Improved Static Resolution of Dynamic Class Loading in Java

Jason Sawin Atanas Rountev
Ohio State University
{sawin,rountev}@cse.ohio-state.edu

Abstract

Modern applications are becoming increasingly more dynamic and flexible. In Java software, one important flexibility mechanism is dynamic class loading. Unfortunately, the vast majority of static analyses for Java handle this feature either unsoundly or overly conservatively. We present a set of techniques for static resolution of dynamic-class-loading sites in Java software. Previous work has used static string analysis to achieve this goal. However, a large number of such sites are impossible to resolve with purely static techniques. 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. In addition, we propose generalizations of string analysis to increase the number of sites that can be resolved purely statically, and to track the names of environment variables. We present an experimental evaluation on 10,238 classes from the standard Java libraries. Our results show that a state-of-the-art purely static approach resolves only 28% of non-trivial sites, while our approach resolves more than twice as many sites. This work is a step towards making static analysis tools better equipped to handle the dynamic features of Java.

1 Introduction

Modern software applications need to be highly adaptable and flexible to stay competitive. 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 such as Eclipse and Tomcat support third-party extensions. These extensions allow these frameworks to stay current and relevant without requiring them to absorb resulting massive development costs. To gauge the demand and success of such extensions, one only has to take note of the number of third-party extensions available for Eclipse.

This increased application flexibility does limit what can statically be determined about a program. One very 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. Prime examples of this situation are Java constructs that allow for *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. Dynamic class loading is used extensively in applications such as Eclipse, Tomcat, EJB application servers, etc. 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 as to render the end result useless.

Some recent work has employed *static string analysis* to resolve instances of these dynamic features. Such an approach statically attempts to determine the value of the string which 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, then 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 fairly 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 to be loaded. 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 libraries revealed that over 40% of the fully-contained instances of dynamic class loading —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* in that their run-time values typically remain the same across multiple execution 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 generalizations of static string analysis that improve the tracking the names of environment variables, as well as increase the number of sites that can be resolved purely statically.

Our approach produces results that are sound with respect to the current execution environment, but do not apply to all possible environments. For many clients of static analyses this is both reasonable and desirable. For example, consider program understand tools such as SHriMP [21] or Rigi [18]. Such tools have the potential to overwhelm their users with too much information [22]. If such tools tried to account for *every* class that could 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 generalizations of string analysis that not only enable our approach to resolve more instances of environment-dependent instances of dynamic class loading, but also allow for a greater number of purely static instances to be resolved.
- We describe an experimental study in which our approach was applied to the entire Java 1.4 standard libraries. The results of this experiment indicate that the approach is able to resolve more then twice the number of client-independent sites currently resolvable by the state-of-the-art static string analysis. Through comprehensive manual investigation we also determined that our approach identifies 91% of all sites that are in fact truly static or environment-variable-dependent, which implies very high analysis precision.

The proposed approach and the experimental results define a significant improvement for the handling of dynamic class loading in static analysis, compared to current techniques. Such 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 a brief overview of the dynamic class loading feature in Java, as well as a high-level description of the state of the art in Java string analysis.

2.1 Dynamic Class Loading in Java

The Java Virtual Machine (JVM) is one the defining components of the Java platform [14]. 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 [13]. This is a very 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

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

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

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 then performs a hierarchical 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 meta-data 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 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 *sun.awt.exception.handler* to the string value representing the fully-qualified name of this class. Method `handleException` in `EventDispatchThread` queries this environment variable to retrieve the specified class name (lines 8 and 9) and stores it in field `handlerClassName`. The custom handler is then loaded at line 13 — method `forName` is one of several methods in the Java libraries that can be used to *dynamically load* classes at run time. 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. ◀

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

2.2 Java String Analyzer

Most static analyses have taken two approaches for the handling of the dynamic features in Java: ignore them or treat them very conservatively. Ignoring these features produces a result that is unsound and may miss vital program entity interactions. Such an approach would render an analysis impractical for use on modern Java applications; for example, there is evidence [15] that significant portions of the program call graph can be omitted by this approach. 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 such as [15] and [24] require that the user manually specify the interactions which occur due to dynamic class loading. However, this technique can be time consuming and error prone. Yet others [15] utilize casting information to narrow the field of what needs to be considered. However, such an approach would fail for the code presented in Figure 1, since no cast of the dynamically loaded class is performed.

Since strings specify the classes that are to be loaded at instances of dynamic class loading, a precise *string analysis* has the greatest potential to precisely resolve such instances without requiring input from the user.

The work in [2, 15, 25] employs various forms of string analysis in an attempt to determine the possible run-time values of these target strings. The most powerful string analysis currently available for Java is in the *Java String Analyzer* (JSA) library described in [2].

The input to JSA is a set of Java classes and a set of expressions (*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 and parse the Jimple intermediate representation [25]. From this representation, JSA builds a *flow graph* that models the flow of string values and the operations which 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: $A_n \rightarrow A_x A_y$. The production for an `Init` node n is $A_n \rightarrow \mathit{reg}$ where *reg* corresponds to a regular language. JSA then utilizes the Mohri-Nederhof algorithm [17] 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 Generalizing JSA

String analyses such as the one presented in Section 2.2 have two points of possible failure when attempting to precisely determine the run-time values a string-typed expression can assume:

1. The value of the expression depends upon values that the analysis does not have access to (e.g., the `args[ ]` array passed to a main method).
2. The analysis is not powerful enough to model the flow and manipulation of the string values.

In this section we present several generalizations to JSA. These generalizations increase both the number of program entities the analysis can access and its overall modeling capabilities. These enhancements greatly improve the analysis' ability to resolve instances of dynamic class loading.

3.1 Semi-Static Analysis

Consider the example code shown in Figure 1. If some JSA client specifies the invocation statement `forName(str,...)` as a hotspot, 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 resulting value indicates that under JSA's model, the parameter could potentially 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, most 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. Under most common programming paradigms, these variables provide the program with information about the type of environment it is operating in. Hypothetically, 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 marked by the 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, library class `java.awt.print.PrinterJob` queries an environment variable to determine which classes to load in order to create a job that the current system's printer will recognize. Such a variable will be consistent across systems that have the same type of printer. It is rare that a system frequently changes its printer, and therefore for a given system the value will essentially be static.

We purpose a generalization 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

```
java.lang.System.getProperty (<string>)
java.lang.System.getProperty (<string>, <string>)
java.security.Security.getProperty (<string>)
sun.security.action.GetPropertyAction (<string>)
```

Figure 2. Some entry points for environment variables.

string values. We present only the end alterations to the graph; for brevity, the details of the intermediate stages are not discussed.

Our approach is based on the set of Java library methods that serve as entry points for the values of environment variables; a subset of these methods is shown in Figure 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 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 to be semi-static since, strictly speaking, it is a dynamic execution of a slice from the application under analysis, but in essence it is a look-up of a "static" entity. The outcome of a look-up will result in one of three possible modifications to the graph, as presented 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.

Multiple value return. Of course, more then 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 key value and a special default value, if required. Since JSA is context-insensitive, if the special default value is ever returned from a look-up, our approach assumes that all default values may be possible. 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 Section 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.

The result of this generalization 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). While this paper employs this technique to resolve dynamic class loading, other static analyses may benefit from the same idea (e.g., by performing partial redundancy elimination based on looked-up values).

3.2 Modeling Generalizations

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 13 in the running example (Figure 1). This is due to JSA's

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

inability to accurately model all possible flows of string values. For example, JSA currently does not precisely track the flow of string values to/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 are only considering 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 that of [3] and [23].

During our manual investigation of the Java libraries, described in the next section, 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 `SunCodec` classes 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.

Even after increasing JSA's ability to model fields, it would still not be able to resolve the possible runtime values of `handlerClassName` from the running example. This is due to the limited number of variables types modeled 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 1, the lookup of environment variable *sun.awt.exception.handler* is accomplished by creating an instance of library class `sun.security.action.GetPropertyAction` (line 9). This is a convenience class that implements interface `java.security.PrivilegedAction`. Instances of `PrivilegedAction` are typically passed to invocations of `AccessController.doPrivileged`. This results in the execution of `PrivilegedAction.run` with privileges enabled. In the case of class `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 is not powerful enough to model objects with a compile-time type of `Object` that are actually of type `String`.

It is a very 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 a generalization through which JSA 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 are (1) assignment to another variable of type `Object`, (2) assignment from a variable with a compile-time type of `Object` that is actually of type `String`, (3) cast to a `String` variable, and (4) 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, the variable is conservatively corrupted, and transitively so are 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 libraries.

4 Experimental Evaluation

We implemented our proposed generalizations of JSA and evaluated the enhanced version's ability to resolve instances of dynamic class loading in the 10,238 classes from the Java 1.4 standard libraries. We identified 13 library methods that are used to dynamically load classes into the JVM; some examples are shown in Figure 3. These methods 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; we will refer to this parameter as the *target string*.

Manual investigation. To establish a "perfect baseline" for our results, we performed a manual investigation of the input 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 exam-

```
java.lang.Class.forName(<string>)
java.lang.ClassLoader.loadClass(<string>)
java.lang.ClassLoader.defineClass(<string>,...)
java.lang.ClassLoader.findClass(<string>)
java.lang.ClassLoader.findSystemClass(<string>)
java.lang.ClassLoader.findLoadedClass(<string>)
```

Figure 3. Some library methods used for dynamic class loading.

ple, a call to `Class.forName` with the string literal `"com.sun.media.sound.JavaSoundAudioClip"` was not included in the set of interesting hotspots, since it is trivial to resolve statically.[2]

Since our investigation was of the Java libraries, it was not possible to use a closed-world assumption. Instead, we present results under three different open-world assumption. Under such assumptions, it is impossible to determine the run-time values of certain method parameters and field variables 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)

Call 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). As mentioned earlier, the values of many target strings flow from methods which access the system's environment variables; call sites that were dependent on such strings were categorized as environment variable dependent (EVD). Finally, all remaining sites depended on run-time values that could not be resolved purely statically, or semi-statically with values of environment variables.

It is important to emphasize that this classification was performed using human intelligence. The results of this manual 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.

Assumptions	SD	EVD	DD
`Assumption 1`	40	33	15
`Assumption 2`	33	33	12
`Assumption 3`	18	30	3

Table 1. Results from manual investigation.

Table 1 shows the results of our manual investigation. `Assumption 1` assumed that client code could only affect the values of target strings through invocations of public methods and manipulations of public fields; further, it was assumed that none of the target string values were affected by the use of reflection either in client code or library code. `Assumption 2` was a much more natural assumption for the Java libraries: it assumed that client code and reflection could effect public and protected entities. `Assumption 3` was a fully open-world assumption. That is, it assumed that through the use of reflection all methods and fields could be manipulated by client code, potentially breaking encapsulation for private and package entities.

The results of our investigation indicate several key characteristic of dynamic class loading in the Java libraries. First, 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 `Assumption 1` to the most open `Assumption 3`, only 10% of the instances originally classified as EVD become client-dependent, as opposed to 55% of those classified SD and 80% for DD. This as a strong indication that such instances are meant to be static for most invocations. The second characteristic is that a large number of client-independent sites are indeed dependent on environment variables — those classified as EVD. Under the most natural `Assumption 2`, over 40% of dynamic class loading sites were classified as EVD. Such sites cannot be resolved by *any* purely static analsysis. To our knowledge, our approach is currently the only analysis that leverages these characteristics.

Evaluation of the proposed approach. Table 2 shows the results of four different versions of JSA applied to the Java 1.4 libraries. These implementations operate under `Assumption 2`. Row `SD` shows how many of the manually-classified SD sites were in fact identified by the analysis as being SD. Similarly, row

Version	JSA1	JSA2	JSA3	JSA4
SD	22	22	22 (67%)	30
EVD	0	14	26	30
TOTAL	22 (33%)	36 (55%)	48 (73%)	60 (91%)

Table 2. Number of resolved sites.

EVD shows the number of manually-classified EVD sites that were reported by the analysis as being EVD. Row TOTAL shows the total number of sites that were resolved by the analysis, either as SD or as EVD. The percentages in this row are relative to the total number of manually-classified SD/EVD sites from Table 1.

The first version was JSA in its original form.[3] The corresponding results are shown in column JSA1. Since this version did not incorporate our semi-static enhancement, it was able to resolve only call sites whose values were completely statically determinable. Thus, this state-of-the-art approach could resolve only 28% of all SD/EVD/DD sites, and only 33% of all SD/EVD sites. Column JSA2 shows the gains from enhancing JSA with the semi-static technique from Section 3.1. This addition enables JSA to resolve 64% more sites. The version from column JSA3 added the type generalization outlined in Section 3.2. Although this version did not increase the number of resolved SD instances, it nearly doubled the number of resolved EVD sites, by allowing more precise tracking of string values for names of environment variables (e.g., as illustrated by the call to `doPrivileged` in Figure 1). The final version, shown in column JSA4, added the more precise treatment of fields described in Section 3.2. As a result, the analysis was able to resolve 36% more SD sites and 15% more EVD sites.

Overall, the most general version resolved 77% of all SD/EVD/DD sites, and an impressive 91% of all SD/EVD sites; for the original version of JSA, the corresponding percentages are 28% and 33%.

The most general version was unable to resolve six 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, such as being parsed by a `StringTokenizer`.

Summary. A manual investigation of the Java libraries determined that over 40% of the client-independnent instances of dynamic class loading depend on values stored in environment variables, and therefore are impossible to resolve by any purely static analysis. Our experiments show that augmenting the current publicly available implementation of JSA with the generalizations proposed in this paper increases the number of resolved sites by 273% and successfully identifies 91% of all manually-classified SD/EVD sites.

[3]With minor error fixes, and some alterations to accommodate the open-world assumption.

5 Related Work

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

Jax [24] 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 [25] employs a simple string analysis technique that resolves calls to `Class.forName(<string>)` only if `<string>` is a string literal. Our work employs a far more powerful string analysis. Spark [12] 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 that the original manual check was performed on.

Our analysis builds upon the powerful string analysis presented in [2]. Christensen et al. recognized that their analysis could be used to resolved instances of dynamic class loading. They present a small case study that investigates their ability to resolve calls to `Class.forName`. Our work considers a much wider range of dynamic class loading methods, and their use in the entire Java library. Also, our generalizations greatly increase JSA's ability to resolve instances of dynamic class loading, as shown in Section 4.

The work of Braux and Noye [1] extends classic partial evaluation techniques [5, 10] to apply it 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 create system configuration specific compilations of applications in a much more automated fashion.

The work of Livshits et al. [15] proposes a tiered approach to the resolution of dynamic class loading and reflection. They present a static analysis algorithm which uses points-to information to determine the objects that could be loaded dynamically. Their algorithm tracks constant string values which flow to instances dynamic class loading and reflection. For string values that are not constant, or their analysis is not powerful enough to model, they use casting information. If casting information is not present, or a more 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.

None of the static analyses listed above are 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 libraries. The proposed semi-static approach was shown to be able to resolve many of these instances.

Some existing work [9, 20, 19] 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. The quality of these results depends heavily on the users' ability to fully exercise the application during the test executions. Consequently, the results of such analyses are not practical for use by some static analysis such as those employed for program transformations.

Many other analyses utilize the JSA library. The creators of JSA have employed it in several tools [3, 11] related to Java web technologies and XML documents. The JDBC-Checker tool [7, 6] builds upon JSA to verify the correctness of dynamically generated SQL query string. Similarly, the AMNESIA tool [8] uses JSA to identify all possible string values of SQL queries to aid in the detection and prevention of SQL-injection attacks. The work in [4] extends JSA in the implementation of their static analysis that recovers possible values of C-style strings in x86 executables. In [16], the JSA library is utilized in the implementation of an approach to understand software application interfaces through string analysis. To the best of our knowledge, no analysis other than [2] has employed JSA to resolve instances of dynamic class loading, nor have we been able to identify any that augment JSA with the generalizations proposed in our work.

6 Conclusions and Future Work

This paper presents a novel semi-static approach that utilizes the system's current configuration information to aid in the resolution of dynamic class loading in Java application. This technique produces results that are tailored to the system under analysis, by relaxing the restrictive and sometimes impractical constraints assumed by most purely static analyses. We further present several generalizations of string analysis, which allow better tracking of class names and environment variable names. By implementing these techniques in the current state-of-the-art string analysis for Java, in our experiments we were able to increase the analysis' ability to resolve instances of dynamic class loading by a factor of 2.7.

In the future we plan to extend our approach to incorporate other sources of system configuration information, such as configuration files. Various generalizations of string analysis could also be pursued, for issues such as context sensitivity, and the handling of 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 semi-static environment-aware approach, by employing techniques such as redundancy elimination or partial evaluation.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] W. G. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Int. Conf. Automated Software Engineering*, pages 174–183, 2005.

[9] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference on Object-Oriented Programming*, LNCS 3086, pages 96–122, 2004.

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

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.