Improving the Static Resolution of Dynamic Java Features

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Jason Sawin
Graduate Program in Computer Science and Engineering

The Ohio State University

2009

Dissertation Committee:

Atanas Rountev, Advisor

Timothy Long

Neelam Soundarajan

ABSTRACT

In Java software, two important flexibility mechanisms are dynamic class loading and reflection. Unfortunately, the vast majority of static analyses for Java handle these features either unsoundly or overly conservatively. Our work targets techniques that will increase static analyses' ability to handle dynamic features in a more precise manner.

Since many of these dynamic features rely on string values to specify their run-time behavior, some static analyses have used string analysis to aid in resolution of such features. There are two main concerns with this practice: (1) often a string analysis is not powerful enough to accurately model the needed string values, and (2) the computing costs associated with a precise string analysis make it impractical to incorporate into many static analysis frameworks. We address the first concern by presenting a novel semi-static approach for resolving dynamic class loading by combining static string analysis with dynamically gathered information about the execution environment. The insight behind the approach is that dynamic class loading often depends on characteristics of the environment that are encoded in various environment variables. Such variables are not static elements; however, their run-time values typically remain the same across multiple executions of the application. Additionally, we propose extensions of string analysis to increase the number of sites that can be resolved purely statically. An experimental evaluation on the Java 1.4

standard libraries shows that a state-of-the-art string analysis, Java String Analyzer (JSA) [19], resolves only 28% of non-trivial sites while our approach resolves 74% of such sites. We also demonstrate how the information gained from resolved dynamic class loading can be used to determine the classes that can potentially be instantiated through the use of reflection. Our extensions of string analysis greatly increase the number of resolvable reflective instantiation sites.

For string analysis to be useful for resolution of dynamic features, it has to exhibit practical cost in term of running time and memory usage. We propose several techniques to improve the scalability of JSA thus making it more practical for incorporation into a static analysis framework. Our approach parallelizes a significant portion of JSA, allowing it to take advantage of modern multi-core architectures. We also present several new simplifications to JSA's internal representation of the flow of string values through an application. These simplifications reduce the amount of irrelevant information processed by JSA. We applied an implementation of our proposed enhancements to 25 benchmark applications. For all benchmarks, our implementation realized a speedup when compared to the original version of JSA. For two benchmarks, the speedup was over 180 times. Moreover, the original version of JSA was unable to complete its analysis of three benchmark applications because it exhausted the allotted 6Gb of heap memory. Our implementation was able to easily complete the analysis of all 25 benchmarks.

With the cost of precise string analysis reduced, we incorporate it and our semi-static approach into a Class Hierarch Analysis (CHA) call graph construction algorithm. A call graph abstractly represents the calling relationships between methods in a program. It is a critical component of many static analyses. We investigate how a

hierarchy of assumptions allow for the incorporation of techniques to resolve instances of certain dynamic features. We implemented a unique CHA call graph construction analysis for each level of the assumption hierarchy. These implementations were applied to 10 benchmark applications in an experimental evaluation of the effects of the assumptions and the corresponding resolution techniques. The results of this study indicate that even a slight relaxing of the fully conservative assumption can lead to a call graph with 44% fewer edges without the aid of any resolution techniques. By incorporating assumptions about casting operations and string values, it is possible to remain conservative and reduce the number of edges in the graph by 54% through the use of various resolution techniques. On average, our most precise implementation was able to resolve 6% of the reflective invocation sites, 50% of dynamic class loading sites, and 61% of reflective instantiation sites encountered by the analysis.

This work is a step toward making static analysis tools better equipped to handle the dynamic features of Java. These include tools that facilitate software development, testing, and understanding. Increasing the precision of these tools can decrease development costs and increase software reliability.

To Mom and Pops

ACKNOWLEDGMENTS

Many people have helped to guide and support me on my journey to earning a Ph.D., none more so than my academic adviser and mentor Atanas Rountev. Without his constant feedback, guidance, and support, I doubt I would have achieved my goal. He has taught me how to be a researcher, and more importantly how to be paranoid about my work. Thank you Nasko.

I also need to thank all the faculty and staff of The Ohio State University's Computer Science and Engineering Department. They have provided me with an excellent education. In particular, I would like to thank Timothy Long and Bruce Weide for teaching me how to be an effective instructor of Computer Science and for their valuable feedback on my research. Neelam Soundarajan has encouraged me and helped to guide my research since my first year at OSU. Thanks guys. I also would like to thank Peg Steele and Nikki Strader for teaching me how to advise undergraduate students and for being so flexible.

Finally, I would like to thank all of my friends and family for their support and help. I couldn't ask for more supportive or better parents. Thanks for everything, Mom and Dad. I took comfort in knowing that my brother and sisters were always there to assist me if I stumbled. I cannot imagine a more supportive, understanding, and loving companion than Heather Bandeen. I am eager to start our future adventures together. Richard Sharp has not only been a great friend but has helped

guide me through graduate school and beyond. When it was midnight and I was burned out, I knew I always had a sympathetic ear with Matt Lang. Thanks Matt. I also need to thank my many friends, and often times proofreaders: David Chiu, Scott Kagan, Lindsay North, Raffi Khatchadourian, Julia Valigore, Gregory Buehrer, Quentin Froemke, Kelly Streeter, Evan Elken, Brenda Sodowsky, Shawn Poindexter, Becki Witherow, Adam Sommers, Mike Steffen and so many others.

Thank you everyone. Your help has meant more to me than I can possibly convey.

The material presented in this dissertation is based upon work supported by the National Science Foundation under Grant No. 0546040. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

June 2008	M.S. Computer Science & Engineering, The Ohio State University
May 1995	B.A. Mathematics and Computer Science, Lewis and Clark College
September 2003 – present	Graduate Research/Teaching/Administrative Associate, The Ohio State University
August 29, 1976	Born – Joseph, Oregon

PUBLICATIONS

Research Publications

J. Sawin, and A. Rountev. Improving Static Resolution of Dynamic Class Loading in Java Using Dynamically Gathered Environment Information. In *International Journal of Automated Software Engineering (JASE)*, volume 16, number 2, pages 357-381, June 2009.

J. Sawin, and A. Rountev. Improved Static Resolution of Dynamic Class Loading in Java. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 143-154, October 2007.

R. Khatchadourian, J. Sawin, and A. Rountev. Automated Refactoring of Legacy Java Software to Enumerated Types. In *IEEE International Conference on Software Maintenance*, pages 224-233, October 2007.

J. Sawin, M. Sharp, and A. Rountev. Generating Run-Time Progress Reports for a Points-to Analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 40-44, October 2006.

J. Sawin, and A. Rountev. Estimating the Run-Time Progress of a Call Graph Construction Algorithm. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 53-62, September 2006.

M. Sharp, J. Sawin, and A. Rountev. Building a Whole-Program Type Analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 6-10, October 2005.

A. Rountev, S. Kagan, and J. Sawin. Coverage Criteria for Testing of Object Interactions in Sequence Diagrams. In *Fundamental Approaches to Software Engineering*, pages 282-297, April 2005.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Software Engineering	Prof. Atanas Rountev
Distributed Systems	Prof. Gagan Agrawal
Database Systems	Prof. Hakan Ferhatosmanoglu

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Modern software applications need to be highly adaptable and flexible to stay competitive. Long-running web systems must be able to swap out and update components without interrupting services. Applications are expected to perform similarly on multiple operating systems, under various execution environments. Software users are demanding the ability to customize their applications to a degree that has never been seen before. To meet this demand, more and more applications support third-party extensions. The use of extensions allow these applications to stay current and relevant without requiring them to absorb the resulting massive development costs.

This increased application flexibility limits what can be determined statically about a program. One significant limitation is the lack of access to code for program components, e.g., third-party extensions that are not available at analysis time, or modules that have yet to be developed. However, even if all code entities are available, most static analyses would not be able to accurately analyze modern software systems. This is because the language constructs that make this unprecedented level of flexibility possible are largely viewed as a nuisance by the static analysis community. We present techniques which will allow static analyses to more precisely address a subset of these features.

1.1 Improved Resolution of Dynamic Class Loading

An example of dynamic constructs in Java are the methods that enable *dynamic class loading*. These powerful language features allow Java applications to load classes into the JVM at run time, requiring only a string representation of the class' fully-qualified name. The newly loaded classes can then be manipulated through the use of reflection. In the most general case, there is no way to determine which entities will be loaded until run time. As a result, many static analyses either choose to ignore dynamic class loading constructs, thus producing an unsound result, or handle them in such a conservative fashion that meaningful results are obfuscated by infeasible interactions.

Some recent work has employed *static string analysis* to allow for a more precise treatment of these dynamic features. Such an approach statically attempts to determine the value of the string that specifies the target class to be loaded. For example, a call `Class.forName(s)` dynamically loads the class with the name represented by the string expression `s`. If through static string analysis, the precise run-time value of `s` could be determined, the statement could be treated as a static initialization of the class specified by `s`. Current string analysis approaches have two potential points of failure when trying to determine the value of `s`: (1) when the value of `s` is not a compile-time constant, and truly depends on the run-time execution, and (2) when the analysis is not powerful enough to model the flow of the string value through the application. Unfortunately, the use of such truly-dynamic values and complex string manipulations is common when designing a flexible application. For example, many applications will inspect environment variables, configuration files, or particular directories to determine which extensions are available. In such cases any purely

static analysis will fail to produce a precise result. Similarly, many applications use data structures and perform string operations that are currently beyond the modeling capabilities of string analyses.

We present a novel semi-static approach which combines static string analysis with dynamically gathered information about the execution environment. We implement this approach as an extension to the current state-of-the-art string analysis for Java: Java String Analyzer (JSA) [19]. The key insight behind this approach is the observation that dynamic class loading often depends on characteristics of the execution environment that are encoded in various *environment variables*. Our investigation of the Java 1.4 standard libraries revealed that over 40% of the client-independent dynamic loading sites—i.e., ones that could not be affected directly by client code—depend upon environment variables. Though such variables are not static elements of an application, they are different from other forms of dynamic input data, because their run-time values typically remain the same across multiple executions of the application. Our approach identifies dynamic class loading sites that depend only on such variables, and resolves them based on the current variable values. As part of this approach, we also propose several extensions of static string analysis that improve the tracking of the names of environment variables. These extensions increase the ability of the string analysis to model the flow of string values, thus increasing the number of sites that can be resolved.

The proposed approach produces results that are sound with respect to the current execution environment and the configuration of the analyzed application, but do not apply to all possible environments and configurations. For many clients of static analyses, this is both reasonable and desirable. For example, consider program

understanding tools such as SHriMP [131] or Rigi [100]. Such tools have the potential to overwhelm their users with too much information [132]. If such tools tried to account for *every* class that can potentially be loaded at dynamic class loading sites for *all* possible combinations of environment variable values, their usefulness may be compromised. Instead, by using our approach, the user can obtain information that is sound for her own local environment (i.e., for the specific environment variable values that capture component configurations, operating system parameters, etc.).

This work makes the following contributions:

- We propose a fully automated semi-static approach that utilizes the system's current configuration information to aid in the resolution of dynamic class loading in Java applications. This approach defines a useful and practical relaxation of purely static approaches for handling of dynamic class loading.

- We present several extensions of string analysis that not only enable our approach to resolve more instances of dynamic class loading sites which depend on environment variables, but also allow for a greater number of purely static instances to be resolved.

- We describe an experimental study in which the approach was applied to the entire Java 1.4 standard libraries. The results of this experiment indicate that the approach is able to resolve 46% more client-independent sites than the state-of-the-art static string analysis, with an increase in analysis cost of a few tenths of a second per thousand statements. Through comprehensive manual investigation we also determined that our approach identifies 87% of all sites

that are, in fact, truly static or environment variable dependent, which implies high analysis precision.

- We present a second study that demonstrates how information gained from resolved dynamic class loading sites can be used to aid in the determination of classes that can potentially be instantiated through the use of reflection. The results of this study show that the additional information gained through the use of our approach increases the number of resolvable reflective instantiation sites from 6 to 37 in the Java 1.4 standard libraries. Moreover, 70% of the resolved instantiation sites transitively depend on environment variables and thus could not be resolved through purely static techniques.

These experimental results indicate that the proposed approach represents a significant improvement for the handling of dynamic class loading in static analysis for Java, compared to current techniques. Such an improvement could be valuable for a range of software tools that employ static analyses to support software understanding, transformation, verification, and optimization. However, before JSA can be integrated with many of these tools, its scalability will need to be improved. Our study applied JSA to the individual packages of the Java 1.4 standard libraries. The resource utilization measurements of the experiments suggest that JSA scales well to this size of input. However, when we attempted to apply JSA to the entire library or even just multiple packages, it exhausted the allotted 1.5 Gb heap and failed to complete its analysis. This lack of scalability implies that JSA, in its current form, may not be practical for use by interprocedural analysis such as call graph construction analyses. These analyses often analyze thousands of classes, an input size to which JSA does not scale well.

1.2 Improving the Scalability of String Analysis

Many static analysis frameworks could benefit from the incorporation of a string analysis. As described above, a string analysis can aid in the resolution of dynamic features such as dynamic class loading, thus increasing the precision of certain static analyses. String analysis has also been used in frameworks attempting to statically validate dynamically generated SQL statements [53,54,60]. It has even been employed to aid in the understanding of software application interfaces [92]. Clearly, being able to answer the question "What are the possible run-time values represented by a string variable?" can benefit many different static analyses.

Unfortunately, many static analyses have been slow to adopt existing string analyses. For example, the analyses presented in [71,126,139,144] could all improve their precision through the careful incorporation of a robust string analysis[1]. However, at best, they incorporate a very imprecise analysis that will only validate hard coded string literals even though there are several publicly available string analysis libraries which are much more powerful. This hesitation to incorporate a more powerful string analyses may be due in part to the computing expense associated with precise modern implementations of string analyses.

Currently one of the most precise string analyses for Java is JSA. The input to JSA is a set of Java classes (*application classes*) and a set of expressions or "*hotspots*". JSA conservatively estimates the possible run-time string values at all instances of those hotspots in the input classes. JSA's design consists of two distinct components: (1) a *front-end* component creates a graph which represents the possible flow of string

[1]All of these analyses rely on or construct a method call graph which does not provide a precise treatment of dynamic class loading and reflection.

values through the input classes and (2) a *back-end* converts the graph into a context-free grammar and ultimately generates an over-approximating finite state automaton for each hotspot discovered in the input classes. Our experiments show that the running time and memory usage of JSA can vary greatly even for inputs of similar size (in terms of number of bytecode instructions). For example, the applications JFlex and JGap both contain a little over 15,000K instructions. JSA can fully analyze JFlex in under 10 seconds using only 30MB of memory, however it will exhaust a 6Gb heap and fail to finish analyzing JGap after hours of running.

We explore a variety of techniques to reduce the memory footprint and running time of JSA. First, we present a series of algorithmic transformations to JSA's front-end component. These transformations allow the majority of the work performed by this portion of JSA to run in parallel. Such a design allows JSA to take advantage of modern multi-core systems.

We also present several new simplifications to the flow graph generated by JSA's front-end. These simplifications preserve the relevant details of the graph but have the potential to greatly reduce the running time of the back-end.

The specific contributions of this work are:

- We propose a series of algorithmic transformations to JSA. These transformations represent an evolution from the most intuitive parallelization of JSA to a more efficient version. This exploration details many of the challenges that developers will face when designing Java applications for modern multi-core architectures. On average, the most advanced transformation enabled JSA to

realize an average speedup of 1.54 times when compared to the front-end building times of the sequential version of JSA. Our transformations also reduced the average memory footprint of JSA by 43%.

- We present three new reductions to the flow graphs created by JSA's front-end. These simplifications preserve all relevant details of the graph while removing much of the extraneous information. These reductions increase the average running time of JSA's front-end by a few 100 milliseconds but have the potential to greatly reduce the running time of the back-end. For two benchmarks applications in our empirical study, these simplifications produced a speedup of over 180 times. Moreover, the simplifications allow JSA to complete an analysis of three applications that previously exhausted memory resources.

- We implemented all of our proposed transformations to JSA. These implementations are applied to 25 benchmark applications in an extensive empirical study. This study not only demonstrates the speedup realized by our proposed enhancements, but also provides insights into some of the challenges faced by the developers of parallel Java applications.

By lowering the overhead associated with precise string analysis, we are making it more feasible to incorporate into existing static analyses frameworks. Frameworks that could benefit from a string analysis include those which create fundamental artifacts such as method call graphs (a basic building block of many other static analyses). Increasing the precision of such an analysis would transitively increase the precision of numerous other analyses.

1.3 Assumption Hierarchy for a CHA Call Graph Construction Algorithm

A call graph construction analysis produces a method call graph. This graph abstractly represents the calling relationships between program methods. The nodes of a call graph represent methods and directed edges represent calls between methods. A call graph is a vital component for numerous interprocedural static analyses, e.g., [19, 41, 42, 45, 46, 62, 71, 78, 104, 107, 109, 111, 128, 143].

The existence of dynamic Java features, whose exact run-time behavior cannot be determined by solely examining the static representation of the code, requires that call graph construction algorithms make assumptions about the possible run-time effects of such features. These assumptions will be reflected in the graphs generated by the algorithm. For example, consider a call graph construction algorithm that assumes the analyzed code does not make use of reflective features. The graph it produces for applications that do use reflection will not be sound (i.e., it will not represent all possible run-time calling relationships) as the reflective calls will not be represented. It has been shown that by disregarding dynamic features, call graphs may not represent a significant portion of the actual run-time calling relationships [89]. It is important for clients of a Java call graph analysis to be cognizant of the assumptions that it makes about dynamic features. Such assumptions could have ramifications on the soundness of a client analysis.

We present a hierarchy of assumptions that a *Class Hierarchy Analysis* (CHA) call graph construction algorithm [27] could make about the dynamic features of Java. At the top of the hierarchy is the most conservative assumption, which provides a sound, yet very imprecise, treatment for certain dynamic features. Each consecutive

level of the hierarchy extends the preceding level by adding more assumptions. These additional assumptions allow for a more refined treatment of certain dynamic features. Consequently, graphs created at each level of the hierarchy are subgraphs of those generated by the preceding levels. The hierarchy terminates in a set of assumptions that allows for the use of (1) type information for cast operations, (2) static string values, and (3) semi-static string values to aid in the precise resolution of certain calls to dynamic class loading methods, reflective instantiation, and reflective invocation.

We also propose a similar assumptions hierarchy for a *May Be Loaded* (MBL) analysis. Given an application P, this analysis estimates the set of classes that may be loaded into the JVM during any execution of P. This analysis is often used to determine the set of classes that a CHA analysis must consider when building a call graph for P. The results of MBL are fundamental to the closed-world assumption under which CHA operates.

This work makes the following contributions:

- We present a detailed discussion of how custom class loaders, dynamic class loading, native methods, and reflection can cause both an MBL analysis and a CHA analysis to be unsound. This discussion highlights many challenges that interprocedural static analyses of Java will have to address in order to be sound.

- We propose a hierarchy of assumptions about these dynamic Java features for a CHA call graph construction algorithm and a MBL analysis. The CHA assumption hierarchy could be extended to create a taxonomy from which existing and future call graph construction algorithms could be categorized. For each level of the hierarchy we specify techniques the analysis can use to address certain dynamic features. These techniques include a novel approach based on the

assumption that dynamic features will respect data encapsulation. They also include using our semi-static string analysis introduced above. We believe this is the most precise string analysis to be incorporated into a CHA analysis to date.

- We implemented a version of the CHA analysis and the MBL analysis for each level of their assumption hierarchies. These implementations were applied to 10 real-world Java applications in an empirical study. This study provides a concrete example of the effects of each assumption and the corresponding resolution techniques on the results of these analyses. On average, the implementation of CHA that incorporated all of our resolution techniques was able to resolve 6% of the reflective invocation sites, 50% of dynamic class loading sites, and 61% of reflective instantiation sites it encountered. This capability enabled this version to generate graphs that, on average, contain 10% fewer nodes and 54% fewer edges than the graphs created by the fully conservative version.

1.4 Outline

The remainder of this dissertation is organized as follows: Chapter 2 presents an overview of certain dynamic Java features. It also provides a description of JSA as well as background information about parallelism in Java. Chapter 3 presents our approach for incorporating dynamically gathered information from the execution environment into a string analysis, and how this information can improve the resolution of certain dynamic Java features. Chapter 4 discusses our proposed techniques for improving the scalability of JSA. Chapter 5 presents our evaluation of the effects of various assumptions pertaining to dynamic features of Java on a MBL analysis and

a CHA call graph construction analysis. Past work which relates to this dissertation is described in Chapter 6. Finally, Chapter 7 describes possible future directions of this work and concludes the dissertation.

CHAPTER 2: BACKGROUND

This chapter provides background information that relates to topics covered in the remainder of the dissertation. It includes an overview of dynamic Java features, a design description of the Java String Analyzer library [19], and a discussion of parallelism in Java.

2.1 Dynamic Features of Java

The flexibility of the Java platform has helped propel it to its current level of popularity. The fact that Java is designed to be executed by a virtual machine means that it is platform-independent. Dynamic features, such as reflection and dynamic class loading, allow Java applications to incorporate third party extensions increasing their customizability. Through the use of native methods, Java programs can interface with legacy applications, access platform specific resources, and execute high-performance code. These and other features, combined with the relative ease of programming, have made Java a natural choice for many large and complex applications.

Though these features are of great benefit to application developers, they pose significant challenges to creators of static analyses. The recent explosion of Java applications that support extensions increases the chance that an analysis will not have access to all relevant code. The interactions that take place between Java applications and native methods will often go unaccounted for, or at best, be poorly modeled due to lack of access to the native method's source code. Even if the source code of a

native method was available, the vast majority of Java analyses would not be able to analyze them since, by definition, they are not written in Java.

Even if an analysis had access to all the program components of an application and was able to address all the language features of each, in the most general case, it would still be impossible to precisely model all possible run-time interactions of an application. This somewhat disheartening situation is due to the truly dynamic nature of some of the Java constructs. This section provides an overview of some of the more commonly used dynamic Java features.

2.1.1 Dynamic Class Loading in Java

The Java Virtual Machine (JVM) is one of the defining components of the Java platform [87]. It interprets Java bytecode, allowing Java applications to be platform independent. It also supports dynamic class loading, which is the ability to load classes at run time [86]. This is a powerful mechanism that allows classes to interface with software components that are specified at run time and, in fact, do not even need to exist at compile time. This feature is a key mechanism that allows modern applications to achieve the desired level of flexibility.

Loading classes into the JVM is the responsibility of class loaders. At its simplest, a class loader takes a string representation of the fully-qualified name of the class that is to be loaded and performs a search for the corresponding class file. Upon finding the class file, the loader loads the bytecode into the JVM and returns a `Class` object. This is a metadata object through which the program can access the class (e.g., to create class instances).

```
1  private final String handlerPropName =
2                "sun.awt.exception.handler";
3  private String handlerClassName = null;
4
5  private boolean handleException(Throwable thrown) {
6     .....
7     /* Get the class name stored in environment
8     * variable sun.awt.exception.handler */
9     handlerClassName = (String) AccessController.doPrivileged(
10                       new GetPropertyAction(handlerPropName));
11    .....
12    /* Load the class and instantiate it */
13    Object h;
14    Class c = Class.forName(handlerClassName,...);
15    h = c.newInstance();
16    .....
17 }
```

Figure 2.1: Sample code from library class `java.awt.EventDispatchThread`.

Example. Figure 2.1 illustrates the flexibility an application can gain from the use of dynamic class loading. We revisit this example several times throughout the rest of the dissertation. The code is from `java.awt.EventDispatchThread` and allows custom-defined event handlers to be loaded in a running application. If a client wishes to use a custom event handler, all she needs to do is create the appropriate class and set the environment variable with the key `sun.awt.exception.handler` to the string representing the fully-qualified name of this class. The method `handleException` contained in `EventDispatchThread` queries this environment variable to retrieve the specified class name (lines 9 and 10) and stores the value in field `handlerClassName`. The custom handler is then loaded at line 14. Method `forName` is one of several methods in the Java libraries that can be used to dynamically load classes. A call to `newInstance` is used to create a new object of the class; this call has the same effect as calling the no-arguments constructor of the class.

```
1 private void printGlobals () throws Exception{
2     Class clz = this.getClass();
3     Field [] flds = clz.getFields();
4     for(int i = 0; i < flds.length; i++){
5       if(Modifier.isFinal(flds[i].getModifiers())){
6         System.out.println(flds[i].getName()+"="+flds[i].get(this));
7       }
8     }
9 }
```

Figure 2.2: Reflection example: prints the name and value of all `final` fields.

2.1.2 Reflection

A Java feature that is commonly used in conjunction with dynamic class loading is *reflection*. Through the use of reflection, Java applications are able inspect their structure and modify their behavior at run time. The reflective aspects of Java are implemented in the standard Java libraries' package `java.lang.reflect`. The entry points for this functionality are instances of class `java.lang.Class`—the return type of dynamic class loading methods. Through `Class` objects, the program can make reflective calls to methods `getDeclaredFields`, `getDeclaredMethods`, `getConstructors`, etc. The objects returned by these methods make it is possible to modify the values of fields, invoke methods, and create new instances of the underlying class.

The `printGlobals` method shown in Figure 2.2 demonstrates how reflection can be leveraged to efficiently (in terms of lines of code) display the values of `final` fields contained in a class. The method first gets the `Class` object for its declaring class via the call to `getClass`. This is a method that every class inherits from `java.lang.Object`. It returns the `Class` meta-data object for the calling instance. The `getFields()` method invoked at line 3 is a reflective call that returns an array

containing `Field` objects. `Field` is a metadata object similar to `Class` that provides information about, and dynamic access to, a single field. The call to `getModifiers` at line 5 returns the Java language modifiers for the field represented by `flds[i]`. If the modifier is `final`, then the name of the field and its current values are printed to `stdout`. The name of the field is retrieved through the call `flds[i].getName()`, and its current value for the object pointed to by `this` is retrieved by `flds[i].get(this)`. (If the field is static, the parameter in the call to `get` is irrelevant.)

Without reflection, the coding of a method that prints all the `final` fields of a class could be a potentially arduous task. Consider a class that declares hundreds of such fields. Without reflection, each field `fld` would require a line of code similar to `System.out.println("<name>"+this.fld)`. Moreover, such a method would have to be updated every time a field was added or removed.

Of course, reflection is much more powerful than just printing field values. For example, through the `Field` method `set(Object,Object)`, it is possible to modify a field's value. Class `Method` contains the method `invoke(Object,Object[ ])` which can be used to execute the method represented by `Method` on `Object` with the parameters specified in `Object[ ]`. It should be noted that `Method` cannot represent constructors or static initializer methods (these are artificial methods which initialize static elements of a class). However, new instances of objects can also be created using the `newInstance` method of `java.lang.reflect.Constructor`, or through invocations of `Class.newInstance()`.

Reflection does *not* need to respect encapsulation. By using `AccessibleObject`, it is possible for objects of type `Field`, `Method`, and `Constructor` to circumvent the attribute visibility rules of Java. The superclass of `Field`, `Method`, and `Constructor`

is `AccessibleObject` from which they inherit the `setAccessible(boolean)` method. By invoking this method with the boolean value `true`, a reflective object can suppress the Java language access checking, thus gaining access to all attributes even if they have `protected`, `package`, or `private` visibility.

2.1.3 Native Methods

Native methods are methods that are written in *native programming languages*. There are many situations where it is necessary for a Java application to interface with native methods. For example, a systems that has migrated from C to Java may still need to be able to interface with legacy components written in C. Often, situations occur where a platform-independent Java application requires access to platform-dependent resources (e.g., modem or I/O systems). Such resources are typically accessed through native methods. It may be desirable to implement computationally expensive components of a system in a more efficient native method. For all of the these reasons and more, the Java platform provides its clients with the *Java Native Interface* (JNI) [85]. JNI is an interoperable interface that allows Java classes to *callout* to methods contained in native libraries. It also allows native code to make *callbacks* to Java methods.

Through the use of JNI, it is possible for a native method to create, inspect, and modify objects in a JVM. Native methods can even load new classes into the JVM; essentially a native method has all capabilities of a standard Java method.

2.1.4 Custom Class Loaders

As stated above, Java is an interpreted language. This means that Java applications differ from applications written in conventional compiled languages such as C

or C++. A Java application is not packaged into a single executable file; rather, it consists of many separate class files. The class files of a single application need not, and often are not, loaded into memory at the same time. Typically, classes are loaded into the JVM on demand.

Class loaders are software components that are responsible for loading Java class files into the JVM. During the loading process a typical class loader first locates the file containing the bytecode for the class that is to be loaded. It then verifies that the class is structurally well-formed and does not perform any actions that are not allowed. If the class passes verification the loader prepares for its initialization, which involves allocation of memory for static members of the class, and creation of various structures such as the method tables and object templates. Finally, the class loader initializes the class which requires the execution of the class' static initialization method.

We say these are the actions of a "typical" class loader because custom class loaders need not follow this behavior. Custom class loaders are user extensions of `java.lang.ClassLoader`. Custom class loaders are commonly used to specify alternative locations from which to load class files, instrument bytecode, and partition user defined classes in servers.

The existence of custom class loaders presents a significant challenge to most static analyses. Since custom loaders can load classes from sources that are not included on the classpath, or even on the current system, it can be difficult for whole-program analyses to determine all the possible classes that may be loaded during the execution of an application. The ability of custom class loaders to instrument the bytecode of a class means that the semantics of the class can change between compile time and run

time. Moreover, custom loaders need not load the class specified but could substitute another class or dynamically create a new class.

2.1.5 JVM Interactions and Implicit Calls

There are many different implementations of the Java Virtual Machine specification [87]. Some of the implementations are designed to allow Java applications to run on specific operating systems, others are designed to allow higher performances. A single Java application given the same user inputs may exhibit different run-time behaviors on two different implementations of the JVM even though both implementations are correct with respect to the specifications. This is due to the nondeterminism present in the specification.

One example of this nondeterminism is the requirement for startup of the VM. The specification states that

> The Java virtual machine starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader.

In other words, it is up to the developer of a specific implementation as to exactly how the JVM should begin execution. Some VM implementations will load and initialize a number of Java library classes that are commonly used by applications [8]. This can increase the efficiency of a running application that accesses one of these classes since the class is already loaded. This nondeterminism means that the VM can perform actions on startup that could possibly affect the run-time behavior of an application.

Another ambiguous requirement is the JVM's invocation of a `finalize` method. The root class `Object` contains a `protected` method `finalize` that can be overridden by subclasses. The JVM specification requires that the VM invoke an object's

```
1  class Grigri{
2     static Boolean useGrigri;
3     static{ //static initializer block
4         if(ATC.useATC){
5            System.err.println("Already␣belaying␣with␣an␣ATC");
6            useGrigri = false;
7         }else{
8            useGrigri = true;
9         }
10    }
11    public static void beginBelay(...){
12         ...
13    }
14 }
```

Figure 2.3: Example of a static initializer block.

`finalize` method before its storage is reclaimed by the garbage collector. The intent of this method is to allow for the release of resources that cannot automatically be released by the garbage collector. However, there is no requirement for exactly when the VM should invoke a `finalize` method. Moreover, the method may never be invoked if the VM exits without performing a final garbage collection.

There are a number of code entities like `finalize` that are meant to be called implicitly by the JVM. For example, some classes contain a static initializer block. This block of code is similar to a static method except that it has no method name, no return type, and no parameters. A common use of static initializer blocks is to initialize static fields, but the code contained in a block can be arbitrarily complex. An example of a static initializer block is shown in Figure 2.3 at lines 3-10. In the example, class `Grigri` first examines a static field of class `ATC` before setting the values of its own static field `useGrigri`. It is possible for a class to declare multiple static initializer blocks. When a class is compiled, the compiler combines all static initializer

blocks and static variable initializers in the order in which they appear in the code into a single static initializer method [39] (for the remainder of the dissertation we will refer to this method as `clinit`). The JVM will invoke any class X's `clinit` only once, immediately before one of the following actions:

- An instance of X is created
- A subclass of X is initialized
- A non-constant static field of X is assigned or used
- A static method of X is invoked
- X is initialized due to certain reflective methods
- X is initialized due to JVM startup

Another method that is implicitly called by the JVM is `Thread.run`. As will be discussed in Section 2.3, a programmer can specify the code that is to run in a thread by extending the `Thread` class and overriding the `run` method. At a call of the form `x.start()`, where `x` is an instance of a subclass of `Thread`, the JVM implicitly calls `x.run()`. This allows the concurrent running of two threads, the current thread (the one calling `x.start()`) and the thread whose execution starts with `x.run()`.

Similar to `Thread.run` is the `run` method of a `PrivilegedAction` object. This method contains computations that are to be performed with privileges enabled, meaning that the code in `run` has access to more resources than the code which caused its invocation. Instances of `PrivilegedAction` are passed to invocations of `AccessControler.doPriviledged`. This method performs a security check to ensure that it is safe to invoke the code with privileges enabled. If the action is allowed, the `run` method is called by the JVM.

2.1.6 Dynamic Features and Static Analysis

The Java features described in Sections 2.1.1–2.1.5 are examples of what we are broadly referring to as "dynamic features". In the general case, these are features whose exact behavior cannot be determined until run time. Dynamic features pose a significant challenge to static analyses.

By definition static analyses analyze static representations of program. These types of analyses are typically designed to discover information about "static" features that appear explicitly in the program code (e.g., method calls and assignments). Traditionally, when a static analysis encountered an instance of a dynamic feature in a program it was analyzing, it would either (1) ignore the feature, possibly producing unsound results, or (2) attempt to model the feature's implicit effects in a conservative manner, often generating a very imprecise result. Recent work has proposed techniques that enable static analyses to use static information to provide a more precise treatment for certain dynamic features.

In the subsequent chapters, we present an investigation of the dynamic features listed above and techniques that allow static analyses to provide a more precise treatment for more instances of them. In Chapter 3 we propose extensions to string analysis that better equip it to precisely resolve string values used at calls to dynamic class loading methods. We also show how the type information from resolved dynamic class loading sites can be used to resolve instances of reflective instantiation (e.g., `Class.newInstance()`). In Chapter 5 we explore several assumptions that a CHA call graph construction algorithm and a MBL analysis could make about dynamic features. The different assumptions allow these analyses to incorporate different treatments and resolution techniques for dynamic class loading, reflection, implicit calls made by the JVM, and native methods. The different treatments for these features can have a profound effect on the results generated by the analyses.

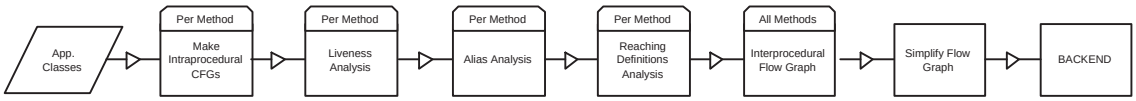

Figure 2.4: Stages of JSA's front-end.

2.2 Java String Analyzer

One of the most powerful string analysis currently available for Java is the *Java String Analyzer* (JSA) library [19]. The input to JSA is a set of Java classes (*application classes*) and a set of expressions (*hotspots*). JSA conservatively estimates the possible run-time string values at all instances of these hotspots in the input classes. It should be noted that JSA uses the Soot analysis framework [144] to generate the Jimple intermediate representation of the application classes. It is this representation that JSA uses to begin its analysis. For the rest of this dissertation this conversion from bytecode to Jimple code is considered a preprocessing step.

JSA's design consists of a *front-end* and a *back-end*. The front-end takes the Jimple representation of the input classes and constructs a flow graph that abstractly represents the flow of string values through the application. This flow graph is the input to the back-end. The back-end builds a context-free grammar based on the information represented in the graph. From the grammar it ultimately generates an over-approximating finite state automaton for each hotspot in the input classes.

2.2.1 The Front-End

Figure 2.4 depicts an overview of the algorithmic design of JSA's front-end. It consists of six stages:

1. **Intraprocedural CFGs**: JSA assumes that all methods in the input classes could be executed at run time. This stage creates a control flow graph (CFG) for each of these methods, including `clinit` methods. The nodes of the graph represent statements of an intermediate representation specific to JSA. This representation is a further abstraction of the Jimple code. Nested Jimple expressions are flattened by using synthetic local variables and assignments. JSA is only concerned with determining string values, thus only operations that pertain to variables of type `String`, `StringBuffer`, `StringBuilder`, and `Array` with a base type of `String`, method calls, and return statements are modeled precisely in the CFGs. All other operations are modeled by special `NOP` nodes. In this stage, JSA uses the class hierarchy analysis available in Soot to determine all possible targets of virtual calls. For each target discovered, JSA creates a unique branch and a unique `Call` node in the CFG.

2. **Liveness Analysis**: This stage performs an intraprocedural liveness analysis [3] on each of the CFGs created in Stage 1. A liveness analysis determines which variables are live at each "point" in a program. A variable is considered live at a specific program point if it can be used in computations past that point. In the case of JSA, the liveness analysis determines which variables are live at each node in the CFG. This information is saved in a `Map` data structure to be used in later stages.

3. **Alias Analysis**: The results from the liveness analysis and the CFGs are inputs to this stage. JSA performs an intraprocedural *must/may* alias analysis on each of the CFGs. This flow analysis identifies pairs of variables that *must* point to the same memory location and pairs that *may* point to the same location. The

results of the liveness analysis are used to identify and eliminate pairs in which one variable is no longer live.

4. **Reaching Definitions Analysis**: The results of the previous two flow analyses are used in an intraprocedural reaching definitions analysis [3]. A definition (a statement) s of variable v reaches a CFG node n if there exists a path from s to n on which v is not redefined. The results of the liveness analysis are used to remove variables that are no longer live. The results of the alias analysis are used to ensure that all definitions that are possible through aliasing relationships are accounted for.

5. **Interprocedural Graph Building**: After the dataflow analyses have been completed, JSA uses the results to aid in the creation of the final flow graph. First JSA creates the nodes of the graph. This is achieved by iterating over all statements in every method contained in the input classes. For each variable defined by a statement (including those which could be affected due to aliasing) JSA creates one of the following nodes depending on the type of statement: `Init`, `Join`, `Concat`, `BinaryOp`, and `UnaryOp`. `Init` nodes represent the initialization of a string (e.g. `String s = "sloper"`). Such a node does not have incoming edges and is associated with a finite state automaton which represents the value of the string. `Join` nodes model assignments and control join points. `Concat` nodes represent string concatenation, `UnaryOp` nodes represent unary string operations such as `reverse`, and `BinaryOp` nodes model binary string operations such as `insert`.

   Once the nodes have been created, JSA adds the edges to the flow graph. The edges represent a *def/use* relationship. They are created by, again, iterating over all statements in all methods. Each statement s has two sets of variables

associated with it: dv (variables defined by s) and uv (variables used by s). For each variable $uv_i \in uv$ an edge is added from the nodes associated with the reaching definitions of uv_i to the nodes associated with the variables in dv. This process is straightforward for the majority of statements. However, this stage introduces the first interprocedural connections.

For *Call* statements of the form $x = y(p_1, p_2, \ldots)$ the node associated with x is linked to all nodes of the reaching definitions for all variables that could be returned by y. Similarly, all arguments p_i of type `Array`, `StringBuffer`, or `StringBuilder` are linked to the reaching definitions of y's corresponding alias parameters ap_i at all of y's return statements. For each *Methodhead* statement (a synthetic entry statement generated by JSA for each method), JSA iterates over all possible call sites of the corresponding method; this information is recorded earlier during the building of the CFGs. The arguments of each call site are linked to the nodes of the parameters of the called method.

Assignments to array elements, of the form $a[i] = s$, also receive special treatment. Since arrays are treated as sets of values represented by one variable a, JSA first links all previous definitions of a to the current node, and then links the definition of s.

6. **Flow Graph Simplification**: After the complete flow graph has been created, JSA performs several reductions on it. It first merges nodes which are equivalent. Two nodes are equivalent if they are the same type (i.e., `Join`, `Concat`, ...) and have the same incoming edges. `Init` nodes are equivalent if their associated automata are equivalent. Equivalent nodes are merged by arbitrarily deleting one node and its incoming edges. The deleted node's outgoing edges are then added to the remaining node.

Next, JSA removes self loops on `Join` nodes. This is accomplished by simply deleting the looping edge. After looping edges have been removed, `Join` nodes are examined to determine if they have more than one incoming edge. If they don't, they are removed by deleting the node and adding its one incoming edge to each of its successor nodes. This has the effect of compressing sequences of single assignments. Lastly, JSA bypasses concatenation nodes if the first argument of the node has exactly one edge and it comes from an initialization node representing the empty string (i.e., `s=""+z;`). This is accomplished by replacing the `Concat` node with a `Join` and deleting the edge to the empty string `Init` node.

The simplification are applied to the graph in an iterative manner until no further simplifications can be performed.

2.2.2 The Back-End

The back-end of JSA takes the flow graph and constructs a context-free grammar. For each node n in the graph, a nonterminal A_n is added to the grammar along with a set of productions corresponding to the incoming edges of n. These productions are determined by the type of n. For example, if n were a `Concat` node and nodes x and y were predecessors of n, the following rule would be added to the grammar: $A_n \rightarrow A_x A_y$. The production for an `Init` node n is $A_n \rightarrow reg$ where reg corresponds to a regular language. JSA then utilizes the Mohri-Nederhof algorithm [98] to transform the grammar into a strongly-regular context-free grammar. The collection of resulting regular languages are then stored in a multi-level automaton structure (MLFA). An MLFA is a hierarchical directed acyclic graph of nondeterministic finite automata that compactly stores the information for all string expressions in the input classes. From this structure an individual finite state automaton can be extracted for a given

hotspot. The language produced by the automaton is a superset of the possible string values that can occur at that hotspot. It should be noted that the extraction of a finite automaton from the MLFA is, in the worst case, a doubly exponential processes, though Christensen et. al. [19] did not observe any such blowups in their experiments.

2.2.3 The Assumptions of JSA

The version of JSA documented in [19] is said to "conservatively take care of interaction with external classes". This appears to be the extent of the published documentation that details JSA's open-world assumption. In this section we infer the details of this assumption through an investigation of the JSA 1.1.3 source code.

Our investigation revealed that if during its analysis JSA discovers a call to a method which is not contained in the set of input classes, the mutable arguments (those of type `StringBuffer`, `StringBuilder` and `Array`) and the return value of the method call are assigned the *anystring* value. This value signifies that JSA assumes these variables can be any Unicode string value. For the remainder of the dissertation if a variable is said to be *corrupted* it is assumed that it can be any Unicode string value. From this we infer that JSA defines external classes to be any class not included in the set of input classes provided by the user. Thus, any library classes which are automatically loaded by Soot are also considered to be external entities.

JSA corrupts formal parameters of type `String`, `StringBuffer`, `StringBuilder`, and `Array` with base type of `String` of `public` methods. It also corrupts all assignments to and from fields. From this it can be inferred that JSA assumes that external code and dynamic features such as reflection could affect all `public` methods and all fields. Since these are the only formal parameters it corrupts, it can be inferred that JSA assumes that the set of classes provided by the user contains all relevant calls

```
1  Class A{
2    ...
3    private native String returnName();
4    void foo(){
5       System.out.println(returnName());
6    }
7  }
```

Figure 2.5: Example of JSA's treatment of native methods.

to `private`, `protected`, and *package-private* methods, and dynamic features of Java will not affect these methods.

JSA does not corrupt the return values on any of the methods contained in the set of user-provided classes. This implies that JSA assumes that the set of input classes contains all methods that could be called through polymorphism and whose return values could affect hotspots.

JSA's treatment of calls to native methods is problematic. Consider the code shown in Figure 2.5. Assume that class `A` is an input class to JSA and instances of `System.out.println` are hotspots. In this example, the method `foo` makes a call to the native method `returnName` which returns a string value. Since JSA does not have access to the code of the native method, it should return the *anystring* value for the `System.out.println` hotspot. However, JSA does not distinguish the Java declaration of native methods from other methods and it will erroneously process `returnName` as it would any other method. Since `returnName` does not contain a code body JSA would simply create a method "shell" for it. The shell would return an empty value so JSA would return an empty string (i.e. `""`) for the hotspot. This problem does not occur if the native method is not declared in the set of input classes, since JSA would then treat it as any other external method. Therefore, it can

be inferred that JSA assumes that none of the input classes declare a native method whose invocation could affect string values at hotspots.[2]

JSA does not make special provisions for reflective calls or calls to dynamic class loading methods. The compile-time types returned by such calls are not of type `String`, `StringBuffer`, `StringBuilder`, or `Array` with base type of `String`; thus, JSA does not precisely model their values. The assumptions JSA makes about the possible implicit effects of such calls are stated above.

To summarize, we have inferred from an examination of JSA 1.1.3 source code that JSA makes the following assumptions: (1) the set of input classes contains all the code that could call `private`, `protected` and *package-private* methods whose formal parameters could affect string values at hotspots; (2) the set of input classes contains all methods that could be called through polymorphism and whose return values could affect hotspots; (3) none of the input classes declare a native method whose invocation could affect the value of a hotspot; (4) reflection and external code will not affect the values of formal parameters of type `String`, `StringBuffer`, `StringBuilder`, and `Array` with base type of `String` of `private`, `protected` and *package-private* methods whose values flow to hotspots.

In Chapter 3 we present several extensions to JSA. One of these extensions incorporates dynamically gathered configuration information. We also present several extensions that increase JSA's ability to statically model the flow of string values. We show how these extensions increase JSA's usefulness in finding string values used at dynamic class loading sites. In Chapter 4 we introduce techniques that increase JSA's scalability, making it more feasible to incorporate into existing static analyses. Finally, in Chapter 5 we show how the information from JSA can aid a CHA call graph construction algorithm to resolve instances of certain dynamic Java features.

[2]For our experiments we altered JSA to corrupt all arguments passed to, and values returned by calls to native methods and reflective calls.

2.3 Multi-Core Architectures and Java

Modern computer designers are running into the upper bounds of Moore's Law [99] due to overheating problems and power consumption. Rather than increasing the frequency of a single processor, many designers are choosing to use a multi-core architecture. The term *multi-core processor* refers to a processor which has two or more processing cores on a single chip. Since all the cores run in parallel, a multi-core processor can achieve application speedup by dividing the working load among the cores.

There are a variety of cache and memory access configurations for multi-core systems. The popular Intel® Core™ Duo processors use a shared-cache architecture [138]. Each core has its own `L1` cache and share the `L2` cache and memory controller. The cores access main memory through a shared bus even if there are multiple chips containing multiple cores. AMD Opteron platforms use a Non-Uniform Memory Architecture (NUMA) [1, 74]. In this configuration every core has a distinct cache hierarchy and therefore there is no shared cache. Cores on the same chip share a memory bus but each chip has its own unique bus.

Different configurations have different strengths and weakness. For example, by not sharing an `L2` cache a NUMA system's caches are closer to the core so access is faster and the contention between cores is reduced. However, the shared design of the Core™Duo allows threads executing on separate cores to share the same cached data. It also means that if only one core is being used there is more cache available.

Java applications can benefit from multi-core systems in a variety of ways. First, the JVM is multi-threaded and therefore its efficiency can increase on the new systems. On a multi-core system, one core can be dedicated to the task of performing garbage collection while another can execute the application. This enables both processes to

run in parallel. Possibly the most benefit for an individual application comes from the Java APIs [52] that allow applications to be multi-threaded.

The Java standard library contains many components designed to aid developers of multi-threaded applications. One of the fundamental structures used to create multi-threaded applications is the `Thread` type. Every thread of execution in a Java application is associated with a `Thread` object. A programmer can specify the code that is to run in a thread by extending the `Thread` type and overriding the `run` method with the code that is to run in parallel. (Alternatively, the `Runnable` supertype could be used.) The `Thread` class includes many useful methods for thread management. A few of the most relevant are: `start`, which initializes the execution of the thread; `sleep`, which causes a thread to pause for a specified amount of time; `interrupt`, which interrupts the thread; and `join`, which instructs the calling thread to wait until the called thread has finished executing.

Multiple `Thread` objects can often access the same objects, fields, and methods which can lead to thread interference. The simplest method to avoid thread interference is to require threads to obtain a lock before using a critical resource. If a thread cannot immediately acquire a lock, it must wait until the thread that does hold the lock releases it. In Java every object contains a lock, and access to this lock is specified through the keyword `synchronized`. A synchronized method or block of code for a given object can only be executed by one thread at a time.

There are several disadvantages to creating a multi-threaded application in Java as opposed to a compiled language such as C or C++. Since the JVM does act as an intermediary between an application and the system hardware, it is very difficult for applications to gain access to low-level memory controls. This can lead to contention

for memory resources. The JVM itself also represents a challenge, for it is a highly-customizable application and can be a difficult to precisely tune. In general there are three common bottlenecks for multi-threaded Java applications:

1. Excessive allocations
2. Synchronization
3. Untuned Java heap configuration

Excessive allocation can lead to increased garbage collection and demands on the memory management system [67]. A large amount of thread interaction that requires synchronization increases the possibility of threads becoming serialized, thus limiting the scalability of the application. Since it is the JVM that performs the memory management, it is vital to properly tune it to limit contention for memory resources.

In Chapter 4 we explore techniques that reduce the execution costs (in terms of time and memory) of JSA. One obvious approach is to parallelize JSA so that it can take advantage of the modern multi-core architectures described above. Since JSA is written in Java, we use the Java libraries APIs in our implementation.

CHAPTER 3: INCORPORATING DYNAMICALLY GATHERED ENVIRONMENT INFORMATION

Most static analyses have taken one of two approaches for the handling of dynamic class loading in Java: (1) ignore it or (2) treat it in an overly conservative fashion. Ignoring these features produces a result that is unsound and may miss vital program entity interactions. It renders an analysis impractical for use on modern Java applications; for example, there is evidence [89] that significant portions of the program call graph can be omitted by a static analysis which disregards dynamic features such as dynamic class loading. Conversely, the conservative approach assumes that any class can be loaded and instantiated. However, the relevant information can be easily obfuscated by the number of infeasible interactions inferred by this technique. Some analyses [89, 139] require that the user manually specify the classes which could be loaded at dynamic class loading sites. However, this technique can be time consuming and error prone. Livshits et al. [89] presents another approach that utilizes casting information to reduce the number of classes that need to be considered. However, such an approach would fail for the code presented in Figure 2.1 since no cast of the dynamically loaded class is performed.

Since string values specify the classes that are to be loaded at dynamic class loading sites, a *string analysis* has the greatest potential to precisely resolve such instances without requiring input from the user. Several approaches [19, 89, 144] employ various forms of string analysis in an attempt to determine the possible run-time values of these target strings. However, such analyses have two points of possible

failure when attempting to precisely determine the run-time values of a string-typed expression: (1) the value of the expression depends upon values that the analysis does not have knowledge of (e.g., the `args[ ]` array passed to a main method), or (2) the analysis is not powerful enough to model the flow and manipulation of the string values.

In this chapter we present several techniques which increases the ability of string analysis to resolve instances of dynamic class loading.

3.1 Extending JSA

To increase JSA ability to resolve instances of dynamic class loading, Section 3.1.1 proposes an extension to JSA which increases the number of relevant string values available to the analysis, and Section 3.1.2 presents several extensions that improve JSA's overall modeling capabilities.

3.1.1 Semi-Static Analysis

Consider the code example presented in Figure 2.1. If some JSA client specifies the call `forName(str,...)` as a hotspot, then JSA will attempt to resolve the possible run-time values of parameter `str`. However, in this example, JSA will return the value *anystring* for `handlerClassName`. This indicates that under JSA's model, the parameter could be any Unicode string. This occurs, in part, due to the fact that JSA views environment variables as run-time inputs to the program and thus assumes that it has no access to the values stored in them.

Unfortunately, applications that utilize dynamic class loading often rely on string values that are not statically contained in their own code. It is rare, however, that a needed string value flows from direct user input (e.g., from `stdin`). A much more

```
1  public static PrinterJob getPrinterJob() {
2    ....
3    return (PrinterJob)java.security.AccessController.doPrivileged(
4        new java.security.PrivilegedAction() {
5        public Object run() {
6        String nm = System.getProperty("java.awt.printerjob");
7        try {
8          return Class.forName(nm).newInstance();
9        } catch (ClassNotFoundException e) {
10   ....
11 }
```

Figure 3.1: Sample code from library class `java.awt.printer.PrinterJob`.

common case is that such values flow from system environment variables, such as in the example above. Environment variables are *key/value* pairs that are stored in the execution environment and can be accessed by all programs. These variables provide the program with information about the type of environment in which it is operating. It is possible that the user could manipulate these values between consecutive runs of an application. This, however, is not the intent of many of these variables. Consider the Java system property with key `os.name`; clearly, this property is not meant to be modified by the user. Moreover, many of these variables will be consistent across a large number of the host environments that the application will be executed on, and certainly across multiple runs on the same host. For example, the library class shown in Figure 3.1 queries an environment variable to determine which class to load in order to create a job that will facilitate printing for the current installation of an application (line 6). Such a variable will be consistent across many systems. A common default `PrinterJob` used to create Win32 print jobs on Windows operating systems is `sun.awt.windows.WPrinterJob`. Unless the application uses a custom extension of `PrinterJob` or the operating system changes, it is likely that the value of `nm` will be constant for a given system. As long as the value of the environment

variable remains constant, the same class will always be loaded and instantiated at line 8.

We propose an extension to JSA that will allow it to make use of the values stored in environment variables. Our approach requires only alterations to the graph model that JSA builds to represent the flow of string values. We present only the end alterations to the graph; for brevity, the details of the intermediate stages are not discussed.

In our approach we identify a set of Java library methods that serve as entry points for the values of environment variables; the set of methods we consider are shown in Figure 3.2. All of these methods take a `key` string parameter which specifies the environment variable that is to be accessed. In the example presented in Figure 2.1, the constant field `handlerPropName` contains the key `"sun.awt.exception.handler"`. Several of these methods take a second `default` string parameter. These methods return the value stored in `default` if the value of `key` does not specify an environment variable with a set value. Since these parameters are strings, we can add a special `env-hotspot` node to the JSA graph for each encountered call to a method that is an environment variable entry point. By leveraging the existing techniques in JSA, it is often possible to resolve the potential run-time values that both the `key` and `default` parameters can assume.

If JSA is able to resolve the `key` and `default` parameters, our approach performs an *analysis time* look-up of the key/value pair in the environment. This look-up is achieved by executing the method call represented by the env-hotspot node. We term this step "semi-static" since all possible `key` values are statically estimated and then the env-hotspot is executed for each unique value. We views this limited hybrid approach to essentially be a look-up of a "static" entity. The values returned from this look-up are treated as constant strings and are propagated to dynamic class loading

```
java.lang.System.getProperty(String)
java.lang.System.getProperty(String,String)
java.security.Security.getProperty(String)
sun.security.action.GetPropertyAction(String)
sun.security.action.GetPropertyAction(String,String)
java.awt.Toolkit.getProperty(String,String)
```

Figure 3.2: Entry points for environment variables.

sites. In general, the idea of examining the values of environment variables could also be applied to other static analyses; Section 3.3 provides a detailed discussion of the assumptions under which such an approach is applicable.

The outcome of a look-up will result in one of three possible modifications to the graph, as described below.

Single value return. The most straightforward case occurs when both the `key` and `default` (if it exists) parameters for an env-hotspot resolve to a single value. In such situations it is guaranteed that the look-up step will return a single string value: if the key/value pair exists it will return the value, and if the pair does not exist it will return the value specified in `default` or `null`.[3] In such cases our approach replaces the env-hotspot node with an `Init` node. The value associated with this `Init` node is the result of the environment variable look-up. Due to this change of the flow graph, all strings that were dependent upon the original method call are now dependent upon the looked-up value.

Multiple value return. Of course, more than one string value may flow to `key`, to `default`, or to both. In such situations the look-up executes the env-hotspot method for every possible pair of a `key` value and a `default` value. Every value that the look-up step discovers, including all defaults when applicable, is assigned to a

[3]JSA does provide treatment of null string values.

new artificial `Init` node. The env-hotspot node is then replaced by a `Join` node and an edge is added from every new `Init` node to this new `Join`. Since `Join` nodes are analogous to ϕ functions (see Section 2.1.1), this has the effect of unioning all the returned look-up values. Thus, all entities that were originally dependent upon the method invocation are now dependent upon the set of possible values that could be returned at run time.

Variable corruption. It is entirely possible that, for some env-hotspot, JSA will not be able to resolve the `key` parameter, the `default` parameter, or both. If the `key` value is unresolvable there is no precise way to determine the appropriate environment variable to look up. Thus, our approach replaces the env-hotspot node with an `Init` node assigned the *anystring* value. This is also the action taken if the `default` parameter is unresolvable and one of the `key` parameter values is an environment variable which is not set (i.e., does not have a key/value pair in the environment). This has the affect of "corrupting" all other strings that are dependent upon the original method call. It should be noted that it is possible to access environment variable through the use of reflection. Our approach does not precisely model `String` values which flow from reflective calls and therefore such values would also be corrupted.

The result of this extension is a solution that is sound with respect to all possible run-time executions during which the configuration values are the same as the values that were observed during the analysis. This semi-static approach differs from both a completely static analysis (which produces a solution describing all possible run-time executions) and a completely dynamic analysis (which produces a solution describing the specific observed run-time execution). Additional discussion of this approach is presented in Section 3.3.

3.1.2 Modeling Extensions

Even with the addition of the semi-static technique described above, the current publicly available version of JSA would still not be able to determine the possible run-time values of `handlerClassName` at line 14 in Figure 2.1. This is due to JSA's inability to accurately model all possible flows of string values. For example, JSA currently does not track the flow of string values to and from fields. All string values that flow from fields are corrupted (i.e., assigned the *anystring* value).

We propose a more precise handling of fields. Our technique models fields similarly to the manner that JSA handles method invocations in that both are treated in a context-insensitive manner. Currently, we only consider private and package-private fields of type `String` and in some special cases, arrays with a base type of `String`. The approach first identifies all accesses to a given field `x` in the input classes. It then unions all values that flow to instances of `x`. In the final flow graph, this is modeled by adding edges from every `Join` node that represents an assignment to `x` to a newly synthesized `Join` node. An edge from this synthesized node is then added to the node representing the field. Consequently all sites that read the value of `x` will be modeled as potentially receiving all possible values that could be assumed by every instance of `x`. This approach of modeling fields is similar to previous work [20, 134]. Note that in the open-world versions of the analysis described in Section 3.3, *anystring* is propagated to fields that could be modified by code outside of the input classes. Thus, our proposed treatment of fields is sound under the assumption that the input classes comprise a complete package.

During our manual investigation of the Java libraries, described in Section 3.4, we discovered several instances of dynamic class loading that depended on string values defined in static final array fields, as illustrated by the following example:

```
private static final String[ ] codecClassNames =
```

{"com.sun.media.sound.UlawCodec","com.sun.media.sound.AlawCode"}

This structure encapsulates the strings specifying the two possible subclasses of SunCodec that could be loaded at run time by class com.sun.media.sound.SunCodec. For such cases, our approach treats the array as a single String field. Synthesized Init nodes are created for each statically defined array entry. These values are unioned together in the fashion described above. Any access of an element in an array is treated as a read of a field. For the above example, if an access of the form x = codecClassNames[0] were discovered, JSA would assume that the value of x could be "com.sun.media.sound.UlawCodec" or "com.sun.media.sound.AlawCode".

Even after increasing JSA's ability to model fields, it would still not be able to resolve the possible run-time values of handlerClassName from the running example. This is due to the limited number of variables types which are given precise treatment by JSA. In its original form JSA only models variables of type String, StringBuffer, StringBuilder and arrays with a base type of String. However, in the code displayed in Figure 2.1, the look-up of the variable sun.awt.exception.handler is accomplished by creating an instance of sun.security.action.GetPropertyAction (line 10). This convenience class implements java.security.PrivilegedAction. Instances of PrivilegedAction are typically passed to invocations of priviledged granting AccessController.doPrivileged method. This results in the execution of PrivilegedAction.run with privileges enabled. In the case of GetPropertyAction, the run method simply wraps an invocation of method System.getProperty. The problem is that the return type of PrivilegedAction.run is java.lang.Object. Even though String is a subclass of Object, JSA does not model objects with a compile-time type of Object which are of type String.

It is a common practice to use an instance of PrivilegedAction to wrap accesses to environment variables. Thus, it is paramount for the success of our semi-static

approach that JSA be able to properly model such occurrences. We propose a extending JSA so that it can conservatively determine variables with compile-time types of `Object` that are actually of type `String`. To achieve this, we augment JSA to also consider variables of type `Object`. Suppose that the only actions performed on such a variable in the input classes are: (1) assignment from a variable with a compile-time type of `Object` that is actually of type `String`, (2) cast to a `String` variable, and (3) assignment from a `String` variable or a string literal. If this is the case, we direct JSA to treat the variable as a `String`. If any action outside of those specified above occurs, such as the `Object` being assigned a dynamic type other than `String`, the variable is conservatively corrupted and, transitively, all string values dependent upon it. This approach is quite conservative and more powerful type inferencing techniques could reveal more instances of `Object` variables which are really of type `String`. Still, our experimental results show that this approach is sufficient to model the flow of most string values which are utilized at dynamic class loading sites in the Java 1.4 standard libraries.

3.2 Resolving Reflective Instantiation

Being able to determine the classes that can be instantiated by dynamic class loading is a valuable ability for many static analyses. Consider the example shown in Figure 2.1. By resolving the instance of dynamic class loading at line 14 the resolved classes will be identified as being part of the application and their static initializers will be identified as being reachable. However, dynamic class loading is just one of Java's dynamic features. Consider the method call `c.newInstance()` at line 15 of Figure 2.1, which is an example of reflective instantiation. As stated earlier, an invocation of `newInstance` has the same effect as calling the no-arguments constructor of the class being represented by `c`.

To illustrate the importance of being able to resolve instances of reflective instantiation, consider the well-known Rapid Type Analysis (RTA) [7] call graph construction algorithm. RTA produces a graph in which the nodes represent methods and edges represent possible calling relationships between methods. The analysis starts at the main method of the program. As calls to methods are discovered, the appropriate nodes and edges are added to the graph and the bodies of called methods are analyzed. RTA maintains a set *Instantiated* of classes that could be instantiated in methods reachable from the main method. Virtual call sites are resolved based on these classes. The set *Instantiated* is updated whenever RTA encounters `new X` expressions. If an update of *Instantiated* implies additional target methods at already-processed call sites, the call graph is updated to reflect the newly discovered relationship.

An implementation of RTA which ignores dynamic instantiations of classes may create a call graph that is unsound for applications utilizing reflection. Since dynamically instantiated classes would not be added to *Instantiated* unless they are instantiated by conventional means elsewhere in the application, RTA will not consider them when resolving virtual call sites. In this case, the resulting call graph would be missing valid call edges. Conversely, if RTA treated even one instance of reflective instantiation conservatively by assuming that all classes could be instantiated at such a site, *Instantiated* would contain all possible classes. This would result in the consideration of all classes when estimating potential receivers of dynamically-dispatched messages. The resulting call graph would be identical to the graph generated by the imprecise CHA [27]. Thus RTA's efforts would be rendered superfluous. Similarly to RTA, many other call graph construction algorithms face problems due to reflective instantiations.

```
1  public SunToolkit() {
2      ....
3      String tgName = System.getProperty("awt.threadgroup", "");
4    ....
5      Constructor ctor = Class.forName(tgName).
6                  getConstructor(new Class[] {String.class});
7      threadGroup = (ThreadGroup)ctor.
8                     newInstance(new Object[] {"AWT-ThreadGroup"});
9      ....
10 }
```

Figure 3.3: Sample code from library class `sun.awt.SunToolkit`

3.2.1 Using Resolved Dynamic Class Loading Sites

The key to determining which classes could be instantiated by an invocation of `Class.newInstance`, is identifying the classes represented by the `Class` object. Thus, a natural extension of the work from Section 3.1 is to track the `Class` objects from resolved dynamic class loading sites to invocations of `Class.newInstance`. If for a given call `x.newInstance()`, where `x` is of type `Class`, all possible values of `x` flow from resolved instances of dynamic class loading sites, then transitively the call to `newInstance` is resolved. All possible entities represented by `x` as determined by string analysis are also the possible classes instantiated by the call `x.newInstance()`.

As shown in Figure 3.3, `Class.newInstance` is not the only reflective method that can dynamically instantiate a class. In this example, the class specified by the `awt.threadgroup` system property is dynamically loaded. The resulting `Class` object is queried to retrieve the class's constructor. The outcome is a `Constructor` object, which represents the constructor of the class that takes a single `String` parameter. The call to `newInstance` at line 8 has the same effect as invoking the constructor, passing it the string `"AWT-ThreadGroup"`.

Determining the classes that could potentially be instantiated by invocations of `Constructor.newInstance` is similar to resolving calls to `Class.newInstance`. For calls of the form `c.newInstance(Object[])`, where the type of `c` is `Constructor`, it is necessary to determine the `Class` objects to which `c` could refer. If all such `Class` objects flow from resolved dynamic class loading sites, they are the ones that could be instantiated by `c.newInstance(Object[])`. It should be noted that we only determine which classes could be instantiated, and we are not concerned with the specific constructor that is being invoked.

This approach for resolving reflective instantiation is another example of the benefits of relaxing the restrictive assumptions made by purely static analyses. Since the foundation for this approach is our semi-static resolution of dynamic class loading, the set of classes returned for resolved reflective instantiation sites will be tailored to the specific system configuration being analyzed. Section 3.4 presents a study of resolving reflective instantiation in the Java 1.4 standard libraries which demonstrates the precision gained by exploiting information from environment variables and by our JSA modeling extensions.

3.3 Examining Assumptions

Assumptions about environment variables. A key assumption of our approach for resolving both dynamic class loading sites and reflective instantiation sites is that values from environment variables will remain the same between analysis time and run time. Our techniques treat these values as constants. This is a relaxation of assumptions made by purely static analyses which view such inputs as purely dynamic. This assumption reduces the number of program states that will be examined by the analysis. States are limited to all possible program executions where the values of environment variables which flow to dynamic class loading sites are the same as

those observed at analysis time. This relaxation increases the precision of the analysis at the cost of soundness. It does introduce two new ways in which the soundness of analysis results could be compromised. First, the execution environment could be modified between analysis time and run time, changing an environment variable whose value is used at a dynamic class loading site. Second, an execution of the application could modify an environment variable at run time and that value could later be used to dynamically load a class.

It is likely that other static analyses can also benefit from such relaxing assumptions. The use of environment variables is not restricted to Java, nor is the application of their values restricted to dynamic class loading. Furthermore, the concept of semi-static input values is not limited to environment variables. These values can originate from any source which remains predominately constant between and during executions of an application. Such sources could include configuration files, file directory structures, and certain database data. Similar to our treatment of environment variables, it may be possible to identify the values being accessed from these sources and treat them as known constants at analysis time.

In Chapter 5 we demonstrate how the precision of a CHA call graph construction algorithm can be increased by making use of environment information. Static analyses which perform program specialization [44] are other obvious examples of analyses that could make use of semi-static inputs. Program specialization is a technique of program optimization where, given a program p and static data d, a transformation is performed to create p_d. This new program, p_d, is formed by precomputing the parts of p that depend only on d, typically creating a more efficient version of the program with respect to d. This is similar to our approach in that p_d has a limited the number of possible execution states when compared to p. As a matter of fact, if a program p were specialized with respect to the environment variable values *env*

and these values were only used at dynamic class loading sites, then our analysis of p would consider the exact state-space of p_{env}. Of course, values from environment variables have many more uses than just designating the classes that are to be loaded at run time. By specializing with respect to the semi-static values, it may be possible to create efficient version of programs which are tailored to a user's current system configuration. Other static analyses such as those that address code testing and verification, program understanding, and code refactoring may also benefit from the use of semi-static inputs.

Open-world assumptions. JSA is designed to analyze partial programs: its input is a set of classes that does not necessarily form a complete program. Our extensions also operate under an open-world assumption. Specifically, it is assumed that certain string values can be affected by unknown client code or by unresolved instances of reflection. For soundness, JSA assumes that the values of these strings could be any Unicode string. In the design of our approach, we take a more nuanced view and consider the following open-world assumptions:

1. *Omnipotent client interactions* (OCI): This is a fully open-world assumption which assumes that through the use of reflection, all methods and fields can be manipulated by client code, potentially breaking encapsulation for private and package-private entities.
2. *Standard client interactions* (SCI): Under this assumption, client code and the use of reflection will respect standard encapsulation practices and will only affect public and protected entities.
3. *Limited client interactions* (LCI): This is an optimistic approach which assumes that client code will only affect the values of target strings through invocations of public methods and manipulations of public fields. Further, it assumes that

```
java.lang.Class.forName(String)
java.lang.Class.forName(String,boolean,...)
java.lang.ClassLoader.loadClass(String)
java.lang.ClassLoader.loadClass(String,boolean)
java.lang.ClassLoader.defineClass(String,...)
java.lang.ClassLoader.defineClass(String,...,ProtectionDomain)
java.lang.ClassLoader.findClass(String)
java.lang.ClassLoader.findSystemClass(String)
java.lang.ClassLoader.findLoadedClass(String)
java.security.SecureClassLoader.defineClass(String)
sun.reflect.misc.ReflectUtil.forName(String)
sun.reflect.misc.MethodUtil.findClass(String)
sun.reflect.misc.MethodUtil.loadClass(String,Boolean)
```

Figure 3.4: Library methods used for dynamic class loading.

none of the target string values could be affected by the use of reflection either in client code or library code.

The assumptions form a hierarchy in which each new assumption embodies more inherent risk than the previous one. The results of the investigation presented in Section 3.4 indicate that willingness to assume greater risk can produce significantly more precise results.

Assumptions about class loaders. We identified 13 Java 1.4 library methods that are used to dynamically load classes into the JVM; they are shown in Figure 3.4. Several of these methods allow users to specify which class loader will be used to load the class into the JVM. The introduction of multiple class loaders significantly complicates not only the precise resolution of dynamic class loading, but also the design of most static analysis algorithms. One complication is the introduction of *namespaces*: classes loaded into the JVM are identified by their fully qualified name *and* by the defining class loader. This means there can be multiple classes with identical fully qualified names present in the JVM. An even greater concern is the

possibility of user-defined class loaders. A user-defined loader can load a completely different class than the one specified by the string parameter of a dynamic class loading method. They can also alter the bytecode of the classes they load. For these reasons, we assume that all classes that could be dynamically loaded can be uniquely identified by their fully qualified names—that is, each such name appears in only one namespace. Further, we assume that all class loaders designated for use at dynamic class loading sites will load the classes specified by the provided string parameter and will not alter the bytecode in a manner that could affect the flow of string values to dynamic class loading sites.

3.4 Experimental Evaluation

We implemented the proposed analysis and evaluated its ability to resolve dynamic class loading and reflective instantiations in the 10,238 classes from the Java 1.4 standard libraries. The methods shown in Figure 3.4 were used as the hotspots input to JSA. A site was considered resolved if JSA returned a finite number of possible string values for the `String` parameter representing the fully-qualified name of the class to be loaded.

3.4.1 Manual Investigation

To establish a "perfect baseline" to which we could compare our results and investigate the affects of the open-world assumptions, we performed a manual investigation of the entire set of library classes. During the investigation we examined all potential hotspots as defined above. Not considered were occurrences where the target

string was a constant string literal. For example, a call to `forName` with the literal `"com.sun.media.sound.JavaSoundAudioClip"` was not included in the set of interesting hotspots, since it is trivial to resolve statically.[4]

The study was conducted under the three open-world assumptions described in Section 3.3. Under such assumptions, it is impossible to determine the run-time values of certain method parameters and fields due to potential future interactions with unknown client code. No analysis technique can resolve such client-dependent sites in the absence of client code. Thus, we focused our investigation on the *client-independent* sites for which the run-time behavior could be completely determined by examining only the library code. Each such site was placed into one of three categories:

1. Static dependent (SD)
2. Environment variable dependent (EVD)
3. Dynamic dependent (DD)

Dynamic class loading sites that were categorized as static dependent (SD) had a target string whose values were statically determinable (i.e., depended only on compile-time constants). The values of many target strings flow from methods which access environment variables; sites that were dependent on such strings were categorized as environment variable dependent (EVD). The remaining sites, which were labeled DD, depended on string values that were not statically contained in the library code or in environment variables but yet were not directly derived from client code. For example, a site whose target string's value flowed from a file read was classified as DD.

[4]This example is from class `sun.applet.AppletAudioClip`, where the call is used to determine if the system has the Java Sound extension installed. If the call fails, a default component is used. In general, checking for the existence of extensions is a common use of dynamic class loading in the Java libraries.

Assumption	SD	EVD	DD	Total
OCI	18	32	3	53
SCI	33	35	12	80
LCI	40	35	15	90

Table 3.1: Manual investigation of the Java 1.4 standard libraries: categorized counts of invocations of dynamic class loading methods under various open-world assumptions.

It is possible for the value of a target string to be dependent on several categories of sources. The categorization of such instances adhered to an intuitive hierarchy imposed over the categories. For example, if a string was dependent on both an environment variable accessed through a call to `System.getProperty` and on input from a configuration file, it would be classified as dynamic dependent (DD). This hierarchy was also applied to the `key` and `default` parameters of methods that are entry points for environment variable values. If such a method's `key` value flowed from a file read and the resulting environment value flowed to a dynamic class loading site, that site would be classified as DD.

It is important to emphasize that this classification was performed using human intelligence. The results of the classification represent the best possible solution that *any* purely-static or environment-variable-aware analysis could hope to achieve. By using these results as a baseline, we can judge how well our analysis performs in absolute terms, instead of simply measuring the improvement over the original JSA.

Table 3.1 shows the results of this manual investigation. Under OCI, the fully open-world assumption, 53 dynamic class loading sites were fully contained within the library code. SCI assumes that client code can only affect string values through the manipulations of public and protected entities. Under this assumption 80 dynamic class loading sites could not be directly manipulated by outside code. In other

words, by assuming that reflection used in client code will respect encapsulation, SCI considers 27 more sites to be fully contained in the library code (i.e., sites that cannot be affect by client code) than OCI. Under the most liberal LCI assumption, which assumes that clients can only affect the values of public entities, 90 dynamic class loading sites present in the library code could not be affected by clients.

The results of this investigation indicate several key characteristics of dynamic class loading in the Java 1.4 libraries. First, assumptions made about client interactions could significantly affect the precision of an analysis attempting to resolve these dynamic features. Second, dynamic class loading that derives the value of the target string from environment variables is usually *closed*. By this, we mean that all entities other than the actual value of the environment variable, including the `key` and `default` parameters, can be determined completely statically and in no way can be affected directly by client code. This is indicated by the fact that between the most restrictive OCI to the most relaxed LCI, only 9% of the instances originally classified as EVD become client-dependent, as opposed to 55% of those classified SD and 80% of those classified DD. Many of the DD sites relied on string values which flowed through parameters and fields, thus OCI assumed they could be affected by client code. The final characteristic is that a large number of client-independent sites are indeed dependent on environment variables—those classified as EVD. Under the most natural SCI assumption, over 40% of dynamic class loading sites were classified as EVD. Such sites cannot be resolved by *any* purely static analysis. To our knowledge, our approach is currently the only analysis that leverages these characteristics.

3.4.2 Resolution of Dynamic Class Loading

Table 3.2 shows the results of resolving dynamic class loading sites in the Java 1.4 standard libraries using four versions of JSA. These four versions operate under

Analysis version	JSA1	JSA2	JSA3	JSA4
SD	22	22	22	27
EVD	0	16	28	32
Resolved	22	38	50	59
% of total	32%	56%	74%	87%

Table 3.2: Precision of string analyses for the Java 1.4 standard libraries: number of SD and EVD dynamic class loading sites resolved by JSA. The percentages are with respect to 68, the total number of SD and EVD sites from the manual investigation.

the SCI assumption, thus a total of 80 dynamic class loading sites were considered (Table 3.1). Of these, the approaches we investigated could only resolve sites that had target string values which could be statically or semi-statically determined—i.e., those which were manually classified as SD or EVD, of which there were 68. Row *SD* shows how many of the manually-classified SD sites were identified by the analysis as being SD. Similarly, row *EVD* shows the number of manually-classified EVD sites that were reported by the analysis as being EVD. Row *Resolved* shows the total number of sites that were resolved by the analysis, either as SD or as EVD. There were 68 manually-classified SD/EVD sites; the last row in the table shows the percentage of these 68 sites that the corresponding version of JSA was able to resolve (i.e., *Resolved*/68).

All four versions of JSA used in this experiment operate under the SCI open-world assumption. To this end, all versions corrupt the values of parameters to public and protected methods since it is assumed values from client code and reflective calls could flow to these entities. The values returned by public methods (these can be overwritten by clients), methods not contained within the code body being analyzed, reflective methods, and native methods are also corrupted. In other words, JSA treats these return values conservatively by assuming they could be any Unicode

string. Since the only values not corrupted by JSA must be fully contained in a single package, we executed the versions of JSA on the individual packages that comprise the Java 1.4 libraries.

The first version was JSA in its original form with minor bug fixes, and some alterations to accommodate the open-world assumption. The corresponding results are shown in column JSA1. Since this version did not use our semi-static extension, it was able to resolve only sites whose string values were completely statically determinable. Thus, this state-of-the-art approach could resolve only 22 of the 80 total SD/EVD/DD sites, which is 32% of the 68 SD/EVD sites. Column JSA2 shows the gains from enhancing JSA with the semi-static technique from Section 3.1.1. This addition enables JSA2 to resolve 16 more sites than JSA in its original form (JSA1). The version from column JSA3 added the type extension outlined in Section 3.1.2. Although this version did not increase the number of resolved SD sites, it was able to resolve an additional 35% of all EVD sites, by allowing more precise tracking of string values that flow from environment variables (e.g., as illustrated by the call to `doPrivileged` in Figure 2.1). The final version, shown in column JSA4, added the more precise treatment of fields described in Section 3.1.2. As a result, the analysis was able to resolve an additional 15% of all SD sites and 11% of all EVD sites.

Overall, the version that contained all our extensions JSA4 resolved 74% of all client-independent sites (SD/EVD/DD) and 87% of all sites classified SD/EVD; for the original version of JSA, the corresponding percentages were 28% and 32%. JSA4 was unable to resolve nine instances that our manual investigation classified as SD or EVD. This was due to some deficiencies in JSA's ability to model the flow of string values. Several of these instances relied on complex data structures, such as `HashMap`, which JSA is currently unequipped to model. The remaining values passed through

Analysis version	JSA1	JSA2	JSA3	JSA4
sec per 1000 Jimple	1.69	1.72	1.74	1.81
MB	20.4	20.4	20.5	21.7

Table 3.3: Analysis cost: running time (seconds per thousand Jimple statements) and memory usage (MB).

operations that were beyond the modeling abilities of JSA, such as being parsed by a `StringTokenizer`.

Analysis cost. The experiments were executed on a PC with an Intel Core Duo T2400 (1.83GHz) processor and 2 Gb of memory, running a Windows XP OS. The JVM heap size (JVM option Xmx) was set to 1.5 Gb. As stated earlier, each version of JSA was executed once for each of the 358 packages comprising the Java 1.4 library. Table 3.3 shows the time and memory used by each version of JSA to conduct the complete experiment. All measurements are the average of three complete runs of the experiment. The time measurements average number of seconds each version took to analyze a thousand Jimple statements (there were over 1.3 million Jimple statements analyzed in total). The time measurements do not include the cost of building the Jimple intermediate representation. The row labeled *MB* displays the maximum memory used by the corresponding version of JSA, averaged over all the packages.

It is important to note that for the versions of JSA incorporating the semi-static extension (JSA2, JSA3, and JSA4), a pre-processing phase was not included in the timing. This pre-processing step resolves the string values of `key` and `default` parameters of methods that are entry points for environment variables. This step can be incorporated into JSA, but doing so efficiently would require a significant engineering effort that is beyond the scope of this work. In the worse case, the pre-processing

Analysis version	JSA1	JSA2	JSA3	JSA4
Resolved	6	19	28	37

Table 3.4: Precision of string analyses for the Java 1.4 standard libraries: number of reflective instantiation sites resolved by JSA.

phase is no more expensive than JSA1 and is easily justified by the increased precision gained through the use of semi-static values. Once the `key/default` values have been determined, our extensions only slightly increase the cost of JSA. Each extension increases the running time of the analysis by a few tenths of a second per 1000 Jimple statements. The biggest cost in terms of both time and memory is the addition of the field extension. This is due to the creation and storage of additional data structures that our implementations uses to track the flow of fields. Overall, the cost of using JSA to resolve instances of dynamic class loading appears to be reasonable when applied at the package scope of the Java libraries.

3.4.3 Resolution of Reflective Instantiations

Section 3.2 presented an approach which tracks `Class` objects obtained from dynamic class loading sites to resolve calls to reflective instantiation methods. We implemented this approach in the four versions of JSA using an intraprocedural flow analysis. This analysis tracks the flow of `Class` objects to calls to `Class.newInstance` and transitively to invocations of `Constructor.newInstance` (for the remainder of this dissertation, references to `newInstance` indicate both the `Class` and the `Constructor` methods). As described in Section 3.2.1, a call `x.newInstance()` is resolved if all possible values for `x` flow from resolved dynamic class loading sites. The extended versions of JSA were applied to the Java 1.4 standard libraries. The results of this evaluation

are shown in Table 3.4. Row *Resolved* shows the number of calls to `newInstance` resolved by each version of JSA.

The ability of each successive version of JSA to resolve more calls to reflective instantiation methods than its predecessor is due to its increased ability to resolve invocations of dynamic class loading. This indicates that at least some of the `newInstance` sites are indirectly dependent on environment variables. The dependency is illustrated in Table 3.5, which shows the number and type of dynamic class loading sites which were used to resolve reflective instantiation. Comparing row *Total* in Table 3.5 with row *Resolved* in Table 3.4, it should be noted that there does not exist a one-to-one relationship between the number of resolved `newInstance` sites and the number of dynamic class loading sites that are needed to resolve them. This is due to the flow of the results of several dynamic class loading sites to the same invocation of `newInstance`.

The original version of JSA resolves dynamic class loading using only purely static string values. The `Class` objects from 8 loading sites flow to 6 distinct calls to `newInstance`. It should be noted that this version was able to resolve 22 instances of dynamic class loading (see Table 3.2). The `Class` objects from the remaining 14 resolved sites did not flow to invocations of `newInstance`. A manual investigation revealed that these objects were used for purposes other than instantiation. Examples of such uses include comparisons—e.g., `ClassX.equals(ClassZ)`—and gaining access to reflective objects such as `Method` and `Field`—e.g., `Class.getMethod(String, Object[])`.

The semi-static extension in JSA2 allow it to resolve additional dynamic class loading sites (Table 3.2). The `Class` objects from 14 of them enabled the resolution of 13 additional invocations of `newInstance`. Thus, the use of environment variable values enabled JSA2 to resolve a total of 19 reflective instantiation sites. Over 57%

Version	JSA1	JSA2	JSA3	JSA4
SD	8	8	8	13
EVD	0	14	25	29
Total	8	22	33	42

Table 3.5: Categorized counts of resolved dynamic class loading sites whose `Class` objects flow to `newInstance`.

of the reflective instantiation sites resolved by JSA2 are indirectly dependent on environment variables.

JSA3 was able to resolve 28 `newInstance` sites (see Table 3.4). The increase is due to the resolution of 11 new EVD dynamic class loading sites which flow to 9 additional calls to reflective instantiation. Finally, JSA4 was able to resolve a total of 59 dynamic class loading sites (Table 3.2). Of these sites, objects from 13 classified as SD and 29 classified as EVD flow to 37 invocations of `newInstance`. Of these 37 reflective instantiation sites, 31 rely on information gained from our modifications to JSA and 26 of them could not have been resolved by a purely static analysis.

3.4.4 Summary of Experiments

A manual investigation of the Java 1.4 libraries determined that over 40% of the client-independent instances of dynamic class loading depend on values stored in environment variables. These instances are impossible to resolve by any purely static analysis. The experiment shows that augmenting the current publicly available implementation of JSA with the extensions proposed in this chapter allowed it to resolve an additional 46% of all client-independent sites. In addition, this augmentation successfully identifies 87% of all sites manually-classified as dependent upon only static

or semi-static (those flowing from environment variables) string values—i.e., SD and EVD sites.

By further extending JSA with a lightweight flow analysis, it is possible to determine the set of classes that can be instantiated at many calls to `newInstance`. One potential use of this information is to make popular call graph algorithms such as RTA [7], XTA, MTA, and FTA [140] more precise when analyzing applications that make use of reflection. Our evaluation showed that the augmented version of JSA was able to resolve 37 `newInstance` calls where as the original version of JSA resolved 6. Moreover, 70% of the total number of resolved `newInstance` sites relied on values that flowed from dynamic class loading sites which were dependent on environment variables. These sites cannot be resolved in a purely static manner.

3.5 Conclusions and Future Work

This chapter presents a semi-static approach that utilizes configuration information to aid in the resolution of dynamic class loading in Java applications. This technique produces results that are tailored to the current execution environment and the configuration of the analyzed application, by relaxing the restrictive and sometimes impractical constraints assumed by most purely static analyses. We also present extensions of string analysis that allow better tracking of class names and environment variable names. In an experimental study conducted on the Java 1.4 standard libraries, our approach was able to resolve 46% more dynamic class loading sites than the state-of-the-art string analysis. We also demonstrate how the information gained from resolved dynamic class loading sites can be used to determine the classes that can potentially be instantiated through the use of reflection. The use of configuration information, and our modeling extensions to JSA, increases the number of resolvable reflective instantiation sites from 6 to 37.

CHAPTER 4: IMPROVING THE SCALABILITY OF STRING ANALYSIS

The precision of many static analyses could be increased if they were able to precisely answer questions such as 'Which classes can be dynamically loaded?", or "Which methods are being invoked through the use of reflection?". As shown in the previous chapter, a string analysis can be employed to aid in deriving the answers to these difficult questions. However, many static analyses either incorporate a very simple string analysis or none at all. For example, the CHA call graph construction algorithm in the popular Soot analysis framework [144] only validates hard-coded string values passed to calls to `Class.forName(string)`. As our experiments in Section 3.4 demonstrated, many instances of dynamic class loading cannot be resolved by such a limited technique. Thus, without direct user input, the call graph generated by Soot will either be unsound or very imprecise at these instances. This is but one example; other analyses that build call graphs for Java applications (either as an end-result or as a component in a larger framework) do not consider string values at all (e.g., [71, 123, 126, 139].)

There are many possible reasons why a static analysis might not integrate a string analysis; one being the difficultly associated with developing a precise string analysis. However, JSA is a very comprehensive and well designed library, so why are not more developers of static analyses choosing to use it? One explanation can be found in the high cost associated with running JSA on certain applications. To put this cost into perspective, in Section 4.3 we present 25 benchmark applications that were used in

our experiments. We executed JSA on each application with the JVM heap size (JVM option Xmx) set to 6 Gb. Of the 25 benchmark application, JSA could not complete an analysis of 3 due to the JVM exhausting its allocated heap memory. This result becomes more alarming when considering that in this experiment we did not include any of the Java standard library classes in the input to JSA. A whole-program analysis would have to consider these library components, possibly increasing the input to JSA by thousands of classes. This level of overhead is simply not practical for many static analyses.

In this chapter we describe techniques that increase the scalability of JSA. We begin in Section 4.1 by presenting several algorithmic transformations to JSA. These transformations parallelize portions of JSA allowing it to leverage modern multi-core architectures. Several of these transformations are also designed to reduce the memory footprint of the analysis. In Section 4.2 we introduce several new simplifications to the intermediate graph generated by JSA's front-end (see Section 2.2 for a description of the design of JSA). Through these simplifications we can greatly reduce the overhead associated with the JSA's back-end for some applications. We implemented our proposed techniques and evaluated their effectiveness on the 25 benchmark applications mentioned above. The results of this empirical study are shown in Section 4.3. Overall, our most advanced version of JSA's front-end realizes an average speedup of 1.54 times relative to the running time of the original version while reducing the average memory footprint by 43%. Incorporating our new simplifications allowed for an overall (front-end + back-end) speedup of over 180 times for two benchmark applications. Moreover, these simplifications allowed JSA to easily complete analysis of the three benchmark applications that previously exhausted the allotted heap memory.

4.1 Design of Parallel String Analysis Algorithm

The trend in modern computer systems is moving away from single processing units in favor of multi-core architectures (see Section 2.3). In such systems, speedup is achieved by dividing the workload across the multiple processors which run in parallel. This architecture allows multiple sequential applications to execute concurrently on the same system without competing for processor resources. Multiple cores also allows multi-threaded applications to realize a significant speedup; just as multiple applications can execute on separate cores in parallel, so can the threads of a single application.

The publicly available version of JSA is a sequential program. Thus, JSA realizes little benefit from the architecture of a multi-core system. Recall that JSA is composed of two components (see Section 2.2): a front-end that takes the input classes and generates a graph representing the flow of string values through the input; and a back-end which takes the flow graph and generates finite state automata for requested hotspots. In this section we propose several transformations to the design of JSA's front-end. These transformations parallelize significant portions of the work performed by this stage of JSA. We focus on the front-end for two reasons: (1) though JSA's back-end is theoretically more expensive, our experimental study of 25 benchmarks show that in practice the front-end requires the majority of the execution time for a large number of our benchmarks; (2) though there may be a way to parallelize the back-end, such a design is far from intuitive. We feel a better use of our effort is to focus on reducing the size of the input graph, thus reducing the amount of work being required of the back-end.

We first introduce a very simple and intuitive parallelization of the front-end. This design introduces what we consider to be the basic unit of work for JSA and

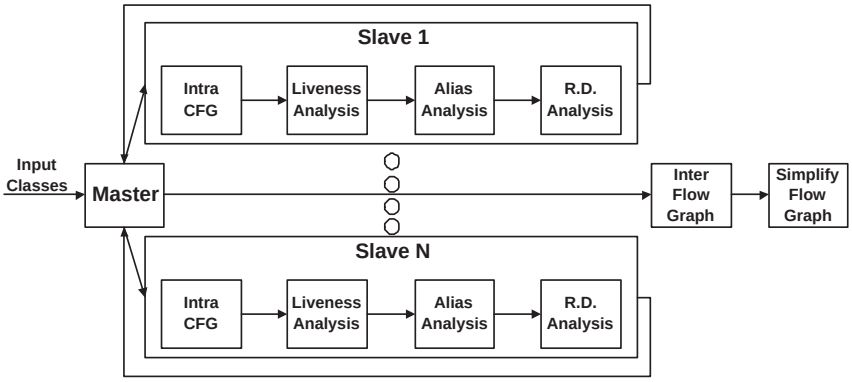

Figure 4.1: Parallel JSA design: intuitive.

the *master/slave* structure being used. We then present a significant refinement to this basic design. As stated in Chapter 2, excessive current memory allocations can tax the memory management system and become a bottleneck for multi-threaded Java applications. This refinement is designed to reduce the memory footprint of the front-end and potentially reduce the amount of concurrent memory allocation. Further, this reduced memory design provides an opportunity to parallelize portions of the graph simplifications performed by JSA.

4.1.1 Intuitive Design

From the description in Section 2.2 it becomes natural to think of a method as a unit of work for JSA. Each stage of JSA's front-end fully completes its analysis of a single method before progressing onto the next one. Prior to the building of the interprocedural flow graph, only a limited amount of information flows between the analysis of one method to another. The only information that flows between the analysis of two methods in these stages occurs in the *Intra CFG* phase (see Section 2.2) when a method invocation is discovered. Recall that a Class Hierarchy

```
input   AppClasses : set of Jimple classes
output  ResultSet : set of results generated by the slave threads
procedure  Master(AppClasses)
 1: ResultSet := workloads := ThreadSet := ∅
 2: hierarchy := NewHierarchy(AppClasses)
 3: workloads := LoadBalance(AppClasses)
 4: while workloads ≠ ∅ do
 5:     n := workloads.RemoveAny()
 6:     slave := newThread(n, hierarchy)
 7:     slave.start()
 8:     ThreadSet := ThreadSet ∪ {slave}
 9: end while
10: while ThreadSet ≠ ∅ do
11:     slave := threadSet.RemoveAny()
12:     slave.join()
13:     ResultSet := ResultSet ∪ slave.getResults()
14: end while
15: return ResultSet
```

Figure 4.2: Algorithm for the master stage.

Analysis of the input classes is performed to determine the targets of the virtual calls. For each target discovered, a new `Call` statement is added to the CFG. A pointer to this newly created `Call` object is then recorded in a data structure associated with the target method. This information is used to identify interprocedural edges during the final construction of the flow graph. This data transfer between methods can be eliminated by annotating `Call` statements with the signature of their target method. During the *Interprocedural Flow Graph* phase, interprocedural edges are discovered by looking up the `Method` object that corresponds to a `Call`'s annotated target in a table that maps a method signatures to a `Method`.

With the change to the identification of interprocedural edges and the stipulation that a method represents a unit of work, a natural algorithmic transformation can be performed to parallelize the front-end of JSA. This intuitive design is shown in Figure 4.1. The design uses the classic *Master-Slave* design pattern [11]. In this pattern, the master thread of execution is responsible for dividing the work into

approximately equal sub-tasks. These subtasks are then delegated to identical slave components which run in separate threads of execution. The master component collects the partial results created by the slaves and combines them in a meaningful manner.

Figure 4.2 presents the pseudocode for the *Master* component of our intuitive parallel design for JSA. It first gathers the class hierarchy information for the entire set of input classes (line 2). It then calls the method *LoadBalance* which takes the entire set of input classes. It is the responsibility of *LoadBalance* to separate the methods of the input classes into roughly equivalent workloads. Our load balancing heuristic weights each method by the number of Jimple statements it contains. It then creates a number of workloads and counts of the number of Jimple statements assigned to each. At first all the workloads contain 0 Jimple statements, so methods are arbitrarily assigned. After all workloads have a method assigned to them, methods are assigned to the workload that contains the least number of Jimple statements.

For each workload a unique *slave* thread is initialized (line 6) and its *start* method is invoked. The ideal number of workloads is equal to the number of hardware threads available on the execution system so that each slave can execute on a unique core. Once all the slaves have started, the master monitors their progress. It randomly selects a slave and invokes *slave.join*() (line 12). This causes the master thread to wait until the selected slave has completed its workload. Once the slave has completed the master gathers its results and selects another slave on which it performs the same actions. Once all the slaves have completed their workloads, the combined result sets are then passed to the *Inter Flow Graph* stage.

The slaves are instances of a class which extend `java.lang.Thread`. Recall that it is by overriding the `run` method of a `Thread` object that a user specifies the code that is to be executed in the new thread. Figure 4.3 presents the pseudocode for the

input *WorkLoad* : set of Jimple method bodies
input *Hierarchy* : CHA hierarchy information for all input classes
procedure run(*WorkLoad*, *Hierarchy*)
1: *CFGSet* := *IntraproceduralCFGs*(*WorkLoad*, *Hierarchy*)
2: *LiveSet* := *LivenessAnalysis*(*WorkLoad*, *CFGSet*)
3: *AliasSet* := *AliasAnalysis*(*WorkLoad*, *CFGSet*, *LiveSet*)
4: *RDSet* := *ReachingDefAnalysis*(*WorkLoad*, *CFGSet*, *LiveSet*, *AliasSet*)

Figure 4.3: Intuitive design: algorithm for slave object's *run* method.

run method of our slave class. The methods invoked by *run* correlate to the flow analysis stages shown in Figure 4.1. The slave first creates an intraprocedural CFG for all the methods in its workload. These graphs are then used in the execution of the liveness analysis. The CFGs and the results of the liveness analysis are then used to perform the alias analysis. Finally, the results of all the prior analyses are used in a reaching definitions analyses. Notice that the slaves execute the flow analyses in a manner that is very similar to the sequential version of JSA. The results generated by these stages are stored in sets until the thread has completed. These sets are then gathered by the master thread.

This design has several advantages. First, it is a fairly simple modification from the sequential version of JSA. Second, there is very limited communication between threads. Each worker thread only communicates with the master thread. This communication occurs during worker's initialization (to receive its workload) and at its conclusion (to report its results). The communication at the end of a thread's life represents the only communication that requires synchronization. This very limited communication model greatly reduces the possibility of a bottleneck due to synchronization.

One potential weakness of this design is the memory resources it requires. This design actually increases the overall memory footprint required by JSA. This increase

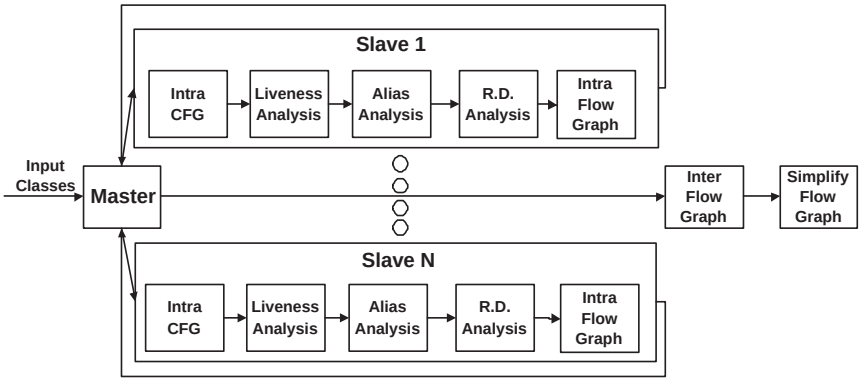

Figure 4.4: Parallel JSA design: reducing memory footprint.

is due to the thread structures, and the redundant data structures maintained by each slave (e.g., each thread maintains a list of all method signatures in the input classes.)

4.1.2 Reducing the Memory Footprint

In the simple parallel version of JSA presented above, the slave components are responsible for building the CFGs and executing the flow analyses. Each of these stage fully analyzes all of the methods in the slave's workload. Once a stage has completed it passes its entire result set to the next stage. The master thread collects the complete result sets from each of the slaves and passes the combined information to the *Inter Flow Graph* stage. Assuming that local memory used by the individual slave stages and the master stage is less than the combined results of slave stages, the *Inter Flow Graph* stage will contain the peak memory use for JSA's front-end.

One possible transformation could leverage the fact that to process a single method m, the stages of the slave component only require the results from the previous stages for m. This implies that one method could be fully analyzed by all four

```
input   WorkLoad : set of Jimple method bodies
input   Hierarchy : CHA hierarchy information for all input classes
procedure   run(WorkLoad, Hierarchy)
 1: while WorkLoad ≠ ∅ do
 2:     curMeth := WorkLoad.RemoveAny()
 3:     curCFG := IntraproceduralCFGs(curMeth, Hierarcy)
 4:     curLive := LivenessAnalysis(curMeth, curCFGS)
 5:     curAlias := AliasAnalaysis(curMeth, curGFG, curLive)
 6:     curRD := ReachingDefAnalysis(curMeth, curCFG, curLive, curAlias)
 7:     curIntraFG := IntraFlowGraph(curMeth, curCFG, curLive, curAlias, curRD)
 8:     IntraFlowGraphs := IntraFlowGraphs ∪ {curIntraFG}
 9:     InterConnectInfo := InterConnectInfo ∪ getInterInfo(curIntraFG)
10: end while
```

Figure 4.5: Reducing memory footprint: algorithm for slave's *run* method.

stages before work on the next method starts. Such a transformation would make it possible to parallelize a portion of the flow graph creation process by creating an intraprocedural flow graph for each method. Figure 4.4 shows the high-level design of such a transformation. The master component of this new design is identical to the master component of the intuitive design. It is the slave component that has been modified.

Figure 4.5 shows the new *run* method for slave objects. The new method iterates over all the methods in the workload. Each method is removed and individually analyzed by each stage. The new procedure call, *IntraFlowGraph*, creates the intraprocedural flow graphs for each method. After a method's intraprocedural flow graph has been created, all non-interprocedural information associated with that method is released for garbage collection (i.e., everything except for calling relationships between methods). The *Intra Flow Graph* stage (Figure 4.4) builds an intraprocedural flow graph for a single method A in exactly the same manner as the *Inter Flow Graph* stage of the intuitive design (Figure 4.1) would if method A was the only input. The

only difference is that *Intra Flow Graph* maintains additional interprocedural information for each method. This information indentifies nodes which may be connected by interprocedural edges in the final interprocedural flow graph. This interprocedural information and the intraprocedural flow graphs are collected by the master thread once a slave completes its workload. The information is passed to the *Inter Flow Graph* stage (Figure 4.4). This stage is now greatly simplified; it iterates over the calling information discovered in the *Intra Flow Graph* stage and connects the appropriate nodes. The graph simplification is same as in the sequential version of JSA.

This design offers several improvements over the previous design. First, a large portion of the building of the interprocedural graph has been parallelized. Secondly, passing a single method through the first four stages could potentially reduce the number of cache misses. In the original design, if method m were the first method analyzed by the *Intra CFG* stage, the corresponding CFG data structures would not be accessed again until the stage had finished analyzing all the methods in the workload. Given a large enough workload this would mean that m's CFG would no longer be in cache when it was first accessed by the *Liveness Analysis* stage. In the worst case every access to a CFG made by the *Liveness Analysis* stage could result in a cache miss. By passing a single method through all the slave stages the relevant objects for that method are more likely to remain in cache. Lastly, the total memory footprint for certain inputs will be reduced. By creating an intraprocedural flow graph for each method, all the purely intraprocedural information from the slave stages can be garbage collected immediately, eliminating the tremendous build up of information being passed to the *Inter Flow Graph* stage.

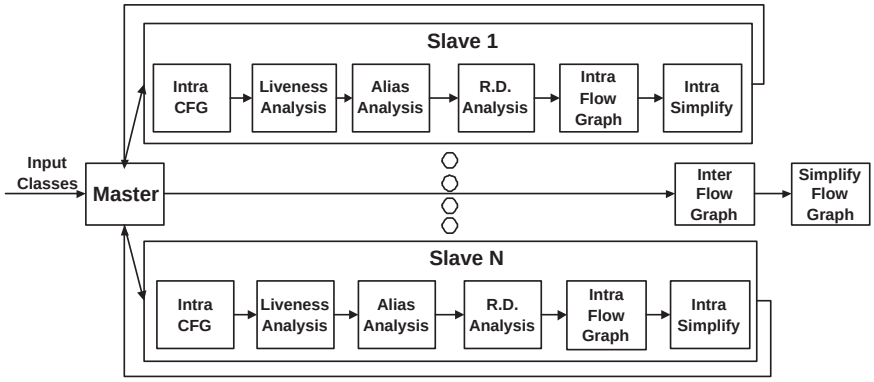

Figure 4.6: Parallel JSA design: parallel flow graph simplification.

4.1.3 Parallel Graph Simplification

The final stage of JSA's front-end is the simplification of the complete interprocedural flow graph. As our experiments will show, the reduction of the flow graph can have a significant impact on the running time of the back-end. In the general cases, to achieve maximum benefit of the simplifications, the entire interprocedural graph must be considered. Currently, the multi-threaded designs proposed above leave the simplification step as a sequential element in the design, thus ensuring that the complete graph is considered. Another option is to introduce a new stage which performs the simplifications on the intraprocedural flow graphs, then performs a second, abbreviated version of the simplification algorithm on the interprocedural graph. This is the design shown in Figure 4.6.

The new stage labeled *Intra Simplify* in the slave components simplifies the intraprocedural flow graphs immediately after they are created. This design allows for a large portion of the graph simplification process to run in parallel potentially reducing the overall running time.

The intraprocedural simplifications must be performed with care in order to preserve the desired semantics of the interprocedural graph. Consider the algorithm JSA uses to simplify the interprocedural flow graph: (1) all nodes are put into a worklist (2) a node n is randomly removed from the worklist, its type (e.g., *JoinNode*) and its incoming edges are inspected to determine if a simplification can be performed (a description of the simplifications can be found in Section 2.2), (3) if a simplification has been performed on n all of its predecessors are added to the worklist (if they are not already on it). Notice that the simplifications performed by JSA depend only on the type of the node being considered and its incoming edges, not the order in which they are simplified. Since each intraprocedural graph is a sub-graph of the interprocedural graph, the only information that is missing is at the nodes that will be connected by interprocedural edges. Therefore, any node in the intraprocedural graphs can be simplified as long as its incoming edges will remain the same in the interprocedural graph. By exempting nodes which could possibly have new incoming edges in the interprocedural graph from the intraprocedural simplification, the semantics of the final graph will be preserved. The nodes which must be exempt are those which represent: (1) formal parameters of methods, (2) variables that are assigned values returned by method calls, (3) and aliased actual parameters.

Figure 4.7 presents the new intraprocedural graph simplification algorithm. It is important to note that we are not introducing any new simplification but specifying how the existing simplification made by JSA can be applied at the intraprocedural level. This procedure takes as input a graph (a set of nodes and edges) and a set of nodes *ExemptNodes*. The nodes contained in *ExemptNodes* are identified in the *Intra Flow Graph* stage as being nodes which could possibly have interprocedural predecessors (incoming edges from other methods). The procedure returns a simplified version of the input graph. The algorithm is driven by a worklist that is initialized with all

```
input  IntraGraph : set of nodes and edges that comprise an intraprocedural graph
input  ExemptNodes : set of nodes that are potentially effected by interprocedural edges
procedure simplify(IntraGraph,ExemptNodes)
 1: worklist := IntraGraph − ExemptNodes
 2: while worklist ≠ ∅ do
 3:    n := worklist.RemoveAny()
 4:    if ContainsEq(n, IntraGraph) then
 5:       eqn := GetEq(n, IntraGraph)
 6:       MergeNodes(n, eqn, IntraGraph)
 7:    else if n instance of JoinNode then
 8:       if ContainsSelfLoop(n, IntraGraph) then
 9:          RemoveSelfLoop(n, IntraGraph)
10:       end if
11:       if ContainsOnePredecessor(n, IntraGraph) then
12:          Condense(n, IntraGraph)
13:       end if
14:    else if n instance of ConcatNode then
15:       if LeftArgEmptyString(n, IntraGraph) then
16:          ReplaceConcat(n, IntraGraph)
17:       end if
18:    end if
19:    if ModifiedOutGoingedges(n, IntraGraph) then
20:       for each suc ∈ GetSuccessors(n, IntraGraph) do
21:          if suc ∉ ExemptNodes then
22:             worklist := worklist ∪ {suc}
23:          end if
24:       end for
25:    end if
26: end while
```

Figure 4.7: Simplification algorithm for intraprocedural flow graphs.

the nodes from the input graph except for the nodes contained in *ExemptNodes*. A node n is removed from the worklist and is checked at line 4 to determine if an equivalent node exists in the graph. Two nodes are equivalent if they are the same type (e.g., both `Join` nodes) and they have the same incoming edges. Two `Init` nodes are equivalent if they contain equivalent automata. If an equivalent node is found, n and the node are merged (lines 5 and 6). If there are multiple equivalent nodes, *GetEq* will arbitrarily return one of them. If no equivalent nodes are found and n is a `JoinNode`, it is checked to determine if it contains a self loop (an edge from n to n.) If such an

```
input  InterGraph : set of nodes and edges that comprise the interprocedural graph
input  StartNodes : all Init nodes and nodes with an incoming interprocedural edge
procedure simplify(InterGraph,StartNodes)
 1: worklist := StartNodes
 2: while worklist ≠ ∅ do
 3:    n := worklist.RemoveAny()
 4:    if ContainsEq(n, InterGraph) then
 5:       eqn := GetEq(n, InterGraph)
 6:       MergeNodes(n, eqn, InterGraph)
 7:    else if n instance of JoinNode then
 8:       if ContainsSelfLoop(n, InterGraph) then
 9:          RemoveSelfLoop(n, InterGraph)
10:       end if
11:       if ContainsOnePredecessor(n, InterGraph) then
12:          Condense(n, InterGraph)
13:       end if
14:    else if n instance of ConcatNode then
15:       if LeftArgEmptyString(n, InterGraph) then
16:          ReplaceConcat(n, InterGraph)
17:       end if
18:    end if
19:    if ModifiedOutgoingEdges(n, InterGraph) then
20:       worklist := worklist ∪ GetSuccessors(n, InterGraph)
21:    end if
22: end while
```

Figure 4.8: Simplification algorithm for interprocedural flow graphs.

edge exists it is removed (lines 8 and 9). Similarly, if n is a `JoinNode` that has only one predecessor, n is removed and its edges are redirected (line 12). Lines 14 through 18 perform the simplification for concatenation nodes whose leftmost argument is the empty string. Finally, if the outgoing edges of n have been modified, then all of the successors will be added back into the worklist since their incoming edges may have been modified (lines 19–25) . The one special case is when the successor is an element of *ExemptNodes*, in which case it is ignored.

When the intraprocedural graphs are joined by interprocedural edges in the *Inter Flow Graph* stage, the complete graph may not be minimal. There are two possible reasons why a node in the interprocedural graph may require further simplification.

First, the node may have new incoming interprocedural edges meaning that it was exempt from the intraprocedural simplification. The second case occurs when, in the new global context, there are new node equivalencies.

The bulk of the algorithm for the *Simplify Flow Graph* stage shown in Figure 4.8 is identical to the intraprocedural simplification algorithm. The main difference is that the interprocedural version specifies a set of seed nodes *StartNodes* from which to start the simplification process. *StartNodes* contain all the nodes that were exempt from the intraprocedural simplification, as well as all the `Init` nodes. No other nodes need to be checked for equivalence since nodes from separate methods that do not have incoming interprocedural edges could not possibly have the same incoming edges (only `Init` nodes do not have incoming edges.) The remainder of the algorithm performs the same simplifications. The seed nodes are removed from the worklist and are examined to determine if a simplification is possible (lines 2–18). If a node is simplified all of its successors are added to the worklist to determine if further simplifications are possible.

4.2 New Flow Graph Simplifications

It has been shown that the simplifications performed by JSA can reduce the running time of the back-end [20]. By performing these simplifications a gamble is being made that the amount of time it takes to perform the reductions will be less than the time saved by performing them. By parallelizing the simplifications of the graph we have reduced the cost of the initial wager. Since the cost of performing the simplifications is now reduced, we can incorporate further simplifications. In this section we present three new graph simplifications which may reduce the cost of the back-end.

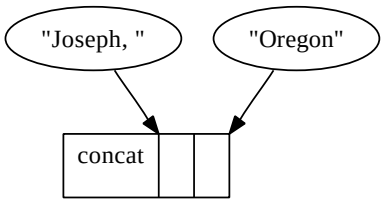

Figure 4.9: Concatenation example: not simplified.

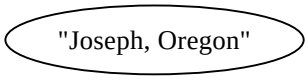

Figure 4.10: Concatenation example: simplified.

4.2.1 Concatenation Simplification

JSA simplifies the flow graph by bypassing `Concat` nodes if the first argument of the node has exactly one edge and it comes from an initialization node representing an empty string value (e.g., `s=""+z;`). Now that the cost of simplification has been reduced we can perform a second reduction similar to this one. If a `Concat` node has exactly one incoming edge for both the first and second arguments and these edges originate from `Init` nodes we can replace the `Concat` node with a new `Init` node. An example of this situation is shown in Figure 4.9. In this example two `Init` nodes are being concatenated. By inlining the concatenation we can remove the `Concat` and replace it with the `Init` node as shown in Figure 4.10. In this example the values of the original `Init` nodes were only being used in the concatenation and we can remove those nodes as well. The next section discusses how this transformation can be accomplished.

This simplification requires a small addition to the algorithm shown in Figure 4.6. Figure 4.11 shows the necessary changes. Line 4 performs a check to determine if the arguments of the `Concat` node are single `Init` nodes. It then creates a new `Init` node in line 5 by concatenating the values of the two original `Init` nodes. This concatenation is achieved by redirecting all of the transitions which directly lead to the accept state in the automaton associated with the left argument to the start state

```
1: if n instance of ConcatNode then
2:    if LeftArgEmptyString(n, IntraGraph) then
3:       ReplaceConcat(n, IntraGraph)
4:    else if LeftArgSingleInit(n, IntraGraph) ⋀ RightArgSingleInit(n, IntraGraph)
      then
5:       nInit := CreateCombinedInit(n.leftArg, n.rightArg)
6:       ReplaceNode(n, nInit)
7:    end if
8: end if
```

Figure 4.11: Concatenation simplification: algorithm for new concatenation simplification.

of the automaton associated with the right argument. Line 6 replaces the original `Concat` node with the new `Init` node. This is achieved by deleting all the `Concat`'s incoming edges and copying all of its outgoing edges to the new node.

4.2.2 Removal of Extraneous Nodes

If a user of JSA wanted to determine all possible error messages displayed when an exception is thrown in her application she might designate the constructors of certain exception types as hotspots[5]. If this were the case, the code sample shown in Figure 4.12 would contain exactly one hotspot at line 31, the instantiation of `IllegalArgumentException`. The only value that flows to this hotspot is `"Error in copy..."`; however, when JSA is applied to this code it creates the flow graph displayed in Figure 4.13. The flow graphs generated by JSA contain the flow of all string values in the input classes regardless of whether these values can affect any hotspots. In this example, the only node that is required is labeled `"Error in copy..."`, and all other nodes can be discarded. By removing extraneous nodes it

[5]Class `java.lang.Exception` and many of its extensions contain constructors that require a `String` argument. This argument specifies the message associated with an exception object.

```
1  public boolean saveCopyIntern ( int modus, boolean confirm )
2  {
3     String title, hstr, titleToken, confirmMsg, confirmOpt;
4     boolean fileExists, isNew, backup, saveAs, copy, save;
5     ......
6     backup = modus == SAVE_BACKUP;
7     saveAs = modus == SAVE_AS;
8     copy = modus == SAVE_COPY;
9     save = modus == SAVE_FILE;
10    if ( backup ){
11       titleToken = "dlg.savebackup";
12       confirmMsg = "msg.confirm.backup";
13       confirmOpt = "confirmBackup";
14       opcode = OP_BACKUP;
15    }else if ( save ){
16       titleToken = "dlg.savefile";
17       confirmMsg = "msg.confirm.savefile";
18       confirmOpt = "confirmSave";
19       opcode = OP_SAVE;
20    }else if ( saveAs ){
21       titleToken = "dlg.saveas";
22       confirmMsg = "msg.confirm.saveas";
23       confirmOpt = "confirmSaveAs";
24       opcode = OP_SAVEAS;
25    }else if ( copy ){
26       titleToken = "dlg.savecopy";
27       confirmMsg = "msg.confirm.savecopy";
28       confirmOpt = "confirmFileCopy";
29       opcode = OP_SAVECOPY;
30    }else
31       throw new IllegalArgumentException("Error␣in␣copy...");
32   ......
33 }
```

Figure 4.12: Sample code from the JPWS benchmark.

may be possible to achieve a speedup in running time for JSA's back-end since it may eliminate useless computations.

Figure 4.14 presents an algorithm that will identity all extraneous nodes in a graph (i.e., those which will not affect the values of hotspot expressions). The algorithm takes as input a set of nodes *EffectNodes*. These are the nodes that are

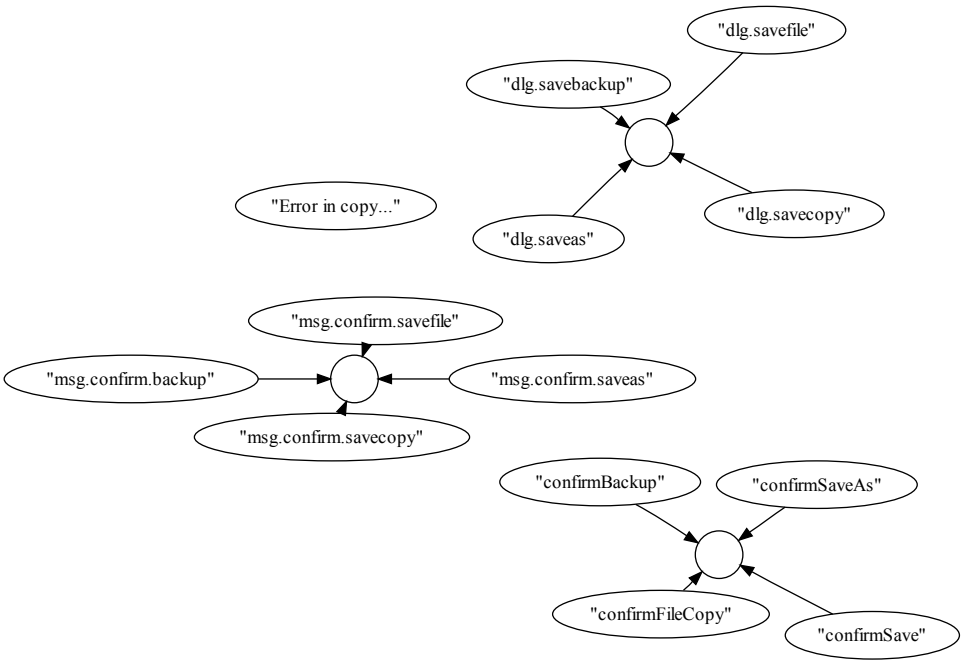

Figure 4.13: JSA flow graph of JPWS sample code.

predetermined to have a possible effect on at least one hotspot. In the complete interprocedural graph *EffectNodes* will contain only the nodes that represent hotspots. These nodes are identified during the construction of the intraprocedural flow graphs. The *removeExtraneous* procedure performs a reverse traversal of all the paths leading to these nodes in the graph. All nodes encountered on these paths have the potential to affect the value of a hotspot. Inversely, all nodes not on observed paths are extraneous and cannot affect a hotspot expression. These nodes can be removed from the graph.

This simplification can be performed on intraprocedural graphs with a few simple additions to the *EffectNodes* set. For an intraprocedural graph, *EffectNodes* should include nodes contained in the graph that represent hotspot expressions. It also must include any values which could possibly escape the method being modeled by the graph as these values could affect hotspots in other methods. Thus, in addition to

input *Graph* : the graph being simplified
input *EffectNodes* : set of nodes that are identified as potentially affecting hotspots
output *RevisedGraph* : simplified graph contain only information that can affect hotspots
procedure removeExtraneous(*Graph*,*EffectNodes*)

```
 1: worklist := EffectNodes
 2: seen := ∅
 3: while worklist ≠ ∅ do
 4:     n := worklist.RemoveAny()
 5:     if n ∉ seen then
 6:         seen := seen ∪ {n}
 7:         preds := GetAllPredecessors(n, Graph)
 8:         worklist := worklist ∪ preds
 9:     end if
10: end while
11: return RevisedGraph := Trim(Graph, seen)
```

Figure 4.14: Algorithm to remove all nodes that cannot affect the value of hotspot strings.

hotspot nodes, *EffectNodes* must also contain: (1) all return nodes, (2) all nodes that represent aliased formal arguments, and (3) parameters to method calls (only to methods that are contained in the original input classes). By performing the simplification at both the intra and interprocedural levels, the memory footprint of JSA may be reduced by discarding unnecessary information as soon as possible.

4.2.3 Propagation of Anystring Values

If a programmer wanted to understand all of JEdit's potential debug messages, JSA could be used to aid in this endeavor. All calls to the `Debug` method could be flagged as a hotspots. Consider the code example shown in Figure 4.15. The call to `Debug` at line 16 would be a hotspot. The intermediate flow graph JSA would generate for this example is shown in Figure 4.16.

As shown, all the possible values for `method` flow to a single `Join` node. The values then flow to the right argument of a `Concat` node whose left hand argument

```
1  public void doSwitchBranch(String [] params, String method){
2    ....
3    if(method == null){
4       String type = params[0];
5       if      ( type.equals("Z") )
6           method = "getBoolean";
7       else if ( type.equals("B") )
8           method = "getByte";
9       else if ( type.equals("C") )
10          method = "getChar";
11      else if ( type.equals("S") )
12          method = "getShort";
13      else
14          method = "getObject";
15   }
16   Debug("Invoking␣method␣"+method);
17   ....
18 }
```

Figure 4.15: Sample code from the JEdit benchmark.

is `"Invoking method "`. From the finite state automaton JSA produces for this flow graph, all that can be determined about this invocation of `Debug` is that it will display a string starting with `"Invoking method "` followed by any Unicode string value. The reason for the infinite number of possibilities can be observed by again examining the flow graph in Figure 4.16. Notice that one of the values flowing to the `Join` node is `<any string>`[6]. Under JSA's open-world assumption (see Section 2.2.3) this value represents that `doSwitchBranch` is a public method and therefore the `method` parameter might be assigned string values from un-analyzed code. Since the *anystring* value subsumes all others, the specific `"getBoolean"`, `"getByte"`, etc. string values are lost after the `Join`. In general, the outgoing value of any `Join` node that has an `<any string>` node as a direct predecessor will be *anystring*. This means that any `Join` node with such a predecessor can be removed, all its incoming edges deleted,

[6] An `<any string>` node is an `Init` node which represents the *anystring* value.

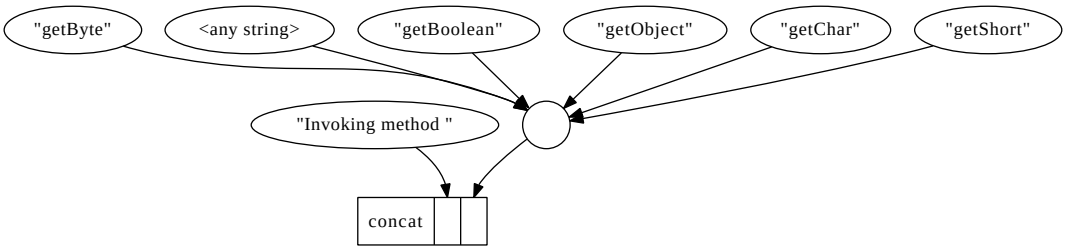

Figure 4.16: JSA flow graph of JEdit sample code.

```
1: if n instance of JoinNode then
2:     if HasAnyStringPred(n, IntraGraph) then
3:         node := GetAnyStringPred(n, IntraGraph)
4:         RemoveIncomingEdges(n, IntraGraph)
5:         RedirectOutgoingEdges(n, node, IntraGraph)
6:     end if
7: end if
```

Figure 4.17: AnyString propagation simplification: algorithm for propagating anystring values through the flow graph.

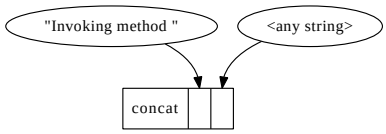

Figure 4.18: Reduced flow graph of JEdit sample code.

and replaced with a much simpler `<any string>` node. Such a simplification can be achieved with an addition to the algorithm shown in Figure 4.6.

Figure 4.17 presents the addition that must be made to the intraprocedural simplification algorithm to propagate the *anystring* value through the flow graph. As the algorithm iterates over all the nodes in the graph, it checks to determine if the current node n is a `Join` node. If it is, the algorithm checks all of its immediate predecessors to see if one is an `<any string>` node (line 2). If such a relationship is discovered, the incoming edges to the `Join` node are removed (line 4) and its outgoing edge is redirected such that it originates from the `<any string>` node (line 5). A very similar addition to the interprocedural simplification algorithm will achieve the

same result in the complete flow graph. Figure 4.18 present the flow graph after the above simplification was applied and the extraneous nodes removed.

4.3 Experimental Evaluation

To evaluate the effectiveness of our proposed parallelization of JSA and new graph simplifications, we performed an empirical study on 25 benchmark applications. This section presents that study. We first discuss the benchmarks and the experimental setup. We then present the evaluation of our parallel designs and we conclude with the results from our investigation of the new flow graph simplifications.

4.3.1 Benchmarks and Experimental Setup

Table 4.1 presents the 25 benchmark applications that were used in this study. For each benchmark **Num Classes** shows the number of class files that are unique to the application; this number does not include library classes that may be referenced by the application. Column **K Jimple Stmt** displays the number of Jimple statements contained in the application's class files. Column **Num Hotspots** presents how many of the following hotspots are found in each application:

- `java.lang.System.out.println(String)`
- `java.lang.System.err.println(String)`
- `java.lang.Class.forName(String)`
- `java.io.File(String)`
- `java.sql.Connection.prepareStatement(String)`
- `java.io.Writer.write(String)`
- `java.sql.Statement.excute(String)`
- `java.sql.Statement.excuteUpdate(String)`
- `javax.swing.Jlabel(String)`
- `javax.swing.JTextArea(String)`
- `java.lang.System.getProperty(String)`

Benchmark	Num Classes	K Jimple Stmt	Num Hotspots
Buoy	188	14586	0
Compress	24	3088	36
DB	15	3119	66
Fractal	25	3718	45
GattMath	55	8241	13
Jack	68	12536	328
Javac	188	26574	51
JavaCup	42	10474	107
Jb61	45	7433	33
JEdit	851	124830	458
Jess	163	13282	51
JFlex	75	15455	39
JGap	174	15331	34
JLex	25	8241	61
Jpws	193	28425	167
Jtar	130	6974	49
Mindterm	135	30626	94
MpegAudio	67	15082	43
Muffin	278	37748	200
Rabbit	76	10148	84
Sablecc	267	36155	101
Sockecho	24	4340	3
Sockproxy	26	4640	3
VietPad	215	24998	78
Violet	130	9959	12

Table 4.1: Benchmarks statistics: number of classes, Jimple statements, and hotspots.

These hotspots have been areas of concern discussed in other papers [19, 20, 120, 121] or are frequently used methods. All of the benchmarks have been well established in previous work [71, 110, 116, 119, 122, 126, 148] and were selected to represent a large variety of applications in terms of function, size, and number of hotspots.

All experiments were performed on a Dell PowerEdge R300 with a Quad Core Intel®Xeon®X3363, 2.83GHz, 2x6M Cache, 1333MHz FSB and 8Gb of RAM. The execution environment was running a Linux 2.6.18 operating system. All experiments were executed on a Java HotSpot(TM) 64-Bit Server VM.

To limit the effects of garbage collection and to maximize resource utilization all experiments were executed with the following JVM options set:

- `-Xmx6000m`: Sets maximum Java heap size to 6Gb.
- `-Xms4000m`: Sets the initial Java heap size 4Gb.
- `-Xss1000m`: Sets the stack size for each thread to 1Gb.

4.3.2 Evaluation of Proposed Algorithms on Front-End Running Time and Memory Usage

To evaluate our proposed parallel designs described in Section 4.1 and the new graph simplifications detailed in Section 4.2, we implemented 5 versions of JSA:

- **JSA-ORIG**: JSA in its original form with minor bug fixes.
- **JSA-INTU**: A parallelized version of JSA based on the intuitive design described in Section 4.1.1.
- **JSA-RMEM**: This version of JSA was parallelized using the algorithmic transformation designed to reduce the memory usage described in Section 4.1.2.
- **JSA-PSIM**: JSA-RMEM with the addition of the parallel graph simplification stage presented in Section 4.1.3.
- **JSA-NSIM**: JSA-PSIM with the addition of the new graph simplifications presented in Section 4.2.

Each version was executed on the set of benchmarks listed in Table 4.1. The input for an execution was the entire set of classes for the benchmark, all the hotspots listed above and the number of slave threads the implementation was to use to build

Implementation	1 Thread	2 Threads	3 Threads	4 Threads	Mem. Red.
JSA-ORIG	1	NA	NA	NA	1
JSA-INTU	.99	1.20	1.21	1.21	-5%
JSA-RMEM	1.08	1.33	1.36	1.30	44%
JSA-PSIM	1.07	1.54	1.52	1.53	43%
JSA-NSIM	1.02	1.39	1.44	1.40	41%

Table 4.2: Front-end time and memory results: Average speedup achieved for 22 benchmark applications. Column **Mem. Red.** displays the average percentage memory reduction achieved; the results of JSA-ORIG were used as the baseline.

the flow graph. Since JSA-ORIG is a serial implementation it was only executed on 1 thread. The remaining implementations were executed using 1, 2, 3, and 4 slave threads. This progression was used so that the speedup of the parallelism could be observed. The maximum of 4 threads was selected because the system used in this study contained 4 cores. If more than 4 threads were used by our parallel versions of JSA, the slaves would begin to compete with each other for processing resources. Each implementation was executed 5 times for each input. During these executions, time measurements for the front-end and the back-end were taken. The lowest measurements observed (being the fastest execution time) during the 5 executions was recorded. Similarly, memory measurements were recorded for the lowest maximum memory footprint observed for each benchmark. In other words, during a single execution the memory usage was monitored and the maximum amount used was noted. Out of 5 executions the lowest maximum was recorded to represent the lowest possible amount of memory needed to completely analyze the benchmark. Due to duplicated structures and information, the memory footprint for the parallel implementations grows with each additional slave, therefore memory measurements for these versions were recorded only when the input required 4 slave threads.

Table 4.2 displays the average speedup each version of JSA was able to achieve (compared to JSA-ORIG) when building the flow graphs for 22 benchmark applications (listed in Table 4.1). Column **Mem.Red.** displays the average percentage of memory reduction (i.e., $1 - (\text{JSA-RMEM}_{\text{MEM}}/\text{JSA-ORIG}_{\text{MEM}})$) observed during the analysis. The results from the JEdit, Jess, JGap benchmark applications are not included in the averages. This is because JSA-ORIG, JSA-INTU, JSA-RMEM, and JSA-PSIM could not complete an analysis of these benchmarks; they all exhausted the 6Gb heap. The averages present a very coarse overview of the results of our experiments. The complete set of results is shown in Appendix A.

JSA-INTU. Notice that, on average, when JSA-INTU runs sequentially (only uses 1 thread of execution) it is a little slower than JSA-ORIG. This is to be expected since it naively executes the load balancing heuristic, and creates special structures that are needed for multiple threads. When concurrent execution is used, this design produces a very modest speedup in the building of the flow graph. Using four slave threads, on average, this version only achieves an average speedup of 1.21 over JSA-ORIG. The `Thread` structures and the duplicated information slightly increases the memory footprint of this version.

There are several reason why a more substantial speedup was not achieved. The first being that the graph simplification portion of the front-end is still sequential in this version. Depending on the input this phase can represent a significant amount of work. Another factor that was dependent upon the input benchmark was the effectiveness of our load balancing heuristic. For the majority of the applications our heuristics appeared to be perform adequately with slave threads completing within a few hundred milliseconds of each other. However, for a few benchmark a single slave would perform the majority of the work. One such example is `Violet`. This application actually ran slower when multiple threads were used. This can be attributed to

poorly balanced workloads. `Violet` contains several methods which contain a large number of `String` objects. These methods may have a similar number of Jimple statements as other methods but require much more analysis time. A load balancing heuristic which weights methods by both number of Jimple statements and `String` objects might produce better balance. However, our heuristic generates the workloads for the largest application, `JEdit`, in less than 15 ms. A more complicated heuristic would take more time which might offset any gains in a large number of applications.

Another likely reason for the underwhelming results is contention for memory resources. As stated in Section 2.3, one of the common causes of bottlenecks in parallel Java applications is excessive object allocation [67]. In this design the *Intra CFG* stage creates multiple objects for each Jimple statement present in the input classes. In this design all slave threads will start in this stage and remain there until all the methods in their workload have been converted to CFGs. This is a prolonged period when all threads are contending for memory resources. Finally, there might have been some contention for processor resources. Even though JSA-INTU used at most 4 threads of execution, these threads may have been contending for processing resources with the JVM, which is a multi-threaded application itself.

JSA-RMEM. This version attempts to reduce contention for memory resources by running a single method through all of the intraprocedural flow analyses before beginning analysis of the next method. This technique produced an average speedup of 1.08 over JSA-ORIG even without using concurrency. This speedup can be attributed to the reduction of memory used by this version. On average JSA-RMEM used 44% less memory than JSA-ORIG. This reduction meant that cache misses and access times to certain data structures (e.g., `HashMaps`) were reduced. However, it proved to be even less scalable than JSA-INTU. The greatest speedup of 1.36 was

achieved when 3 threads were used. However, when 4 threads were used the average speedup actually declined.

This version suffers from many of the same problems that JSA-INTU incurred. It still executes all of the graph simplification as a sequential step. Even though the heavy object allocation performed by the *Intra CFG* stage has been diffused, ultimately the same number of objects will be created thus creating contention for memory resources. These issues, load balancing inconsistencies, contention for processing resources, and the new *Intra Flow Graph* stage meant that this version ran a little slower on average when 4 threads were used, compared to using 3 threads.

JSA-PSIM. This version parallelizes a portion of the flow graph simplification performed by the front-end of JSA through the introduction of a *Intra Simplify* stage. The addition of this stage only slightly increases the memory footprint and sequential running time of this version as compared to JSA-RMEM. However, this version is able to achieve a maximum speedup of 1.54 when compared to JSA-ORIG. The biggest gain is seen when only 2 slave threads are used. It is likely that contention for memory and processor resources is too great to realize scalability for these benchmarks on this experimental system.

JSA-NSIM. This implementation is the same as JSA-PSIM with the addition of the new graph simplifications proposed in Section 4.2. These additions increase both the building time of the flow graph and the memory footprint when compared to JSA-PSIM. This is to be expected since the new simplifications introduces more work. However, it is important to notice that even with the increased work JSA-NSIM completes the building of the flow graph in a shorter amount of time than JSA-ORIG, meaning that using concurrency can offset the cost of the new simplifications.

4.3.3 Back-End Running Time

The flow graphs generated by the front-end of JSA-ORIG, JSA-INTU, JSA-RMEM, and JSA-PSIM for a given application will be identical. Since we made no effort to modify the back-end, the running times for this portion of these implementations are approximately identical (times can vary due to garbage collection and other interactions with system resources.) JSA-NSIM produces a reduced flow graph that preserves the details of the graph that are needed for hotspots but removes certain irrelevant information. The additional reductions performed by JSA-NSIM have the potential to significantly reduce the running time of the back-end for certain inputs.

Table 4.3 shows the timing results for JSA-ORIG and JSA-NSIM (using 4 slave threads to build the flow graph) for the 22 benchmark applications that JSA-ORIG was able to fully analyze. As can be seen, for the majority of the applications it takes JSA-ORIG longer to build the flow graph than it does for the back-end to compute the finite state automata for the hotspots. The exceptions are Javac, JWPS, Mindterm, and Muffin. For JWPS and Muffin the difference is very significant; it takes the back-end 147 times longer for JWPS and 109 times longer for Muffin. This imbalanced relationship is likely due to the potential for doubly-exponential blowup when the back-end converts the MFLAs to DFAs. This kind of blowup does not occur often but is more likely to happen in complicated flow graphs which contain numerous hotspots that are connected. Since our simplification which propagates *anystring* values breaks connected graph components, this exponential blowup is less likely to occur in our graphs. As shown in the table, JSA-NSIM's back-end completes 1138 times faster than JSA-ORIG's for JPWS and 1073 times faster for Muffin. This time savings implies that JSA-NSIM achieves an overall speedup of 230 for Jpws and 184 for Muffin.

	JSA-ORIG			JSA-NSIM (4 Threads)			
Apps	**Front**	**Back**	**Total**	**Front**	**Back**	**Total**	**Speedup**
Buoy	1502	23	1525	1274	9	1283	1.18
Compress	650	325	975	621	314	935	1.04
DB	690	373	1063	527	319	846	1.25
Fractal	664	512	1176	993	174	1167	1.00
GattMath	1319	72	1391	989	19	1008	1.37
Jack	1701	1362	3063	1287	502	1789	1.71
Javac	3021	7325	10346	2189	211	2400	4.31
JavaCup	2861	1795	4656	2421	502	2923	1.59
Jb61	1322	623	1945	932	373	1305	1.49
JFlex	6129	961	7090	5443	37	5480	1.29
JLex	1340	415	1755	725	222	947	1.85
Jpws	4023	594057	598080	2072	522	2594	230.56
Jtar	1755	763	2518	1381	400	1781	1.41
Mindterm	3138	3478	6616	2418	94	2512	2.63
MpegAudio	2760	311	3071	2302	208	2510	1.22
Muffin	5496	604409	609905	2729	568	3297	184.98
Rabbit	1579	973	2552	1472	276	1748	1.45
Sablecc	5611	47034	52645	2723	45695	48418	1.08
Sockecho	759	264	1023	557	190	747	1.36
Sockproxy	754	234	988	622	227	849	1.16
VietPad	5949	4121	10070	2796	113	2909	3.46
violet	1375	52	1427	985	25	1010	1.41

Table 4.3: Running time (in ms) for JSA-ORIG and JSA-NSIM.

JSA-ORIG, JSA-INTU, JSA-RMEM, JSA-PSIM were unable to analyze the JEdit, Jess, and JGap benchmarks on the system where our experiments were conducted. For these applications the back-end stage of JSA experienced a similar blowup as it did for JWPS and Muffin. However, for these benchmarks it exhausted the allotted heap memory before the analysis could be completed. Table 4.4 shows the time results recorded for JSA-NSIM when it was applied to these benchmarks. Again, our new simplifications greatly reduced the resources needed by the back-end. It is also interesting to note that for JEdit the concurrent building of the flow graph appears to scale better than other benchmark applications (achieving over a 3 times speedup with 4 slave threads). This indicates that our proposed parallel algorithms may be

Benchmark	1 Thread	2 Threads	3 Threads	4 Threads	Back-end
JEdit	20451	9533	6877	6053	1021
Jess	1730	1418	1363	1350	457
JGap	1995	1591	1764	1873	556

Table 4.4: Running time results in ms for JSA-NSIM.

more relevant for large applications or whole-program analyses where library classes must also be considered.

4.4 Conclusion

This chapter presents multiple techniques to increase the scalability of the Java String Analyzer (JSA) library. These techniques include algorithmic transformations which parallelize portions of JSA's front-end, allowing it to leverage modern multi-core architectures. In an empirical study of 25 benchmark applications it was shown that for 22 of the benchmarks our most advanced parallel version of JSA's front-end was, on average, able to achieve a speedup of 1.54 over the original sequential version of JSA. Moreover, the parallel design reduced the average memory footprint for these 22 applications by approximately 43%. For the remaining 3 applications all version of JSA exhausted the allotted heap memory.

JSA's front-end builds a flow graph which represents the flow of string values through the input classes. This flow graph is the input to the back-end. We present three new simplifications to this graph which preserve relevant details at expressions of interest (hotspots) while eliminating several sources of superfluous information. These reductions have the potential to decrease the amount of work that must be performed by the back-end. On average, these new simplifications added 200 ms to the running

time of the front-end of the most advanced parallel version of JSA. However, for two applications, these simplifications reduced the running time of the back-end from over 590,000 ms to under 600 ms. Moreover, our simplifications allowed JSA to complete an analysis of three benchmark applications that previously exhausted a 6Gb heap.

CHAPTER 5: ASSUMPTION HIERARCHY FOR A CHA CALL GRAPH CONSTRUCTION ALGORITHM

Performing interprocedural static analysis of software written in an object-oriented programming language, such as C++ or Java, requires information about the calling relationships between program methods. This information is abstractly represented in a *call graph* where the nodes of the graph represent methods and directed edges represent calls between methods. These representations are often critical components of modern static analyses, e.g., [19,41,42,45,46,62,71,78,88,104,107,109,111,128,141, 143]. The building of a call graph for OO languages is complicated by the existence of polymorphism and virtual calls. There has been extensive work conducted in the area of call graph construction which addresses this issue with varying degrees of precision and expense, e.g., [2, 7, 9, 14, 27, 28, 57, 58, 61, 77, 83, 84, 94, 95, 112, 130, 133, 134, 140, 148, 149].

Class Hierarchy Analysis (CHA) [27] was one of the first, and simplest, call graph construction algorithms. For a virtual call site $x.m()$ where C is the compile-time type of x, CHA assumes that all subtypes of C are possible run-time receiver types of the call. As shown in Figure 5.1, a call graph created with CHA may contain infeasible edges. In this example the graph has an edge from `main` to `Z.m` even though an instance of `Z` is never created (for this example it can be assumed that the only calls made are shown in the code). In spite of its inherent imprecision, many static analyses use CHA to fulfill their need for a call graph, e.g., [19, 41, 62, 104, 128]. This

```
class A{
   public static void main(...){
       X x = new X()//c1
       if(...) x = new Y();//c2
       x.m();//c3
    }
}
class X { void m(){...}}
class Y extends X { void m(){...}}
class Z extends X { void m(){...}}
```

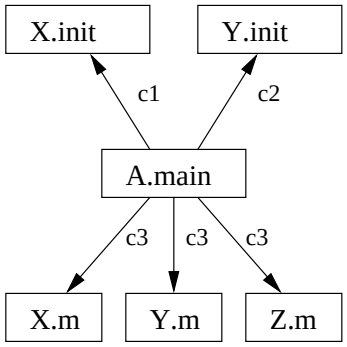

Figure 5.1: CHA imprecision example: code and resulting call graph.

is often due to the relative ease of implementing the algorithm and its fast running time ($\mathcal{O}(N)$ where N is the size of program).

Although the graph shown in Figure 5.1 contains an infeasible edge (`main`→ `Z.m`), it is sound—i.e., it contains all feasible edges (ignoring any implicit calls made from the JVM). The "perfect" call graph, the one that represents only and all possible run time calls between methods, is a sub-graph of the graph shown. A call graph is *unsound* if it does not represent all possible run-time calling relationships. For example, if the call graph in Figure 5.1 did not contain the edge `main`→ `Y.m` it would be unsound.

An unsound call graph could have disastrous affects on the analysis relying on it. Consider the very simple analysis that removes dead code from the production version of an application. Assume this analysis defines dead code as any `private` method which is neither reachable from `main` nor called implicitly by the JVM. Such an analysis could be implemented by removing `private` methods that are not reachable from `main` (i.e., the call graph does not contain a path from `main` to the method) or reachable due to calls made by the JVM. For such an analysis, an unsound call graph

could result in a method mistakenly being identified as dead code. In such a scenario the method would be removed, potentially causing failure of the application.

Unfortunately, the truly-dynamic nature of Java (Section 2.1) presents a significant challenge for *any* static call graph construction algorithms. In the general case it is impossible to statically determine the run-time actions of the Java constructs that allow for reflection, dynamic class loading, and interfacing with native methods. This fact requires developers of call graph construction algorithms to make assumptions about the effects of these dynamic features. These assumptions fall on a continuum.

At one extreme of the continuum, the algorithm simply ignores dynamic features. This action will produce an unsound call graph whenever it is applied to applications that use such features. This option was leveraged heavily in the past when relatively few applications were using dynamic constructs. However, this solution is no longer viable for modern applications which are more and more frequently supporting dynamic architectures. As shown in Chapter 3, the Java libraries also make use of these dynamic features and thus almost *any* application indirectly employs such features due to its use of library classes.

The other extreme assumption results in the call graph construction algorithm very conservatively estimating all possible effects of dynamic features. For example, consider a program `P` with a method `m` which contains a statement `stmt` that calls the reflective method `Method.invoke`. Recall from Section 2.1.2 that `Method.invoke` results in the invocation of the method being represented by `Method`. Since it is possible for reflection to break encapsulation, it could be conservatively assumed that a call to `Method.invoke` could result in any method in `P` being invoked. Thus in the call graph an edge would be added from `m` to every other method in `P`. A similar treatment would be applied to all dynamic features. Such a conservative treatment will create a sound result (under a closed-world assumption), but for most static

analyses the relevant information will be obfuscated by infeasible edges. Clearly, such an approach would be impractical for the example analysis above which removes dead code from applications. For any application that uses reflection in a method that is reachable from `main` (including library methods), the overly conservative call graph would assume that all methods could be reachable, and consequently no method could possibly be identified as dead code.

In this chapter we explore the effects of different assumptions made about dynamic features on two related static analyses. Our study first examines a hierarchy of assumptions about dynamic features that can be applied to a *May Be Loaded* (MBL) analysis. This analysis is typically thought of as preprocessing step of a CHA analysis. Its goal is to determine the set of classes that must be considered when the CHA analysis is being performed. We then examine assumptions that can be applied to a CHA call graph construction algorithm.

The remainder of this chapter is organized as follows: Section 5.1 provides an overview of an MBL analysis and describes how certain dynamic features of Java can render the results of the analysis unsound. We provide an overview of a CHA call graph construction algorithm in Section 5.2 and describe how dynamic features can introduce unsoundness into the graphs it generates. Section 5.3 discusses the specific assumptions about dynamic features that we will be examining. Section 5.4 presents our experimental study of both the MBL analysis and the CHA call graph construction algorithm. The chapter is summarized in Section 5.5.

5.1 May Be Loaded Analysis

A whole-program analysis, such as a CHA call graph construction algorithm, operates under a closed-world assumption. In other words, the analysis assumes that it has access to all relevant code entities. In the case of CHA, the analysis is

performed on a set of classes *AppClasses*. To be a sound analysis for an application P, the following must hold: $AppClasses_P \subseteq AppClasses$ where $AppClasses_P$ is the set of all classes which could possibly be loaded into the JVM during any execution of P. Typically, *AppClasses* will be much larger than the set of classes written by the application developers of P, for it will also include all the library classes that are directly and transitively accessed by P. For many applications, *AppClasses* will contain thousands of classes. Due to its large size, and the lack of access to the source code of libraries, it is impractical to expect a CHA client to manually specify *AppClasses*. Conversely, including every class accessible from the classpath could generate an *AppClasses* set that includes an unwieldy number of irrelevant classes. A large set could significantly increase the memory and time needed to complete the CHA analysis. It could also lead to numerous infeasible edges being added to the call graph depending upon how dynamic features are represented (as can be seen in Section 5.3.2).

The use of a *May Be Loaded* analysis is one approach of automatically estimating the classes that need to be included in *AppClasses*. A MBL analysis examines the code of an applications looking for actions that could cause a class to be loaded into the JVM. According to the JVM specification [87] the following actions will cause a class X to be loaded (*Loading Actions*):

1. An instance of X is created
2. A non-constant static field of X is assigned or used
3. A static method of X is invoked
4. A subclass of X is initialized
5. X is initialized due to certain reflective methods
6. X is initialized due to JVM startup

```
input   cm : class containing main
input   Cls : set of all classes available on the classpath
output  MBLCls : set of classes that may be loaded by P
procedure MaybeLoaded(cm,Cls)
 1: ClassWorklist := {cm}
 2: while ClassWorklist ≠ ∅ do
 3:    c : class := ClassWorklist.RemoveAny()
 4:    if c ∉ MBLCLs then
 5:       MBLCLs := MBLCLs ∪ {c}
 6:       for each stmt : statement ∈ c do
 7:          cl : class := null
 8:          if stmt instanceof new X then
 9:             cl := X      /*Loading Action (1)*/
10:          else if stmt instanceof X.fld then
11:             cl := X      /*Loading Action (2)*/
12:          else if stmt instanceof X.m(...) then
13:             cl := X      /*Loading Action (3)*/
14:          end if
15:          if cl ≠ null then
16:             superCls := GetSuperClasses(cl, Cls)      /*Loading Action(4)*/
17:             LCLs := LCLs ∪ {cl} ∪ superCls
18:             ClassWorklist := ClassWorklist ∪ {cl} ∪ superCls
19:          end if
20:       end for
21:    end if
22: end while
23: return MBLCLs
```

Figure 5.2: Simple May Be Loaded (MBL) analysis.

Figure 5.2 present an algorithm for a simple MBL analysis. The algorithm takes as input the class *cm* containing the `main` method for an application P, and the set of all classes in the "world"[7]. Since all executions of P start with `main`, *cm* is guaranteed to be loaded into the JVM. The MBL analysis initializes a standard worklist with *cm*. The algorithm continues as long as the worklist is not empty. It

[7]In practice this set is usually all classes available on the system classpath and the application classpath. This practice could be unsound for applications that use custom class loader which can load classes from alternate locations.

removes any class from the worklist. If the class has not previously been analyzed, all of its statements are examined (lines 3–6). MBL assumes that all code in a loaded class could be reached during an execution of P. If a statement instantiates a new class X or accesses a static field/method of X, then X and all of X's superclasses are added to the worklist (lines 6–20).

Note. An analysis similar to the one presented in Figure 5.2 can be conducted by leveraging the `constant_pool` structure contained inside every class file [87]. The `constant_pool` is a table of structures that represent entities that are referred to by the code contained in the class. Accessing this structure would be more efficient than inspecting every statement. However, we present the above algorithm for clarity and for further refinement to address certain dynamic features that could not be addressed by solely using the `constant_pool`.

5.1.1 Effects of Dynamic Features

The MBL analysis described above would be able to determine all classes loaded by P due to

1. An instance of X is created
2. A non-constant static field of X is assigned or used
3. A static method of X is invoked
4. A subclass of X is initialized

For the remainder of this dissertation we will refer to this set of actions as *statically determinable class loading actions*. Unfortunately, it would not be sound for many Java applications (i.e. $AppClasses_P \not\subseteq AppClasses$). The above analysis does not consider the dynamic features of Java which can cause classes to be loaded into the JVM.

The analysis does not address the classes that are loaded by the JVM upon startup. Recall, from Section 2.1, that different implementations of the JVM will perform different actions upon startup, including default loading of commonly used Java library classes. Not only are these classes loaded into the JVM, but their static initializer methods are invoked. Thus, even if these classes are not explicitly used by P their code could still affect P's run-time behavior.

Similarly, the above analysis does not contain any mechanisms to address dynamic class loading methods. As discussed in Section 2.1.1, in the general case it is impossible to determine the classes that will be loaded by methods such as `Class.forName`. However, there are techniques that can be used to aid in the resolution of certain invocations of these methods.

The set *AppClasses* calculated by the analysis would not include classes that may be loaded by native methods. As described in Section 2.1, native methods essentially have all capabilities of Java methods including the ability to load classes into the JVM. The algorithm shown in Figure 5.2 makes no provisions for such interactions.

The unpredictability of custom class loaders introduces another potential source of unsoundness. Class loaders developed by third party users are not required to adhere to the conventional actions of standard Java loaders. They can instrument the code of any class Z during the loading process, possibly altering the semantics of Z and the set of classes that could be loaded by Z. Custom loaders can load classes from sources that are not listed on the classpath. They can even substitute a specified class with an arbitrary replacement at loading time.

Any class Q loaded by the dynamic features listed in this section will not be present in the set *AppClasses* calculated by the above MBL analysis unless Q would have been loaded in a statically determinable manner elsewhere in P (e.g., an instance of Q is created using the `new` operator.) Missing even a single dynamically loaded

class has the potential to greatly impact the results of a MBL analysis, as that class could load a number of other class, and those classes will load yet other classes, and so on.

In Section 5.3 we present a hierarchy of assumptions that a MBL analysis could make about the interactions of such dynamic features. We specify how the simple algorithm presented in Figure 5.2 can be modified to remain sound under each assumption.

5.2 Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) was formally introduced by Dean et. al. [27] as a technique to conservatively estimate the possible receivers of dynamically-dispatched messages. This information could then be used by optimizing compilers to replace certain dynamic calls with far less expensive static ones. While this work considered a language which was dynamically typed, in Java the existence of static types simplifies the analysis to the following basic operation: given a call site $x.m()$, where the declared type of x is C, the possible run-time type of x must be a non-abstract subtype of C and the run-time target method for each such subtype can be determined by a simple bottom-up traversal of the class hierarchy.

In this section we present a CHA call graph construction algorithm which is motivated by the CHA call graph construction algorithm in Soot. The pseudocode represents Soot's algorithm at a high level. However, the presented version is relatively simple and does not fully address the complexities of the Java language. Later we will describe how this version can be altered to remain sound under certain assumptions about the dynamic features of Java.

- P = the program being analyzed
- *AppClasses* = set of classes (including library classes) that comprise P
- *DoPrivileged* = set of doPrivileged methods
- *JVMStartupMethods* = set of methods called by the JVM on startup
- *JVMStartupClinits* = set `clinit`s for all classes loaded by the JVM on startup
- *MainClinit* = `clinit` for the class declaring the `main` method

Figure 5.3: CHA algorithm notation.

5.2.1 CHA Call Graph Construction Algorithm

To create a call graph for an application P, CHA takes as input a set of Java classes *AppClasses* (this set is the result of running an MBL analysis on P) and a representation of P's `main` method. From these inputs CHA generates:

- A set of reachable methods *Reach*. These are methods which may be invoked during an execution of P starting in `main`.
- A call graph with nodes $m \in Reach$ and edges $e \in Reach \times Reach \times CallSites$. A call graph edge (m, n, c) indicates that method m contains a call site c at which one possible run-time target method is n.

We now present the pseudocode for a CHA call graph construction algorithm. Figure 5.3 provides definitions of several sets and variables that will be referenced in the algorithm.

CHAMain. The algorithm employs a standard worklist approach (Figure 5.4). The worklist is initialized with the `main` method, the `clinit` for the class containing the `main` method, the `clinit` methods for the JVM startup classes[8], and a set of methods that are invoked implicitly by the JVM upon startup (line 1). Method n is

[8]These are the same startup classes as discussed in Section 5.1

```
input  AppClasses : the set of all application classes
input  MainMethod : the main method of the application
output Reach := ∅ : set of reachable methods
output CallGraph := (∅, ∅) : call graph
procedure CHAMain(AppClasses,MainMeth)
 1: worklist := {MainMethod} ∪ {MainClinit} ∪ JVMStartupClinits ∪ JVMStartupMethods
 2: while worklist ≠ ∅ do
 3:     n := worklist.RemoveAny()
 4:     if n ∉ Reach then
 5:         Reach := Reach ∪ {n}
 6:         methSet := ProcessBody(n)
 7:         worklist := worklist ∪ methSet
 8:     end if
 9: end while
```

Figure 5.4: Main procedure for CHA.

arbitrarily removed from the worklist and, if it has not already been analyzed, it is added to *Reach* and is passed to *ProcessBody*. This procedure determines the set of methods which could possibly be called by n. These newly discovered methods are added to the worklist. The worklist iteration continues until $worklist = \emptyset$ indicating that all methods that are reachable from `main` have been analyzed.

ProcessBody. The *ProcessBody* procedure shown in Figure 5.5 analyzes a method m to determine the set of methods *methSet* that can be called by m. First, m is examined to determine if it is a constructor for some class C. If it is and C overrides `java.lang.Object.finalize`, an edge is added from m to $C.finalize$ and $C.finalize$ is added to *methSet* (lines 4–7). This edge represents that if an instance of C is created, the JVM may invoke its `finalize` method[9]. A more complete discussion of finalizer methods is available in Section 2.1.5.

[9]This is the same treatment that Soot 2.2.3 provides for `finalize` methods. It can lead to an unsound call graph since it does not consider situations where C does not declare a `finalize` method but some superclass of C does. In this situation if an instance of C is created the `finalize` method of its superclass will be invoked when the instance of C is garbage collected.

```
input   m : method
output  methSet : set of method which could be called from m
procedure ProcessBody(m)
 1: methSet := ∅
 2: if isConstructor(m) then
 3:     decClass := getDeclaringClass(m)
 4:     if hasFinalizeMethod(decClass) then
 5:         fin := getFinalizeMethod(decClass)
 6:         AddCallGraphEdge(m, fin)
 7:         methSet := methSet ∪ {fin}
 8:     end if
 9: end if
10: for each st : statement ∈ m do
11:     if (st instanceOf v = new X) ∨ (st contains X.fld) ∨ (st contains X.n()) then
12:         methSet := methSet ∪ AddClinits(m, st, X)
13:     end if
14:     if containsMonomorphicCall(st) then
15:         n := getInvokedMethod(st)
16:         methSet := methSet ∪ {n}
17:         AddCallGraphEdge(m, n, st)
18:         if n ∈ DoPrivileged then
19:             arg0 := getFirstArg(n)
20:             methSet := methSet ∪ ResolveVirtualCall(m, arg0, getRunMeth(arg0), st)
21:         end if
22:     end if
23:     if containsVirtualCall(st) then
24:         n := getInvokedMethod(st)
25:         rec := getCompileTimeReceiver(st)
26:         methSet := methSet ∪ ResolveVirtualCall(m, rec, n, vc)
27:         if isThreadStartMeth(n) then
28:             methSet := methSet ∪ ResolveVirtualCall(m, rec, getRunMeth(rec), st)
29:         end if
30:     end if
31: end for
32: return methSet
```

Figure 5.5: Processing the bodies of newly discovered reachable methods.

At line 10 *ProcessBody* begins to iterate over every statement *st* in m. If *st* requires a class X to be loaded into the JVM due to a statically determinable class loading action, then X is passed to the *AddClinits* procedure (lines 11–12).

AddClinits. Procedure *AddClinits* (shown in Figure 5.6) adds a call graph edge from m to all the appropriate `clinit` methods. Though the JVM implicitly

```
input   m : calling method
input   st : statement causing X to be loaded
input   X : new loaded class
output  clinitSet : set of called clinits
procedure AddClinits(m,st,X)
 1: clinitSet := ∅
 2: if containsClinit(X) then
 3:     clinit := getClinit(X)
 4:     clinitSet := clinitSet ∪ {clinit}
 5:     AddCallGraphEdge(m, clinit, st)
 6: end if
 7: if hasSuperType(X) then
 8:     Y = getSuperType(X)
 9:     clinitSet := clinitSet ∪ AddClinits(m, st, Y)
10: end if
11: return clinitSet
```

Figure 5.6: Adding edges to static initializer methods (i.e. `clinits`).

calls `clinit` methods, the algorithm creates an edge from the method m to $X.clinit$ to depict that the actions of m caused the invocation of $X.clinit$. When the static initializer method of a class is invoked, so too is the static initializer method of its super type, if the super type is has not already been initialized. This possibility of additional `clinits` being invoked is addressed in lines 7 and 9. *AddClinits* returns all `clinit` methods that have been added to the graph so that they may be analyzed.

ProcessBody. After *st* as been analyzed for interactions with `clinits`, it is checked for monomorphic calls—i.e. invocations with JVM opcode `InvokeStatic` or `InvokeSpecial` [87]—(Figure 5.5 line 14). If is contains such a call, an edge is added from m to the called method (lines 14–22).

Each monomorphic site is examined to determine if it contains a call to an `AccessController.doPrivileged` method. The `doPrivileged` methods are used for access control operations and decisions. They allow or reject access to critical system resources. The methods in the set *DoPrivileged* all take an argument of type

```
input   m : method
input   rec : class or interface
input   n : method
input   c : call site
output  recSet : set of possible reciever methods
procedure ResolveVirtualCall(m,rec,n,c)
 1: recSet = ∅
 2: subTypes = getSubTypes(rec)
 3: if isInterface(rec) then
 4:     for each X ∈ subTypes do
 5:         recSet := recSet ∪ ResolveVirtualCall(m, X, n, c)
 6:     end for
 7: else
 8:     for each X ∈ subTypes do
 9:         dn := Dispatch(X, n)
10:         recSet := recSet ∪ {dn}
11:         AddCallGraphEdge(m, dn, c)
12:     end for
13:     return recSet
14: end if
```

Figure 5.7: Resolving all possible targets of a virtual call.

`PrivilegedAction` or `PrivilegedExceptionAction`. Objects of these types declare a method `run`. This method contains the code that requires access to the requested system resource. If access to the resource is granted the `run` method is implicitly invoked. Thus, an edge is added from `m` to `run`. Since `run` is a virtual call, dynamic dispatch has to be accounted for as can be seen at line 20.

The algorithm addresses virtual calls in lines 23–30. For each virtual call site (those with JVM opcodes `InvokeVirtual` or `InvokeInterface`) found in the method, *ProcessBody* determines the compile-time target method and the compile-time receiver type of the call. This information is passed to *ResolveVirtualCall* (Figure 5.7). A special case is needed for calls to `void start()` methods belonging to the class `Thread`. When invoked, these methods cause the thread to begin execution. Upon invocation of a `start` method the JVM calls the `run` method of the specified thread.

```
input   X : class
input   n : method
output  z : method
procedure Dispatch(X, n)
 1: if z ∈ X ∧ Signature(z) = Signature(n) then
 2:     return z
 3: else if hasSuperType(X) then
 4:     Y := getSuperType(X)
 5:     Dispatch(Y, n)
 6: end if
 7: return ERROR {should never reach here}
```

Figure 5.8: Virtual dispatch.

The additional call to *ResolveVirtualCall* at line 28 ensures that an edge is added to the implicitly called `run` method.

ResolveVirtualCall. As shown in Figure 5.7, *ResolveVirtualCall* implements a simple Class Hierarchy Analysis. If the compile-time receiver type *rec* is an interface, *ResolveVirtualCall* recursively calls itself for all subtypes of *rec* (line 5). If it is a class, *ResolveVirtualCall* iterates through every subtype X of *rec* (including *rec* if it is a concrete class) and calls *Dispatch* on each (line 9). *Dispatch*, which is shown in Figure 5.8, ascends the class hierarchy starting at X searching for a method which has a signature that matches n's. Once a match is found, it is returned to *ResolveVirtualCall*. An edge is then added for all possible receivers at line 11. It should be noted that the intent of $Signature(z) = Signature(n)$ at line 1 of the *Dispatch* procedure is to check whether the method names, return types, and parameter types lists of z and n match. This simulates how Java 1.4 resolved virtual dispatch. However, in Java 1.5, and subsequent versions, the dispatch model changed slightly. In Java 1.5, an overriding method can also return a subtype of the type returned by the overridden method [52].

5.2.2 Effects of Dynamic Features

The call graph construction algorithm presented above addresses some, but not all, of Java dynamic features. It does make allowances for the actions taken by the JVM upon startup. These accommodations are realized through the use of the sets *JVMStartupMethods* and *JVMStartupClinits*. The first includes methods which the JVM can invoke on startup; these can include `initializeSystemClass()` from the `System` class and `runFinalizer()` from the `Finalizer` class. *JVMStartupClinits* includes the `clinits` for all the classes the JVM automatically loads on startup. These sets are given as inputs to the algorithm.

The algorithm also models several implicit calls made by the JVM, including calls to `run` methods of `Thread` and `PrivilegedAction` objects. Even though these calls are made by the JVM, the above algorithm depicts the calling edge as coming from the application code. This representation is apt in that actions in the client code (e.g., invocations of a `Thread` object's `start` method) caused the JVM call.

Similarly, the algorithm represents edges to `X.clinit` as orienting from the client code that could cause `X` to be loaded into the JVM. This treatment of `clinit`s can lead to a large number of spurious edges in the graph. A class's `clinit` is only called once when the class is loaded into the JVM. However, a call graph generated by the above algorithm could depict numerous calls to the same `clinit`. Some of these multiple invocations will be valid whereas others will be infeasible. Consider the example shown in Figure 5.9 and assume that the only calls to `jumar` and `fifiHook` are at lines 4 and 5, respectively. Also, assume that all fields are static. First notice that method `fifiHook` accesses two static fields of the `Hammer` class. In a call graph generated by the above algorithm, each access would result in a an edge being added from `fifiHook` to `Hammer.clinit` even though the `clinit` would only be invoked on the first access.

```
1   public class AidClimber {
2     public void etrier(){
3       double ps = Piton.strength;
4       if(ps > 3.0){this.jumar();}
5       this.fifiHook();
6       ....
7     }
8     private void jumar(){
9       int pw = Piton.weight;
10      int hw = Hammer.weight;
11      ....
12
13    }
14    private void fifiHook(){
15      boolean pl = Piton.hasLoop;
16      boolean hl = Hammer.hasLoop;
17      int hw = Hammer.weight;
18      ....
19    }
20  }
```

Figure 5.9: Example that demonstrates complexities of when `clinits` are invoked.

Edges to `clinit` methods can be invalid due to interprocedural relationships as well. Notice that all three methods in Figure 5.9 access a static field of the class `Piton`. In the call graph created by the algorithm above, every method of `AidClimber` would have an edge to `Piton.clinit`. However, the edges from `jumar` and `fifiHook` are infeasible since the only paths of execution on which they are invoked pass through `etrier`. Since `Piton.clinit` can be invoked at most once, and it must be invoked prior to `etrier`'s access of `strength`, it will not be invoked again in `jumar` or `fifiHook`. The call graph will also contain edges from `jumar` and `fifiHook` to `Hammer.clinit`. Even though `Hammer.clinit` will only be invoked once, both edges are valid since there are paths of execution where its first invocation can be in either `jumar` or `fifiHook`.

To eliminate some of these infeasible edges to `clinit` methods, the CHA call graph construction algorithm in Soot 2.2.3 employs an intraprocedural dominance analysis. This analysis identifies statement dominance in a method's control-flow graph. Statement `B` is dominated by statement `A` if there is no path of execution that executes `B` without first executing `A`. This analysis is applied individually to each method in the call graph. Edges to `clinit` methods that are found to be infeasible due to dominance relationships are removed from the graph.

The algorithm makes no attempt to address the ramifications of dynamic features in the code. For example, it does not provide a treatment for calls to dynamic class loading methods. Many of these loading methods invoke a class's `clinit` immediately after loading it. Ignoring such calls has the potential to generate an unsound call graph since calls to `clinit`s may not be represented. Similarly, reflective instantiation is not addressed. As described in Section 2.1.2, calls to `Class.newInstance` and `Constructor.newInstance` can create new instances of a class. These calls result in an implicit invocation of a class's constructor. These indirect calls will not be represented in a call graph created by the algorithm above. Further unsoundness will be introduced if a method is indirectly invoked through the use of a reflective object `Method`. Again, these implicit calls are not represented in the call graph.

The algorithm also does not address native methods. This means that the graph will not represent the native code that could be executed. The transition from Java to a native language is an unrealistic expectation for most static analysis tools. One of the biggest challenges with incorporating native methods into an analysis is the lack of source code, much less addressing different language features and semantics. However, by ignoring native methods the CHA algorithm might miss callbacks to the Java code. Recall that native methods have the capabilities of Java methods: they can load new classes into the JVM (resulting in `clinit` methods being invoked),

create instances of classes, and call Java methods. These calling relationships will not be represented by the above algorithm.

Finally, the call graph created by the algorithm may not present a sound abstraction of possible calling relationships if the application under analysis makes use of a custom class loader. The ability of a custom class loader to instrument the code it loads makes it extremely difficult for any static analysis to generate a sound result. In the case of call graphs, a custom loader could load unexpected classes which could result in calls to `clinit` methods not being represented. It could alter or remove exciting method calls or insert new invocations. Such actions would not be represented in the call graph.

5.3 Assumption Hierarchies

Making conservative assumptions about the effects of unknown external code is a common practice for many static analyses (e.g. [19, 63, 108, 113, 115, 129, 141, 150]). Even if not stated explicitly, many static analyses for Java applications make implicit assumptions about the effects of the language's dynamic features. In this section we present two hierarchies of assumptions. The first hierarchy is for the MBL analysis, and the second is for the CHA call graph construction analysis. These assumptions pertain to the dynamic features of Java which can cause the analysis to be unsound. It should be noted that the assumptions presented do not constitute all the possible assumptions such analyses could make.

5.3.1 MBL Assumptions

In Section 5.1 we described the dynamic features which could load a class into the JVM. Presented below are a range of assumptions that would be practical for an MBL

analysis to make about these dynamic features. These assumptions form a hierarchy starting with the most relaxed assumption and building to the most conservative. The result set generated under each assumption is a superset of the result set generated under the preceding assumption.

Overly Optimistic

For a program P, it may be possible to determine all the classes that could be loaded into the JVM by performing a straightforward static analysis of P's source code using the algorithm presented in Figure 5.2. Therefore a static analysis could make the following assumption:

> ***S**tatically **D**iscernable **L**oading **A**ssumption* (**SDLA**): This is an overly optimistic assumption that assumes all classes loaded in the JVM will be loaded by the set of statically discernable loading actions defined above. It also assumes that this set can be determined by starting the analysis at the class which contains the `main` method for P.

Notice that this assumption does not automatically indicate that the analysis will produce unsound results for applications that use dynamic features. It is possible that every class that could be loaded by a dynamic feature could also be loaded by statically determinable means. For example, consider an application that contains the call `Class.forName("X")`. If that same application contains the call `X x = new X()`, then `X` will be in the set returned by the MBL analysis without the need to explicitly address that dynamic class loading method. This, of course, is a very optimistic assumption and will likely be unsound for the majority of Java applications.

JVM Startup

The MBL algorithm presented in Figure 5.2 starts its analysis at the class containing the `main` method for an application. However, many implementations of the

JVM will pre-load classes before executing `main`. It is unlikely that every application will explicitly make use of these pre-loaded classes. Thus, a slightly more conservative assumption than SDLA would be:

> ***J**VM **S**tartup **L**oading **A**ssumption* (**JSLA**): This assumption recognizes that an implementation of the JVM may pre-load classes. However, it assumes all other classes loaded into the JVM can be determined statically.

This assumption can be accommodated with a simple addition to the algorithm presented in Figure 5.2. The new version not only initializes the worklist with the class containing `main` as before but also includes a set of classes *JvmSt*. This set represents the classes that the JVM loads on startup.

There are several techniques that could be used to determine the classes that need to be included in *JvmSt*. These techniques require slight refinements to JSLA. One of these refinements is:

> ***JSLA**-**G**lobal **F**it* (**JSLA-GF**): This refinement of JSLA assumes a static set of classes will encompasses all possible classes that will be loaded by any implementation of JVM on startup. If a particular implementation of the JVM pre-loads a class not in *JvmSt* than the analysis assumes that the application will statically load it.

This technique uses a one-size-fits-all solution. This is the approach used by the popular Soot framework. The analysis uses a hard-coded set of classes that it assumes all implementation of the JVM will load upon startup. This solution will neither be sound nor precise for all systems and implementations of the JVM. However, it will likely be more sound than simply ignoring the fact that the JVM loads classes on startup.

Another approach would require the MBL analysis to incorporate a pre-processing stage. This stage would examine the JVM which will be executing the MBL analysis. This examination would determine the set of classes pre-loaded by that particular

instance of the JVM. This technique can be combined with the global set technique under the following assumption:

> ***JSLA-Semi-Static*** (**JSLA-SS**): This assumption extends JSLA-GF. It assumes that the set of startup classes loaded by the JVM currently executing the MBL analysis combined with a static set of classes (as defined by JSLA-GF) represents all classes that could possibly be loaded by the JVM upon startup.

This technique is similar to the use of semi-static sources of string variables described in Chapter 3. Using this technique will not guarantee global soundness of the results generated by a MBL analysis. Instead, it will tailor *JvmSt* to the system and the JVM currently under analysis. This approach is made less system-specific by combining its results with the statically defined set of JSLA-GF.

There are several approaches that could be used to determine the set of classes a particular JVM loads on startup. It is possible to run the JVM in *verbose* mode. In this mode, the JVM will notify the user of every class it loads. By executing an application that consists of an empty `main` method (i.e., one that does not explicitly load any classes) on a JVM in verbose mode, a list of startup classes can be determined. Another method used to estimate the set JVM startup classes and method invocations is to use a JVM profiling tool such as *JVMPI* [70]. JVMPI can determine the classes the JVM loads and which methods are invoked during the execution of an application.

It is interesting to note that we used both a JVMPI agent and the JVM verbose option when executing a version of the JVM on an application consisting of an empty `main`. The set of classes generated by techniques varied slightly. To be safe, an MBL analysis could use the union of the two result sets. The set of startup classes defined in Soot 2.2.3 (the version used in the experiments presented later) was a subset of both result sets.

```
1     Gene thisGeneObject;
2     try {
3         thisGeneObject =(Gene) Class.forName(geneClassName).newInstance();
4     }catch (Exception e) { .... }
```

Figure 5.10: An example of dynamic class loading from the JGap benchmark.

All Referenced Types

As shown in Chapter 3, the Java libraries make use of dynamic class loading. These same dynamic features are available to all Java applications. The classes loaded by these features will not be included in the set *MBLCLs* by the algorithm from Figure 5.2.

For example, consider the code shown in Figure 5.10. Notice that line 3 loads a class specified by the string `geneClassName`, creates an instance of that class using `newInstance`, and casts the resulting object to the type `Gene`. If `Gene` is not loaded in a statically determinable manner elsewhere in JGap, it will not be included in *MBLCLs*.

One assumption an MBL analysis could make that would catch such instances is:

> ***A**ll **R**eferenced **T**ypes **A**ssumption* (**ARTA**): This assumption extends JSLA-SS by assuming that all types referenced in a class will be loaded.

An implementation that operates under ARTA could use the type information in the `constant_pool` of a class[10]. By considering all type references in the `constant_pool`, ARTA will include type information from cast operations, parameter declarations, field declarations, etc. Some of these additional types may represent classes that are loaded via dynamic class loading methods, such as `Gene` above.

[10]Soot uses a similar technique which considers all type information available in a class file.

```
1    Method methodGetComponentOrientation =
2           Class.forName("java.awt.Container").getMethod
3                    ("getComponentOrientation", new Class[0]);
```

Figure 5.11: An example of a call to a dynamic class loading method from the JGap benchmark.

Using String Values

There are several instances when a class may contain a dynamic class loading site but its `constant_pool` does not contain a reference to the dynamically loaded class. For example, consider the code in Figure 5.10 again. Assume the class that is loaded is a subtype of `Gene`. Since the subtype is not directly referenced in the class, its type information will not be contained in the `constant_pool`.

A second example of when the `constant_pool` may not contain type information about dynamically loaded classes can be seen in Figure 5.11. In this example, `java.awt.Container` is dynamically loaded and a `Method` object representing its `getComponentOrientation` method is retrieved. At no point in time is a cast of the loaded class performed. Therefore there is no type information to find in the `constant_pool`.

By employing a static string analysis to resolve dynamic class loading sites as described in Chapter 3, an MBL analysis may be able to provide treatment for cases where type information is not available in the `constant_pool`.

The following assumption allows for the incorporation of an intraprocedural string analysis to aid in the resolution of dynamic class loading sites:

> ***S**tatically **R**esolvable **P**arameter **A**ssumption* (**SRPA**): This assumptions extends ARTA by assuming that invocations of dynamic class loading methods will not throw a `ClassNotFoundException`. It further assumes, that any string values which flow from method parameters, method returns, and fields can be any string value.

The assumption that dynamic class loading methods will not produce a run-time exception ensures that a string value that can be resolved specifies a legitimate class. The second part of the assumption limits the string analysis to intraprocedural scope. This is necessary since without being based on a sound MBL analysis, interprocedural analyses cannot guarantee soundness.

Truly Dynamic Loading

As stated previously, in the general case it is impossible to determine which classes can be loaded. Moreover, because of custom class loaders and native methods, this assumption has to be extended to applications which do not make use of dynamic class loading features explicitly. The only assumption that could be applied soundly to every application is:

> ***F**ully **C**onservative **L**oading **A**ssumption* (**FCLA**): This is a fully conservative assumption that assumes that any class in the world could be loaded by an application due to Java features which dynamically load classes.

It is, of course, impossible to know every class in the world. A reasonable approximation is to limit the "world" to the classpath of the system and applications under analysis.

5.3.2 CHA Assumptions

In Section 5.2.2 we described the dynamic features of Java that could cause a simple CHA call graph construction algorithm to be unsound. Presented below is a range of assumptions that would be practical for a CHA analysis to make about these dynamic features. It is important to note that all of these assumptions are predicated on the assumption that the input *AppClasses* contains all possible classes that could be loaded during any execution of P.

The assumptions form a hierarchy. Unlike the hierarchy presented for MBL, CHA's is rooted at the most conservative assumption and becomes progressively more relaxed. The assumption at each level in the hierarchy contains the assumptions made by the preceding level. Consequently, the graphs generated under by one assumption are subgraphs of those generated under the preceding assumption.

Well-Behaved Loaders

Custom class loaders can dynamically load classes from locations not on a system's classpath, load classes other than those specified, and alter the code of any class. Thus, the only conservative assumption that can be made in the presence of such a loader is that every method could call every other method in the world. Luckily, most applications will use class loaders that do not alter the semantics of the classes that they load. A more reasonable assumption about loaders would be:

> ***W**ell-**B**ehaved **C**lass **L**oaders* (**WBCL**): It is assumed that all classes that are loaded can uniquely be identified in a single name-space, and the loaders will not alter the semantics of the classes they load.

This assumption does not address other dynamic features of Java. To be fully conservative under this assumption, certain implicit assumptions are being made. Specifically, given a use of a dynamic feature `dyno`, the following assumptions are made depending on the feature represented by `dyno`:

- **Dynamic class loading features**: `dyno` can invoke any `clinit` in *AppClasses*.
- **Reflective instantiation—Class.newInstance**: `dyno` can call any default constructor (one that takes no arguments) in *AppClasses*.
- **Reflective instantiation—Constructor.newInstance**: `dyno` can call any constructor in *AppClasses*.
- **Reflective invocation—Method.invoke**: `dyno` can call any concrete method in *AppClasses* excluding `clinit`s and constructors.

- **Call to a native method**: An edge is added to `NativeMethod`, a synthetic method. This method contains an edge to every `clinit`, constructor, and method in *AppClasses*. This represents potential callbacks from native methods.

Respectful Reflection

Under **WBCL** it is assumed that reflection will break encapsulation. Some uses of reflection will break encapsulation, as is often the case in frameworks that use reflection to conduct unit testing. However, for many applications reflection will respect encapsulation. For such applications the following assumption is appropriate:

> ***E**ncapsulation **R**especting **D**ynamic **F**eatures* (**ERDF**): This assumption extends WBCL. In addition to well-behaved class loaders it assumes that all dynamic features will respect encapsulation.

In Java access level modifiers determine the visibility of program components. There are two levels of access control: (1) Top level—non-nested classes can be declared `public` or *package-private* (the default when no modifier is specified) and (2) Member level—fields, methods, and nested classes can be declared `public`, `protected`, `private` and *package-private* (no modifier).

The `public` modifier specifies that the entity is accessible to all classes in the world. *Package-private* entities are only accessible in their own package. The modifier `protected` designates that members are accessible in their own package and by a subclass of its class in another package. A member marked as `private` can only be accessed within its own class.

Top level modifiers trump member level ones. For example, if class `C` is declared as *package-private* but contains a member method `m` which is declared to be `public`, `m` will still only be accessible from its own package.

Under this assumption great care must be taken with the nested classes. In Java it is possible to declare a class in the body of another; such classes are called nested classes [87]. There are four categories of nested classes:

- Static member classes
- Member classes
- Local classes
- Anonymous classes

A *static member class* has the same properties as an static member of the enclosing (i.e., outer) class. It can access all other static members of its top-level classes (recursive nesting is allowed), even those marked as `private`. It is also possible for a static member class to declare an instance of an outer class to gain access to all instance members. A *member class* is very similar to a static member class except that it is instance specific and can therefore access all members of an outer class without creating an instance of it.

As a class member, both static member classes and member classes can be given a visibility modifier (i.e., `private`, `protected`, `public` or *package-private*). Just as nested classes can access all the members of outer classes, outer classes have similar access to all members declared in nested classes (even recursively nested classes). The one crucial difference is, nested classes can access inherited `protected` methods of their encapsulating classes. However, an outer class cannot access the inherited `protected` methods of nested classes[11].

Local classes and *anonymous classes* are declared in a block of code, typically a method. They are only accessible within the surrounding block just like any other

[11]Assuming that the superclass of the nested class resides in a different package than the outer class.

local variable. The difference between the two is, an anonymous class is declared without a name and is only accessible at the point of its declaration. Just like local variables, local and anonymous classes cannot be assigned a modifier.

Consider an invocation of a dynamic feature `dyno` in a method `m` contained in a class `C` which resides in a package `p`. Depending on the feature `dyno` represents, under ERDF the following assumptions are made :

- **Dynamic class loading features**: Since the concept of encapsulation really does not apply to `clinit` methods[12], it is assumed `dyno` can still invoke any `clinit` in *AppClasses*.
- **Reflective instantiation—Class.newInstance**: `dyno` can call any `public` default constructor, any *package-private* default constructor in `p`, any `private` default constructor declared in `C` (including all declared in nested static member classes). If `C` is a nested class, then `dyno` could invoke all `protected` and `private` default constructors of the outer classes of `C` and any default constructor of local classes declared in `m` (anonymous classes do not contain named constructors).
- **Reflective instantiation—Constructor.newInstance**: `dyno` can call any `public` constructor, any *package-private* constructor in `p`, any `private` constructor declared in `C` (including all declared in nested static member classes). If `C` is a nested class, then `dyno` could invoke all `protected` and `private` constructors of the outer classes of `C` and any constructor of local classes declared in `m`.
- **Reflective invocation—Method.invoke**: `dyno` can call any `public` method, any *package-private* method in `p`, any `private` method declared in `C` (including all declared in nested member classes). If `C` is a nested class, then `dyno` could invoke all `protected` and `private` methods of the outer classes of `C` (including those visible through inheritance) and any method of local classes declared in `m`.
- **Call to a native method**: There are several different assumption that could be made about native methods. We choose to think of native members as external code entities and therefore assume that they will only access `public` methods. Thus `dyno` could invoke all `clinit`s, and all `public` constructors and methods.

[12]The JVM invokes `clinit` methods, not the application code, and the concept of encapsulation does not apply to calls made by the JVM.

The above assumed actions for `dyno` are essentially the same actions that an analogous conventional object-oriented code could take. Since these actions respect encapsulation we call them *encapsulation safe*.

Correct Casting

As discussed earlier, a common use of dynamic class loading methods is to create an instance of the loaded class using reflective instantiation and casting the newly created object to the appropriate type. This casting information could be leveraged under the following assumption:

> ***C**orrect **C**asting **I**nformation **A**ssumption* (**CCIA**): This extension of ERDF assumes that casts of reflectively instantiated objects will not cause a run-time exception[13].

This assumption makes it possible to resolve dynamic class loading sites with the following characteristics:

1. The statement *DynoSite*, which contains the dynamic class loading site, is of the form `x = Class.forName(...)` (or a similar call to another dynamic class loading method) and *DynoSite* is not a reaching definition of `x`.
2. A statement, *NewSite*, of the form `o = x.newInstance()` where `x` is of type `Class` post-dominates *DynoSite*. In other words, there are no paths of execution starting at *DynoSite* that do not pass through *NewSite*. The only reaching definitions of `x` at *NewSite* is *DynoSite*, and *NewSite* is not a reaching definition of `o`.
3. *NewSite* is post-dominated by a casting statement of the form `q = (CastType)o` and the only reaching definition of `o` is *NewSite*.

Since *DynoSite* is post-dominated by *NewSite*, and it is the only reaching definition of `x` at *NewSite*, and *DynoSite* is not a reaching definition of `x` at *DynoSite* (this can

[13]Livshits et al. [89] make a similar assumption so that they can resolve instances of dynamic class loading in their points-to analysis.

happen in loops), it must hold that every execution of *DynoSite* is followed by an execution of *NewSite*. It is also true that the class being instantiated by *NewSite* is the same one that was load by *DynoSite*. A similar relationship is true for *NewSite* and the casting operation that post-dominates it.

This set of characteristics is easily extended to dynamic class loading sites which are post-dominated by a *NewSite* of the form `o = con.newInstance()` where `con` is of type `Constructor`. For such instances the following relationships must hold: (1) *DynoSite* must be post-dominated by a statement *ConSite* of the form `con = x.getConstructor(...)`, (2) the only reaching definition of `x` at *ConSite* is *DynoSite*, (3) *DynoSite* is not a reaching definition of `x` at *DynoSite*, (4) *ConSite* is not a reaching definition for `con` at *ConSite*, (5) *ConSite* must be post-dominated by *NewSite*, and (6) the only reaching definition for `con` at *NewSite* is *ConSite*.

For dynamic class loading sites that meet these requirements, it can be inferred that the loaded classes must be of type `CastType` or a subtype of `CastType`. We call this set of types (`CastType` and its subtypes) the *resolving class set*, and we call the dynamic class loading sites *cast resolvable*.

Notice that the *NewSite* statements above are also resolved. If *NewSite* is of the form `x.newInstance()` where `x` is of type `Class`, then it is assumed that *NewSite* can implicitly invoke any default constructor in the resolving class set (if a class does not declare a default constructor, its superclasses are searched until one is found). If `x` is of type `Constructor` it is assumed *NewSite* could invoke any constructor declared by a class, or a superclass of a class, in the resolving set.

The type information from resolved dynamic class loading sites can be used to resolve reflective invocations. Consider a statement of the form `m = c.getMethod(...)` and assume that the only reaching definitions of `c` are a set of resolved dynamic class loading sites. Each of these resolved dynamic class loading sites has a set of resolved

classes associated with it; we call the union of these sets *ResolvedDyno*. The method represented by `m` must be declared in a class, or a superclass of a class, contained in *ResolvedDyno*. Carrying this reasoning forward, if for a statement of the form `x = m.invoke(...)` all the reaching definitions of `m` come from resolved instances of `getMethod`, then it is possible, using similar logic, to determine the set of methods that could possibly be invoked; we call this set of methods the *resolving method set*.

With the capability to resolve certain dynamic features, the CHA call graph construction algorithm's treatment of these features becomes more precise. Consider an invocation of a dynamic feature `dyno` in a method `m` contained in a class `C` which resides in a package `p`. Depending on the feature `dyno` represents, under CCIA the following assumption are made:

- **Dynamic class loading features**: If `dyno` is cast resolvable, it is assumed that it could invoke any `clinit` in its resolving class set. Otherwise, it is assumed that `dyno` can invoke any `clinit` in *AppClasses* (as defined in ERDF).
- **Reflective instantiation—Class.newInstance**: If `dyno` is cast resolvable, it is assumed that `dyno` could invoke any default constructor in its resolving class set. If `dyno` is not resolvable then it is assumed that it has the same access as under the ERDF assumption.
- **Reflective instantiation—Constructor.newInstance**: If `dyno` is cast resolvable, it is assumed that `dyno` could invoke any constructor declared by a class, or a superclass of a class, in the resolving class set. Otherwise, it is assumed that it has the same access as under the ERDF assumption.
- **Reflective invocation—Method.invoke**: If `dyno` is cast resolvable, it is assumed that `dyno` can call any encapsulation safe method in its resolving method set (e.g., `dyno` can call all `public` methods in the set). If `dyno` is not resolvable then it is assumed that it has the same access as under the ERDF assumption.
- **Call to a native method**: The treatment for `dyno` is the same as under ERDF.

Correct String Values

The use of casting information will not be able to resolve all instances of dynamic class loading and reflective instantiation since not all uses of these features will be post-dominated by a cast. Even if such features are resolved, the resulting resolving class set could be quite large if the casting type has many subtypes. It may be possible to resolve more instances of dynamic class loading and reduce the size of some resolving class sets by making the following assumption:

> ***C**orrect **S**tring **I**nformation **A**ssumption* (**CSIA**): This extension of CCIA assumes that (1) features such as reflection will not affect the string-typed formal parameters of `private` and *package-private* visible methods and `private` fields whose values flow to dynamic class loading sites and (2) dynamic class loading sites will not throw a `ClassNotFoundException`.

This assumption contains all the assumptions made by CCIA (and transitively those made by ERDF and WBCL) which includes the assumption that reflection will respect encapsulation. CSIA further assumes that reflection contained in a package or class will not affect a very limited number of encapsulation safe method parameters and fields—specifically, string-typed formal parameters of `private` and *package-private* visible methods and `private` fields whose values flow to dynamic class loading sites. It also assumes that all string values which flow to such sites designate a valid class which could be loaded.

The CSIA assumption allows the CHA call graph construction algorithm to incorporate information from our version of JSA presented in Chapter 3 (modulo the use of semi-static environment variables). This version of JSA is very conservative. It considers the *AppClasses* generated by the FCLA version of the MBL analysis (i.e., all classes on the application and system classpaths). It assumes that all methods in *AppClasses* are reachable, and uses CHA to resolve only virtual calls that may affect the values of string variables. It further assumes that the only entities not affected by

dynamic features are local string variables, string formal parameters of `private` and *package-private* methods, `private` fields and string variables returned by `private` and *package-private* methods, whose values flow to dynamic class loading sites. Since the assumptions JSA operates under are more conservative than CSIA, a call graph operating under CSIA can use the information for JSA without a loss of soundness.

The results of JSA can be integrated into the call graph algorithm to aid in the resolution of dynamic class loading sites. For example, consider a statement *loadSite* of the form `x = Class.forName(s)` where `s` is of type `String`. Assume that JSA is able to determine a finite set *classNames* of run-time values for `s`. Due to the imprecision of JSA not every string value in *classNames* may represent a fully-qualified class name that is feasible for *loadSite*. Under the assumptions made by CSIA it is possible to remove some of these infeasible values. Since CSIA is based on a closed-world assumption (as all our assumptions are), strings in *classNames* that do not specify the fully-qualified names of classes contained in *AppClasses* can be discarded. Since CSIA extends CCIA, it assumes that type information from cast operations are correct. Therefore, if *loadSite* is cast resolvable, then strings in *classNames* that do not specify the fully-qualified names of classes contained in the cast resolved class set of *loadSite* can also be discarded. After infeasible values have been removed, it can be inferred that the *loadSite* can load any class whose name is in *classNames*. We call this set of classes the *string resolved class set* and we refer to *loadSite* as *string resolvable*.

Section 3.2 specifies how the resolution of reflective instantiation can be achieved using string resolved dynamic class loading sites. Reflective invocation sites are resolved in the same manner as under CCIA.

Consider an invocation of a dynamic feature `dyno` in a method `m` contained in a class `C` which resides in a package `p`. Depending on the feature `dyno` represents, under CSIA the following assumption are made:

- **Dynamic class loading features**: If `dyno` is string resolvable, it is assumed that it could invoke any `clinit` in its string resolving class set. Otherwise, it is treated as specified by CCIA.
- **Reflective instantiation—Class.newInstance**: If `dyno` is string resolvable, it is assumed that `dyno` could invoke any default constructor in its string resolving class set. Otherwise, it is treated as specified by CCIA.
- **Reflective instantiation—Constructor.newInstance**: If `dyno` is cast resolvable, it is assumed that `dyno` could invoke any constructor declared by a class, or a superclass of a class, in its string resolving class set. Otherwise, it is treated as specified by CCIA.
- **Reflective invocation—Method.invoke**: If `dyno` is cast resolvable, it is assumed that `dyno` can call any encapsulation safe method in its string resolving method set (e.g., `dyno` can call all `public` methods in the set). Otherwise, it is treated as specified by CCIA.
- **Call to a native method**: The treatment for `dyno` is the same as under ERDF.

Dynamically Gathered Environment Information

As shown in Chapter 3, the incorporation of dynamically gathered environment information can increase JSA's ability to resolve dynamic class loading and reflective instantiation sites in the Java 1.4 standard libraries. The following assumption allows these same semi-static values to be used to increase the precision of the CHA algorithm:

> ***S**emi-**S**tatic **E**nvironment **A**ssumption* (**SSEA**): This extension of CSIA assumes that (1) dynamic features will not affect the string-typed formal parameters of `private` and *package-private* visible methods and `private` fields whose values flow to environment variable access methods and (2) the values of environment variables which can effect dynamic class loading site will be the same at analysis time and at run time.

This assumption allows the incorporation of information from our semi-static version of JSA presented in Chapter 3. Just like the static version of JSA and CSIA, the semi-static version of JSA operates under a much more conservative assumption than SSEA so its information can be used by a call graph analysis operating under SSEA without a loss of soundness. As discussed in Section 3.3, making this assumption about environment variables means that the generated call graph will be tailored to the system under analysis. Thus, it will no longer be sound in a global context. It will only be sound for systems where the environment variables are the same as those observed by the call graph analysis.

Under this assumption there is no change needed to the algorithm's treatment of dynamic features. It only enables the string analysis to consider more sources of string values, increasing the number of dynamic class loading sites that can be precisely resolved.

Client Resolved

In the general case it is impossible to statically resolve any of Java's dynamic features. Several static analyses rely on their clients to specify the actions of such features. This shifts the onus of soundness from the analysis to its user. This shift of responsibility is captured under the following assumption:

> ***C**lient **R**esolved **F**eatures **A**ssumption* (**CRFA**): This extension of SSEA assumes that if the algorithm is not able to resolve a dynamic feature using casting information, or string values (including semi-static values), the user will specify the correct treatment for that instance.

Under this assumption every time the call graph algorithm encounters a dynamic feature it will first attempt to resolve it using the same techniques as SSEA. However, if CRFA fails to resolve the instance it will query the user for the set of methods which

that feature could be implicitly invoke. It will then add the appropriate edges to the call graph.

5.4 Experimental Evaluation

To evaluate how the assumptions outlined in Section 5.3 will affect the results of the MBL and CHA analyses, we performed an empirical study. The results of the study are presented in this section. We first describe the implementations of both the MBL analysis and the CHA call graph construction algorithm used in the study. Then, a brief description the benchmark applications and the experimental setup is provided. The sections concludes with a discussion of the results of the study.

5.4.1 Implementations

MBL Analysis

Experiments were conducted on 7 implementations of the MBL analysis (see Section 5.1). The implementations that were used are:

- **SDLA.** This is an exact implementation of the MBL algorithm presented in Section 5.1. It does not explicitly address classes that are loaded by either the JVM at startup, or classes loaded through calls to dynamic class loading methods.

- **JSLA-GF.** This implementations operates under the assumption that a static set of classes will represent all the classes loaded by the JVM. This particular implementation uses the same static set of classes as defined in Soot. This is a set of 49 commonly used classes. The MBL implementation adds these classes to the initialization of the worklist, ensuring they will be processed.

- **JSLA-SS.** This implementation is a slight modification of JSLA-GF. In addition to the static set of classes used by JSLA-GF, it incorporates a pre-processing phase which monitors the current JVM's startup. It applies both a JVMPI agent and the JVM `-verbose` option to a Java 1.4.2 HotSpot(TM) Client VM executing a program consisting of an empty `main` to determine the classes that are loaded by the JVM. The set of observed classes are added to the static set of startup classes from JSLA-GF for a total of 294 classes.

- **ARTA.** This implementation operates under the assumption that all referenced types in a class will be loaded. It extends JSLA-SS, but when it removes a class from the worklist it accesses the `constant_pool` of the class. All type information found in the `constant_pool` is then added to the worklist.

- **SRPA.** This implementation operates under the assumption that an intraprocedural string analysis will be able to determine classes that are loaded by dynamic class loading sites. SRPA extends ARTA by including a intraprocedural version of the JSA string analysis. This version of JSA corrupts all non-local string values when analyzing a method; otherwise it is identical to the original version. If JSA is able to resolve a dynamic class loading site, the set of resolved classes are added to the worklist. Note that due to the imprecision of JSA, not every string value it returns may specify a class on the classpath. These values are ignored.

- **FCLA.** This version implements the fully conservative assumption. It simply returns the set of classes that are available on the system and application classpaths.

- **SOOT.** This is the MBL analysis used by the CHA call graph construction algorithm in Soot 2.2.3. This implementation is approximately ARTA without the semi-static approach to the JVM startup classes.

CHA Analysis

Two categories of CHA call graph construction algorithms were used in this study. The first category consists of 6 implementations, each of which operates under a different assumption presented in Section 5.3.2. These implementations use the SRPA version of the MBL analysis to calculate the set of *AppClasses* they analyze. Recall that the CHA algorithms requires the user to define two sets: *JVMStartupClinits* and *JVMStartupMethods* (see Section 5.2). The set *JVMStartupClinits* used by these implementations consists of all `clinit` methods which are invoked due to the loading of the JVM startup classes as defined by the JSLA-SS MBL analysis. *JVMStartupMethods* consists of method invocations that were discovered by applying a JVMPI agent to a Java 1.4.2 HotSpot(TM) Client VM executing a program consisting of an empty `main` method. There were 476 methods discovered.

The 6 implementations of this category are:

- **WBCL.** This implementation operates under the assumption that custom class loaders will not alter the semantics of the classes they load. It is built on top of Soot's CHA call graph construction algorithm. When a dynamic feature is identified this version adds the appropriate edges as specified in Section 5.3.2.

- **ERDF.** This implementation operates under the assumption that reflection will not break encapsulation. It provides the encapsulation safe treatment for dynamic features as specified in Section 5.3.2.

- **CCIA.** This implementation extends ERDF by incorporating a post-dominance analysis and a reaching definitions analysis. The information from these analyses is used to resolve instances of calls to dynamic class loading methods, `newInstance` methods, and `Method.invoke`. The conditions for resolution are described in Section 5.3.2.

- **CSIA.** This version incorporates JSA to aid in the resolution of dynamic class loading methods. This version does not make use of any semi-static values. **Note**: The version of JSA used by this implementation incorporates all the extensions of string analysis described in Chapter 3, save the use of semi-static values. It also incorporates all the graph simplification techniques described in Section 4.2 and the parallel design described in Section 4.1.3.

- **SSEA.** This implementation is the same as CSIA, except that the version of JSA used by this implementation incorporates dynamically gathered environment information as described in Chapter 3.

- **CRFA.** This implementation is the same as SSEA except it ignores all unresolved dynamic features. It operates under the assumption that the user will specify the actions that should be taken at such points. This version generates the same result as if the user specified that no action should be taken for unresolved features.

The second category of implementations used in this study consist of two versions of the Soot 2.2.3 CHA call graph construction algorithm. These implementations consider the *AppClasses* set generated by the SOOT version of the MBL analysis. The set *JVMStartupClinits* used by these implementations consists of all `clinit` methods which are invoked due to the loading of the JVM startup classes (this is the same set of 49 commonly used classes statically defined in the JSLA-GF version of

the MBL analysis). The set of startup classes is statically defined and hard coded into the algorithm. The set *JVMStartupMethods* is also statically defined and hard coded into the algorithm. This set contained 13 methods.

The 2 implementations of this category are:

- **SCON.** This version represents Soot's CHA call graph construction algorithm in its most conservative setting. In this setting, Soot provides treatment for dynamic class loading calls of the form `Class.forName(Sting)`, and `newInstance` calls of the form `Class.newInstance()`. It resolves `Class.forName(lit)` calls where `lit` is a string literal value. If it is unable to resolve a call to `forName`, it will add an edge to all `clinit` methods in *AppClasses*. For `Class.newInstance` calls, it adds an edge to every default constructor in *AppClasses*. It does not provide any treatment for calls to native methods, reflective invocations, calls of the form `Constructor.newInstance(...)`, or dynamic class loading methods other than `Class.forName`.

- **SOOT.** This is Soot executed in its default settings. By default, Soot only provides treatment for calls of the form `Class.forName(lit)` where `lit` is string literal value. It adds an edge to the `clinit` of the class specified by `lit`. All other dynamic features are ignored. This version is most comparable to our CRFA implementation.

5.4.2 Benchmarks and Experimental Setup

Table 5.1 presents the 10 benchmark applications that were used in this study. They are a subset of the applications used in the empirical study presented in Section 4.3. For each benchmark **Classes** shows the number of class files that are unique

App	Classes	Meths	K Jimple	Dyno	NewInst	Invoke	Native
DB	15	175	3119	0	0	0	0
Javac	188	1320	26574	0	0	0	0
JEdit	851	6206	124830	313	6	16	0
JGap	174	1035	15331	25	8	4	0
Jpws	193	1616	28425	5	0	0	0
Mindterm	135	1072	30626	5	5	0	0
Muffin	278	2258	37748	11	4	0	0
Sablecc	267	2248	36155	2	0	0	0
VietPad	215	914	24998	22	5	8	3
Violet	130	636	9959	2	4	2	0

Table 5.1: Benchmarks statistics: number of classes, methods, Jimple statements, invocations of dynamic class loading methods, `newInstance` methods, `Method.invoke`, and native methods.

to the application; this number does not include library classes that may be referenced by the application. Column **Meths** presents the number of methods (including constructors and `clinit`s) contained in the application classes. Column **K Jimple** displays the number of Jimple statements contained in the application's class files. **Dyno** presents the number of invocations of dynamic class loading methods (see Figure 3.4 for the list of methods) present in the Jimple representation of the application's classes. Similarly, **NewInst** and **Invoke** display the number of invocations of `newInstance` (both `Class` and `Constructor`) and `Method.invoke`, respectively. Column **Native** displays the number of native methods declared in the application.

The benchmarks DB was selected because it is small and contains no calls to any dynamic features. All dynamic features encountered during an analyses of it must be contained in the library classes. Javac, JEdit, Jpws, Mindterm, Muffin, Sablecc and Vietpad represent the 7 largest applications used in experiments presented in Section 4.3 (they all contain over 20,000K Jimple statements). JGap and Violet were included because they contained interesting uses of reflection and dynamic class loading. The JEdit benchmark contains the most invocations of dynamic features,

including 313 calls to dynamic class loading methods. This number may be misleading. It is largely due to JEdit's use of *class literal expressions* [51]. A class literal expression consists of the name of a type followed by a `"."` followed by the keyword `class`. It is used to gain access to the `Class` object representing the specified type. For example, `X.class` returns the `Class` object which represent `X`. Some compilers replace class literal expressions with calls of the form `Class.forName(s)` where `s` is a string literal representation of `X`'s fully qualified name. Thus, JEdit has many more invocations of `forName` in its Jimple representation (which is derived from bytecode) than in its Java source representation. JEdit contains 32 invocations of dynamic class loading methods in its Java source representation.

All of the benchmarks were compiled using a compiler that was compliant with the specifications for Java 1.4 and the Java 1.4 standard libraries. For experiments conducted on any application *bench*, the classpath was set to contain only the classes of *bench* and the Java 1.4 standard libraries. The environment variables consider by the semi-static string analysis were from a Microsoft Windows XP (Service Pack 3) OS.

5.4.3 MBL Results

Table 5.2 presents the results of our empirical study of the MBL analysis. The numbers in the table represent the number of classes each implementation of MBL determined could be loaded into the JVM for the given benchmark application. Column **SDLA** presents the results for the implementation of MBL which operates under the most relaxed assumption (i.e., all classes will be loaded in a statically determinable manner). We use this version as our baseline. The results it generates are a subset of the results returned by the other implementation. Row **AVEΔ** displays the average percentage increase in result set size over SDLA.

Apps	SDLA	JSLA-GF	JSLA-SS	ARTA	SRPA	FCLA	SOOT
DB	2278	2293	2356	3429	3538	10253	3389
Javac	2447	2461	2524	3597	3706	10426	3557
JEdit	3576	3590	3653	4225	4340	11089	4185
JGap	2598	2613	2676	3511	3620	10412	3471
Jpws	2927	2942	2989	3664	3773	10431	3641
Mindterm	2323	2338	2401	3485	3641	10373	3445
Muffin	2454	2469	2532	3592	3701	10516	3545
SableCC	1312	1328	1394	1737	1746	10505	1694
VietPad	2857	2872	2919	3641	3759	10453	3618
Violet	2665	2680	2743	3584	3693	10368	3544
AVEΔ	–	0.6%	3.1%	26.4%	28.4%	75.8%	25.5%

Table 5.2: MBL analysis results: number of classes that will be loaded. **AVEΔ** is the average percentage increase with respect to column **SDLA**.

The addition of the static set of JVM startup classes only slightly increases the size of the result set, as can be seen in column **JSLA-GF**. The 49 startup classes increased the result set by about 0.6% on average. This limited growth can be attributed to the fact that most of the benchmark applications will load these classes in a statically determinable manner. The addition of the semi-static JVM startup classes increased the set of startup classes to 294 classes. However, as can be seen in column **JSLA-SS**, the addition of these classes only added on average about 3% more classes to the baseline's result set.

A sizeable average increase of 26% is realized when all the type information available in the `constant_pool` is incorporated into the MBL analysis, as can be seen in column **ARTA**. This increase is due to the fact that the `constant_pool` may contain references to types that are not actually loaded. It may also contain references to types that are loaded dynamically through calls like `Class.forName(...)`. A large number of these dynamically loaded classes may not be loaded in a statically determinable manner elsewhere in the code.

As shown in column **SRPA**, the addition of the intraprocedural string analysis increased ARTA's result set by slightly over 100 classes for most benchmarks (SableCC being the one exception). The additional classes arise directly or transitively from instances of dynamic class loading calls for which the `constant_pool` contains no type information. The fully conservative solution, which assumes that all classes on the classpath can be loaded, is significantly larger than the other solutions.

Soot's implementation of MBL most closely resembles ARTA. ARTA produces a slightly large set due to the inclusion of the semi-static JVM startup classes. The result set generated by SOOT is missing between 50 and 200 classes for each benchmark when compared to SRPA, our second most conservative implementation. The majority of these missing classes represent classes that SRPA was able to resolve using a string analysis. If these resolved dynamic class loading sites are executed, these additional classes will be loaded, implying SOOT is unsound for these cases.

5.4.4 CHA Results

Table 5.3 presents the number of nodes and edges in the call graphs created by the implementations of the call graph analysis. Column **WBCL** presents the results of the most conservative implementation. This version is used as the baseline to which the other implementations are compared. The graph generated by WBCL for a particular application is a supergraph of all other implementations' graphs for that same application. Row **AVE**Δ_N displays the average reduction in the number of nodes in the graphs generated by the corresponding implementation, compared to WBCL. Row **AVE**Δ_E contains similar information but with respect to the number of edges.

	Number of Nodes							
Apps	**WBCL**	**ERDF**	**CCIA**	**CSIA**	**SSEA**	**CRFA**	**SCON**	**SOOT**
DB	32537	29060	29046	29045	29044	19242	18738	17705
Javac	33662	30180	30167	30166	30165	20356	19850	18818
JEdit	39083	35640	35625	35624	35624	25855	25043	24048
JGap	32890	29404	29391	29390	29388	19763	19251	18232
Jpws	34663	31162	31148	31148	31145	21558	21214	20256
Mindterm	33268	29779	29764	29762	29761	19601	19370	18343
Muffin	33976	30416	30400	30400	30399	20305	19770	18728
SableCC	33807	30364	30349	30348	30347	20833	9341	8761
VietPad	33647	30157	30144	30144	30143	21869	20334	19338
Violet	33577	30143	30130	30129	30126	20187	19686	18667
AVEΔ_N	–	10.2%	10.2%	10.2%	10.2%	38.6%	43.6%	46.5%
Number of Edges								
DB	1696889	921170	831580	779502	774273	238596	297279	214758
Javac	1761197	967652	874079	820521	815220	259640	318844	235447
JEdit	2626960	1513423	1401168	1075290	1069333	338787	423598	314391
JGap	1792150	975737	880632	819586	814309	252615	318010	228600
Jpws	1823623	1005794	907518	848529	842952	280133	347504	257897
Mindterm	1743252	951832	854017	800711	795385	247825	310511	219768
Muffin	1800537	985564	886112	826637	821207	260815	324478	235154
SableCC	1466806	845363	749460	730955	725507	280994	143121	115234
VietPad	2019676	1076309	981160	920010	914611	279680	332829	243409
Violet	1834669	997052	896791	842490	836950	259232	334939	235037
AVEΔ_E	–	44.8%	50.2%	54.1%	54.4%	85.3%	83.1%	87.6%

Table 5.3: CHA call graph construction algorithm results: number of nodes and edges in the graph created by the corresponding version. **AVE**Δ is the average percentage decrease with respect to column **WBCL**.

The Nodes

The nodes of the call graph represent methods that are reachable from the `main` method of an application. By assuming that dynamic features will respect encapsulation, an average of 10% of the nodes are removed from the fully conservative graph. With the addition of each technique—using (1) type information from casting operations, (2) constant string values, and (3) semi-static string values—the number of nodes in the corresponding graphs are reduced, but not significantly. This trend can be seen in columns **CCIA**, **CSIA**, and **SSEA**. The reason that more nodes are

not removed is due to the treatment of unresolved `Method.invoke` calls and calls to native methods. Starting with the ERDF assumption, it is assumed that all calls to native methods will have an edge added to a synthetic node representing native code. This synthetic node will have an edge added to every `public` method in *AppClasses*, representing that native code could potentially call all public methods. Similarly, all unresolved `Method.invoke` calls will have edges added to all public methods (in addition to all other encapsulation safe methods.) This has the effect of making all public methods reachable. Consequently the only methods that will not be reachable (and not represented in the graph) are those not reachable from a public method. Appendix B presents the number and type of dynamic features encountered by the analysis for each benchmark and each implementation. Every execution of the analysis encountered both unresolved instances of `Method.invoke` and calls to native methods.

If the calls to native methods and `Method.invoke` are ignored, a substantial reduction in the number of graph nodes can be realized (though the call graph could and likely would be unsound). This fact can be observed in columns **CRFA**, **SCON**, and **SOOT**. These implementations provide no treatment for native methods and unresolved `Method.invoke` calls. For example, the graphs generated by CRFA have, on average, over 38% fewer nodes than those of WBCL.

This reduction in nodes cannot be entirely attributed to ignoring instances of reflective invocation and calls to native methods. The CRFA implementation will ignore all unresolved dynamic features. Therefore, it may be missing nodes which represent `clinit` methods due to unresolved dynamic class loading site, as well as nodes representing constructors due to unresolved calls to `newInstance`.

SCON provides a conservative treatment for instances of `Class.forName`. Thus, all the `clinit`s in its *AppClasses* should be represented in the graph. However, both

SCON and SOOT are using Soot's internal MBL analysis so the *AppClasses* they considered were smaller than the *AppClasses* considered by the other implementations. Thus, there may be fewer total `clinit` methods (as well as fewer constructors and methods). SCON provides a fully conservative treatment for instances of `class.NewInstance`, so all of the default constructors in its *AppClasses* will be represented. However, it ignores calls to `Constructor.newInstance`. Thus, non-default constructors may be missing from its graph.

SOOT ignores all dynamic features except for calls to `Class.forName(lit)` where `lit` is a string literal. On average, SOOT's graphs contain approximately 1000 fewer nodes for each application than CRFA.

The Edges

Edges in a call graph represent calling relationships between methods. Unlike the nodes, there was a dramatic reduction in the number of edges created by consecutive versions of the analysis. By simply assuming that dynamic features will respect encapsulation, an average of 44% of the edges can be trimmed from the fully conservative graphs (column **ERDF**). By using information from cast operations to resolve instances of reflection and dynamic class loading in addition to the assumption that reflection will respect encapsulation, an average of 50% of the edges can be removed (column **CCIA**). The use of a static string analysis by CSIA allows it to produce graphs that, on average, contain 54% fewer edges than the fully conservative graphs. By including semi-static values in the string analysis, the number of edges is further reduced (column **SSEA**). Thus, by assuming that (1) reflection will respect encapsulation, (2) cast operations and dynamic class loading will not generate exceptions, (3) dynamic features will not affect certain string values which flow to dynamic class loading sites, and (4) values of environment variables which are used in dynamic class

Features	CCIA	CSIA	SSEA	SCON
Dyno Loading	16%	46%	50%	23%
newInstance	56%	58%	61%	–
Invoke	0%	6%	6%	–

Table 5.4: Resolutions results: average percentage of resolved instance of dynamic features.

loading operations will remain constant, it is possible to generate call graphs which, on average, will contain 54% fewer edges than the fully conservative call graph.

CRFA, SCON and SOOT all produce unsound call graphs for the above benchmark applications. SCON attempts to produce a conservative graph but does not provide adequate treatment for all dynamic features. Thus, when SCON is compared to SSEA—our most liberal implementation that does not require user input—the graphs generated by SCON have, on average, 62% fewer edges than those produced by SSEA.

CRFA and SOOT produce graphs which are comparable since they both only provide treatment for dynamic features which they can resolve. However, the graphs created by CRFA contain, on average, 12% more edges than those produced by SOOT. These additional edges can be attributed to CRFA's ability to resolve more dynamic class loading sites and calls to `newInstance` and `Method.invoke`, which SOOT does not have.

Resolution of Dynamic Features

CCIA, CSIA, SSEA, and SCON all make an effort to resolve certain dynamic features. Table 5.4 presents the average percentage of reflective sites each version was able to resolve for all benchmark applications (CRFA and SOOT were not included

since they have the same resolution capabilities as SSEA and SCON, respectively). Row *Dyno Loading* presents the average percentage of dynamic class loading sites each version was able to resolve. Row *newInstance* presents the percentage of calls to `Class.newInstance` and `Constructor.newInstance` that were resolved. Row *Invoke* presents the percentage of `Method.invoke` calls that were resolved. Since SCON does not attempt to resolve instances of `newInstance` and `Method.invoke`, these table cells are empty. The actual number of sites encountered, and sites resolved for each benchmark are included in Appendix B.

The use of casting information appears to be only moderately effective at resolving instances of dynamic class loading. CCIA could only resolve an average of 16% of such sites. However, it is very successful at resolving calls to `newInstance` (on average CCIA was able to resolve 56% of such sites). It was expected that this technique would be more effective for `newInstance` sites. For casting information to be relevant for dynamic class loading sites, the loading site must be post-dominated by a `newInstance` site which in turn must be post-dominated by a casting operation. Therefore, for every resolved dynamic class loading site, there must be a corresponding resolved `newInstance` call. One of the reasons this technique was not more successful in resolving instances of dynamic class loading is due to the fact that the post-dominance and reaching definitions analyses used by CCIA were intraprocedural. In most instances, if a reflective instantiation of a class is used in a casting operation, it is instantiated and casted in the same method. However, it is not uncommon for the `Class` objects from dynamic class loading sites to flow through multiple methods before being instantiated. Furthermore, many of the dynamic class loading site were not post-dominated by cast operations. For example, many of the `Class` objects from the `Class.forName` calls in Jimple representation of JEdit, which correspond to

statements of the form `X.class` in the source representation, were not instantiated at all.

CSIA's incorporation of a static string analysis enabled it to be much more successful at resolving dynamic class loading sites. On average, it resolved 46% of the dynamic loading sites encountered. This increased precision enabled it to resolve 58% of the `newInstance` sites it encountered. The dynamic class loading sites it was not able to resolve depended on (1) string values that flowed from formal parameters of `public` or `protected` methods which were corrupted under the assumption that these values could be affected by unresolved reflective calls and native code, (2) string values passed through structures, such as `HashMaps`, which our string analysis is not powerful enough to model, or (3) string values that flowed from dynamic sources. Interestingly, several of the classes resolved by this technique were not found in set of *AppClasses* returned by the MBL analysis. In fact these classes were not included in the classpath for the system or in the application classpath. Several of these classes only existed in the Java 1.3 standard libraries and the code encountered by the analyses was deprecated. Several were RMI [93] stub classes that the library code assumed the client code would create. None of the benchmark applications included any such stubs. This library code was most likely reached via an infeasible path. These instances were considered unresolved by the analysis and were treated conservatively.

SSEA extends CSIA with a semi-static string analysis. This version performs a lookup of environment variables whose values flow to dynamic class loading sites (see Chapter 3 for a detailed description.) The addition of dynamically gathered environment information allowed SSEA to resolve 50% of all dynamic class loading sites and 61% of `newInstance` sites. SSEA consistently encountered the same 5 references to environment variables in the library code that were not set on the

experimental system (i.e. JSA's look-up of these values failed). These missing values were represented by the *anystring* value. Consequently, the dynamic class loading sites these values flowed to were not string resolvable. The fact that these variables were not set could imply that this code was reached via an infeasible path (at least for the experimental execution environment), or that these values would be set at run time. Again, several of the resolved classes were not in *AppClasses*. However, some of these classes were available on the classpath. This indicates that the SRPA MBL analysis was not sound for these instances. These missing classes correspond to dynamic class loading sites that were resolved using semi-static string values. SRPA used a weaker form of string analysis and thus did not have knowledge of these classes.

SCON resolved calls to `Class.forName` that rely on string literal values. This technique allowed it to resolve an average of 23% of all dynamic class loading sites it encountered. Again, several of the resolved classes were not in SCON's *AppClasses*, but were available on the classpath. This indicates that Soot's version of MBL is not sound either[14].

None of the implementations were effective at resolving calls to `Method.invoke`. We performed a manual investigation of the 27 unresolved instances that SSEA discovered for the DB benchmark. All 27 were located in the Java 1.4 standard libraries. Two of them relied on `Method` objects which flowed from dynamic class loading sites contained in the same method. The string analysis was unable to resolve these dynamic class loading sites due to values flowing from formal parameters of `public` methods. The remaining 25 `Method.invoke` sites relied on `Method` objects that were created in other procedures, meaning that our intraprocedural analysis was not capable of tracking their flow.

[14]Soot does allow its users to specify a set of classes that they believe will be loaded dynamically; however, here we study only the fully automated results.

5.5 Conclusion

This chapters presents a hierarchy of assumptions a *Class Hierarchy Analysis* call graph construction algorithm could make about the dynamic features of Java. At the top of the hierarchy is the most conservative assumption which generates an imprecise call graph for an application making use of dynamic features. Each consecutive level of the hierarchy represents a slight relaxation of the preceding level. Consequently, a graph created under each level of the hierarchy is a subgraph of the one generated by the preceding level. These relaxations allow the algorithm to incorporate various techniques that attempt to precisely resolve instances of dynamic class loading, reflective invocation, and reflective instantiation. These techniques include using information from casting operations and string analyses similar to the ones presented in Chapter 3.

Also presented was a similar assumption hierarchy for a *May Be Loaded* (MBL) analysis. Given an application P, this analysis estimates the set of classes that may be loaded into the JVM during any execution of P. This analysis is often used to determine the set of class that a CHA analysis must consider when building a call graph for P. The results of MBL are fundamental to the closed-world assumption under which CHA operates.

We implemented a version of the CHA analysis and the MBL analysis for each level of their assumption hierarchies. These implementations were applied to 10 real-world Java applications in an empirical study. This study provides a concrete example of the effects of each assumption and the corresponding resolution techniques on the results of these analyses. On average, our most precise implementation of CHA was able to resolve 50% of dynamic class loading sites, 61% of reflective instantiation sites, and 6% of reflective invocation sites. This capability enabled the implementation to

generate graphs that, on average, contain 10% fewer nodes and 54% fewer edges than the graphs generated by the fully conservative implementation.

CHAPTER 6: RELATED WORK

In this chapter we present work that is related to the subjects discussed in this dissertation. We first provide a survey of static and dynamic analyses that explicitly address some of the dynamic features of the Java language (Section 6.1). As demonstrated in Chapters 3 and 5, string analyses such as JSA can be very powerful and useful tools. In Section 6.2 we present an overview of other analyses that use JSA. We also discuss alternative string analyses. In Chapter 3 we presented a hybrid extension to string analysis. In Section 6.3 we provide an overview of other hybrid analyses. In Chapter 4 we presented our techniques to increase the scalability of JSA by modifying its algorithmic design to take advantage of modern multi-core architectures. In Section 6.4 we discuss work related to these techniques. Finally, in Section 6.5 we present an overview of relevant work related to the construction of method call graphs.

6.1 Analyses That Address Dynamic Features

Similar to the work presented in Chapters 3 and 5, other static analyses attempt to resolve instances of dynamic features in Java applications using techniques of various degrees of sophistication. In this section we present a few of the most relevant approaches.

Jax [139] is a Java application compression tool. It performs a variety of code transformations that reduce the overall size of an application. To preserve program

semantics the user must document, in a configuration file, all instances of dynamic class loading and reflection in the application. Our work presents a fully automated approach.

The class hierarchy analysis (CHA) call graph construction in the Soot analysis framework [144] employs a rudimentary string analysis that resolves calls to `Class.forName(String)` only if the parameter is a string literal. Our work employs a more powerful string analysis and, as shown in Section 5.4, our most precise version of CHA is able to resolve many more instances of dynamic features. Spark [83] is a points-to analysis engine implemented in Soot; it provides a hand-compiled list of reflective call sites that appear in the standard libraries. These possible targets are automatically accounted for in the analysis. However, such a solution is only compatible with the library version and system configuration on which the original manual check was performed.

Our analysis builds upon the powerful string analysis by Christensen et al. [19]. The authors of this work recognized that their analysis could be used to resolve instances of dynamic class loading. They present a small case study that investigates the ability to resolve calls to `Class.forName`. Our work considers a much wider range of dynamic class loading methods, as well as their use in the entire Java library. In addition, our extensions presented in Section 3.1 greatly increase JSA's ability to resolve instances of dynamic class loading, as shown in Section 3.4.

The work of Braux and Noye [10] extends classic partial evaluation techniques [22, 69] to apply them to the Java reflection API. Their work aims to replace invocations of the reflection API with conventional object-oriented syntax. This specialization relies on type constraints which must be completed by hand. Conceivably, a similar approach could be coupled with our work, in order to automatically create compilations of applications which are specific to a system's configuration.

Sreedhar et al. [129] propose a framework for interprocedural optimizations of programs that use dynamic class loading called the *extant analysis framework*. It performs a static analysis that classifies objects as either (1) `unconditionally extant` implying that their run-time type can statically be guaranteed, or (2) `conditionally extant` meaning that their run-time type cannot be guaranteed—this situation can arise due to dynamic class loading. They guard any optimizations dependent on `conditionally extant` objects with dynamic checks. Nguyen and Xue [101] use a similar classification approach in a side-effect analysis. Their analysis identifies points-to and modification sets which may not be complete due to dynamic class loading. It may be possible to incorporate our techniques for the resolution of dynamic class loading methods into these analyses. This extension could allow the analyses to provide a more precise treatment for certain dynamically loaded objects (e.g., identifying `conditionally extant` objects that could be treated as `unconditionally extant`).

The work of Livshits et al. [89] proposes a tiered approach to the resolution of dynamic class loading and reflection that is similar to our approach presented in Section 5.3.2. They present a static analysis algorithm that uses points-to information to determine the objects that could be loaded dynamically. Their algorithm tracks constant string values that flow to instances of dynamic class loading and reflection. For cases where they are unable to resolve the target string's value, they utilize casting information. If such information is not present, or a precise solution is required, their approach relies on user specifications. We use similar techniques in a CHA call graph construction algorithm. Our techniques could enhance the automation and precision of their analysis. We employ a more advanced string analysis and incorporate information that currently has to be manually provided to their analysis by the user. Our encapsulation safe treatment for unresolved instances of dynamic features also provides an alternative to user specifications.

The static analyses listed above are not able to automatically and accurately resolve instances of dynamic class loading that depend on environment variables. Our work presented in Chapter 3 shows that such instances constitute a large number of sites in the Java 1.4 libraries. The proposed used of semi-static values was shown to be able to resolve many of these instances.

Some existing work [65, 66, 75, 103, 106, 135] circumvents the typical shortcomings of static analyses by developing online algorithms. Pechtchanski and Sarkar [103] present a generic approach to performing online interprocedural analysis. Their approach identifies when a method analysis should be triggered and when optimistic assumptions are invalidated. Lee et al. [79] explore how different classloading strategies affect the performance of online interprocedural analyses. Hirzel et al. [65, 66] present an online version of a subset-based points-to analysis. Qian and Hendren [106] describe an online version of an XTA call graph construction algorithm. An online version of escape analysis is presented by Kotzmann and Mossenbock [75]. Sundaresan et al. [135] present techniques for performing aggressive code patching and collecting accurate profiles in multi-threaded applications that use dynamic class loading. All of the above approaches require either (1) modifications to the JVM services that handle dynamic class loading and reflection or (2) instrumentation of application code. These alterations allow the analyses to observe the actual execution of an application, which can be used to resolve any ambiguity introduced by the use of dynamic class loading. However, as with any purely-dynamic analysis, the results are unsound and represent only properties of the observed execution, not of all possible executions. Our approach has a more restricted form of unsoundness, as defined by the assumptions from Chapters 3 and 5.

6.2 Analyses Related to JSA

Many other analyses utilize the JSA library, although to the best of our knowledge we are the only ones to incorporate it into a call graph construction algorithm. The creators of JSA have employed it in several tools [20, 73] for Java web technologies and XML documents. The JDBC-Checker tool [53, 54] builds upon JSA to verify the correctness of dynamically generated SQL query strings. The AMNESIA tool [60] uses JSA to identify all possible string values of SQL queries to aid in the detection and prevention of SQL-injection attacks. SAFELI [43] also uses a JSA-like string analysis to detect possible SQL-injection attacks. Similarly, Wassermann and Su [145] presents a static analysis framework designed to prevent SQL command injection attacks. Their framework is built upon JSA. The work by Christodorescu et al. [21] extends JSA in the implementation of their static analysis that recovers possible values of C-style strings in x86 executables. The JSA library has also been used in the implementation of an approach to understand software application interfaces through string analysis [92]. To the best of our knowledge, no analysis other than the one by Christensen et al. [19] has employed JSA to resolve instances of dynamic class loading, nor have we been able to identify any techniques that augment JSA with the extensions proposed in our work, nor have any parallelization approaches been proposed for building of the front-end flow graph.

There are many other forms of string analysis that have been studied. For example, Tabuchi et al. [136] introduce an approach where string expressions are typed by regular languages. The work of Thiemann [137] utilized a type system for string analysis based on a context-free grammar. The approach used by Minamide [96] is based on the techniques of JSA but does not approximate CFGs to FSAs. Wassermann and Su adapt Minamide's approach to track taint information [146]. HAMPI [72]

is a solver for string constraints over bounded string variables. It takes constraints in the form of context-free languages and outputs a string that satisfies all constraints. Similarly, Emmi et al. [35] use a constraint solver that allows constraints over unbounded regular languages to automatically generate test cases for database applications. Godefroid et. al. [48] use a similar approach for generating test cases for compilers and interpreters. We used JSA as the foundation for our approach because it provides an open-source, well documented library that directly applies to Java applications. It is also widely accepted and used, as described above. However, other string analyses may be able to make use of our semi-static approach. For example, the work by Choi et al. [18] presents an abstract-interpretation-based approach to string analysis which uses a heuristic widening method to overcome the technical problem of recursive constraint solving encountered by Tabuchi et al. [136]. Their empirical study suggests that the precision of their analysis is comparable to that of JSA's. Their approach also provides treatment for fields and is context sensitive. By incorporating dynamically gathered environment information, their analysis may be able to generate comparable results to our extended version of JSA.

To the best of our knowledge, none of the above string analyses have been incorporated into a CHA call graph construction algorithm with the purpose of resolving calls to reflective or dynamic class loading methods. Furthermore, these existing analyses do not appear to be able to directly benefit from the multi-core architecture of most modern computing systems.

6.3 Hybrid Analyses

In Chapter 3 we demonstrated how a hybrid string analysis could aid in the resolution of dynamic class loading. Recent work has focused on similar hybridization

of dynamic and static analyses. Ernst [37] provides an overview of early work in this area. We now briefly describe some of the most relevant work.

Gupta et al. [59] introduced a hybrid approach to program slicing. Their approach uses dynamic information gathered by a debugging tool to increase the precision of their static slices. Specifically, they use the execution information from breakpoints and method calls/returns to prune spurious control flow while statically calculating slice information. Similarly, Mock et al. [97] use dynamic points-to information to reduce the number of infeasible data dependences considered by their static program slicing tool Sprite.

Groce and Joshi [56] use program traces to improve their static model checking for C programs. They use information from failure traces to generate code slices that eliminate certain control flow. This pruning has the potential to dramatically reduce the size of the SAT instances used in their model checking.

Dufour et al. [29,30] introduced the *blended program analysis* paradigm for performance understanding of applications built with frameworks. This paradigm combines dynamic representations of a program's calling structure with targeted static analyses. The dynamic analysis isolates an execution of interest and the static analysis is applied only to that execution. They demonstrate this paradigm by implementing a blended escape analysis.

Artzi et al. [6] present a "pipelined" mutability analysis. Each stage of the pipeline represents a lightweight mutability analysis. These analyses classify parameters as `mutable`, `immutable`, or `unknown`. Once a stage completes its result set is passed to the next stage. Once a parameter has been identified as `mutable` or `immutable` no succeeding stage can change that classification. One of the stages in the pipeline is a dynamic analysis which monitors the execution of the program and identifies `mutable` parameters.

Recent work in the area of automatic test case generation has focused on a hybrid approach (e.g., [4, 5, 12, 25, 36, 47–50, 68, 90, 91, 124, 142]). The majority of this work extends and refines an approach which combines the symbolic and concrete executions of a program. A symbolic execution of a program is a simulated execution where the program is supplied symbolic constraints for inputs. Every assignment along an execution path updates the program state with a symbolic expression. Every conditional encountered on the path generates a constraint in terms of symbolic inputs. The constraints generated by the symbolic execution are simplified using values observed during the concrete execution. A constraint solver is then used to generated concrete inputs that satisfy the simplified constraints. Csaller et al. [24, 25] similarly combine symbolic and concrete executions to discover likely program invariants and to support debugging.

Our semi-static version of JSA, like all the analyses listed in this section, uses dynamic analysis to increase the precision of a static analysis. However, the dynamic portion of our analysis is extremely lightweight (limited to the gathering of environment variable information) when compared with the analyses listed above. A more comprehensive dynamic analysis could be combined with JSA to increase its precision. However, such a coupling comes at the cost of soundness and performance.

6.4 Scalable Static Analyses

A limited amount of work has been conducted on parallelizing data flow analyses. Notably, Lee and Ryder [81] defined three types of parallelism inherent in dataflow problems: independent-problem parallelism, separate-unit parallelism, and algorithmic parallelism. Independent-problem parallelism is the concurrent execution of dataflow problems that are not related. This form of parallelism does not apply

to JSA as each dataflow analysis it uses depends upon the results of its predecessor. Lee and Ryder define separate-unit parallelism as existing in intraprocedural dataflow analysis of multiple-procedure applications. We describe how this form of parallelism may be exploited for JSA in Chapter 4. The third form of parallelism they define is algorithmic, where fixed-point calculations of a dataflow problem are parallelized. Several researchers have focused on this level of parallelism for different dataflow problems using different partitioning techniques, e.g., [34,76,80,82,125,151]. Algorithmic parallelism could also be applied to the dataflow analyses used by JSA, though the communication and load balancing costs associated with such a design would likely be higher than the separate-unit approach presented in this dissertation.

The work of Dwyer and Martin [32] is similar to our work in that they parallelized an existing analysis system written in Java in an attempt to increase scalability. They parallelized FLAVERS [31], a toolset that uses flow analysis to verify user-specified correctness properties of concurrent systems. FLAVAERS uses different flow analyses than those used by JSA. Their parallelization exploited algorithmic parallelism whereas our parallelization of JSA exploited separate-unit parallelism. Their empirical study used classic parallel programs as benchmark applications (e.g., *readers-writers*) while we used publicly-available, production-level applications in our study.

Our parallelization of JSA relies on modularity in the analyses performed by JSA. Other techniques have exploited modularity in static analysis. We outline a few of the most relevant approaches below. A more complete discussion of modular static analysis is available in [23].

There are several analyses that make use of summary functions, e.g., [14–17, 40, 117, 147–149]. Most employ a bottom-up traversal of the call graph, and compute summary functions using the functions computed for the visited procedures. Another

technique that exploits modularity computes partial analysis results for each component, combines the results for all components and completes the rest of the analysis; examples include [22, 26, 38, 102, 114]. It should be possible to parallelize both of the these techniques since there is a clear partitioning of independent work.

Several approaches have relied on pre-computed library summary information to increase the scalability of whole-program analyses; examples include [13,110,116,126]. This work demonstrates that significant speedups can be achieved through the use of library summaries. It may be possible to develop similar pre-computed library summaries for JSA.

6.5 Call Graph Construction Algorithms

There has been a substantial amount of research conducted in the area of call graph construction for object-oriented programs. This section briefly describes a few of the approaches that are most related to our work.

We proposed enhancements to a CHA call graph construction algorithm [27]. For a virtual call site $x.m()$ where C is the compile-time type of x, CHA assumes that all subtypes of C are possible run-time receivers of the call to m. CHA is context and flow insensitive making it extremely efficient in both time and space but imprecise. The Soot analysis framework [144] provides an implementation of this analysis for Java programs and provides a minimal attempt to resolve instances of dynamic class loading and a conservative treatment of `newInstance` calls. In Chapter 5 we demonstrated how our proposed treatments were able to resolve many more instances of dynamic features. JAN [105] also provides an implementation of CHA. JAN relies on user inputs to resolve dynamic features. We presented a range of fully automated treatments for these features.

Rapid Type Analysis [7] is similar to CHA in that it takes a whole program as input and produces a context and flow insensitive call graph. It differs from CHA in that it constructs a global set which approximates the set of all types instantiated in the program. It uses this set to reduce the number of candidate types considered for each virtual call site. For a virtual call site $x.m()$ where C is the compile-time type of x, RTA assumes that all instantiated (as determined by membership in the global instantiation set) subtypes of C are possible run-time receivers of the call to m. The call graph generated by RTA has been shown to be more precise than that of CHA [7, 105]. JAN presents an implementation of RTA for Java applications. It relies on inputs from the users to precisely treat the dynamic features of Java. Sharp et al. [126] present an RTA implementation for Java applications as a plugin for the popular Eclipse IDE [33]. This implementation provides no treatment for dynamic features such as reflection or dynamic class loading. The Soot analysis framework provides an implementation of RTA with similar treatment of the dynamic features as their CHA implementation. Our approaches presented in Chapter 5 could be extended to an RTA analysis enabling it to more precisely model certain uses of dynamic Java features.

Tip and Palsberg [140] present a series of call graph construction algorithms which refine RTA's approach of tracking instantiated types. Rather than creating a global set for the entire application, they create sets for individual program entities, thus creating a more precise local view. The most precise algorithm they present is XTA. XTA creates a distinct set for every method and field contained in the application. These sets contain the possible types instantiated in the method and pointed to by the field they are associated with. Tip and Palsberg state that for detecting unreachable methods the more advanced algorithms perform only marginally better than RTA, but are more precise at determining call sites with a single target. They

mention that the implementations of their analyses, used in their experiments, rely on the actions of dynamic constructs such as reflection and dynamic class loading being specified manually. Our techniques for resolving dynamic features could potentially be generalized to an XTA call graph construction algorithm. The variable-type analysis (VTA) presented by Sundaresan et al. [134] computes type information for variables and fields with the help of a type propagation graph. They state that they manually summarize the effects of native methods on their analysis but make no mention of other dynamic features of Java.

There is a range of more expensive analyses for call graph construction. One algorithm is 0-CFA [127], which creates one set of approximate run-time values per expression. Another is k-CFA, which uses multiple sets per expression, thus modeling context-sensitivity. Grove et al. [57, 58] present a parameterized algorithmic framework for call graph construction of object-oriented languages. In this framework they implemented a number of call graph construction algorithms, including 0-CFA and k-CFA. They state that their framework does not address dynamic class loading. It would be interesting to see how a context-sensitive approach to string analysis could increase its precision.

Points-to analysis determines the set of memory locations to which a pointer variable may refer. This information can be used to resolve virtual dispatch and to precisely determine possible calling relationships between methods. There has been extensive work conducted in the area of points-to analysis, as summarized in [64,118]. The points-to analysis of Livshits et al. [89] described above currently provides the most precise treatment of dynamic features in Java for this type of analysis. Our techniques for dynamic feature resolution, including our semi-static approach to string analysis, could potentially increase the precision of points-to analyses for Java.

To the best of our knowledge, our incorporation of JSA into a CHA call graph construction algorithm represents the most precise string analysis to be used to resolve instances of dynamic class loading for this type of analysis. None of the analyses cited above use an encapsulation safe approach for conservative treatment of unresolved dynamic features. To date, the empirical study presented in Chapter 5 is the most comprehensive study of the effects of assumptions about dynamic features on a CHA call graph construction algorithm.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

The existence of dynamic features, such as dynamic class loading, reflection and, native methods, allow modern Java applications to be extremely flexible. Unfortunately, these same features pose a significant challenge to static analyses. In the general case, it is impossible to statically determine the precise run-time behavior of these dynamic features. Many static analyses take one of two approaches in their treatment of dynamic Java features: (1) ignore such features and generate an unsound result when applied to applications that use them or (2) treat them in a sound but overly conservative manner possibly obscuring relevant results with spurious information. Our work presents techniques that will enable static analyses to more precisely handle some of the most commonly used dynamic features in Java.

7.1 Incorporating Dynamically Gathered Information

A key dynamic feature commonly used by Java applications that support plugin architectures are the methods that enable dynamic class loading. Given a string representation of a class' fully-qualified name, these methods will load the specified class into the JVM at run time. For example, a call `Class.forName(s)` dynamically loads the class with the name represented by the string expression `s`. If a static analysis is able to resolve the possible run-time values of `s`, it could treat the call to `forName` as a static initialization of the classes specified by `s`. A string analysis can be used to attempt to resolve `s`. However, current string analyses will fail to

precisely resolve `s` if it is not a compile-time constant. They will also fail if they are not powerful enough to model the flow of the string value through the application.

We present a hybrid extension to string analysis that incorporates configuration information gathered at analysis-time to aid in the resolution of dynamic class loading in Java applications. This approach enables the string analysis to consider values which are typically treated as dynamic values. A manual investigation of the Java 1.4 standard library revealed that over 40% of the nontrivial client independent dynamic class loading sites depended on this configuration information. We also present extensions of string analysis that allow it to precisely model the flow of static string values. In an experimental study conducted on the Java 1.4 standard libraries, our approach was able to resolve 2.6 times more dynamic class loading sites than the state-of-the-art string analysis. We also demonstrate how the information gained from resolved dynamic class loading sites can be used to determine the classes that can potentially be instantiated through the use of reflection. The use of configuration information and our extensions to JSA increase the number of resolvable reflective instantiation sites from 6 to 37.

In the future this work can be extended to incorporate other sources of configuration information, such as configuration files. Various generalizations of string analysis could also be pursued, such as context sensitivity and more precise handling of value flow through containers (e.g., sets, maps, and lists). It would also be interesting to investigate other forms of static analysis that can benefit from a similar environment-aware approach, by employing techniques such as program specialization.

7.2 Increasing the Scalability of String Analysis

Though it has been shown that a precise string analysis can improve the precision of some static analyses, many that could benefit have not incorporated any of the

existing string analysis algorithms. One of the reasons that many analyses have not adopted a precise string analysis might be related to the high costs (in terms of memory and time) associated with performing such an analysis.

We present multiple techniques to increase the scalability of the Java String Analyzer (JSA) library [19]. The input to JSA is a set of Java classes and a set of hotspots (i.e., expressions of interest). JSA conservatively estimates the possible run-time string values at all instances of those hotspots in the input classes. JSA's design consists of a *front-end* and a *back-end* component. The front-end creates a graph which abstractly represents the flow of string values through the input classes. The back-end converts the flow graph into a context-free grammar and ultimately generates a finite state automaton for each hotspot expression in the input classes.

The techniques presented include algorithmic transformations which parallelize portions of JSA's front-end allowing it to leverage modern multi-core architectures. Our empirical study of 25 benchmark applications showed that for 22 of the benchmarks our most advanced parallel version of JSA's front-end was, on average, able to achieve a speedup of 1.54 over the sequential version of JSA. The parallel design also reduced the average memory footprint for these 22 applications by approximately 43%. For the remaining 3 applications, all version of JSA (both parallel and sequential versions) exhausted the allotted heap memory.

We present three new simplifications to the graphs created by JSA's front-end. These simplifications preserve relevant details at expressions of interest (hotspots) while reducing the size of the overall graph . Since the flow-graph is the input to the back-end, this reduction can decrease the amount of work that must be performed by the back-end. On average, these new simplifications added 200 ms to the total running time of the front-end of the parallel version of JSA. However, for two applications, the simplifications reduced the running time of the back-end from over 590,000 ms to

under 600 ms. These simplifications also enable JSA to complete the analysis of the three applications that previously exhausted a 6Gb heap.

Our experiments indicate that for some applications, our parallel implementations were not realizing their theoretical speedup. This can be attributed, in part, to contention for memory resources between the slave threads. In the future it would be interesting to see if the use of object pools [55] could reduce this contention. The experiments also indicated that, though adequate for the benchmarks used in the experiments, the load balancing heuristic used by the parallel implementations could be improved. Recall that we used a heuristic that weighted methods by the number of Jimple statements they contained. It would be interesting to explore more advanced heuristics that provide considerations for other code features such as the number of string variables or virtual calls. It would also be interesting to see if the scalability of JSA can be increased through the use of pre-computed library summaries.

7.3 Assumption Hierarchy for a CHA Call Graph Construction Algorithm

By increasing the modeling power of JSA, demonstrating that this new expressiveness can substantially increase its ability to resolve dynamic class loading sites, and reducing its execution costs, we have made it practical to incorporate JSA into a static analysis. We demonstrated this fact by employing it as part of a call graph construction algorithm.

Class Hierarchy Analysis (CHA) [27] call graph construction algorithms are common components of many static analyses. Call graphs abstractly represent the calling relationships between methods of an application. The nodes of the graph correspond to methods and directed edges represent calls between methods. This information

is commonly used by interprocedural analyses. The existence of dynamic feature in Java, whose exact run-time behavior cannot be determined purely statically, makes it very challenging to create a sound call graph.

We present a systematic exploration of the effects that dynamic features can have on the results produced by a CHA call graph algorithm. We also explore the assumptions that such an algorithm can make about dynamic features, and present a novel hierarchy of such assumptions. The hierarchy is rooted at the most conservative assumption. Call graphs built under this assumption will be sound but will likely be very imprecise. Each consecutive level of the hierarchy represents a slight relaxation of the preceding level. Consequently, the graph created at each level of the hierarchy is a subgraph of the one generated by the preceding level. These relaxations allow the algorithm to incorporate various techniques in an attempt to precisely resolve instances of dynamic class loading, reflective invocation, and reflective instantiation. These techniques include using information from casting operations, as well as the use of our semi-static string analysis. Our techniques to improve string analysis scalability make it practical to include in the call graph construction algorithm.

Also presented is a similar assumptions hierarchy for a *May Be Loaded* (MBL) analysis. Given an application P, this analysis estimates the set of classes that may be loaded into the JVM during any execution of P. This analysis is often used to determine the set of classes that a CHA analysis must consider when building a call graph for P. An unsound result generated by a MBL analysis could violate the closed-world assumption under which CHA operates.

We implemented a version of the CHA analysis and the MBL analysis for each level of their assumption hierarchies. These implementations were applied to 10 real-world Java applications in an empirical study. On average, the most precise implementation of CHA, which uses our semi-static string analysis, was able to resolve 6% of the

reflective invocation sites, 50% of dynamic class loading sites, and 61% of reflective instantiation sites encountered by the analysis. This capability enabled this version to generate graphs that, on average, contain 10% fewer nodes and 54% fewer edges than the graphs generated by the fully conservative implementation. These results indicate that by allowing a few very practical relaxations to the fully conservative assumption, a call graph's precision can be greatly increased.

Our experiments indicate that the technique for resolution of reflective calls through `Method.invoke` was greatly hampered by its intraprocedural nature. It would be interesting to see how many more instances could be resolved if this limitation were removed through the use of interprocedural post-dominance and reaching definitions analyses. Another alternative would be to use a string analysis to identify the name of the method being represented by the `Method` object in order to reduce the number of methods being considered.

An obvious extension of this work would be to apply similar assumptions and resolution techniques to a more precise call graph construction algorithm. A natural choice would a Rapid Type Analysis [7] call graph construction algorithm. RTA maintains a set *Instantiated* of classes that could be instantiated in methods reachable from the main method. Virtual call sites are resolved based on these classes. *Instantiated* is updated whenever RTA encounters `new X` expressions. If an update of *Instantiated* implies additional target methods at already-processed call sites, the call graph is updated to reflect the newly discovered relationship. RTA is arguably more sensitive to dynamic features than CHA, since even a single conservative treatment of reflective instantiation would result in RTA generating the same graph as CHA.

7.4 Conclusion

We have demonstrated that by reducing the cost of string analysis and increasing its modeling capabilities, it can successfully be incorporated into a CHA call graph construction algorithm to aid in the resolution of dynamic Java features. The call graphs produced by this algorithm contain significantly fewer edges than the graphs generated by an algorithm that provides an overly conservative treatment for dynamic features. By increasing the precision of call graphs our work transitively increases the precision of many static analyses which depend on them. Moreover, our semi-static and dynamic feature resolution techniques can be incorporated into other static analyses, increasing their precision directly.

The work presented in this dissertation is a step toward making static analysis tools better equipped to handle the dynamic features of Java. These include tools that facilitate software development, testing, and understanding. Increasing the precision of these tools can decrease development costs and increase software reliability.

APPENDIX A: COMPLETE RESULTS FOR PARALLEL JSA EXPERIMENTS

This appendix presents the complete time and memory measurements for the experiments conducted on the parallel implementations of JSA presented in Chapter 4. The following implementations were used in the experiments: JSA-INTU, JSA-RMEM, JSA-PSIM, and JSA-NSIM. Tables A.1, A.2, A.3, and A.4 present the results of this study. The **App** column shows the benchmarks used in the experiments. Columns **1 Thread – 4 Threads** contain the time, in milliseconds, it took the implementation's front-end to complete the construction of the flow-graph using the specified number of slave threads. The **Memory** column presents the maximum memory footprint observed, in Mb, during the execution.

App	1 Thread	2 Threads	3 Threads	4 Threads	Memory
Buoy	1529	1493	1562	1791	123.6
Compress	693	524	505	478	41.2
DB	747	607	528	557	44
Fractal	657	452	721	723	39.9
GattMath	1352	1259	1190	1253	86.8
Jack	1725	1276	1304	1255	116.4
Javac	3286	2476	2561	2792	285.8
JavaCup	2750	2481	2612	2887	145.9
Jb61	1370	1116	1037	1153	73.1
JEdit	20933	13788	12331	12475	935.2
Jess	2321	1471	1589	1425	112.9
JFlex	6330	6067	5908	5808	155.7
JGap	2057	1623	1609	1632	141.3
JLex	1274	986	814	787	57.2
Jpws	4051	2798	2871	2704	187.3
Jtar	1769	1615	1201	1443	81
Mindterm	3145	2418	2506	2369	217.1
MpegAudio	2833	2557	2384	2354	132.7
Muffin	5572	4355	4507	4389	235.3
Rabbit	1512	1434	1664	1472	115.45
Sablecc	5697	4119	4126	4325	351.9
Sockecho	728	610	534	536	47
Sockproxy	766	625	739	571	44.7
VietPad	6149	4754	4427	3919	325.8
Violet	1403	1636	1490	1604	101.3

Table A.1: Front-end time and memory results for the implementation of the intuitive parallel design (JSA-INTU). Time results are in ms and memory results are in Mb.

App	1 Thread	2 Threads	3 Threads	4 Threads	Memory
Buoy	1368	1173	1203	1487	68.56
Compress	648	573	554	580	28.2
DB	710	588	582	568	30.23
Fractal	990	780	837	701	23.6
GattMath	1090	860	947	975	44.4
Jack	1585	1252	1204	1273	61.5
Javac	2980	2375	2351	2485	113.4
JavaCup	2477	2327	2336	2236	54.6
Jb61	1250	1028	1022	1114	36.8
JEdit	20123	12583	9856	9803	408.1
Jess	1927	1532	1620	1591	64.2
JFlex	6635	6014	5772	5757	69.2
JGap	2012	1535	1526	1492	72.3
JLex	1126	791	721	782	29.7
Jpws	3261	2570	2681	2659	112.6
Jtar	1190	963	889	1103	36.9
Mindterm	2855	2268	2233	2209	102.7
MpegAudio	2814	2414	2364	2286	88.3
Muffin	5569	4094	3956	4084	145
Rabbit	1340	1465	1021	1532	51.7
Sablecc	5374	4040	4255	4020	137
Sockecho	725	578	603	593	26.7
Sockproxy	760	579	609	570	27.7
VietPad	4389	3336	3236	3200	177.5
Violet	1251	1082	1014	1180	54

Table A.2: Front-end time and memory results for the implementation of the reduced memory parallel design (JSA-RMEM). Time results are in ms and memory results are in Mb.

App	1 Thread	2 Threads	3 Threads	4 Threads	Memory
Buoy	1381	1198	1224	1377	74.2
Compress	703	555	467	548	26.17
DB	736	583	550	569	27.6
Fractal	964	810	908	880	24.5
GattMath	1035	769	774	802	44.1
Jack	1577	1110	1051	1144	63.2
Javac	3035	2170	1994	1946	111.3
JavaCup	2555	2236	2299	2201	52.6
Jb61	1271	912	882	907	33.4
JEdit	20069	9652	6512	5821	402.7
Jess	1756	1405	1293	1302	61.0
JFlex	6264	5265	5070	5087	72.5
JGap	1812	1485	1677	1679	71.3
JLex	1143	749	734	745	33.1
Jpws	3046	1920	2102	1943	116.7
Jtar	1298	952	1296	948	36.1
Mindterm	2724	1667	1462	1713	107.4
MpegAudio	2969	2363	2321	2186	90.1
Muffin	5564	2662	2592	2541	145.7
Rabbit	1301	1026	1308	971	53.4
Sablecc	5312	2849	2538	2603	140.5
Sockecho	740	535	529	550	28.1
Sockproxy	787	499	544	546	26.22
VietPad	5315	2833	2982	3047	181.9
Violet	1284	945	1051	1047	57.4

Table A.3: Front-end time and memory results for the implementation of the parallel graph simplification design (JSA-PSIM). Time results are in ms and memory results are in Mb.

App	1 Thread	2 Threads	3 Threads	4 Threads	Memory
Buoy	1409	1149	1156	1274	72
Compress	716	565	552	621	25.1
DB	781	595	552	527	33.1
Fractal	922	763	763	993	27.8
GattMath	1172	991	902	989	47.1
Jack	1739	1292	1217	1287	68.6
Javac	3119	2292	2064	2189	118.63
JavaCup	2521	2278	2226	2421	52.8
Jb61	1325	1013	816	932	32.2
JEdit	20451	9533	6877	6053	415.8
Jess	1730	1418	1363	1350	61.5
JFlex	6457	5393	5057	5443	75.4
JGap	1995	1591	1764	1873	74.12
JLex	1153	774	696	725	31.9
Jpws	3090	1952	2134	2072	116
Jtar	1428	1361	1318	1381	39.8
Mindterm	3943	2742	2387	2418	110.7
MpegAudio	2815	2330	2324	2302	90.1
Muffin	4889	2756	2779	2729	149.5
Rabbit	1467	1295	1480	1472	56.6
Sablecc	5319	2956	2652	2723	135.3
Sockecho	786	576	578	557	38.4
Sockproxy	761	576	603	622	27.6
VietPad	5593	3438	3005	2796	152.7
Violet	1283	923	1056	985	56.2

Table A.4: Front-end time and memory results for the implementation of the parallel JSA with three new graph simplifications (JSA-NSIM). Time results are in ms and memory results are in Mb.

APPENDIX B: COMPLETE RESOLUTION RESULTS OF THE CHA CALL GRAPH EXPERIMENTS

This appendix contains additional results from the experiments described in Chapter 5. The following seven implementations of a CHA call graph construction algorithm were used in these experiments: WBCL, ERDF, CCIA, CSIA, SSEA, SCON and SOOT. Tables B.1, B.2, and B.3 present the number of dynamic features that each implementation encountered in its analysis of a benchmark. The tables also show how many of these features each implementation was able to resolve. The **App** column contains the applications used in the experiments. The sub-columns under **Dyno Loading** present information about calls to dynamic class loading methods the implementation encountered. The sub-columns under **newInstance** show information relating to calls to `Class.newInstance` and `Consturctor.newInstance`. Information pertaining to calls to `Method.invoke` is found under the column heading **Method.invoke**. Information relating to native methods is presented under the heading **Native**. The sub-columns **Calls** indicate the number of instances of the dynamic feature the analysis encountered and the sub-columns **Res** indicates the how many of these calls the implementation was able to resolve. Since none of the implementations attempt to precisely resolve calls to native methods, there is no sub-column **Res** for this category.

	WBCL - Fully Conservative						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	161	0	101	0	34	0	666
Javac	161	0	101	0	34	0	666
Jedit	449	0	108	0	44	0	669
Jgap	182	0	104	0	36	0	666
Jpws	166	0	101	0	34	0	666
Mindterm	166	0	106	0	34	0	666
Muffin	170	0	104	0	34	0	666
Sablecc	145	0	98	0	23	0	662
Vietpad	183	0	106	0	42	0	669
Violet	166	0	108	0	36	0	666
	ERDF - Encapsulation Safe						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	150	0	95	0	29	0	636
Javac	150	0	95	0	29	0	636
Jedit	437	0	102	0	39	0	636
Jgap	171	0	98	0	31	0	636
Jpws	155	0	95	0	29	0	636
Mindterm	155	0	100	0	29	0	636
Muffin	159	0	98	0	29	0	636
Sablecc	134	0	92	0	18	0	632
Vietpad	172	0	100	0	37	0	639
Violet	158	0	104	0	31	0	636
	CCIA - Correct Casting						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	149	24	94	52	29	0	636
Javac	149	24	94	52	29	0	636
Jedit	436	28	101	59	39	0	636
Jgap	170	27	97	55	31	0	636
Jpws	154	24	94	52	29	0	636
Mindterm	154	29	99	57	29	0	636
Muffin	158	25	97	55	29	0	636
Sablecc	133	24	91	52	18	0	632
Vietpad	171	25	99	53	37	0	639
Violet	157	28	103	59	31	0	636

Table B.1: Number of dynamic features encountered and resolved by the WBCL, ERDF, and CCIA implementations of the CHA call graph construction analysis.

	CSIA - Static Strings						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	149	62	94	54	29	2	636
Javac	149	62	94	54	29	2	636
Jedit	436	342	101	61	39	5	636
Jgap	170	77	97	57	31	2	636
Jpws	154	67	94	54	29	2	636
Mindterm	154	67	99	59	29	2	636
Muffin	158	70	97	57	29	2	636
Sablecc	133	48	91	54	18	0	632
Vietpad	171	74	99	55	37	2	639
Violet	157	66	103	61	31	2	636
	SSEA - Semi-Static Strings						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	149	67	94	57	29	2	636
Javac	149	67	94	57	29	2	636
Jedit	436	347	101	64	39	5	636
Jgap	170	82	97	60	31	2	636
Jpws	154	72	94	57	29	2	636
Mindterm	154	72	99	62	29	2	636
Muffin	158	75	97	60	29	2	636
Sablecc	133	53	91	57	18	0	632
Vietpad	171	79	99	58	37	2	639
Violet	157	71	103	64	31	2	636
	CRFA - Ignore All Not Resolved						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	109	40	79	51	16	0	0
Javac	109	40	79	51	16	0	0
Jedit	374	298	89	59	28	4	0
Jgap	123	52	82	54	18	0	0
Jpws	114	45	79	51	16	0	0
Mindterm	114	45	84	56	16	0	0
Muffin	117	45	82	54	16	0	0
Sablecc	111	42	79	51	16	0	0
Vietpad	152	70	92	54	34	2	0
Violet	116	44	88	58	18	0	0

Table B.2: Number of dynamic features encountered and resolved by the CSIA and SSEA implementations of the CHA call graph construction analysis.

	SCON - Conservative Soot						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	116	17	80	0	16	0	486
Javac	116	17	80	0	16	0	487
Jedit	370	257	89	0	28	0	497
Jgap	130	26	83	0	18	0	489
Jpws	121	22	80	0	16	0	489
Mindterm	121	20	85	0	16	0	490
Muffin	122	21	82	0	16	0	495
Sablecc	70	13	55	0	7	0	254
Vietpad	140	39	88	0	24	0	492
Violet	123	19	89	0	18	0	485
	SOOT - Default Soot						
	Dyno Loading		newInstance		Method.invoke		Native
App	Calls	Res	Calls	Res	Calls	Res	Calls
Db	111	17	76	0	16	0	471
Javac	111	17	76	0	16	0	473
Jedit	365	257	85	0	28	0	482
Jgap	125	26	79	0	18	0	474
Jpws	116	22	76	0	16	0	475
Mindterm	116	20	81	0	16	0	475
Muffin	118	21	79	0	16	0	483
Sablecc	66	13	52	0	7	0	245
Vietpad	135	39	84	0	24	0	477
Violet	118	19	85	0	18	0	470

Table B.3: Number of dynamic features encountered and resolved by the SCON and SOOT implementations of the CHA call graph construction analysis.

BIBLIOGRAPHY

[1] `http://lse.sourceforge.net/numa/faq/`.

[2] G. Agrawal. Demand-driven construction of call graphs. In *International Conference on Compiler Construction*, pages 125–140, 2000.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, 2008.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis*, pages 261–272, 2008.

[6] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 104–113, 2007.

[7] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[8] J. Barker. *Beginning Java Objects: From Concepts To Code*. Apress, 2005.

[9] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[10] M. Braux and J. Noye. Towards partially evaluating reflection in Java. In *ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 2–11, 1999.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[12] C. Cadar, P. Twohey, V. Ganesh, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[13] R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-382, Rutgers University, 1999.

[14] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[15] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[16] S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *International Conference on Compiler Construction*, pages 172–186, March 2007.

[17] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[18] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Asian Symposium on Programming Languages and Systems*, pages 374–388, 2006.

[19] A. S. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Static Analysis Symposium*, LNCS 2694, pages 1–18, 2003.

[20] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[21] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 88–95, 2005.

[22] M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 451–464, 1993.

[23] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, pages 159–178, 2002.

[24] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. In *International Symposium on Software Testing and Analysis*, pages 1–37, 2008.

[25] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering*, pages 281–290, 2008.

[26] M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[27] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[28] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[29] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *International Symposium on Software Testing and Analysis*, pages 118–128, 2007.

[30] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 59–70, 2008.

[31] M. B. Dwyer, L. A. Clarke, G. Naumovich, and J. M. Cobleigh. Data flow analysis for verifying properties of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, 1994.

[32] M. B. Dwyer and M. Martin. Practical parallelization: Experience with a complex flow analysis. Technical Report KSU CIS TR 99-4, Kansas State University, 1999.

[33] Eclipse project. `www.eclipse.org`.

[34] M. Edvinsson and W. Löwe. A parallel approach for solving data flow analysis problems. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2008.

[35] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[36] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[37] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *International Workshop on Dynamic Analysis*, pages 24–27, 2003.

[38] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, Mar. 1999.

[39] D. Flanagan. *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[40] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

[41] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, 2005.

[42] C. Fu and B. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *International Conference on Software Engineering*, pages 230–239, 2007.

[43] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *Computer Software and Applications Conference*, pages 87–96, 2007.

[44] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1971.

[45] N. Glew and J. Palsberg. Method inlining, dynamic class loading, and type soundness. *Journal of Object Technology*, 4(8):33–53, 2005.

[46] N. Glew, J. Palsberg, and C. Grothoff. Type-safe optimisation of plugin architectures. In *Static Analysis Symposium*, pages 135–154, 2005.

[47] P. Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, 2007.

[48] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[49] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[50] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*, 2008.

[51] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[52] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3 edition, 2005.

[53] C. Gould, Z. Su, and P. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *International Conference on Software Engineering*, pages 697–698, 2004.

[54] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *International Conference on Software Engineering*, pages 645–654, 2004.

[55] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[56] A. Groce and R. Joshi. Exploiting traces in program analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 379–393, 2006.

[57] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[58] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[59] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6:370–397, 1997.

[60] W. G. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, 2005.

[61] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, June 2007.

[62] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 312–326, 2001.

[63] M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.

[64] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[65] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems*, 29(2):11, 2007.

[66] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference on Object-Oriented Programming*, pages 96–122, 2004.

[67] W. Huang, W. Srisa-an, and J. M. Chang. Object allocation and memory contention study of Java multithreaded applications. In *International Performance, Computing, and Communications Conference*, pages 375–382, 2004.

[68] K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 425–428, 2007.

[69] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[70] *Java Virtual Machine Profiler Interface (JVMPI)*, 2007.

[71] R. Khatchadourian, J. Sawin, and A. Rountev. Automated refactoring of legacy Java software to enumerated types. In *IEEE International Conference on Software Maintenance*, pages 224–233, 2007.

[72] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[73] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[74] A. Kleen. A NUMA API for Linux. Novel Inc., 2005.

[75] T. Kotzmann and H. Mossenbock. Escape analysis in the context of dynamic compilation and deoptimization. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120, 2005.

[76] R. Kramer, R. Gupta, and M. Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):805–813, 1994.

[77] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, June 2007.

[78] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *International Conference on Compiler Construction*, pages 287–304, 2005.

[79] K. Lee, Q. Ali, and S. P. Midkiff. Efficient classloading strategies for inter-procedural analyses in the presence of dynamic classloading. In *International Workshop on Dynamic Analysis*, pages 6–13, 2007.

[80] Y. Lee, T. Marlowe, and B. Ryder. Performing data flow analysis in parallel. In *ACM/IEEE Conference on Supercomputing*, pages 942–951, 1990.

[81] Y. Lee and B. Ryder. A comprehensive approach to parallel data flow analysis. In *International Conference on Supercomputing*, pages 236–247, 1992.

[82] Y. Lee, B. G. Ryder, and M. E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. *IEEE Transactions on Software Engineering*, 21:913–926, 1995.

[83] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[84] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[85] S. Liang. *The Java Native Interface. Programmer's Guide and Specification.* Addison-Wesley, 1999.

[86] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 36–44, 1998.

[87] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.

[88] Y. Liu and A. Milanova. Static analysis for dynamic coupling measures. In *Conference of the Center for Advanced Studies on Collaborative Research*, pages 119–130, 2006.

[89] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, LNCS 3780, pages 139–160, 2005.

[90] R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.

[91] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 134–143, 2007.

[92] E. Martin and T. Xie. Understanding software application interfaces via string analysis. In *International Conference on Software Engineering*, pages 901–904, 2006.

[93] Sun Microsystems. *RMI Specification.* 2002.

[94] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[95] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[96] Y. Minamide. Static approximation of dynamically generated web pages. In *International Conference on World Wide Web*, pages 432–441, 2005.

[97] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 71–80, 2002.

[98] M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord, editors, *Robustness in Language and Speech Technology*, pages 251–261. Kluwer Academic Publishers, 2000.

[99] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, 1965.

[100] H. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *International Conference on Software Engineering*, pages 80–86, 1988.

[101] P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Australasian Conference on Computer Science*, pages 9–18, 2005.

[102] N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *European Conference on Object-Oriented Programming*, pages 329–349, 1992.

[103] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 195–210, 2001.

[104] M. Pistoia, R. Flynn, L. Koved, and V. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *European Conference on Object-Oriented Programming*, pages 362–386, 2005.

[105] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with Java. In *Conference of the Center for Advanced Studies on Collaborative Research*, pages 19–34, 1998.

[106] F. Qian and L. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.

[107] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[108] A. Rountev. Precise identification of side-effect-free methods in Java. In *IEEE International Conference on Software Maintenance*, pages 82–91, 2004.

[109] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[110] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[111] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Fundamental Approaches to Software Engineering*, LNCS 3442, pages 282–297, 2005.

[112] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

[113] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004.

[114] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, LNCS 2027, pages 20–36, 2001.

[115] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

[116] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, LNCS 4959, pages 53–68, 2008.

[117] E. Ruf. Effective synchronization removal for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[118] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, LNCS 2622, pages 126–137, 2003.

[119] J. Sawin and A. Rountev. Estimating the run-time progress of a call graph construction algorithm. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 53–62, 2006.

[120] J. Sawin and A. Rountev. Improved static resolution of dynamic class loading in Java. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 143–154, 2007.

[121] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in Java using dynamically gathered environment information. *International Journal of Automated Software Engineering*, 16(2):357–381, June 2009.

[122] J. Sawin, M. Sharp, and A. Rountev. Generating run-time progress reports for a points-to analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 40–44, 2006.

[123] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, 2003.

[124] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[125] V. Sgro and B. G. Ryder. Differences in algorithmic parallelism in control flow and call multigraphs. In *Workshop on Languages and Compilers for Parallel Computing*, pages 217–233, 1994.

[126] M. Sharp, J. Sawin, and A. Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 6–10, 2005.

[127] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[128] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *International Conference on Software Engineering*, pages 336–345, 2004.

[129] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[130] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[131] M.-A. Storey and H. Müller. Manipulating and documenting software structures using SHriMP views. In *IEEE International Conference on Software Maintenance*, pages 275–284, 1995.

[132] M.-A. Storey, K. Wong, and H. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, 2000.

[133] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, Sept. 2000.

[134] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[135] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 87–97, 2006.

[136] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In *International Workshop on Types in Programming*, pages 1–18, 2002.

[137] P. Thiemann. Grammar-based analysis of string expressions. In *ACM SIGPLAN Workshop on Types in Languages Design and Implementation*, pages 59–70, 2005.

[138] T. Tian and C.-P. Shih. Software techniques for shared cache multi-core systems. Intel Software Network, 2007.

[139] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1999.

[140] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[141] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems*, 24(6):625–666, 2002.

[142] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *International Symposium on Software Testing and Analysis*, pages 97–107, 2007.

[143] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *IEEE International Conference on Software Maintenance*, pages 159–168, 2003.

[144] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[145] G. Wassermann and Z. Su. An analysis framework for security in web applications. In *FSE Workshop on Specification and Verification of Component-Based Systems*, pages 70–78, 2004.

[146] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[147] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[148] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *International Symposium on Software Testing and Analysis*, pages 225–236, 2008.

[149] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122, 2009.

[150] J. Xue and P. H. Nguyen. Completeness analysis for incomplete object-oriented programs. In *International Conference on Compiler Construction*, pages 271–286, 2005.

[151] A. Zobel. Parallel interval analysis of data flow equations. In *International Conference on Parallel Processing*, pages 9–16, 1990.