Autom Softw Eng (2009) 16: 357–381
DOI 10.1007/s10515-009-0049-9

Improving static resolution of dynamic class loading in Java using dynamically gathered environment information

Jason Sawin · Atanas Rountev

Received: 25 January 2009 / Accepted: 2 February 2009 / Published online: 19 February 2009

Abstract In Java software, one important flexibility mechanism is dynamic class loading. Unfortunately, the vast majority of static analyses for Java treat dynamic class loading either unsoundly or too conservatively. We present 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. Thus, the string values reported by our technique are tailored to the current installation of the system under analysis. Additionally, we propose extensions of string analysis to increase the number of sites that can be resolved purely statically, and to track the names of environment variables. An experimental evaluation on the Java 1.4 standard libraries shows that a state-of-the-art purely static approach 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. This work is a step towards making static analysis tools better equipped to handle the dynamic features of Java.

Keywords Static analysis · String analysis · Dynamic class loading · Reflection

This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040.

J. Sawin (✉) · A. Rountev
The Ohio State University, 395 Dreese Labs, 2015 Neil Ave, Columbus, OH 43210, USA
e-mail: sawin@cse.ohio-state.edu

A. Rountev
e-mail: rountev@cse.ohio-state.edu

1 Introduction

Modern software applications often need to be highly adaptable and flexible. They 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 difficult, secondary concern by the static analysis community. An example of these constructs are the Java features that allow for *dynamic class loading*. These powerful language features allow Java applications to load classes into the Java Virtual Machine (JVM) (Lindholm and Yellin 1999) at run time, requiring only a string representation of the fully-qualified name of the class. 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 (e.g., spurious data dependences and control-flow paths).

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 that is 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.

In this paper we present a novel semi-static approach which combines static string analysis with dynamically gathered information about the execution environment. 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 (Storey and Müller 1995) or Rigi (Müller and Klashinsky 1988). Such tools have the potential to overwhelm their users with too much information (Storey et al. 2000). 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, 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 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 study 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 the information gained from resolved dynamic class loading sites can be used to aid in the determination of the 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.

2 Background

This section provides an overview of dynamic class loading in Java, as well as a high-level description of string analysis for Java.

2.1 Dynamic class loading in Java

The Java Virtual Machine (JVM) is one of the defining components of the Java platform (Lindholm and Yellin 1999). 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 (Liang and Bracha 1998). 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).

Example Figure 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 paper. The code is from the Java 1.4 standard library class `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. 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.

```
private final String handlerPropName =
              "sun.awt.exception.handler";
private String handlerClassName = null;

private boolean handleException(Throwable thrown) {
     .....
     /* Get the class name stored in environment
      * variable sun.awt.exception.handler */
     handlerClassName
       = (String) AccessController.doPrivileged(
       new GetPropertyAction(handlerPropName));
     .....
     /* Load the class and instantiate it */
     Object h;
     Class c = Class.forName(handlerClassName,...);
     h = c.newInstance();
     .....
}
```

Fig. 1 Sample code from library class `java.awt.EventDispatchThread`

Similar examples can be found throughout the entire JDK code. Frameworks heavily use dynamic class loading features to implement their component models; the same is true for various Java-based servers. The uses of these mechanisms will only become more prevalent as the complexity of Java applications grows. It is critical for the static analysis community to aggressively address the problem in handling such features.

2.2 Java string analyzer

Most static analyses have taken one of two approaches for the handling of dynamic features in Java: (1) ignore them or (2) treat such features 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 (Livshits et al. 2005) that significant portions of the program call graph have been omitted by a static analysis which disregards dynamic features such as dynamic class loading. Conversely, the conservative approach assumes that any class will be loaded and instantiated. However, the relevant information will be obfuscated by the number of infeasible interactions inferred by this technique. Some analyses (Tip et al. 1999; Livshits et al. 2005) require that the user manually specify the classes which could be loaded at dynamic class loading sites. However, this technique is time consuming and error prone. Livshits et al. (2005) 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 Fig. 1.

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 (Christensen et al. 2003a; Livshits et al. 2005; Vallée-Rai et al. 2000) employ various forms of string analysis in an attempt to determine the possible run-time values of these target strings. One of the most popular string analyses for Java is available in the *Java String Analyzer* (JSA) library (Christensen et al. 2003a).

The input to JSA is a set of Java classes and a set of expressions, or "*hotspots*." JSA conservatively computes the possible run-time string values at all instances of those hotspots in the input classes. The analysis utilizes the Soot analysis framework to generate the Jimple intermediate representation (Vallée-Rai et al. 2000). From this representation, JSA builds a *flow graph* that models the flow of string values and the operations that manipulate them. The nodes of the graph represent variables and expressions; the edges are directed def-use edges that represent the possible flow of data. The graph contains five types of nodes: `Init` nodes represent the initial construction of string values, `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`. In essence, this graph is a static single assignment form where the join nodes are analogous to ϕ functions.

From the flow graph JSA 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: $A_n \to A_x A_y$. The production for an `Init` node n is $A_n \to reg$ where *reg* corresponds to a regular language. JSA then utilizes the Mohri-Nederhof algorithm (Mohri and Nederhof 2000) to transform the grammar into a strongly-regular context-free grammar. The result can be accurately modeled by a finite state automaton. Such an automaton is created for each node in the graph that represents a hotspot. The language produced by the automaton is a superset of the possible string values that can occur at that hotspot.

3 Improving the precision of JSA

String analyses such as the one outlined in Sect. 2.2 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. To address some of these limitations, Sect. 3.1 proposes an extension to JSA which increases the number of relevant string values available to the analysis, and Sect. 3.2 presents several extensions that improve JSA's overall modeling capabilities.

3.1 Semi-static analysis

Consider the example in Fig. 1. If some client of JSA 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

```
 1  public static PrinterJob getPrinterJob() {
 2    ....
 3    return (PrinterJob)java.security.AccessController.do-
 4    Privileged(
 5        new java.security.PrivilegedAction() {
 6        public Object run() {
 7        String nm
 8          = System.getProperty("java.awt.printerjob");
 9        try {
10          return Class.forName(nm).newInstance();
11        } catch (ClassNotFoundException e) {
12    ....
13  }
```

Fig. 2 Sample code from library class `java.awt.printer.PrinterJob`

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 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 Fig. 2 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.

```
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)
```

Fig. 3 Entry points for environment variables

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 Fig. 3. All of these methods take a `key` string parameter which specifies the environment variable that is to be accessed. In the example presented in Fig. 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 view 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 sites. This approach assumes that calls to these methods do not affect the string analysis in any other way. In general, the idea of examining the values of environment variables could also be applied to other static analyses; Sect. 5 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`.[1] 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.

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

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 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 Sect. 2.2), 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 variables through the use of reflection. Our approach conservatively corrupts all String values which flow from reflective calls.

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 Sect. 5.

3.2 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 Fig. 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 consider only 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 union 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 (Christensen et al. 2003b; Sundaresan et al. 2000). Note that in the versions of the analysis described in Sect. 5, *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 Sect. 6, 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 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 variable types which are given precise treatment by JSA. In its original form JSA only models variables of type `String`, `StringBuffer`, and `StringBuilder`, and arrays with a base type of `String`. However, in the code displayed in Fig. 1, the look-up of environment variable `sun.awt.exception.handler` is accomplished by creating an instance of library class `sun.security.action.GetPropertyAction` (line 10). This convenience class implements `java.security.PrivilegedAction`. Objects of type `PrivilegedAction` are most commonly passed to invocations of `AccessController.doPrivileged`. This results in `PrivilegedAction.run` being executed with privileges enabled. In the case of `GetPropertyAction`, the `run` method simply wraps an invocation of method `System.getProperty`. We treat `doPrivileged` as a primitive operation and assume that its execution will not effect string values. 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 wrap accesses to environment variables in a `PrivilegedAction`. Thus, it is paramount for the success of our semi-static approach that JSA be able to properly model such occurrences. We propose to extend 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 the actions are thus limited, 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.

4 Resolving reflective instantiation

Being able to determine the classes that can be initialized by dynamic class loading is a valuable ability for many static analyses. Consider the example shown in Fig. 1. By resolving the instance of dynamic class loading at line 14 the resolved classes will be identified as part of the application and their static initializers will be identified as reachable. However, dynamic class loading is just one of Java's dynamic features. Consider the method call `c.newInstance()` at line 15 of Fig. 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 referred to by `c`.

To illustrate the importance of being able to resolve instances of reflective instantiation, consider the well-known Rapid Type Analysis (RTA) (Bacon and Sweeney 1996) 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 that there are additional target methods called at already-processed call sites, the call graph is updated to reflect the newly discovered relationships.

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

```
 1  public SunToolkit() {
 2       ....
 3       String tgName
 4         = System.getProperty("awt.threadgroup", "");
 5       ....
 6       Constructor ctor = Class.forName(tgName).
 7                    getConstructor(new Class[] {String.class});
 8       threadGroup = (ThreadGroup)ctor.
 9                       newInstance(new Object[]
10                       {"AWT-ThreadGroup"});
11       ....
12  }
```

Fig. 4 Sample code from library class `sun.awt.SunToolkit`

imprecise CHA (Dean et al. 1995). Thus RTA's efforts would be rendered superfluous. Similarly to RTA, many other call graph construction algorithms face problems due to reflective instantiations.

The key to determining which classes could be instantiated at a call to `Class.newInstance` is identifying the classes represented by the `Class` object. Thus, a natural extension of the work from Sect. 3 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 Fig. 4, `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 `java.lang.reflect.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 and passing it the string "`AWT-ThreadGroup`".

Determining the classes that could potentially be instantiated by calls to `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 determine only those classes that could be instantiated, and we are not concerned with the specific constructor that is 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 analyzed. Section 6 presents a study that uses these techniques to resolve reflective instantiation in the Java 1.4 standard libraries. It demonstrates the precision gained by exploiting information from environment variables and by our JSA modeling extensions.

5 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. Future studies will need to examine how often such situations occur in different application domains.

It is likely that other static analyses can also benefit from such 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.

Static analyses which perform program specialization (Futamura 1971) are 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 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 versions 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.

Assumptions about unknown code interactions 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 conservative assumptions about unknown code interactions. Specifically, it is assumed that certain string values can be affected by unknown client code, by native methods, or by unresolved 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 categories of assumptions:

1. *Omnipotent client interactions* (OCI): This is a fully conservative assumption which assumes that through the use of reflection and native methods, 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 and native methods (including reflection used by the input classes) 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 none of the target string values could be affected by the use of reflection or native methods 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 Sect. 6 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 Fig. 5. 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

```
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)
```

Fig. 5 Library methods used for dynamic class loading

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 byte-code in a manner that could affect the flow of string values to dynamic class loading sites.

6 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 Fig. 5 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. Four versions of JSA were investigated in this study:

1. **JSA-ORIG**: JSA in its original form with minor bug fixes, and some alterations to accommodate the assumptions under which our experiments were conducted.
2. **JSA-ENV**: JSA-ORIG enhanced with the semi-static technique presented in Sect. 3.1.
3. **JSA-OBJ**: JSA-ENV enhanced with the type extension outlined in Sect. 3.2.
4. **JSA-FLD**: JSA-OBJ enhanced by the technique for more precise treatment of fields described in Sect. 3.2.

The experiments presented in this section were executed on a PC with an Intel Core Duo T2400 (1.83 GHz) processor and 2 GB of memory, running a Windows XP OS. The JVM heap size (JVM option Xmx) was set to 1.5 GB. The execution environment remained stable for all experimental executions and the values of all referenced environments variables remained constant.

6.1 Establishing baseline results

To establish a "perfect baseline" to which we could compare our results and investigate the affects of the various 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.[2]

The study was conducted under the assumptions described in Sect. 5. 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.

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.

[2]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.

Table 1 Manual investigation of the Java 1.4 standard libraries: categorized counts of invocations of dynamic class loading methods under various assumptions

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

Table 1 shows the results of this manual investigation. Under OCI, the most conservative 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. 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 can in no way be affected directly by client code. This is indicated by the fact that between the most restrictive OCI and 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. 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.

6.2 Resolution of dynamic class loading

Table 2 shows the results of resolving dynamic class loading sites in the Java 1.4 standard libraries using the four versions of JSA described above. These four versions operate under the SCI assumption, thus a total of 80 dynamic class loading sites were considered (Table 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 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).

Table 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

Analysis version	JSA-ORIG	JSA-ENV	JSA-OBJ	JSA-FLD
SD	22	22	22	27
EVD	0	16	28	32
Resolved	22	38	50	59
% of total	32%	56%	74%	87%

Table 3 Analysis cost: running time (seconds per thousand Jimple statements) and memory usage (MB)

Analysis version	JSA-ORIG	JSA-ENV	JSA-OBJ	JSA-FLD
Sec per 1000 Jimple	1.69	1.72	1.74	1.81
MB	20.4	20.4	20.5	21.7

To accommodate the SCI assumption, all versions of JSA used in this experiment 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 and protected methods (these can be overridden by clients), methods not contained within the code body under analysis, 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 standard libraries.

The results for the unaltered version of JSA are shown in column JSA-ORIG. Since this version did not use our semi-static extension, it was only able to resolve 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. The addition of the semi-static technique enabled JSA-ENV to resolve 16 more sites than JSA in its original form (JSA-ORIG). Although JSA-OBJ did not increase the number of resolved SD sites, it was able to resolve an additional 35% of all EVD sites, due to its ability to more precisely track string values that flow from environment variables (e.g., as illustrated by the call to `doPrivileged` in Fig. 1). The final version, shown in column JSA-FLD, added the more precise treatment of fields described in Sect. 3.2. 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, JSA-FLD, 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%. JSA-FLD 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 operations that were beyond the modeling abilities of JSA, for example, values returned by a `StringTokenizer`.

Table 4 Precision of string analyses for the Java 1.4 standard libraries: number of reflective instantiation sites resolved by JSA

Analysis version	JSA-ORIG	JSA-ENV	JSA-OBJ	JSA-FLD
Resolved	6	19	28	37

Analysis cost As stated earlier, each version of JSA was executed once for each of the 358 packages comprising the Java 1.4 standard libraries. Table 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 the 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 (JSA-ENV, JSA-OBJ, and JSA-FLD), 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 worst case, the pre-processing phase is no more expensive than JSA-ORIG 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.

6.3 Resolution of reflective instantiations

Section 4 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 invocations of `Class.newInstance` and transitively to calls to `Constructor.newInstance` (for the remainder of this paper, references to `newInstance` indicate both the `Class` and the `Constructor` methods). As described in Sect. 4, 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 4. Row *Resolved* shows the number of calls to `newInstance` resolved by each version of JSA.

Table 5 Categorized counts of resolved dynamic class loading sites whose `Class` objects flow to `newInstance`

Version	JSA-ORIG	JSA-ENV	JSA-OBJ	JSA-FLD
SD	8	8	8	13
EVD	0	14	25	29
Total	8	22	33	42

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 5, which shows the number and type of dynamic class loading sites which were used to resolve reflective instantiation. Comparing row *Total* in Table 5 with row *Resolved* in Table 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 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 JSA-ENV enables the resolutions of additional dynamic class loading sites (Table 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 JSA-ENV to resolve a total of 19 reflective instantiation sites. Over 57% of the reflective instantiation sites resolved by JSA-ENV are indirectly dependent on environment variables.

JSA-OBJ was able to resolve 28 `newInstance` sites (see Table 2). 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, JSA-FLD was able to resolve a total of 59 dynamic class loading sites (Table 2). Of these sites, objects from 13 sites 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.

6.4 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 paper 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 by certain invocations of `newInstance`. One potential use of this information is to make popular call graph algorithms such as RTA (Bacon and Sweeney 1996), XTA, MTA, and FTA (Tip and Palsberg 2000) 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 whereas 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 not be resolved in a purely static manner.

7 Related work

Many static analyses attempt to resolve instances of dynamic class loading in Java applications using techniques of various degrees of sophistication. In this section we present a few of the most relevant ones along with a brief description of other analyses employing the JSA library.

Jax (Tip et al. 1999) 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 (Vallée-Rai et al. 2000) 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. Spark (Lhoták and Hendren 2003) is a points-to analysis engine implemented in Soot; it provides a hand-compiled list of call sites using reflection inside 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. (2003a). 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 modifications greatly increase JSA's ability to resolve instances of dynamic class loading, as shown in Sect. 6.

The work of Braux and Noye (1999) extends classic partial evaluation techniques (Codish et al. 1993; Jones et al. 1993) 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 to automatically create compilations of applications which are specific to a system's configuration.

The work of Livshits et al. (2005) proposes a tiered approach to the resolution of dynamic class loading and reflection. They present a static analysis algorithm that uses points-to information to determine the classes 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. Our work 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.

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 shows that such instances constitute a large number of sites in the Java 1.4 libraries. The proposed new approach was shown to be able to resolve many of these instances.

Some existing work (Hirzel et al. 2004, 2007; Qian and Hendren 2004; Pechtchanski and Sarkar 2001; Sundaresan et al. 2006) circumvents the typical shortcomings of static analyses by developing online algorithms. This approach typically requires modifications to the JVM services that handle dynamic class loading and reflection. 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 Sect. 5.

Many other analyses utilize the JSA library. The creators of JSA have employed it in several tools (Christensen et al. 2003b; Kirkegaard et al. 2004) for Java web technologies and XML documents. The JDBC-Checker tool (Gould et al. 2004a, 2004b) builds upon JSA to verify the correctness of dynamically generated SQL query strings. The AMNESIA tool (Halfond and Orso 2005) uses JSA to identify all possible string values of SQL queries to aid in the detection and prevention of SQL-injection attacks. Similarly, Wassermann and Su (2004) presents a static analysis framework designed to prevent SQL command injection attacks. Their framework is built upon JSA. The work by Christodorescu et al. (2005) 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 Martin and Xie (2006). To the best of our knowledge, no analysis other than the one by Christensen et al. (2003a) has employed JSA to resolve instances of dynamic class loading, nor have we been able to identify any that augments JSA with the extensions proposed in our work.

There are many forms of string analysis that have been studied. For example, Tabuchi et al. (2002) introduce an approach where string expressions are typed by regular languages. The work of Thiemann (2005) utilized a type system for string analysis based on a context-free grammar. The approach used by Minamide (2005) is based on the techniques of JSA but does not approximate CFGs to FSAs. Wassermann and Su (2007) adapt Minamide's approach to track taint information. 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. (2006) presents a string analysis approach based on abstract interpretation which uses a heuristic widening method to overcome the technical problem of recursive constraint solving encountered by Tabuchi et al. (2002). Their empirical study suggests that the precision of their analysis is comparable to that of JSA's. Their approach also provides a precise treatment of fields and is context sensitive. By incorporating dynamically gathered environment information, their analysis may be able to generate results that are comparable to our extended version of JSA.

8 Conclusions and future work

This paper 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 of our modeling extensions to JSA, increases the number of resolvable reflective instantiation sites from 6 to 37.

In the future we plan to extend our approach 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). We also plan to investigate methods of reducing the overall cost of JSA both in terms of memory usage and time. 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.

References

Bacon, D., Sweeney, P.: Fast static analysis of C++ virtual function calls. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 324–341 (1996)

Braux, M., Noye, J.: Towards partially evaluating reflection in Java. In: ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation, pp. 2–11 (1999)

Choi, T.H., Lee, O., Kim, H., Doh, K.G.: A practical string analyzer by the widening approach. In: Asian Symposium on Programming Languages and Systems, pp. 374–388 (2006)

Christensen, A.S., Møller, A., Schwartzbach, M.: Precise analysis of string expressions. In: Static Analysis Symposium, pp. 1–18 (2003a)

Christensen, A.S., Møller, A., Schwartzbach, M.I.: Extending Java for high-level Web service construction. ACM Trans. Program. Lang. Syst. **25**(6), 814–875 (2003b)

Christodorescu, M., Kidd, N., Goh, W.H.: String analysis for x86 binaries. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 88–95 (2005)

Codish, M., Debray, S., Giacobazzi, R.: Compositional analysis of modular logic programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 451–464 (1993)

Dean, J., Grove, D., Chambers, C.: Optimizations of object-oriented programs using static class hierarchy analysis. In: European Conference on Object-Oriented Programming, pp. 77–101 (1995)

Futamura, Y.: Partial evaluation of computation process—an approach to a compiler-compiler. Syst. Comput. Controls **2**, 45–50 (1971)

Gould, C., Su, Z., Devanbu, P.: JDBC checker: A static analysis tool for SQL/JDBC applications. In: International Conference on Software Engineering, pp. 697–698 (2004a)

Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: International Conference on Software Engineering, pp. 645–654 (2004b)

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

Hirzel, M., Diwan, A., Hind, M.: Pointer analysis in the presence of dynamic class loading. In: European Conference on Object-Oriented Programming, pp. 96–122 (2004)

Hirzel, M., Dincklage, D.V., Diwan, A., Hind, M.: Fast online pointer analysis. ACM Trans. Program. Lang. Syst. **29**(2), 11 (2007)

Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)

Kirkegaard, C., Møller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Trans. Softw. Eng. **3**(3), 181–192 (2004)

Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: International Conference on Compiler Construction, pp. 153–169 (2003)

Liang, S., Bracha, G.: Dynamic class loading in the Java virtual machine. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 36–44 (1998)

Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1999)

Livshits, B., Whaley, J., Lam, M.: Reflection analysis for Java. In: Asian Symposium on Programming Languages and Systems, pp. 139–160 (2005)

Martin, E., Xie, T.: Understanding software application interfaces via string analysis. In: International Conference on Software Engineering, pp. 901–904 (2006)

Minamide, Y.: Static approximation of dynamically generated web pages. In: International Conference on World Wide Web, pp. 432–441 (2005)

Mohri, M., Nederhof, M.J.: Regular approximation of context-free grammars through transformation. In: Junqua, J.C., van Noord, G. (eds.) Robustness in Language and Speech Technology, pp. 251–261. Kluwer Academic, Norwell (2000)

Müller, H., Klashinsky, K.: Rigi—a system for programming-in-the-large. In: International Conference on Software Engineering, pp. 80–86 (1988)

Pechtchanski, I., Sarkar, V.: Dynamic optimistic interprocedural analysis: A framework and an application. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 195–210 (2001)

Qian, F., Hendren, L.: Towards dynamic interprocedural analysis in JVMs. In: Virtual Machine Research and Technology Symposium, pp. 139–150 (2004)

Storey, M.A., Müller, H.: Manipulating and documenting software structures using SHriMP views. In: IEEE International Conference on Software Maintenance, pp. 275–284 (1995)

Storey, M.A., Wong, K., Müller, H.: How do program understanding tools affect how programmers understand programs? Sci. Comput. Program. **36**(23), 183–207 (2000)

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallee-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 264–280 (2000)

Sundaresan, V., Maier, D., Ramarao, P., Stoodley, M.: 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, pp. 87–97 (2006)

Tabuchi, N., Sumii, E., Yonezawa, A.: Regular expression types for strings in a text processing language. In: Proceedings of International Workshop on Types in Programming, pp. 1–18 (2002)

Thiemann, P.: Grammar-based analysis of string expressions. In: ACM SIGPLAN Workshop on Types in Languages Design and Implementation, pp. 59–70 (2005)

Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 281–293 (2000)

Tip, F., Laffra, C., Sweeney, P., Streeter, D.: Practical experience with an application extractor for Java. In: ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 292–305 (1999)

Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the Soot framework: Is it feasible? In: International Conference on Compiler Construction, pp. 18–34 (2000)

Wassermann, G., Su, Z.: An analysis framework for security in web applications. In: Workshop on Specification and Verification of Component-Based Systems, pp. 70–78 (2004)

Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 32–41 (2007)