98 benchmarks. An SSP length of 1 denotes methods without context, as in Chapter 3.

are available), the possible profile is larger than 38,000 for SSP=1 but expands rapidly to 1.45×10^8 for SSP=2 and 1.94×10^{73} if one looks 16 frames down. In practice, of course, the actual sandbox size is much smaller, and grows linearly instead of exponentially.

A similar calculation was carried out by Somayaji in calculating pH's profile sizes [97]. His eventual representation, look-ahead pairs, is on the order of several hundred thousand, depending on window size. Based on that example, one would expect that an SSP length of 2 produces a large enough space for fault detection. However, the proper length is an empirical calculation and may be dependent on application behavior. Consider *javac*, for example. Its behavior is vastly different from the other benchmarks. It is different because it is a compiler. Unlike the other benchmarks which operate as a loop doing calculations on an external set of data, compilers are constructed as large recursive case statements, creating a large possible state space. This indicates that a stable profile is hard to provide for larger SSP lengths for these types of programs. For other programs, however, a stable profile for larger SSPs may be achievable.

Benchmarks are small programs that do not provide faults. To test a fault detection system based on method sequences, a real application is required, and I therefore introduce

our test case: a complex simulation of unmanned aerial vehicles.

6.3 The Metron UAV simulation

Previous tests of dynamic sandboxing used either synthetic programs or standard benchmarks. Here I examine the Metron UAV simulation used extensively in the DARPA Taskable Agents Software Kit (TASK) program in which my lab participated. The program is a composition of two separate simulations. We find it useful because it is an examplar of a real, commercially developed program.

The Metron UAV application simulates an Unmanned Aerial Vehical (UAV) in two modes of operation: search and surveillance. In surveillance mode, the UAV is assigned to keep track of known targets. It does this by solving the travelling salesperson problem for its list of known target locations and exploring that circuit. If the target has moved beyond the sensor range of its last known location, the UAV performs a spiral movement until it is found. If it is not found, then the target is placed on a list of unknown targets for the search UAVs. In search mode, UAVs use a probability map of likely locations of targets to decide their paths. Once a target is found, it is assigned to one of the surveillance UAVs to maintain. UAVs can be transferred from one mode to the other depending on the ratio of found to missing targets.

The simulation consists of two applications linked together. The surveillance and search modes are complete simulations on their own and share little code beyond primitives for thread synchonization. Such synchronization is necessary because the combined simulation is aggresively multi-threaded, especially on the search side. Because of this, runs are not replicable due to race conditions. Figure 6.3 shows a screenshot of the simulation in GUI mode.

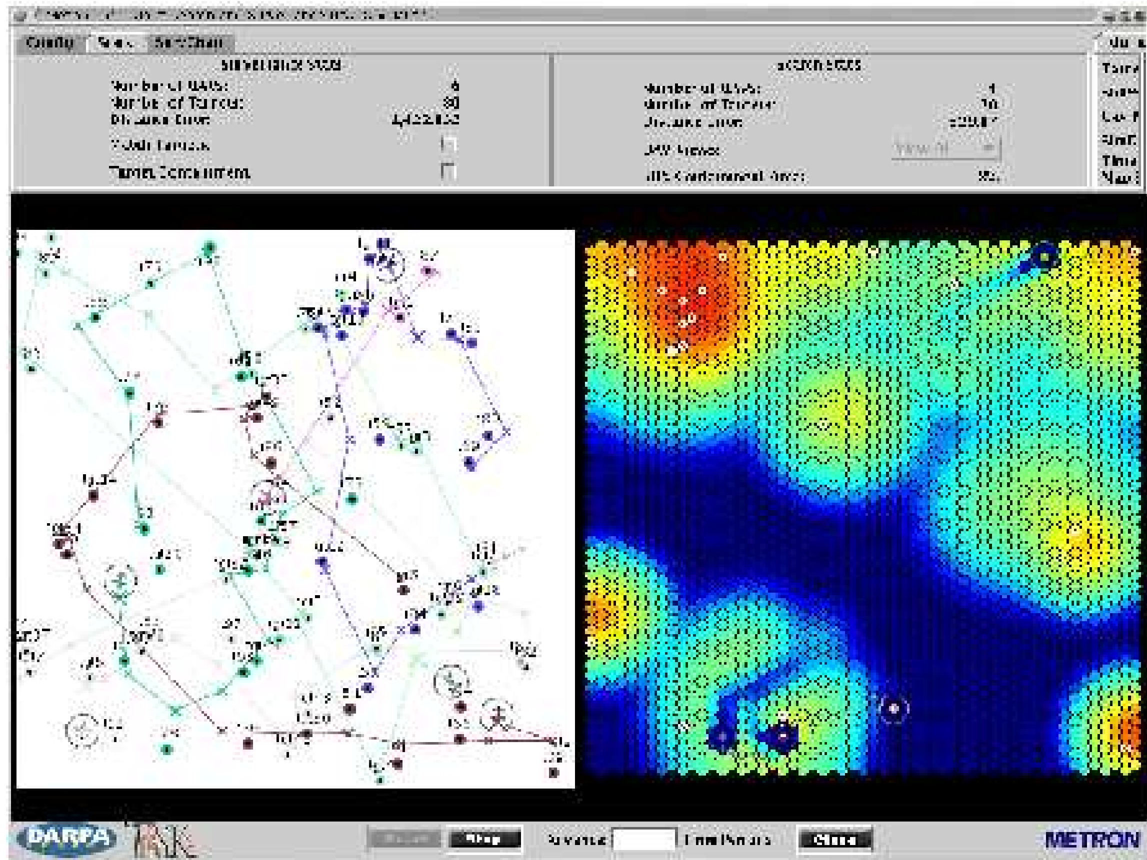

Figure 6.3: A screenshot of the Metron UAV simulation with the graphical interface. The surveillance sim is on the left and the search sim is on the right.

6.4 Scenarios

Faults, called anomaly scenarios, were added to the Metron UAV simulation by programming hooks directly into the simulation itself. These were as unobtrusive as possible. The scenarios are controlled by the anomaly scenario subsystem, a package I added to the Metron UAV simulation. On simulation initialization, the subsystem is alerted to the presence of an anomaly scenario description file. If one exists, the subsystem is switched on and the description file is parsed. From then on, the subsystem is entered only if the

subsystem is on and the various hooks at specific points in the simulation are exercised. These points are located at configuration and specific points during each timestep of the simulation.

There are two basic types of anomaly scenarios: UAV and sensor. In the UAV scenarios, the total behavior of the UAV is changed. In the sensor scenarios, the anomaly framework merely changes the values provided by the UAVs sensors. Six of the scenarios change the behavior of one UAV. In only one scenario (random_movement_all) does the behavior of all of the UAVs change.

Descriptions of the anomaly scenarios follow:

- **The Null Scenario (nop)**
  This scenario loads the anomaly scenario subsystem and nothing more. This minimal amount of activity is useful for comparison to the other scenarios.

- **UAV Faults**
  These scenarios simulate faults in specific UAVs.

  - **A Security Fault (exec_process)**
    An external process, in this case an x-terminal, is launched in this scenario. This scenario simulates a successful intrusion into the UAV.

  - **A UAV Crash (crash_uav)**
    This scenario contains the most intrusive code in the simulation. It removes a UAV in a quick and unclean way.

  - **Random Movement in all UAVs (random_movement_all)**
    All UAVs randomly search the environment instead of moving up the probability gradient.

  - **Random Movement in one UAV (random_movement_one)**

This scenario differs from the previous one in that only one UAV roams randomly.

- **Sensor Faults**

  These scenarios demonstrate sensor faults.

  - **Search UAV that never finds targets (search_fnr)**

    The value that determines the probability of a UAV finding a target in search mode is set to 0.

  - **Surveillance UAV that always reports a target (always_yes)**

    The probability of inaccurately reporting a target is found is set to 1.

  - **Surveillance UAV that never finds its target (no_true_positives)**

    The probability of a UAV in surveillance mode actually finding a target is set to 0.

Before running the experiments, I predicted that some of these scenarios would be easier to detect than others. This was born out in the experiments, with a few surprises. Before we consider the results, here is the order of difficulty, from easiest to hardest, that I expected: exec_process, crash_uav, random_movement_all, always_yes, random_movement_one, search_fnr, and finally no_true_positives. Scenario exec_process should be the easiest to detect because it initiates a complicated cascade of method invocations that ends in a child process. Next is crash_uav, which forces the removal of a UAV in an unclean manner. The standard way to remove a UAV from the simulation is a complicated affair that requires it to message the rest of the sim and launch a helper thread to distribute various goals among the remaining UAV before cleaning up after itself. Random_movement_all causes a radical change in behavior in all the search UAVs. Always_yes, and random_movement_one, search_fnr each affect the behavior of a single UAV in a radical way, and ought to be detectable. Detecting no_true_positives was predicted to be difficult because its behavior could realistically model an unlucky UAV.

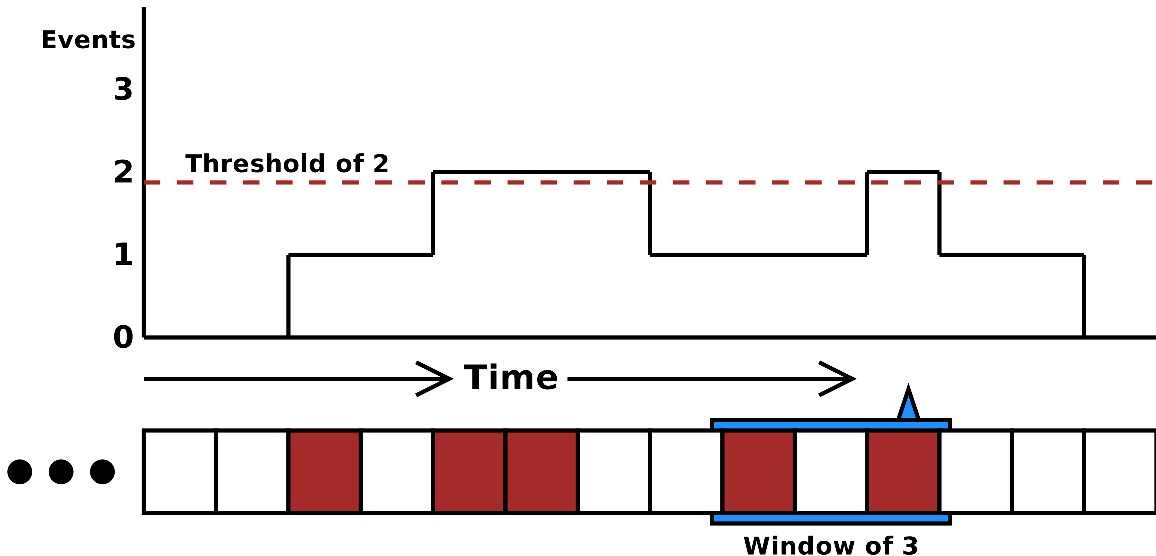

Figure 6.4: Reducing false positives. An anomaly is triggered only if two events are correlated in time. In this example, an anomaly is defined as 2 events that occur within a 3 method invocation span of time.

6.5 Dynamic Sandboxing

In Chapter 3, I defined a dynamic sandbox that allowed only methods invoked during training to be invoked during sandboxing. Here, I introduce several variants of dynamic sandboxing. First I extend the execution feature from methods to method sequences. Other additions are required to deal with the change.

The first is a variant to extend the analysis beyond single anomaly events. This allows us to control false positives. I introduce two new parameters: the invocation-window, and the event-threshold. Figure 6.5 diagrams how this works during execution. The invocation-window specifies a number of method invocations. The threshold determines the number of anomaly faults within the event-threshold that constitute an actual anomaly event.[2] In

[2]This is similar to the *locality frame* in pH, Somayaji's IDS that analyzes system call information.

the previous chapter these parameters are both implicitly set to one. Insisting that events be correlated in time produces a higher threshold for triggering an anomaly, suppressing false positives. It is hoped that true positives produce a series of events, pushing the event count past the threshold and triggering an anomaly.

Another variant involves monitoring individual threads instead of the application as a whole. Although the previous variants are inherently thread specific, the number of total anomalies is a global value. We can instead count anomalies separately for each thread, labelling a *thread* as anomalous instead of the application. These variants can be combined.

A final variant allows us to monitor only certain methods. In the analysis to follow, the goal is to find faults within the UAVs, not the simulation as a whole. Therefore, the system can ignore all method invocations that originate outside UAV code.

Filtering method invocations may be useful beyond the Metron simulation. DEEs in many ways act as operating systems, and separate threads and classes can act as different, separate applications. This filtering method allows selective sandboxing.

6.6 Implementation

The dynamic sandbox variations using the stack string prefix rely on determining the exact state of the stack at any time. A system that received only method invocation events would have difficulty if it could not distinguish between different threads.[3] Access to VM internals allows direct access to the state of the stack, but this research was done in simulation using an external monitor. The solution was to access VM internals through Sun's semi-standard debugging interface (Java Debug Interface)[76].[4] A short application

[3]This is similar to the difficulty pH has with programs using user-space threading.
[4]Unfortunately, this interface is substantially different in Java 1.5.

produces a trace which can then be analyzed in real-time by the sandboxing engine or saved to a file.

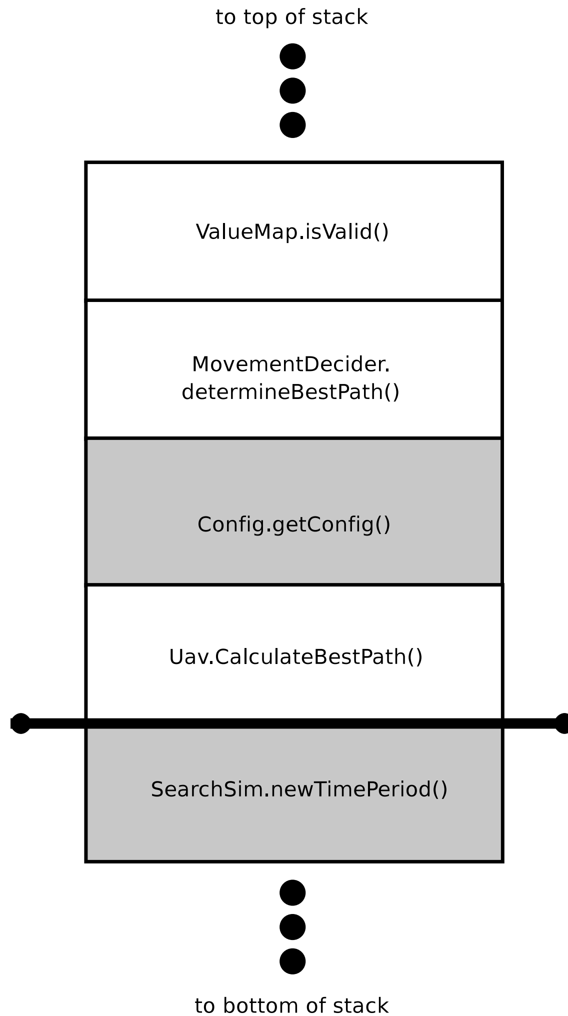

Figure 6.5: Filtering method invocations. No method invocations are logged until a method from the trigger list is invoked (Uav.calculateBestPath in this example). Config.getConfig() is on this ignore list and is ignored. The other methods are recorded because they were invoked within a method on the trigger list and are not on the ignore list. Thus the stack visible to the analysis engine is, from top to bottom: *ValueMap.isValid(), MovementDecider.determineBestPath(), Uav.CalculateBestPath()*. All methods invoked above SearchSim.newTimePeriod will continue to be recorded until the stack falls below that level. This example is not taken from the simulation code.

The trace consists of method invocation event records. Each record contains whether the event is a method entry or exit, the thread identifier, the class and method signature, and the stack height. The stack height is required for us to filter out simulation specific method calls. For one set of experiments method invocations from the simulation are filtered out—only method invocations that originate within UAVs are examined. The straightforward way to do this is to record method entries and exits. Unfortunately, this is not possible in Java because methods that exit by exceptions are not recorded. This is why the stack height is part of the trace record. The stack height is recorded whenever the system enters a method belonging to a UAV. These classes are called triggers.[5] All method invocations are then recorded, unless they are on an ignore list, until the stack height falls below the original entrance height. Only the anomaly scenario subsystem, consisting of only the ds.Config class, was ignored. Figure 6.5 demonstrates how this is calculated in the prototype. For a description of a more efficient implementation integrated with the VM, see Section 7.2.

6.7 Experiments and Discussion

I examined several variants of dynamic sandboxing: stack-based, per-thread, and variations of window and threshold size. Each variant includes results for filtering out simulation methods, labelled "All simulation method invocations", and with filtered UAV only methods "UAV class originated simulation invocations". I present results only for windows and thresholds of 1-1 and 2-2. I conducted experiments with windows of 1 to 8, 16, 32, 64, 128, 512 and thresholds of 1 to 4, 8 and 16 but increasing the window or threshold past 2 did not improve results; the best results were obtained with parameters of 1-1 or

[5]These classes are searchsim.UAV, searchsim.SearchBot, searchsim.Ugs, searchsim.UavInfo, searchsim.UgsInfo, uavsim.bot.UavBot, uavsim.bot.UavBrain, and uavsim.bot.UavSensor.

2-2.[6]

Differences in training were also examined. Columns labeled "Separate" contain averages of 25 runs, containing the cross of 5 profiles with 5 sandboxes. "Combined" runs use a sandbox profile containing the union of the 5 sandboxes with the separate runs. This is not difficult to do since profiles, like all the dynamic sandboxes previously described, are unordered sets of method sequences.

The tables include the ratio of events of anomaly scenarios to nop. This is necessary to give a basis as to what is truly anomalous. Without nop, it would be difficult to establish the number of events that define a scenario as anomalous. The nop scenario gives us a baseline for the number of false positives that should be expected. If significantly more anomalies are observed than that number we can conclude that the scenario is truly anomalous. Thus, ratios close to or below one indicate an inability to identify the scenario as anomalous. Ratios far greater than one mean that they can be identified easily.

6.7.1 Sandboxing with SSP=1

Sandboxes with an SSP=1 are similar to the sandbox presented Chapter 3. The difference is the addition of the window and threshold parameters. The basic sandbox is tested when windows and thresholds are 1.

Table 6.2 shows the results of experiments using this sandbox. The non-filtered experiments exhibit unimpressive results. In both the combined and separate sets of experiments, the non-filtered case only reliably identifies random_movement_all as anomalous out of the seven anomaly scenarios. The experiments examining only method invocations originating within the UAV class instances (the filtered experiments, shown in the lower half of

[6]Given an SSP length, a window and threshold value can be determined that identifies each anomaly routinely. These values are different for each scenario, however. As a whole, only 1-1 and 2-2 produced consistently good results.

All simulation method invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	2397	1	3054	1
exec_process	2503	1.0	3230	1.0
crash_uav	2210	0.9	2849	0.9
random_movement_all	6931	2.9	7414	2.5
random_movement_one	3861	1.6	4430	1.5
search_fnr	2399	1.0	3138	1.0
always_yes	2397	1.0	3002	1.0
no_true_positives	2397	1.0	2898	1.0
UAV class originated simulation invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	0	1	662	1
exec_process	54	∞	785	1.2
crash_uav	37	∞	680	1.0
random_movement_all	7500	∞	7995	12
random_movement_one	417	∞	1265	1.9
search_fnr	0	1	742	1.1
always_yes	0	1	608	0.9
no_true_positives	0	1	509	0.8

Table 6.2: Data from SSP=1 (non-stack) based detection of anomalies using a window and threshold of 1.

the table) show somewhat different behavior. In the cases not using a combined sandbox profile, the results are similar to the previous case. However, four scenarios are identified as anomalous using combined sandbox profiles. These scenarios, crash_uav, exec_process, random_movement_all, and random_movement_one, are UAV fault scenarios.

The total number of events for nop is an indicator of the suitability of the sandbox profile. A large average number of events (hundreds), indicates that the sandbox does not embody enough normal behavior to distinguish between nop and the anomaly scenarios. This accounts for the results of the experiments using separate sandboxes.

Table 6.3 shows the data corresponding to Table 6.2 with different values for the window and threshold. The results of these experiments are better than in the 1 and 1 case with

All simulation method invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	0	1	3.3	1
exec_process	79	∞	86.6	26
crash_uav	4.2	∞	12.7	3.8
random_movement_all	0	0	18.6	5.6
random_movement_one	0	0	4.2	1.3
search_fnr	0	0	4.7	1.4
always_yes	0	0	2.8	0.8
no_true_positives	0	0	6.6	2

UAV class originated simulation invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	0	1	4	1
exec_process	38	∞	47	11.9
crash_uav	23	∞	32	8.2
random_movement_all	1512	∞	1539	389
random_movement_one	0	0	144	36
search_fnr	0	0	5.3	1.3
always_yes	0	0	3.3	0.8
no_true_positives	0	0	7	1.8

Table 6.3: Data from SSP=1 (non-stack) based detection of anomalies using a window and threshold of 2.

the exception of the quadrant using filtered results and combined sandbox profiles (lower left). There are no longer problems with excessive numbers of events (false positives) for nop. Interestingly, instead of too little behavior embodied in the profile, in this case the opposite occurs—the sandbox profile is too inclusive for the combined cases. Events appear only for crash_uav and exec_process when looking at all invocations, and additionally random_movement_all in the filtered case.

When using sandboxes based on individual runs, an average number of 3.3 and 4 events are observed for non-filtered and filtered nop, respectively, compared to the hundreds and thousands seen in Table 6.3. The system is able to classify with ease crash_uav, exec_process, and random_movement_all for both the filtered and unfiltered experiments.

The ratios for no_true_positives and search_fnr, while greater than one, are not large enough to consistently determine the run as anomalous. The data from individual runs reveals bi-modal behavior for those scenarios. The sandbox detects them on some occasions but on others they produce few anomalies. In all cases, the sandbox was unable to identify always_yes.

All simulation method invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	6309	1	16474	1
exec_process	8279	1.3	18604	1.1
crash_uav	5952	0.9	10253	0.6
random_movement_all	26791	4.2	56638	3.4
random_movement_one	12999	2.06	31742	1.9
search_fnr	6342	1.0	7913	0.5
always_yes	6225	1.0	16286	1.0
no_true_positives	6110	0.9	8722	1.1
UAV class-originated simulation invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	3138	1	357784	1
exec_process	5150	1.6	339538	0.95
crash_uav	2905	0.9	298886	0.83
random_movement_all	8850	2.8	312503	0.87
random_movement_one	23689	7.5	360540	1.01
search_fnr	3258	1.0	379137	1.06
always_yes	3040	1.0	344980	0.96
no_true_positives	3016	1.0	338906	0.95

Table 6.4: Data from SSP=2 detection of anomalies using a window and threshold of 1.

To summarize, four out of the seven anomaly scenarios are detected by the dynamic sandbox. Filtering out invocations that did not originate within UAV instances improves our ability to identify them. False positives, defined here as large numbers of events for nop, are a problem when using classical dynamic sandboxing (windows and thresholds of 1). The "sweet spot" used windows and thresholds of 2 with separate sandboxes and filtering.

All simulation method invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	2498	1.0	11960	1.0
exec_process	2604	1.0	12162	1.0
crash_uav	2277	0.9	5894	0.5
random_movement_all	12927	5.2	42709	3.6
random_movement_one	5359	2.1	23496	1.9
search_fnr	2415	1.0	3213	0.3
always_yes	2506	1.0	11919	1.0
no_true_positives	2437	1.0	14680	1.2
UAV class originated simulation invocations				
	Combined		Separate	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	4	1.0	280932	1
exec_process	133	33	264214	0.94
crash_uav	91	23	233342	0.83
random_movement_all	1883	470	245534	0.87
random_movement_one	10497	2624	280520	1.00
search_fnr	1.4	0.4	297580	1.06
always_yes	1.4	0.4	270487	0.96
no_true_positives	4.6	1.2	265924	0.95

Table 6.5: Data from SSP=2 based detection of anomalies using a window and threshold of 2.

6.7.2 Sandboxing with SSP=2

Next, I present the results of experiments using the variant of dynamic sandboxing employing an SSP with length 2. Table 6.4 contains the results using a window and threshold of 1. They are not an improvement over the SSP=1 case. It is clear that not enough normal behavior is embodied in the sandbox profile—the smallest average number of events for nop was 3138. Only the random movement scenarios produced the large number of events necessary to discriminate between themselves and nop. This is unexpected because the changes made to the sim for nop could not have introduced the novelty for the filtered run in a conventional way. It is likely that the modifications in nop changed the usual timing of threads, and consequently the order of method invocations. Thus, the UAV sim exhibited

different behavior, even when no configuration changes were made.

All simulation method invocations				
	SSP=1		SSP=2	
Benchmark	1–1	2–2	1–1	2–2
exec_process		✓		
crash_uav		✓		
random_movement_all	✓		✓	✓
random_movement_one			✓	✓
search_fnr				
always_yes				
no_true_positives				

UAV class originated simulation invocations				
	SSP=1		SSP=2	
Benchmark	1–1	2–2	1–1	2–2
exec_process	✓	✓		✓
crash_uav	✓	✓		✓
random_movement_all	✓	✓	✓	✓
random_movement_one	✓		✓	✓
search_fnr				
always_yes				
no_true_positives				

Table 6.6: Sensitivity of variants using combined training data. A checkmark indicates that the system reliably identified the scenario as anomalous.

The large number of false positives in the 1-1 case suggests that correlating events in time might increase the signal. Using a threshold and window of 2 decreases the number of events, as Table 6.5 clearly shows. There is still a problem with false positives for three of the quadrants, however. The lower left quadrant shows a good signal for the four UAV faults but not for the sensor faults.

Table 6.6 summarizes the results of the variants that use combined sandbox profiles from several runs for training data. The results of the experiments with separate training runs are not shown because the combined sandbox results are better. The two sandboxing techniques (SSPs of 1 or 2) generally show equal results. Both are able to detect the four

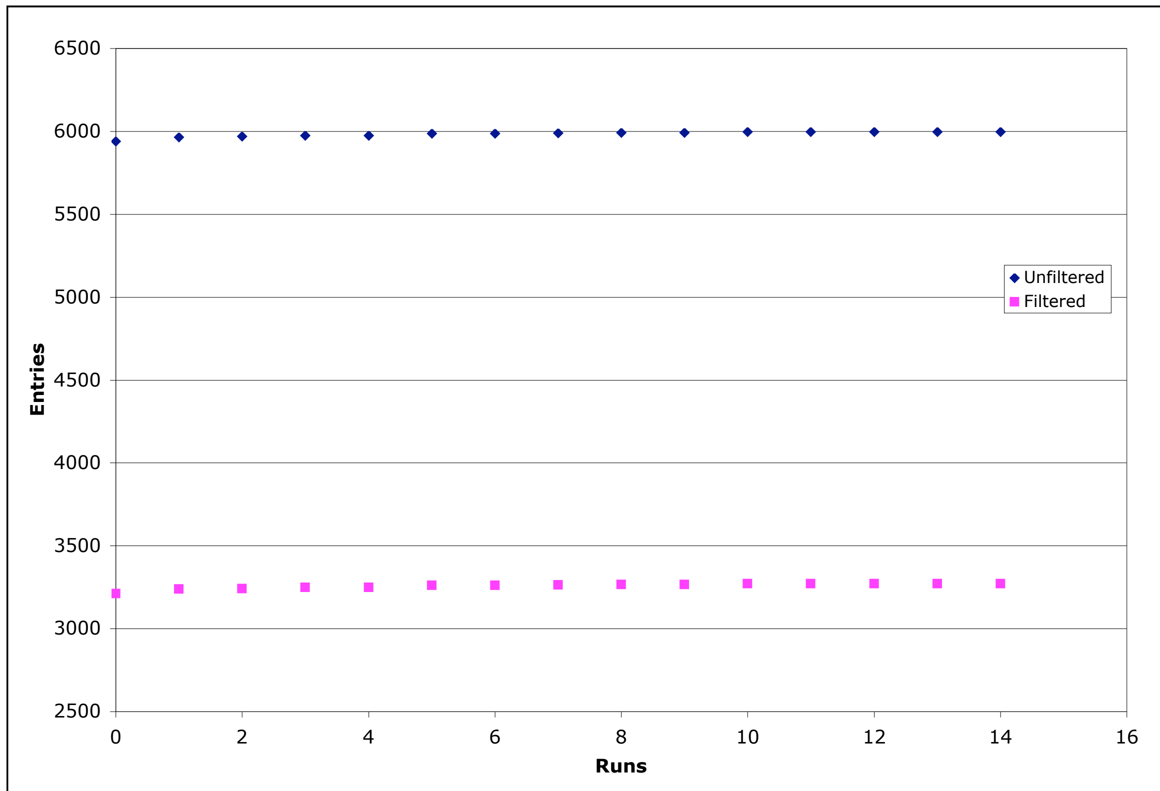

Figure 6.6: Growth in profile size over repeated training. The profiles grow very slowly. Sandbox size, however, is only loosely related to the number of events produced during a simulation using the sandbox.

UAV faults, but show less sensitivity for the sensor faults.

The table also shows that the effective parameter settings are predictable. Both are most effective when extraneous method invocations are filtered out. However, sandboxes with an SSP length of 1 are most effective when events are not filtered, while sandboxes using the stack are most effective when they are filtered. This makes sense. In the first case, false positives are not a problem, so filtering filters out true positives. In the second, the filtering does helps filter out false positives. Unfortunately, a longer SSP does not seem to increase the sensitivity of the dynamic sandbox.

The primary function of using the SSP is to increase the granularity of the sandbox

profile. The space of the sandbox becomes the number of methods available to the application raised to the length of the SSP. For classical dynamic sandboxing, the SSP length is one and the the number of methods available is about 3000. In my experiments, using a window of 2, this computes to 9 million. In practice the growth in observed behavior behaves similarly to the benchmarks. It is much more linear: sandbox sizes grow from 1500 to 3200 entries in the unfiltered case and 2000 to 6000 in the filtered case. The larger number of events should therefore be expected.

We would generally expect the increase in granularity to require more training. To test this expectation, sandboxes were constructed from 15 separate runs. Figure 6.7.2 shows the growth in sandbox size over the runs for both the filtered and unfiltered cases. The sandbox grows very slowly from runs 1 to 8 and stops after run 9. The total growth is less than 2% for both. When this sandbox was tested against anomaly scenarios, results were qualitatively similar. Therefore, because extra training neither increases the size of the profile nor improves results, this test indicates that insufficient training time is responsible for the false positives.

From the differences in the “Combined” and “Separate” columns it is clear that even small differences in sandbox size can have large effects on the number of anomalous events. The results indicate two possibilties: much longer training times are required, or that the minimal code changes to nop produce idiosyncratic events even when all first order effects are removed. The first would be unfortunate, because few would want to use systems that take many times longer to train than to use. The second is perhaps the more likely, but proof would require a study of the application’s race behavior.

The factor of “raciness” in the system is not likely to be a problem with other large applications. Races occur when the relative order of method invocations between threads can affect the behavior of a system. This means that different runs produce different results, even with identical initial conditions. The sandbox profile is therefore incomplete. Slight changes to the code, as with nop, can change the usual timing behavior, leading to

even more novelty. This unpredictability is very unusual in an application and is usually avoided by design. Races are seen as bugs, not features. If the sandbox is indeed influenced by the races it is, in a sense, properly detecting faults. There are several anomaly detection systems aimed at detecting nothing but race conditions [86, 23].

Moving from parameter settings to the individual scenarios, one might question, "Why are the four UAV faults detectable and the sensor faults not?" For the most part, the initial analysis of the scenarios holds true. The sensor faults are fundamentally harder to detect. Scenarios crash_uav and exec_process cause a large amount of disruption to the system over a short period of time. The random movement scenarios cause UAVs in search to behave in a fashion that would be visually odd to one watching the GUI version over the entire course of the simulation. The three anomaly scenarios the sandbox cannot detect affect one UAV in a particular mode over the course of its run.

6.7.3 Per-Thread Sandboxing

The next variation I implemented was per-thread sandboxing. Instead of counting a global number of anomalies for each application, the dynamic sandbox keeps a count for each thread. This makes sense because in the Metron UAV sim many anomalies affect only one UAV. This is not a custom sandbox for each thread type, it is merely using the same profile independently for each thread.

To test it, I gathered both more and larger traces (2.5 times the usual run length). Because of the longer traces, combined sandboxes were not used. Table 6.7 shows the results using a window and threshold of 1, and Table 6.8 is the equivalent with a window and threshold of 2. For many of the parameter settings that false-positives still wash out the signal. The results are markedly better for some configurations, however. In particular, windows and thresholds of 2 produce at least a weak signal for all the anomalies except in

All simulation method invocations				
	SSP of 1		SSP of 2	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	2728	1	5559	1
exec_process	2738	1.0	8142	1.5
crash_uav	2499	0.9	5080	0.9
random_movement_all	2741	1.0	5557	1.0
random_movement_one	2743	1.0	34806	6.4
search_fnr	2744	1.0	5513	1.0
always_yes	2676	1.0	5560	1.0
no_true_positives	2730	1.0	5561	1.0
UAV class originated simulation invocations				
	SSP of 1		SSP of 2	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	3.9	1	2526	1
exec_process	37.6	9.6	5098	2.0
crash_uav	42.6	10.9	2300	0.9
random_movement_all	1258.8	321.9	3756	1.5
random_movement_one	731	187.1	35517	14.1
search_fnr	16.6	4.2	2550	1.0
always_yes	5.4	1.4	2526	1.0
no_true_positives	6.9	1.8	2527	1.0

Table 6.7: Per-thread sandboxing for a window and threshold of 1. Events reflect the average number of events in the maximum thread of each scenario.

the unfiltered case with an SSP of 2.[7]

My results are strongly affected by the highly multi-threaded nature of the simulation. I was aware that the sim was multi-threaded before I started the analysis and took some steps to alleviate those complications (see Section 6.6). I did not anticipate, however, that the sim would generate hundreds or thousands of threads during a standard run. When only one thread is anomalous, its signal is washed out by the hundreds of other threads that appear normal.

The Metron UAV simulation can be seen as worst-case test of dynamic sandboxing.

[7] The individual traces indicate the system does not identify all cases of always_yes and no_true_positives.

All simulation method invocations				
	SSP of 1		SSP of 2	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	1.2	1	2752	1
exec_process	15.9	13.9	2823	1.0
crash_uav	3.5	6.0	2512	0.9
random_movement_all	10.7	9.3	2751	1.0
random_movement_one	4.8	4.1	2755	1.0
search_fnr	3.0	2.6	2729	1.0
always_yes	2.8	2.5	2752	1.0
no_true_positives	4.1	3.5	2753	1.0
UAV class originated simulation invocations				
	SSP of 1		SSP of 2	
Scenario	Events	ratio/nop	Events	ratio/nop
nop	1.2	1	6.3	1
exec_process	17.3	14.2	64.9	10.3
crash_uav	10.9	8.9	62.9	10.0
random_movement_all	266.8	218.7	1507	239
random_movement_one	147.4	120.8	941	149
search_fnr	7.6	6.2	27.8	4.4
always_yes	2.7	2.2	10.0	1.6
no_true_positives	4.0	3.2	12.9	2.0

Table 6.8: Per-thread sandboxing for a window and threshold of 2. Events reflect the average number of events in the maximum thread of each scenario.

Earlier versions of the Metron application used a single-threaded model and the behavior was quite dissimilar. My experience with benchmarks and the earlier version of the simulation led me to expect low false positives and predictable behavior. It was only the extremely varied behavior of this simulation which required per-thread sandboxing.

6.8 Intrusion Detection

Since the dynamic sandbox of Chapter 3 stops the four intrusions introduced earlier it is clear that these variations would stop them as well. However, adding context to the sandbox might prevent some types of intrusions that try to avoid a simpler sandbox.

StrangeBrew, for example, could be reimplemented to introduce no new methods. Instead of adding help methods, it could simply add its code directly to the existing constructors. The simple sandbox would not be capable of stopping the virus if the boxed application also used the common file I/O methods also used by the virus. A dynamic sandbox using a SSP length of 2 would only fail to identify the intrusion if the infected constructor itself used those file I/O methods, a much more unlikely situation.

Generalizing from the example, adding context by increasing the number of observed stack frames can make some attacks significantly more difficult. So-called "mimicy" attacks take advantage of the inner workings of applications to attack while still behaving similarly to the attacked application [109]. Adding context makes this more difficult because it restricts the attacker to using only methods usually invoked from within the enclosing method that the exploit attacks, instead of any method invoked by the entire application.

6.9 Possible Extensions and Conclusions

Stopping mimicry attacks was a principle motivation for testing the system. The case study study examining the Metron UAV simulation was used to determine whether the system could identify more subtle errors. That is, could the system be used in wider settings than just anomaly intrusion detection?

The case study reveals the strengths and weakness of dynamic sandboxing. It shows that it can reliably detect several of the anomaly scenarios, and that different variations of the dynamic sandboxing can improve detection. The weaknesses are in the required length of training requirement, the difficulty in determining training length by observing profile size, and the amount of parameter "tweaking" to produce a good result.

On the whole, these results reinforce my confidence in dynamic sandboxing's design.

With proper but intuitive parameter settings the sandboxes with different SSP lengths were able to detect 4 out of the 7 anomalies in a challenging application. Although further research is needed to discover applications beyond intrusion detection in which the longer SSP length will be useful, proper parameter settings indicate that false positives can be controlled. Furthermore, the per-thread sandboxes showed some ability to detect more than the four UAV faults.

Using the stack frame as a feature seems to be necessary for catching some anomalies but the behavior can be described as "brittle". The training weaknesses could be lessened by greater analysis during profile generation or at runtime. The system could analyze the events as they occur to determine if the profile is incomplete. New anomalies could then be added to the profile instead of generating events. Somayaji's pH includes such a system [97].

A possible extension of per-thread boxes is to create a customized sandbox for each thread type. The standard Java security mechanisms envision per-thread custom sandboxes. An individual Java sandbox, called a "protection domain" can exist for each thread. A short examination of this idea is presented in Section 7.1.4.

Chapter 7

Other Features and a Complete System Design

The previous four chapters described case studies that investigated the dynamics of several features of execution observable within DEEs. In this chapter I briefly describe some features that did not receive the same scrutiny as those of the case studies. Then I describe how a complete and practical system could be implemented. It would duplicate the total functionality of the previously described ones, add the ability to observe other features, and allow for responses to anomalies.

7.1 Other Features

I mentioned several features in the Introduction that were not addressed by the case studies. Four features that one might logically observe are types, method arguments, method frequencies, and per-thread custom sandboxes.

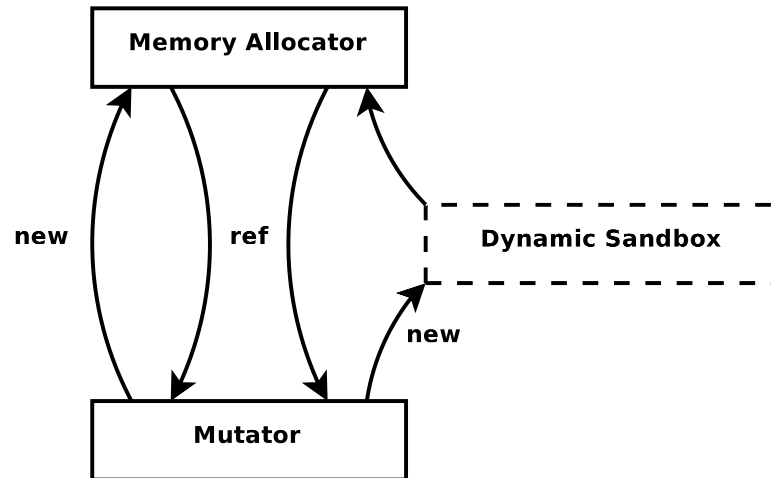

Figure 7.1: Dynamic sandboxing using types. The sandbox interposes itself between the mutator (the application) and the allocator. In normal operation, the allocator is given a type and returns a reference to a region of memory to place an instance of that type. A sandbox, implemented as an allocator proxy, would allow only allocation of types allowed by its profile.

7.1.1 Types

The object lifetime prediction system described in Chapter 5 is highly dependent on sophisticated implementations of garbage collection. One might imagine a simpler system, more useful for dynamic sandboxing, that interposed itself between the allocator and the mutator's requests for memory (these are indicated by `new` instructions). This is depicted in Figure 7.1. When training, the sandbox examines allocation requests and records the type information accompanying it in the profile. When sandboxing, the system prevents any allocation that is not in the sandbox profile.

The main argument for not building the system is that it is redundant. The simple method invocation sandbox designed in Chapter 3 sandboxes by types already—a special `<clinit>` method is called each time a class is initialized. It was possible to observe the behavior I was interested in without implementing another system.

Redundancy is not a always a disadvantage, however. Because it operates on a different interface from the method invocation sandbox, bugs in one sandbox would not affect the other, turning redundancy into a positive. Another argument in favor of such a system is its potential performance. A sandbox that only impacted allocation could be fast. It would not be as fast as the single method sandbox of Chapter 3, but faster than the method sequence sandboxes described in Chapter 6 that can only be implemented through stack introspection.

Benchmark	App Classes	Library Classes	Total
_201_compress	7	20	27
_202_jess	9	20	29
_209_db	7	20	27
_213_javac	13	20	33
_222_mpegaudio	19	20	39
_227_mtrt	8	20	28
_228_jack	8	20	28

Table 7.1: Number of application and library classes used during a run of the SPEC JVM98 benchmarks.

Another possible advantage is the precision of the granularity. False positives became a problem once execution context was added to the dynamic sandbox. The source may have been the decreased granularity (larger size) of that feature space. Although that problem did not occur with the simpler sandbox, future versions of Java or other DEEs may suffer from a similar problem, even excluding context. Tables 7.1 and 3.1 suggest that the space of types and the resulting type sandbox sizes are about the tenth the size of the method sandbox profiles. As Java and other DEEs grow in size, finding profiles of less precise granularity may help reduce false positives.

7.1.2 Method Arguments

Unlike the case for types, there are several strong reasons for not observing method arguments. First, the feature is observable but not part of the standard profiling infrastructure for DEEs. Cataloging argument values is possible through debugging extensions, but it would incur enormous performance penalties. Another problem is in semantics. Determining similarity for primitives, such as integers, doubles, and strings, is not difficult, but how should compound objects be handled? As an initial investigation I computed the hash functions for objects as a proxy for value using the Kaffe JVM. This approach leads to the largest problem: profile blowup. This is because each method needs to be stored with the set of legal arguments (either primitives or hash values). The Permission sandbox of Chapter 4 can be seen as a subset of this sandbox. It covers only one particular method—`checkPermission()`.

Sandboxing by method arguments records *data*, and data naturally change from run to run. Generalization is required for this to work. How to generalize? Again, for most primitives there are distance metrics to make this possible. For compound objects, the obvious handle on identity is through a hash function, which is only comparable for equality, making generalization difficult.

Darko Stefanovic recently suggested abandoning values and analyzing the *types* of the arguments passed to methods. Types are of course specified by the parameter lists in method signatures, but in Java and other object-oriented languages an object may be of several types. Sandboxing on the most specific subtype might be feasible, but this has not been explored.

7.1.3 Method Frequency

From the start, method arguments seemed difficult to categorize. Method frequencies, by contrast, seem easier. Anomaly detection based on method frequencies is simply a different representation of the successful systems described in earlier chapters. It is also the primary feature used by the JIT compilation system to improve performance.

The obvious problem is the presence of working sets during execution. Working sets are caused by modality in program execution. A program typically executes using a small set of its codebase and then switches to another set. The dramatic change in the frequency of method execution during a program's run. Under a binary representation of behavior for sandbox profiles, the large changes in frequency embodied by switches between working sets of methods are ignored, so there were fewer false positives.

Another solution is to use different bounds, based on the frequency variance, for classifying methods as anomalous. I used the traces gathered for Chapter 6 to analyze the behavior of a potential system.

The sandbox profile for the dynamic sandbox employing method frequencies consists of a set of records. Each record contains the method signature, the average method frequency during a run, and the standard deviation computed from the training runs. Obviously, this variant requires several training runs to generate the sandbox.

An anomaly score can be generated from three components:

1. A method invoked in the test trace is not in the sandbox profile (standard model).
2. A method not invoked in the trace is in the profile (the opposite case).
3. A method is in the sandbox profile and is invoked during the trace but with significantly different frequencies.

The sum of all frequencies that match case 1 is the α component. The sum of all frequences that fall under case 2 is the β component. The frequencies of case 3 are referred to as the γ component and are parameterized by the *spread*: its value is the sum of all differences of frequencies and the profile for those method frequencies that lie outside plus or minus the *spread* times the standard deviation of their sandbox frequency:

$$\gamma = \sum_{i=1}^{n} x_i, \quad \text{where} \quad x_i = \begin{cases} 0, & \text{if } dif_i \leq spread \times stdev(profile_i) \\ dif_i, & \text{otherwise.} \end{cases}$$

$$\text{and} \quad dif_i = |profile_i - frequency_i|$$

I do not try to calculate a total score in this exploration. Instead, I investigate the implications of the individual components.

	All methods		UAV methods	
Scenario	$\alpha \times 10000$	ratio/nop	$\alpha \times 10000$	ratio/nop
nop	1.67	1	0.00	1
exec_process	1.78	1.06	0.0140	∞
crash_uav	1.65	0.99	0.00977	∞
random_movement_all	3.47	2.08	3.37	∞
random_movement_one	2.00	1.20	0.18	∞
search_fnr	1.56	0.94	0	1
always_yes	1.72	1.03	0.00	1
no_true_positives	1.74	1.04	0	1

Table 7.2: Average frequencies for α (method invocations that do not appear in the sandbox profile) and their ratio over nop.

The algorithm was tested against the anomaly scenarios presented in Chapter 6. Tables 7.2 and 7.3 present the portion of scores for α and β, respectively, using composite profiles from five runs. Looking at the α component, we see, similarly to the results presented above, that discrimination is successful only when filtering is enabled. When it is

	All methods		UAV methods	
Scenario	$\beta \times 100000$	ratio/nop	$\beta \times 100000$	ratio/nop
nop	3.58	1	5.05	1
exec_process	3.55	0.99	4.86	0.99
crash_uav	3.57	0.99	5.04	0.99
random_movement_all	16300	4570	28100	5560
random_movement_one	3.57	0.99	5.00	0.99
search_fnr	3.56	0.99	5.00	0.99
always_yes	3.57	0.99	5	0.99
no_true_positives	3.55	0.99	5.03	0.96

Table 7.3: Average frequencies for β (methods that do appear in the sandbox profile but not in the trace) and their ratio over nop.

not enabled, a moderately strong result is seen for random_movement_all, compared with the usual four in the filtered case.

The α component corresponds to the dynamic sandbox of Chapter 3: it gauges the proportion of method invocations that were never invoked during training. The β component tries to gauge the opposite—methods that were invoked during training, but not during the run, which cannot be computed online. The β component results are particularly weak. Only random_movement_all gives a good score. The differences between the α and β component scores indicate that the design for the simple dynamic sandbox is robust. The α component, using filtered invocations, gives results as good as those described above for non-frequency based detection, as one would expect. The β component, adding little, strengthens the conclusion that the representations used in Chapter 3 and 6 are sufficient to capture behavior.

Of course, the most important part of frequency-based detection is the comparisons of non-zero frequencies. These results are presented in Table 7.4. The portion of the score based on non-zero frequencies in both training and testing is dependent on γ and *spread*. Table 7.4 shows only the ratio of average events of the anomaly scenarios over nop for different values of *spread*. There are three things to note from the table. First, unlike the

All simulation method invocations					
Scenario	1	2	4	8	16
exec_process	1.44	1.71	4.11	17.9	215.65
crash_uav	1.62	2.01	2.78	6.15	1.88
random_movement_all	13.6	20.4	74.9	1261	251437
random_movement_one	1.94	2.48	2.34	15.4	649
search_fnr	1.42	1.73	1.53	6.12	29.67
always_yes	1.09	0.99	1.43	1.20	32.85
no_true_positives	1.57	2.11	3.41	2.78	1.56
UAV class originated simulation invocations					
Scenario	1	2	4	8	16
exec_process	1.73	2.87	2.74	23.6	1399
crash_uav	1.71	3.26	1.54	0.01	1.40
random_movement_all	15.1	44.1	90.4	2064	322551
random_movement_one	1.95	2.16	0.48	1.06	59.14
search_fnr	1.61	2.23	1.53	8.42	1.27
always_yes	1.19	1.50	0.36	1.10	1.08
no_true_positives	1.66	2.64	1.44	2.25	1.25

Table 7.4: Presents the ratio of each scenario with nop and the specified *spread* (1, 2, 4, 8 , 16) with a γ of 1.

other experiments, the γ component benefits from seeing all method invocations, rather than only those originating in UAV classes. Second, the component's score benefits from larger values of γ. Finally, there is some ability to distinguish the sensor-based scenarios from nop, at least when the run is not filtered.

Detection using non-zero frequencies seems effective for high values of *spread*. This is misleading. First, the results hold only when non-filtered traces are used. This is "cheating" as the goal was to simulate the behavior within individual UAVs. When non-UAV invocations are filtered out, only three of the usual four scenarios were identified. Finally, the large value of *spread* is puzzling. It is not clear whether the results are robust. Their qualitative similarity to the earlier approaches, however, supports the contention that they do in fact distinguish those scenarios. This result, along with the α and β scores, gives confidence in the previous systems.

From the current set of results it cannot be concluded that method frequency is useful in determining program behavior abnormality. Frequency monitoring in the Metron UAV sim produced interesting results for only a few configurations. The simulation is very different from most applications, however. Its multi-threaded nature, and the behavior differences in types between those threads, creates an obstacle to frequency based approaches. Even if method frequencies are stable for each thread type, the number of each type may differ in each run, making runs seem anomalous.

More research is needed to determine if there are enough regularities in method frequencies to make anomaly detection worthwhile.

7.1.4 Per-thread Custom Sandboxes

The Metron UAV simulation made several problems apparent in the method invocation sandboxing schemes. Chief among them was the profile instability caused by the large number of threads produced by the program.

One solution I presented in Section 6.7.3 used individual anomaly counts for each thread. This was partially effective but suffered from higher false positive rates. A further extension to this would be to use different sandboxes for each thread type as well. This would increase the granularity of the profile space considerably, allowing a more precise profile.

Such an approach essentially treats each thread as a different program. This may not make sense for multi-threaded applications like Apache, in which all threads operate similarly [38], but in cases where different threads do in fact behave differently, a per-thread sandbox may be appropriate.

I investigated the behavior of the Metron UAV Sim by recording profiles for each thread in the format of the simple method invocation sandbox. Figure 7.6 shows a short

run of the simulation. The run produced about 250 threads and called about 2600 unique methods. The threads were sorted using k-means clustering with a cluster size of 9 [34]. The clustering reveals, as one would expect from the simulation's source, that there are distinct behaviors for several thread types.

I did not attempt to build such a system. Per-thread sandboxing increases the complexity of training enormously. The dynamic sandbox would become more like a traditional host-based system, examining every application, than a sandbox specific to an application. Faced with the increased difficulty in training, I decided not to pursue such a system. Still, a per-thread sandbox system may be practical for programs that behave with more regularity than the Metron application. As in the method frequency case, more research is needed.

7.1.5 Feature Fusion

One idea for an observable feature that I have not tested in this work is the combination of features. False positives might be reduced by correlating anomalies by feature. I have not and cannot test a combined system because each of the systems presented in the previous series of chapters was designed and tested independently. Furthermore, they were implemented using several different JVMs, and the behavior between them is different enough that comparisons between different chapters cannot readily be made.

An integrated system would be the next logical step. A new system, engineered from the individual systems described here, would allow us to observe global regularities between features, as well as introduce responses to anomalies. In the next section I describe how this system could be implemented.

7.2 The Complete System

Each of the case studies introduced applications or simulations that were not of production quality. Three different JVMs were used out of the four examples. To continue this research forward, a unified VM capable of simultaneously observing all the interesting features is required. This dynamic sandbox would also be capable of response.

To construct this VM, three capabilities are required: a custom security manager, a new garbage collector and allocator, and a stack introspector. In non-DEE environments, these functions are integrated into each application and are not easily observed or modified. In DEEs, these components are common to all applications and could be, with access to source code, observed, replaced, or modified.

7.2.1 Stack Introspection

The method sequence sandbox of Chapter 6 was the slowest of the dynamic sandboxes because it was the only one running entirely in simulation. A small library was used to output method invocation information and a second process simulated and analyzed the stack behavior.

An implementation entirely within the VM could be much faster. Modelling the stack is not necessary because the entire stack is already available. The sandbox would use the JIT compiler to generate a small amount of code in the method prologue to examine the stack and compare it to the profile. More specifically, at each method invocation during training the prologue would scan the top of the stack, observe the integer valued method identifiers, and interleave them together to form a new hash to add to the profile. During testing, the prologue's code would generate the hash again and compare it to the sandbox, and initiate a response if necessary.

The number of stack lookups could be one less than the SSP length because the pro-

logue would provide the first method identifier in the code itself. A hash table would be used to store the profile. In Chapter 6 we saw typical profile sizes of SSP=2 were under 3000 for most of the benchmarks but up to 6000 for the Metron UAV sim. In the larger case, a hash table of perhaps 32KB or 64KB, allowing for collision detection, could be used. Another possibility would be to ignore collisions and allow for a much sparser hash table, using bits instead of bytes. Cache performance considerations would determine the exact representation.

Stack introspection-based approaches are not as fast as the system built out of just the top frame, like the sandbox implemented with ORP. That system incurred no overhead after a method's first invocation. Indeed, there is a performance reason to implement such a system, even in the presence of an implementation of stack introspection. The reason is that native code cache, the collection of methods that has been translated from bytecode to native instructions, can be used as the profile. Instead of recording method signatures or their hashes, the code cache could be saved to disk. On startup, the VM does not need to involve the JIT compilation system. Instead, the native versions of methods could be loaded and linked immediately. This would provide both a boost in performance and an increased level of security.

7.2.2 Windowing

Another variant introduced in Chapter 6 correlated anomalous events in time before triggering an alert. This used the parameters window length and trigger threshold.

A straightforward implementation uses a circular array and a variable holding the current count. After a method is invoked and its status determined, the next element of the array is overrwritten and the count recomputed, with its result depending on the old and new element value. A count greater than the threshold would trigger a response.

More generally, windowing could be subsumed by a configurable component analyz-

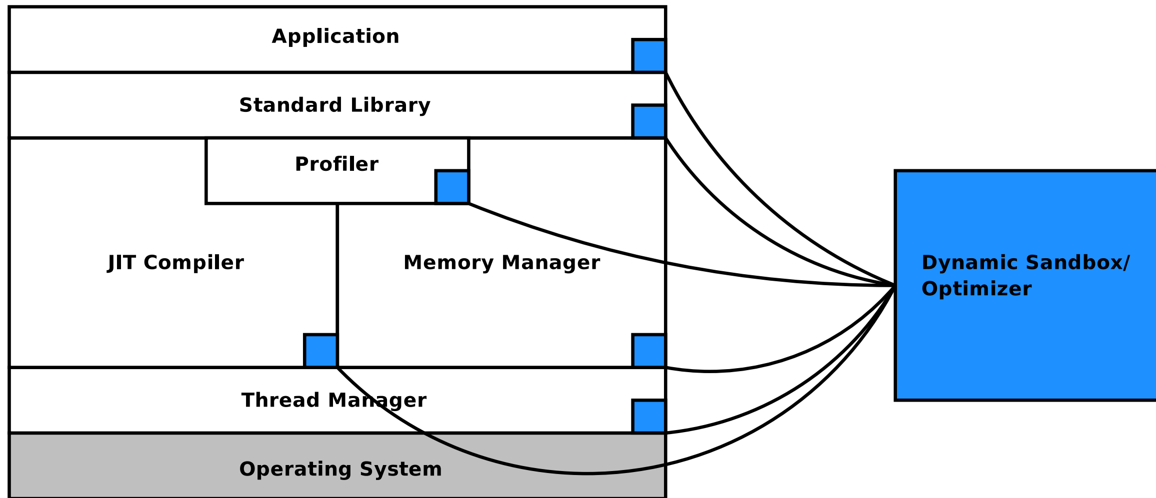

Figure 7.2: Unified dynamic sandbox. A central repository analyzes behavior from interface proxies. Anomalous behavior is analyzed and then the proper response is initiated through the appropriate proxy.

ing events from many different feature sandboxes. A centralized store and analyzer for anomalous events allows for feature fusion, and a smarter, most context dependent response mechanism.

Figure 7.2, a reworking of the standard DEE from Figure 1.2, depicts the sandboxing system interacting with all the DEE's components. As one would expect, different features all interact with the centralized sandbox, which would store profiles, correlate events, and initiate responses.

7.2.3 Response

My research has concentrated on detection—the classification of various features of execution into the predictable or unpredictable. The prototypes used simple responses, such as shutting down the system or logging the event. These responses are not appropriate to a production system. Shutting down the system does not address false positives, and doing

nothing but logging an anomaly allows intrusions and other faults to succeed.

Any response beyond logging would affect program execution. Although the responses would begin in the dynamic sandbox, all of them would appear to come from someplace else to preserve the original semantics (the lines in Figure 7.2 indicate information flow in both directions.) Responses would take the form of exceptions.

Exception handling is a method of dealing with error conditions raised during operation. It was first popularized in the mid 1970s for languages like Ada, but has become a requirement for DEE languages like Java [41]. The details of the exception response would decide in which part of the DEE the exception occured. This exception would be a subtype of a standard error thrown through the same interface. For example, an anomaly thrown through the method sequences sandbox could be a StackOverflowError. Normally, these errors result in program termination.

Applications could be modified, however, to trap these errors. Information about the anomaly, ignored by the standard application, could be analyzed by adding exception handlers. That information would be added by the sandbox by creating subtypes of the usual exceptions and inserting details of the anomaly into the exception object. Programs would run without modification while allowing developers to add code to determine whether anomalies were false positives.

The existence of false positives complicates response. Analysis would be needed to decide whether an anomaly was a true or a false positive, and what the proper response should be. This would be helped by the inclusion of the stack trace in the Exception. Analysis could be divided between the dynamic sandbox and the application, allowing the dynamic sandbox to provide a confidence level, perhaps, to the program.

The program's response could be very simple. If it were an interactive application, it could ask the user to determine a response. A more sophisticated response could borrow from Somayaji's pH. pH uses exponentially increasing delays of subsequent system calls

in the executing process. Small numbers of anomalies have small delays, and go unnoticed. Large numbers of temporally clustered anomalies, produced when programs execute novel code paths, have such large delays that the program essentially freezes.

The appropriate response strategy depends on the exploits and the monitored programs. The dynamic sandbox described here seems best suited to server or middleware applications, where security is most needed and behavior usually limited. Interactive applications, which are closer to the user, might have functionality that is invoked at infrequent intervals, creating larger numbers of false positives. The first response strategy, providing more information for specific anomalies, might work well in server applications while the strategy of delay is more appropriate for interactive applications.

7.2.4 Memory Prediction

If dynamic sandboxing was implemented with types, responses would come from raising already expected exceptions (such as OutOfMemory or Instantion errors). The case study from Chapter 5, though it involved the memory manager, was not a dynamic sandbox. The lifetime predictor responds to regularities instead of anomalies, and the result is the Death-Ordered-Collector. Earlier, I analyzed the best-case performance of the Death-Ordered Collector, the memory management system built to exploit lifetime prediction. It could not operate as described because it assumes that a misprediction is never made. Although true prediction revealed our accuracy to be very good ($>$ 99% for many benchmarks), there is still the rare occurrence when objects live longer than forecasted.

A realistic implementation would need to relax that one assumption of the DOC implementation. Without that assumption, Section 5.7.1 demonstrated that the DOC is superior to a standard semispace collector if the overhead is small. Here I argue that a realistic implementation will be similar to that of a generational collector.

Let us first consider the overhead of allocation. Allocation is more expensive in the

DOC because the allocator must decide whether to allocate into the standard heap or the KLS. Placement requires the examination of the SSP, which in some cases is quite long. During execution, the SSP would be available only as separate values in individual stack execution frames, or perhaps as a separate display-like structure.

To show how allocators incorporating prediction are constructed, I provide an example. At the point of an allocation (a *new* in Java), the compiler would generate inline code for the allocation. By way of example, consider this point of execution in the diagram below (Figure 7.3):

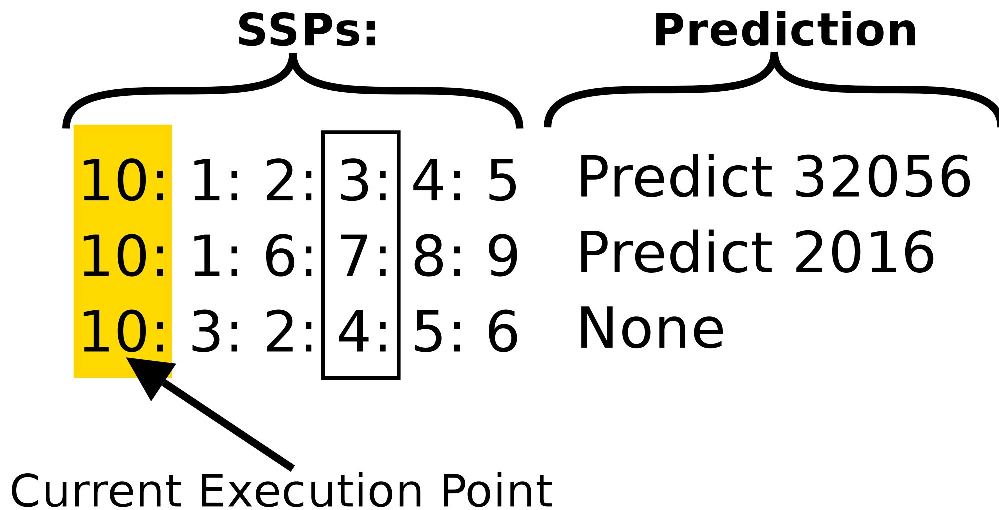

Figure 7.3: The 3 SSPs that need to be considered for an allocation at execution point 10 with their associated predictions. Although the SSP length is 6, the allocator need consider at most 2 positions for linear comparisons, and only 1 for random access.

The application is at execution point 10, and has three options for allocation. The top two SSPs lead to predictions, the third does not. The task at hand is to compare the current SSP, embodied by the execution stack, with those in the predictor. It is impractical to consider the entire SSP as a single entity because each value within the SSP is a full integer in length. Instead, consider the individual values which make up the SSP: the

method ID and position within the method, which is equivalent to the return address. Since the generated code for this allocator is customized to the allocation site (execution point 10), it need not consider any SSP starting with a value other than 10. Nor, since all SSPs we need consider start with 10, need it consider the starting SSP position. Instead, the allocator simply needs to look at enough values of its own stack string to uniquely distinguish between the three SSPs. To accomplish this, the allocator is constructed as a tree. At the first SSP value, it considers values for the second SSP position, which are 1 and 3. For SSPs beginning 10:3, there is no prediction, so the compiler generates code for the normal allocation path. For the two SSPs, the compiler then generates code that examines the second value, which makes the prediction of 32056 bytes for 10:1:2 and 2016 bytes for 10:1:6.

The example is depicted in Figure 7.4. Note that these need not be binary trees. For each position, there are as many edges leaving the node as there are unique SSP values at that position. If we assume that each SSP is observed at this allocation point an equal number of times then the average depth of search is 5/3.

The previous example considers an implementation that assumes that application stack examination is similar to walking a list. If a display-like structure is available, where access to any value in the SSP is in constant time (random access), the allocator needs only one node with 3 values, that for position 3 (outlined in the diagram) and the average depth for search drops to 1.[1] Choosing an optimal sequence of positions when random access is available is in general a hard problem. However, the following heuristic works well: choose the position with the largest number of unique values. This breaks the set down into that number of subsets. Repeat with this heuristic on each subset until no ambiguities exist.

Because all of the stack strings are recorded during the profiling run, the allocator can

[1]Displays are used in languages that allow nesting of functions. The display allows constant time access to variables within containing functions [1].

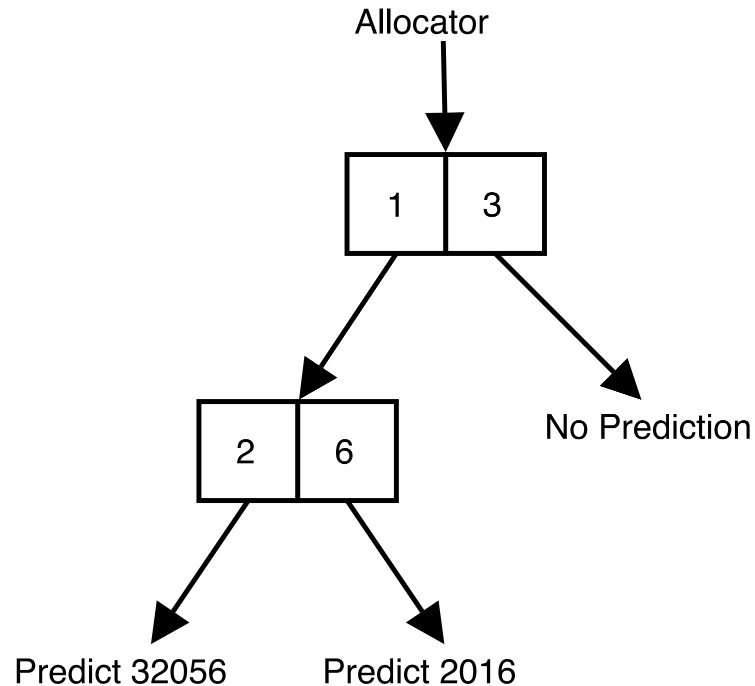

Figure 7.4: Linear allocator generated for the execution point 10 shown in Figure 7.3. The allocator needs to consider only two positions of the SSP to uniquely distinguish them.

calculate the average search depth required, as shown in Table 7.5 for my set of benchmarks. The column labeled "linear" shows results for the algorithm described above that assumes that the stack is similar to a list, where execution frames that are further away take longer to examine. However, assuming constant time random access to the execution frames, many lookups are eliminated, as shown in the column labeled "random". In each, instead of the full SSP length, for most benchmarks only 1 to 2 comparisons are required for each allocation. Even for very long SSP lengths, like those in *jess* and *javac*, the average number of lookups is 4 or less in the linear case and less than that in the random access case. Thus, allocation can be made efficient in the DOC system.

The overhead of collection in the DOC system is similar to that of a generational collector. Like all generational collectors, the DOC system requires write barriers and

Benchmark	Fully Precise			Logarithmic		
	ssp	linear	random	ssp	linear	random
compress	20	1.92	1.28	10	1.92	1.28
jess	24	3.08	2.12	26	3.08	2.12
db	20	0.98	1.86	3	0.96	1.85
raytrace	20	2.00	1.45	4	1.62	1.07
javac	32	3.84	3.27	32	3.84	3.27
mpegaudio	20	2.09	1.40	8	2.06	1.37
mtrt	20	2.01	1.46	3	1.25	0.99
jack	20	4.02	3.21	20	4.02	3.21
pseudojbb	20	2.50	1.89	14	2.48	1.86

Table 7.5: The average number of hash table queries that are required to identify an allocation context for a given ssp and benchmark. The two sets of data shown are for the SSP lengths used in Table 5.6. For each set, I calculate the average depth using hash tables constructed using the SSP with linear access and with random access.

remembered sets. The heaps are arranged as described in the previous subsection, and shown in Figure 7.5. Remembered sets are required between the two heaps, as any multi-space collector requires. A remembered set is also required for objects in the KLS pointing to objects with smaller predicted lifetimes. This feature makes the KLS similar to Barret and Zorn's generational collector with a dynamic/threatening boundary [10]. Like that scheme, the DOC can collect a variable amount of space to "tune" pause times. It is likely that the remembered sets for objects being collected in the KLS are small, due to the high accuracy. However, if set size is a problem, one might unify the two heaps into one looking very similar to the Barret and Zorn collector described above, in which my predictions are used as a parameterized pretenuring scheme, with the object lifetime predictions used to determine their placement in the list. This would have overheads very similar to Barret and Zorn's collector. The choice of heap arrangement is an empirical question that involves the relative performance of various parts of the memory management system.

In the worst case, the DOC's overhead will be similar to that of a standard heap, because its overhead is similar to that of a generational collector. When the DOC can use its

KLS, it will outperform a traditional collector. Implementing the DOC and measuring its performance experimentally is an area of future investigation.

7.3 Summary

This chapter explored how the results of Chapters 3-6 could be incorporated into an operable system. The resulting system would have production class performance, but would not be a production system. It also discussed other potential execution features that have not been explored in depth. I have discovered features and representations that would likely be useful, but the space of parameters that could tune the system in the areas of response, generalization, and sandbox granularity still need to be explored. Only then, when the complete behavior of target applications is examined, could a proper system be built to improve security and performance.

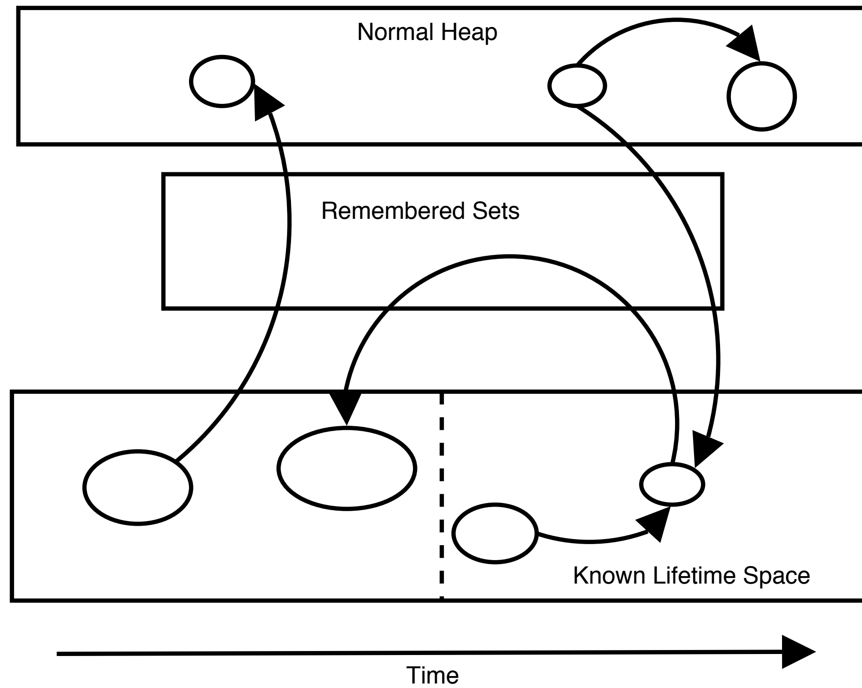

Figure 7.5: A visual representation of the death-ordered collector. The ovals within the top and bottom rectangles represent objects within the normal heap and the known lifetime space (KLS), respectively. Objects in the KLS are arranged in order of predicted death. Arrows indicate references from one object to another. Arrows that overlap the remembered sets rectangle must be considered as roots when a collection occurs (though the sets do not have to be unified in any particular implementation). The dotted line indicates the current time. A DOC starts a collection with objects at the far left, the ones with the earliest predicted time-of-death, up to the current time. Remembered sets are necessary because objects not considered for collection, those in the normal heap and those not predicted to be dead, are assumed to be alive and thus any objects pointed to by them must also be considered alive.

Figure 7.6: Differences between threads in the Metron UAV simulation. Each row corresponds to a thread and each column corresponds to a method. Pixel (i, j) is white if method i is invoked at least once during the run of thread j. There are approximately 250 threads and 2600 methods shown.

Chapter 8

Conclusion

In this work I have discovered regularity in the behavior of several features of execution and shown that such regularity can be exploited. This was accomplished by developing several prototype systems. In this chapter I summarize the contributions of my research and comment about the differences in my views compared to when I began this line of research.

8.1 Contributions

The contributions of my research are in the areas of computer security, garbage collection, and virtual machine technology. I have also provided evidence for my thesis for several of the features I examined:

> *The behavior of applications in DEEs is regular, and that regularity can be exploited.*

Although the regularity of DEE behavior depends on each program examined, I have shown that many common benchmarks produce regular behavior in a variety of ways, and

I have shown that the regularity is exploitable for performance and security purposes.

Individually, these are the contributions:

- **Chapter 1** introduced the term *dynamic execution environments* (DEEs), defined as virtual machines with the following characteristics: verifiable instruction sets, Just-In-Time (JIT) compilation, garbage collection, large standard libraries, and runtime profiling.

- **Chapter 2** described related work and introduced the tools and data used for this research. It also reported on the current state of Java security and analyzed all Java vulnerabilities since 1999. Such analysis has not been carried out previously.

- **Chapter 3** introduced the first and simplest implementation of the *dynamic sandboxes*. By controlling the JIT compilation interface to allow only previously invoked methods to be compiled, the sandbox stopped intrusions without hindering performance in real applications.

- **Chapter 4** built a system around Permissions, the feature used for access control in Java. This dynamic sandbox had finer granularity than the previous ones, and exhibited more false positives. Its sandbox profile was expressed as a standard security policy for the virtual machine, allowing easy manual modification.

- **Chapter 5** moved away from dynamic sandboxing to address optimization in the memory management system. I showed that for some applications many object lifetimes are predictable and then presented a design and analysis of the potential performance of a garbage collector designed to use those predictions to improve efficiency. Called the Death-Ordered-Collector (DOC), it would collect items as soon as they died, minimizing overhead and maximizing free space. I also showed that objects that are born and die before the next object is allocated form a significant

portion of all objects, a fact that was previously unknown. This is important because it might be used as advice for for other memory management optimizations.

- **Chapter 6** extended the method invocations sandbox by examining several variants in the context of a large multi-threaded simulation in the application of fault detection. One variant increased sensitivity by adding execution context. Another was used to reduce false positives by correlating anomalies in time. These variants were only partially successful when tested against the Metron UAV Simulation. The simulation showed a large amount of novelty, even when incorporating large amounts of training data. However, the variant sandboxes were effective enough to conclude that they would be more fruitful when used with conventional programs.

- **Chapter 7** described more cursory investigations of method arguments, method frequencies, types, and thread groups. It also described the implementation of a system that would integrate several of the dynamic sandboxes as well as the DOC.

The results of this research are supplemented by datasets and software. The datasets consist of traces from several sets of benchmarks from the experiments in Chapters 6 and 5. The lifetime prediction traces are particularly useful. They record benchmark behavior with exact lifetime data, and they can no longer be duplicated with recent versions of Jikes.

The datasets are complemented by software. The four exploits that were used to test the dynamic sandboxing software in various chapters are not generally available to researchers. I reimplemented the three that were discovered in the wild, and HTTPTrojan, the synthetic trojan horse, is arguably the most interesting of the four. Other software consists of analysis tools. These tools were developed for Chapters 5 and 6. The rest of the software consists of the systems themselves: the patched version of ORP from Chapter 3 and the reimplemented SecurityManager and utilities from Chapter 4.

8.2 DEEs and Anomaly Detection

The original goal of my research, beyond proving the thesis, was to construct an integrated system more capable and complex than the one described in Chapter 7.

Constructing and testing these systems was more involved than I anticipated. Jikes, in particular, was complicated to use and modify. In the end, I was never able to collect traces for certain benchmarks and configurations.[1]

Data collection and analysis was another challenge. By recording traces I was able to simulate the behavior of several different sandboxes or predictors. The actual trace used, however, was often hundreds or thousands of megabytes, even when compressed. Simulations often took hours.

Beyond the implementation challenges, I was overly optimistic about the progress of architecture and optimization with DEEs. I believed that platforms like Transmeta's Crusoe [64] would incorporate the dynamic optimization of VMs like Dynamo [105] and specialization like that of Massalin's Synthesis [82] to produce radically different machines. Such a machine would monitor itself constantly, recoding and rewriting both itself and the application to improve performance.

DEEs, however, have advanced slowly. The most complex DEE to date, the Java HotSpot VM, was first introduced in 2000 and has had only incremental improvements since then. Its primary method of feedback directed optimization is still method frequencies. A recent paper showed that simple heuristics for JIT compilation could improve both startup and throughput times, suggesting that even currently available profiling information is being ignored [59]. Transmeta no longer makes processors, and most VM work is

[1] My system was implemented on Jikes 2.0.3. While newer versions of Jikes purportedly support Merlin, none run all the SPEC or DaCapo benchmarks with the finest lifetime granularity. Also, at no time have benchmarks suites completed using Jikes's optimizing compiler as the JIT compiler. This makes my traces particularly valuable for garbage collection research.

aimed at simulating IA32 processors.

The research projects originating from my group often compare computers with biological systems. This is becoming a more apt analogy. Even with the slow advancement of DEEs, these systems' behaviors are now too complex to understand from first principles. It is impossible to predict or understand the behavior of our systems from their design and configuration. Biologists struggle with a similar problems in ontogeny—knowing the genetic code of an organism does not provide much information on its final structure or behavior. Because the problems we face in studying program behavior are so similar to biological ones, we should employ biologically inspired techniques, The advantage computer scientists have is that we can easily observe detail that biologists cannot. We should take advantage of it.

Appendices

Appendix A

Classification of Java Exploits from the CVE Dictionary

The Common Vulnerabilities and Exposures (CVE) Dictionary is a list of all publicly known vulnerabilities [27]. It is maintained by the MITRE corporation to provide a common name for these exploits. The following table excerpts all vulnerabilities strictly related to the Java platform from the CVE dictionary. This includes all Java plugins, libraries, and virtual machines but does not include Enterprise Edition libraries or other applications written in Java.

The descriptions are excerpted from the CVE dictionary and copyrighted by the MITRE corporation. They are used here with permission.[1] The classification of entries into VM, library, or policy is my own work.

There are 51 exploits listed. There are 6 VM , 2 policy, and 43 library vulnerabilities. See Chapter 2 for a description of Java security and its vulnerabilities.

[1]The CVE web site states: "You may search or download CVE, copy it, redistribute it, reference it, and analyze it, provided you do not modify CVE itself. CVE is publicly available and free to use." [28].

CVE Number	Description	VM	Library	Policy
CVE-1999-0141	Java Bytecode Verifier allows malicious applets to execute arbitrary commands as the user of the applet.	✓		
CVE-1999-0142	The Java Applet Security Manager implementation in Netscape Navigator 2.0 and Java Developer's Kit 1.0 allows an applet to connect to arbitrary hosts.		✓	
CVE-1999-0440	The byte code verifier component of the Java Virtual Machine (JVM) allows remote execution through malicious web pages.	✓		
CVE-1999-0766	The Microsoft Java Virtual Machine allows a malicious Java applet to execute arbitrary commands outside of the sandbox environment.		✓	
CVE-1999-1262	Java in Netscape 4.5 does not properly restrict applets from connecting to other hosts besides the one from which the applet was loaded, which violates the Java security model and could allow remote attackers to conduct unauthorized activities.		✓	
CVE-2000-0162	The Microsoft virtual machine (VM) in Internet Explorer 4.x and 5.x allows a remote attacker to read files via a malicious Java applet that escapes the Java sandbox, aka the "VM File Reading" vulnerability.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CVE-2000-0327	Microsoft Virtual Machine (VM) allows remote attackers to escape the Java sandbox and execute commands via an applet containing an illegal cast operation, aka the "Virtual Machine Verifier" vulnerability.	✓		
CAN-2000-0563	The URLConnection function in MacOS Runtime Java (MRJ) 2.1 and earlier and the Microsoft virtual machine (VM) for MacOS allows a malicious web site operator to connect to arbitrary hosts using a HTTP redirection, in violation of the Java security model.		✓	
CVE-2000-0676	Netscape Communicator and Navigator 4.04 through 4.74 allows remote attackers to read arbitrary files by using a Java applet to open a connection to a URL using the "file", "http", "https", and "ftp" protocols, as demonstrated by Brown Orifice.		✓	
CVE-2000-0711	Netscape Communicator does not properly prevent a ServerSocket object from being created by untrusted entities, which allows remote attackers to create a server on the victim's system via a malicious applet, as demonstrated by Brown Orifice.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CVE-2000-1061	Microsoft Virtual Machine (VM) in Internet Explorer 4.x and 5.x allows an unsigned applet to create and use ActiveX controls, which allows a remote attacker to bypass Internet Explorer's security settings and execute arbitrary commands via a malicious web page or email, aka the "Microsoft VM ActiveX Component" vulnerability.		✓	
CVE-2000-1099	Java Runtime Environment in Java Development Kit (JDK) 1.2.2_05 and earlier can allow an untrusted Java class to call into a disallowed class, which could allow an attacker to escape the Java sandbox and conduct unauthorized activities.		✓	
CAN-2000-1117	The Extended Control List (ECL) feature of the Java Virtual Machine (JVM) in Lotus Notes Client R5 allows malicious web site operators to determine the existence of files on the client by measuring delays in the execution of the getSystemResource method.		✓	
CAN-2001-0068	Mac OS Runtime for Java (MRJ) 2.2.3 allows remote attackers to use malicious applets to read files outside of the CODEBASE context via the ARCHIVE applet parameter.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2001-0324	Windows 98 and Windows 2000 Java clients allow remote attackers to cause a denial of service via a Java applet that opens a large number of UDP sockets, which prevents the host from establishing any additional UDP connections, and possibly causes a crash.		✓	
CVE-2001-1008	Java Plugin 1.4 for JRE 1.3 executes signed applets even if the certificate is expired, which could allow remote attackers to conduct unauthorized activities via an applet that has been signed by an expired certificate.		✓	
CAN-2002-0058	Vulnerability in Java Runtime Environment (JRE) allows remote malicious web sites to hijack or sniff a web client's sessions, when an HTTP proxy is being used, via a Java applet that redirects the session to another server, as seen in (1) Netscape 6.0 through 6.1 and 4.79 and earlier, (2) Microsoft VM build 3802 and earlier as used in Internet Explorer 4.x and 5.x, and possibly other implementations that use vulnerable versions of SDK or JDK.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CVE-2002-0076	Java Runtime Environment (JRE) Bytecode Verifier allows remote attackers to escape the Java sandbox and execute commands via an applet containing an illegal cast operation, as seen in (1) Microsoft VM build 3802 and earlier as used in Internet Explorer 4.x and 5.x, (2) Netscape 6.2.1 and earlier, and possibly other implementations that use vulnerable versions of SDK or JDK, aka a variant of the "Virtual Machine Verifier" vulnerability.	✓		
CVE-2002-0865	A certain class that supports XML (Extensible Markup Language) in Microsoft Virtual Machine (VM) 5.0.3805 and earlier, probably com.ms.osp.ospmrshl, exposes certain unsafe methods, which allows remote attackers to execute unsafe code via a Java applet, aka "Inappropriate Methods Exposed in XML Support Classes."		✓	
CVE-2002-0866	Java Database Connectivity (JDBC) classes in Microsoft Virtual Machine (VM) up to and including 5.0.3805 allow remote attackers to load and execute DLLs (dynamic link libraries) via a Java applet that calls the constructor for com.ms.jdbc.odbc.JdbcOdbc with the desired DLL terminated by a null string, aka "DLL Execution via JDBC Classes."		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CVE-2002-0867	Microsoft Virtual Machine (VM) up to and including build 5.0.3805 allows remote attackers to cause a denial of service (crash) in Internet Explorer via invalid handle data in a Java applet, aka “Handle Validation Flaw.”		✓	
CVE-2002-0941	The ConsoleCallBack class for nCipher running under JRE 1.4.0 and 1.4.0_01, as used by the TrustedCodeTool and possibly other applications, may leak a passphrase when the user aborts an application that is prompting for the passphrase, which could allow attackers to gain privileges.		✓	
CAN-2002-0979	The Java logging feature for the Java Virtual Machine in Internet Explorer writes output from functions such as System.out.println to a known pathname, which can be used to execute arbitrary code.		✓	
CVE-2002-1257	Microsoft Virtual Machine (VM) up to and including build 5.0.3805 allows remote attackers to execute arbitrary code by including a Java applet that invokes COM (Component Object Model) objects in a web site or an HTML mail.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2002-1258	Two vulnerabilities in Microsoft Virtual Machine (VM) up to and including build 5.0.3805, as used in Internet Explorer and other applications, allow remote attackers to read files via a Java applet with a spoofed location in the CODEBASE parameter in the APPLET tag, possibly due to a parsing error.		✓	
CVE-2002-1260	The Java Database Connectivity (JDBC) APIs in Microsoft Virtual Machine (VM) 5.0.3805 and earlier allow remote attackers to bypass security checks and access database contents via an untrusted Java applet.		✓	
CAN-2002-1286	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to steal cookies and execute script in a different security context via a URL that contains a colon in the domain portion, which is not properly parsed and loads an applet from a malicious site within the security context of the site that is being visited by the user.		✓	
CAN-2002-1287	Stack-based buffer overflow in the Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to cause a denial of service via a long class name through (1) Class.forName or (2) ClassLoader.loadClass.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2002-1288	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to determine the current directory of the Internet Explorer process via the getAbsolutePath() method in a File() call.			✓
CAN-2002-1289	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to read restricted process memory, cause a denial of service (crash), and possibly execute arbitrary code via the getNativeServices function, which creates an instance of the com.ms.awt.peer.INativeServices (INativeServices) class, whose methods do not verify the memory addresses that are passed as parameters.		✓	
CAN-2002-1290	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to read and modify the contents of the Clipboard via an applet that accesses the (1) ClipBoardGetText and (2) ClipBoardSetText methods of the INativeServices class.		✓	
CAN-2002-1291	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to read arbitrary local files and network shares via an applet tag with a codebase set to a "file://%00" (null character) URL.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2002-1292	The Microsoft Java virtual machine (VM) build 5.0.3805 and earlier, as used in Internet Explorer, allows remote attackers to extend the Standard Security Manager (SSM) class (com.ms.security.StandardSecurityManager) and bypass intended StandardSecurityManager restrictions by modifying the (1) deniedDefinitionPackages or (2) deniedAccessPackages settings, causing a denial of service by adding Java applets to the list of applets that are prevented from running.		✓	
CAN-2002-1293	The Microsoft Java implementation, as used in Internet Explorer, provides a public load0() method for the CabCracker class (com.ms.vm.loader.CabCracker), which allows remote attackers to bypass the security checks that are performed by the load() method.		✓	
CAN-2002-1295	The Microsoft Java implementation, as used in Internet Explorer, allows remote attackers to cause a denial of service (crash) and possibly conduct other unauthorized activities via applet tags in HTML that bypass Java class restrictions (such as private constructors) by providing the class name in the code parameter, aka "Incomplete Java Object Instantiation Vulnerability."	✓		

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CVE-2002-1325	Microsoft Virtual Machine (VM) build 5.0.3805 and earlier allows remote attackers to determine a local user's username via a Java applet that accesses the user.dir system property, aka "User.dir Exposure Vulnerability."			✓
CAN-2003-0111	The ByteCode Verifier component of Microsoft Virtual Machine (VM) build 5.0.3809 and earlier, as used in Windows and Internet Explorer, allows remote attackers to bypass security checks and execute arbitrary code via a malicious Java applet, aka "Flaw in Microsoft VM Could Enable System Compromise."	✓		
CAN-2003-0525	The getCanonicalPath function in Windows NT 4.0 may free memory that it does not own and cause heap corruption, which allows attackers to cause a denial of service (crash) via requests that cause a long file name to be passed to getCanonicalPath, as demonstrated on the IBM JVM using a long string to the java.io.getCanonicalPath Java method.	✓		

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2003-0896	The loadClass method of the sun.applet.AppletClassLoader class in the Java Virtual Machine (JVM) in Sun SDK and JRE 1.4.1_03 and earlier allows remote attackers to bypass sandbox restrictions and execute arbitrary code via a loaded class name that contains "/" (slash) instead of "." (dot) characters, which bypasses a call to the Security Manager's checkPackageAccess method.		✓	
CAN-2003-1123	Sun Java Runtime Environment (JRE) and SDK 1.4.0_01 and earlier allows untrusted applets to access certain information within trusted applets, which allows attackers to bypass the restrictions of the Java security model.		✓	
CAN-2004-0651	Unknown vulnerability in Sun Java Runtime Environment (JRE) 1.4.2 through 1.4.2_03 allows remote attackers to cause a denial of service (virtual machine hang).			
CAN-2004-0723	Microsoft Java virtual machine (VM) 5.0.0.3810 allows remote attackers to bypass sandbox restrictions to read or write certain data between applets from different domains via the "GET/Key" and "PUT/Key/Value" commands, aka "cross-site Java."		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2004-1489	Opera 7.54 and earlier does not properly limit an applet's access to internal Java packages from Sun, which allows remote attackers to gain senstive information, such as user names and the installation directory.		✓	
CAN-2004-1503	Integer overflow in the InitialDirContext in Java Runtime Environment (JRE) 1.4.2, 1.5.0 and possibly other versions allows remote attackers to cause a denial of service (Java exception and failed DNS requests) via a large number of DNS requests, which causes the xid variable to wrap around and become negative.		✓	
CAN-2004-1753	The Apple Java plugin, as used in Netscape 7.1 and 7.2, Mozilla 1.7.2, and Firefox 0.9.3 on MacOS ✓10.3.5, when tabbed browsing is enabled, does not properly handle SetWindow(NULL) calls, which allows Java applets from one tab to draw to other tabs and facilitates phishing attacks that spoof tabs.		✓	
CAN-2005-0223	The Software Development Kit (SDK) and Run Time Environment (RTE) 1.4.1 and 1.4.2 for Tru64 UNIX allows remote attackers to cause a denial of service (Java Virtual Machine hang) via object deserialization.		✓	

Continued on next page

Table A.1 – continued from previous page

CVE Number	Description	VM	Library	Policy
CAN-2005-0418	Argument injection vulnerability in Java Web Start for J2SE 1.4.2 up to 1.4.2_06, on Mac OS X, allows untrusted applications to gain privileges via the value parameter of a property tag in a JNLP file. NOTE: it is highly likely that this item will be MERGED with CAN-2005-0836.		✓	
CAN-2005-0471	Sun Java JRE 1.1.x through 1.4.x writes temporary files with long filenames that become predictable on a file system that uses 8.3 style short names, which allows remote attackers to write arbitrary files to known locations and facilitates the exploitation of vulnerabilities in applications that rely on unpredictable file names.		✓	
CAN-2005-0836	Argument injection vulnerability in Java Web Start for J2SE 1.4.2 up to 1.4.2_06 allows untrusted applications to gain privileges via the value parameter of a property tag in a JNLP file.		✓	
CAN-2005-1080	Directory traversal vulnerability in the Java Archive Tool (Jar) utility in J2SE SDK 1.4.2, 1.5 allows remote attackersto write arbitrary files via a .. (dot dot) in filenames in a .jar file.		✓	
CAN-2005-1105	Directory traversal vulnerability in the MimeBodyPart.getFileName method in JavaMail 1.3.2 allows remote attackers to write arbitrary files via a .. (dot dot) in the filename in the Content-Disposition header.		✓	

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 1988.

[2] M. Almgren and U. Lindqvist. Application-integrated data collection for security monitoring. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 22–36. Springer, October 2001.

[3] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, New York, NY, USA, 1999. ACM Press.

[4] J. Anderson. Information security in a multi-user environments. *Advances in Computers*, 12, 1972.

[5] James Anderson. Computer security threat modelling and surveillance. Technical report, James P Anderson Company, April 1980.

[6] Apache. BCEL - byte code engineering library. `http://jakarta.apache.org/bcel/manual.html`.

[7] Algirdas Avizienis. The methodology of n-version programming. In Michael Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley & Sons Ltd., 1995.

[8] Stefan Axelsson. Intrusion detection systems: A taxomomy and survey. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, March 2000.

[9] Gabriela Barrantes. *Automated Methods for Creating Diversity in Computer Systems*. PhD thesis, University of New Mexico, 2005.

[10] David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *SIGPLAN Notices*, pages 301–314, La Jolla, CA, June 1995. ACM Press.

[11] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[12] S. M. Blackburn, K.S. McKinley, J. E. B. Moss, E. D. Berger, P. Cheng, A. Diwan, S. Guyer, M. Hirzel, C. Hoffman, A. Hosking, X. Huang, A. Khan, P. McCachey, D. Stefanovic, and B. Wiedermann. The dacapo benchmarks. `http://ali-www.cs.umass.edu/DaCapo/Benchmarks`, 2004.

[13] Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, PLDI'02, Berlin, June, 2002*, volume 37(5) of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[14] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of SIGPLAN 2001 Conference on Object-Oriented Programming, Languages, & Applications*, volume 36(10) of *ACM SIGPLAN Notices*, pages 342–352, Tampa, FL, October 2001. ACM Press.

[15] J.M. Bradshaw, S. Dutfield, P. Benoit, and J.D. Woolley. *Software Agents*, chapter KAoS: Towards an Industrial Strength Generic Agent Architecture, pages 375–418. AAAI Press/MIT Press, 1997.

[16] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *OOPSLA*, pages 353–366, 2001.

[17] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A. V. Surendran, and D. M. Martin Jr. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II '01)*, volume 2, June 2001.

[18] B. Cahoon and K. McKinley. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*, October 1999.

[19] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 264–273, 2000.

[20] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[21] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, volume 33 of *SIGPLAN Notices*, pages 162–173, Montreal, Québec, Canada, June 1998. ACM Press.

[22] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 99)*, November 1999.

[23] M. Christiaens. Dynamic techniques for the optimization of data race detection. In *Program Acceleration through Application and Architecture driven Code Transformations: Symposium Proceedings*, pages 73–75, Edegem, 9 2002.

[24] Lap chung Lam and Tzi cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[25] David A. Cohn and Satinder Singh. Predicting lifetimes in dynamically allocated memory. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 939. The MIT Press, 1997.

[26] M. Condell, C. Lynn, and J. K. Zhao. Security policy specification language (spsl). Internet Draft.

[27] MITRE Corporation. Common vulnerabilities and exposures. `http://www.cve.mitre.org/`.

[28] MITRE Corporation. CVE terms of use. `http://www.cve.mitre.org/cve`.

[29] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, Bristol, UK, 2001. Springer-Verlag.

[30] Dorthy Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222, 1987.

[31] A. Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 358–371, 1997.

[32] Sylvia Dieckman and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In Erik Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 92–115. Springer-Verlag, 1998.

[33] Sylvia Dieckmann and Urs Hölzle. The allocation behavior of the SPECjvm98 Java benchmarks. In Rudi Eigenman, editor, *Performance Evaluation and Benchmarking with Realistic Applications*. The MIT Press, 2001.

[34] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley Interscience, 2 edition, 2001.

[35] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Symposium on Operating System Principles*, pages 57–72, October 2001.

[36] Robert P. Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the Second International Symposium on Memory Management (ISMM)*, pages 111–120, 2000.

[37] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[38] Apache Software Foundation. Apache http server project. `http://httpd.apache.org/`.

[39] Free Software Foundation. Classpath. `http://www.classpath.org`.

[40] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 2 edition, 1993.

[41] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[42] P. H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6), November 1983.

[43] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[44] Timothy L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the Second International Symposium on Memory Management (ISMM)*, pages 127–136, 2000.

[45] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of SIGPLAN 1991 Conference on Object-Oriented Programming, Languages, & Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–40, Phoenix, AZ, October 1991. ACM Press.

[46] Barry Hayes. *Key Objects in Garbage Collection*. PhD thesis, Stanford University, Stanford, California, March 1993.

[47] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems*, volume 30(1) of *ACM Performance Evaluation Review*, pages 140–151, Marina Del Rey, CA, June 2002. ACM Press.

[48] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *Proceedings of the Third International Symposium on Memory Management (ISMM)*, pages 36–49, 2002.

[49] Steven Hofmeyer. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[50] Steve Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[51] Steven Hofmeyr and Matthew Williamson. Primary response technical whitepaper. `http://www.sanasecurity.com/resources/perspectives.php?source=web-resources-resp`.

[52] Jerry Honeycut. Microsoft Virtual PC. `http://download.microsoft.com/download/c/f/b/cfb100a7-463d-4b86-ad62-064397178b4f/Virtual_PC_Technical_Overview.doc`.

[53] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the New Security Paradigms Workshop 2002*. ACM Press, 2002.

[54] Hajime Inoue and Stephanie Forrest. Inferring Java security policies through dynamic sandboxing. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers*. CSREA Press, 2005.

[55] Hajime Inoue, Darko Stefanovic, and Stephanie Forrest. On the prediction of Java object lifetimes. Submitted to IEEE Transactions on Computers, 2003.

[56] Ecma International. Standard ecma-335: Common language infrastructure (CLI). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[57] Inc Internet Security Systems. Realsecure OS sensor. `http://www.iss.net/{securing}_e-business/security_products/intrusion_detection/realsecure_ossensor/`, 2001.

[58] Landing Camel Intl. Codebreakers. `http://www.codebreakers.org`, 1998.

[59] Azeem Jiva and Rober Chun. Compilation scheduling for the Java virtual machine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers*. CSREA Press, 2005.

[60] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, Chichester, 1996.

[61] Maria Jump and Ben Hardekopf. Pretenuring based on escape analysis. Technical Report TR-03-48, University of Texas at Austin, November 2003.

[62] L. Kagal. Rei: A policy language for the me-centric project (hpl-2002-070). Technical report, HP Labs, 2002.

[63] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[64] A. Klaiber. The technology behind Crusoe processors. `http://www.transmeta.com/crusoe/dnload/pdf/crusoetechwp.pdf`, 2000.

[65] Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[66] Benjamin A. Kuperman and Eugene Spafford. Generation of application level data via library interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, West Lafayette, Indiana 47907-1398, October 1999.

[67] Intel Microprocessor Research Labs. Open runtime platform. http://www.intel.com/research/mrl/orp/, 2000.

[68] Terran Lane. *Machine Learning Techniques for the Computer Security Domain of Anomaly Detection*. PhD thesis, Purdue University, August 2000.

[69] James R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[70] H. Lee. BIT: Bytecode instrumenting tool, 1997.

[71] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, 1999.

[72] Lime Wire LLC. Limewire. http://www.limewire.com/english/content/home.shtml.

[73] Carla Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *New Security Paradigms Workshop 2000*, Cork, Ireland, 2000.

[74] Microsoft. Ms02-069: Flaw in microsoft VM may compromise Windows. http://support.microsoft.com/kb/810030/EN-US/, 2002.

[75] Sun Microsystems. Java 2 Platform, Enterprise Edition. http://java.sun.com/j2ee/index.jsp.

[76] Sun Microsystems. The Java debug interface. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/.

[77] Sun Microsystems. Java technology. http://java.sun.com/.

[78] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Technical report, Cornell University, 1999.

[79] Gleb Naumovich and Paolina Centonze. Static analysis of role-based access control in J2EE applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[80] EIQ Networks. Systemanalyzer. http://www.eiqnetworks.com/products/systemanalytics.shtml.

[81] Tim O'Reilly. On understanding the technology book market. http://www.oreillynet.com/pub/wlg/5573.

[82] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems 1*, 1:11–32, 1988.

[83] IBM Research. Jikes research virtual machine (JikesRVM). `jikesrvm.sourceforge.net`.

[84] Anne Rogers, Martin Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[85] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63(9), pages 1278–1308, September 1975.

[86] Stefan Savage, Michael Burrows, Greg Nelson, and Patrick Sobalvarro. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[87] D. Scott and R. Sharp. Spectre: A tool for inferring, specifying, and enforcing web-security. Technical report, Cambridge University, 2002.

[88] Matthew L. Seidl and Benjamin Zorn. Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, University of Colorado, 1997.

[89] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.

[90] Julian Seward and Nick Nethercote. Valgrind, an open-source memory debugger for x86-gnu/linux. `http://developer.kde.org/~sewardj/`.

[91] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Symposium on Principles of Programming Languages*, pages 295–306, 2002.

[92] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS/Performance*, pages 194–205, 2001.

[93] Stephen Smaha. Haystack: An intrusion detection system. In *In Proceedings of the 4th Aerospace Computer Security Applications Conference*, pages 37–44, December 1988.

[94] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, Boston, MA, January 2000.

[95] Psionic Software. Logsentry. `http://sentrytools.sourceforge.net`, 2003.

[96] S. Soman, C. Krintz, and G. Vigna. Detecting malicious Java code using virtual machine auditing. In *Proceedings of the Twelfth USENIX Security Symposium*, 2001.

[97] Anil Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 2002.

[98] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[99] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[100] Symantec. Security response: Javaapp.strangebrew. `http://securityresponse.symantec.com/avcenter/venc/data/javaapp.strangebrew.html`.

[101] Symantec. Security response: Java.beanhive. `http://securityresponse.symantec.com/avcenter/venc/data/java.beanhive.html`.

[102] Transvirtual Technologies. Kaffe 1.0.6. `http://www.kaffe.org`, 2000.

[103] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *Proceedings of the International Semantic Web Conference (ISWC 03)*, Sanibel Island, Florida, 2003.

[104] `Perl.org`. Parrot. `http://www.parrotcode.org/`.

[105] B. Vasanth, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming, Language Design and Implementation*, 2000.

[106] VirusList.com. not-a-virus: Javaclass.port25. `http://www.viruslist.com/en/viruses/encyclopedia?virusid=62347`.

[107] VMWare. VMWare an EMC company. `http://www.vmware.com/`.

[108] David Wagner and Drew Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy*, 2001.

[109] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM Press.

[110] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[111] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[112] David Welton. Programming language popularity. `http://www.dedasys.com/articles/language_popularity.html`.