Assumption Hierarchy for a CHA Call Graph Construction Algorithm

Jason Sawin
Mathematics and Computer Science
University of Puget Sound

Atanas Rountev
Computer Science and Engineering
The Ohio State University

***Abstract*—Method call graphs are integral components of many interprocedural static analyses which are widely used to aid in the development and maintenance of software. Unfortunately, the existences of certain dynamic features in modern programming languages, such as Java or C++, can lead to either unsoundness or imprecision in statically constructed call graphs. We investigate a hierarchy of assumptions that a Class Hierarchy Analysis (CHA) call graph construction algorithm can make about dynamic features in Java. Each successive level of the assumption hierarchy introduces new relaxations of suppositions. These relaxations allow the call graph algorithm to treat some uses of dynamic features more precisely and still remain sound. The hierarchy includes a novel assumption that dynamic features will respect encapsulation. We present an empirical study in which a unique call graph algorithm is implemented for each level of the assumption hierarchy. This study shows that assuming that dynamic features will respect encapsulation can lead to a call graph with 44% fewer edges than the fully conservative graph. By incorporating assumptions about casting operations and string values, it is possible to remain conservative and reduce the number of graph edges by 54% and graph nodes by 10% through the use of various resolution techniques. This work demonstrates that even a slight relaxation of assumptions can greatly improve the precision of a call graph. It further articulates the exact assumptions that a CHA call graph construction algorithm must make in order to use advanced resolution techniques.**

I. Introduction

Many modern software developers have come to rely heavily on static analyses. These analyses are employed in many modern compilers, IDE, and testing frameworks. A vital component for numerous interprocedural static analyses is a method call graph (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]). A call graph construction analysis produces a method call graph. This graph abstractly represents the calling relationships between program methods. The nodes of a call graph represent methods and directed edges represent calls between methods. Without a call graph, many interprocedural analyses simply cannot function.

The existence of dynamic Java features, whose exact run-time behavior cannot be determined by solely examining the static representation of the code, requires that call graph construction algorithms make assumptions about the possible run-time effects of such features. These assumptions will be reflected in the graphs generated by the algorithm. For example, consider a call graph construction algorithm that assumes the analyzed code does not make use of reflective features. These features allow an executing application to inspect itself and modify internal program properties. The graphs produced by this naive algorithm will be unsound for applications which do use reflection as the run time result of the reflective calls will not be represented. It has been shown that by disregarding dynamic features, call graphs may not encode a significant portion of the actual run-time calling relationships [12]. It is important for clients of a Java call graph analysis to be cognizant of assumptions that it makes about dynamic features. Such assumptions can have ramifications on the soundness of a client analysis.

We present a novel structure for exploring the consequences of the assumptions made by a call graph construction algorithm. Specifically, we present a hierarchy of assumptions that a *Class Hierarchy Analysis* (CHA) [13] call graph construction algorithm could make about certain dynamic features of Java. The hierarchy is rooted at the most conservative assumption, which provides a sound yet very imprecise treatment for dynamic features. Each consecutive level of the hierarchy extends the preceding level by adding more assumptions. These additional assumptions allow for a more refined treatment of dynamic features. Consequently, graphs created at each level of the hierarchy are subgraphs of those generated by preceding levels. The hierarchy terminates in a set of assumptions that allow for the use of (1) type information from cast operations, (2) static string values, and (3) dynamically gathered environment information to aid in the precise resolution of calls to dynamic class loading methods, reflective instantiation, and reflective invocation.

This work makes the following contributions:

- We propose a hierarchy of assumptions about dynamic Java features for a CHA call graph construction algorithm. This structure highlights the wide range of assumptions such an algorithm could make and the fact that many of these assumptions are interconnected. It is our belief that the presented assumption hierarchy could be extended to create a taxonomy from which existing and future call graph construction algorithms could be categorized.
- For each level of the hierarchy, we specify techniques the analysis can use to address certain dynamic features. These techniques include a novel approach based on the assumption that dynamic features will respect data encapsulation. They also include the use of a hybrid

string analysis [14]. We believe this to be the most precise string analysis to be incorporated into a CHA analysis to date.

- We implemented a version of the CHA analysis for each level of the assumption hierarchies. These implementations were applied to 10 real-world Java applications in an empirical study. This study demonstrates the effects of each assumption and the corresponding resolution techniques on the results of the analysis. The most precise implementation created graphs that on average, contain 10% fewer nodes and 54% fewer edges than those created by the fully conservative version.

This work demonstrates a carefully reasoned approach to evaluating the types of assumptions that most static analyses must make about dynamic features in Java. The hierarchical structure allows for the impact of each additional assumption to be evaluated separately.

II. CHA and Dynamic Java Features

Call graphs are often created statically by analyzing a representation of a program's source code. In object-oriented languages, such as Java and C++, *virtual method calls* adds a degree of difficulty to the static construction of a call graph. Virtual method calls arise due to the use of polymorphism through which it is possible for a subtype to override methods defined in its supertypes. The actual target of a virtual call is determined at run time. This poses a problem for a static analysis, since it does not benefit from execution information and must statically determine a conservative approximation of the possible run-time targets of virtual calls.

CHA is a technique that conservatively estimates possible receivers of dynamically-dispatched messages. For Java, CHA uses the following basic operation: given a call site $x.m()$, where the declared type of x is C, the possible run-time type of x must be a non-abstract subtype of C. The run-time target method for each such subtype can be determined by a bottom-up traversal of the class hierarchy.

Dynamic dispatch is but one problem when attempting to build an accurate call graph for complex programming languages. Below we describe several other dynamic features of Java which pose a significant challenges to a CHA call graph construction algorithm.

Custom class loaders are user extensions of `java.lang.ClassLoader`. Class loaders are software components that are responsible for loading Java class files into the Java Virtual Machine (JVM). They verify the structure of the class and initialize it by executing its static initialization method [15] (further referred to as `clinit`). Custom loaders are commonly used to specify alternative locations from which to load class files, instrument bytecode, and partition user defined classes in servers. Since they can redefine the semantics of a class at run time most static analyses are unsound in the presence of custom loaders.

Dynamic class loading features allow applications to load classes at run time. A commonly used dynamic class loading method is `Class.forName(x)` where `x` is a `String` specifying the fully qualified name of the class that is to be loaded (see [14] for a list of dynamic class loading methods available in the Java 1.4 libraries). These features return a `Class` which is a metadata object that describes the newly loaded class. They can also invoke the static initialization method `clinit` of the loaded class. As `clinits` can be arbitrarily complex, a call graph algorithm which ignores dynamic class loading could miss vital method interactions.

Reflective instantiation makes it is possible to create a new instance of any class loaded into the JVM by invoking `x.newInstance()` where `x` is of type `Class`. This action invokes the no argument constructor of the class represented by `x`. A class can also be instantiated by gaining access to a `java.lang.Constructor` object using `Class.getConstructor` then calling `Constructor.newInstance`. These invocations result in an implicit execution of a class constructor. These indirect calls will not be represented in a call graph created by a naive call graph construction algorithm.

Reflective invocation allows applications to dynamically invoke the methods of any class loaded into the JVM. Consider an object `x` of type `Class` that represents a class `C`. Through a call to `x.getMethod(...)`, it is possible to get a `Method` object for any method in `C`. `Method` is a reflective object that can be used to invoke the method it represents via `Method.invoke(Object,Object[ ])`. It implicitly invokes the represented method on `Object` argument with the parameters specified in `Object[ ]`. By ignoring reflection, these implicit method calls will not be represented in the call graph.

Native methods are methods that are written in *native programming languages*. To allow for across language interfacing, the Java platform provides its clients with the *Java Native Interface (JNI)* [16]. JNI is an interoperable interface that allows Java classes to *callout* to methods contained in native libraries. It also allows native code to make *callbacks* to Java methods. Through JNI, native method can load classes, create, inspect, and modify objects in a JVM. Essentially, a native method has all capabilities of a standard Java method.

It is important to note that both reflection and call backs from native methods do not have to respect encapsulation. It is possible for these features to circumvent the attribute visibility rules of Java (e.g., `private`).

III. CHA Assumption Hierarchy

Making conservative assumptions about effects of unknown external code is a common practice for static analyses (e.g., [17], [18], [19], [2], [20], [21], [22], [23]). Many

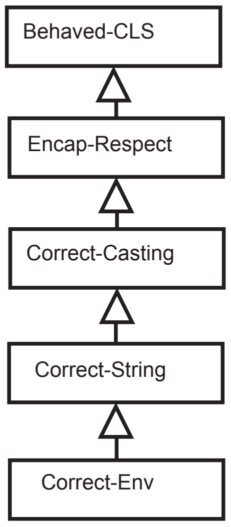

Figure 1. Assumption Hierarchy

static analyses for Java applications make assumptions (explicitly or implicitly) about the effects of the language's dynamic features. In this section, we present a hierarchy of assumptions for a CHA call graph construction analysis. These assumptions pertain to the dynamic features of Java discussed above. The hierarchy is displayed in Figure 1. As shown, each level of the hierarchy extends the preceding level. Thus, each level is the culmination of the assumption presented at that level and of all the assumptions made in the preceding levels. Note that the assumptions presented do not constitute all possible assumptions such analyses could make about these features nor do they cover all possible Java features which could cause the analysis to be unsound. Our assumptions are predicated on the assumption that the analysis has access to all possible classes that could be loaded during any execution of the program under analysis. This set of classes will be henceforth known as $AppClasses$.

A. Well-Behaved Loaders

Custom class loaders can alter the code of any class it loads. The only conservative assumption that can be made in the presence of such a loader is that every method could call every other method in the world. Fortunately, most applications will use class loaders that do not alter the semantics of the classes. A reasonable assumption about loaders would be:

> *Well-Behaved Class Loaders* (Behaved-CLs): It is assumed that the loaders will not alter the semantics of the classes they load and that all classes can be uniquely identified in a single namespace.

This assumption does not address other dynamic features of Java. To be fully conservative, certain implicit assumptions are being made. Specifically, the following assumptions are made for the given features:

- **Dynamic class loading features**: can invoke any `clinit` in $AppClasses$.
- **Reflective instantiation**: can call any constructor in $AppClasses$. It should be noted that calls to `Class.newInstance` are limited to default constructors and should be represented appropriately.
- **Reflective invocation**: can call any concrete method in $AppClasses$ excluding `clinits` and constructors.
- **Call to a native method**: An edge is added to `NativeMethod`, a synthetic method. This method contains an edge, representing a potential callback, to every `clinit`, constructor, and method in $AppClasses$.

B. Respecting Encapsulation

Behaved-CLs assumes that reflection and native methods will break encapsulation (e.g., frameworks that use reflection to conduct unit testing). However, for many applications reflection will respect encapsulation. For such applications the following assumption is appropriate:

> *Encapsulation Respecting Dynamic Features* (Encap-Respect): In addition to well-behaved class loaders it assumes that all dynamic features will respect encapsulation.

In Java, access level modifiers determine the visibility of program components. There are two levels of access control: (1) Top level—non-nested classes can be declared `public` or `package-private` (the default when no modifier is specified) and (2) Member level—fields, methods, and nested classes can be `public`, `protected`, `private` and `package-private`. The `public` modifier specifies that the entity is accessible to all classes. Entities which are `package-private` are only accessible in their own package. The modifier `protected` designates that members are accessible in their own package and by its subclasses. A member marked as `private` can only be accessed within its own class.

Under this assumption a little extra consideration must be given to nested classes. As a class member, member classes are given a visibility modifier. Nested classes can access all the members of outer classes, and outer classes have similar access to all members declared in nested classes. However, nested classes can access inherited `protected` methods of their encapsulating classes. An outer class cannot access the inherited `protected` methods of nested classes.

Under Encap-Respect the following assumptions must be made:

- **Dynamic class loading features**: it is assumed that these features can invoke any `clinit` in $AppClasses$[1].

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

- **Reflective instantiation**: can invoke any `public` constructor, any constructor marked `package-private` in its package, and any constructor declared in the surrounding class (including nested member classes and outer classes). If the surrounding class is a local nested class then it can call any constructor of other local classes declared in the surrounding method.
- **Reflective invocation**: can call any `public` method, any `package-private` method in its package, any method in the surrounding class (including all declared in nested member classes). If it resides in a nested class, then `Method.invoke` can call all `protected` and `private` methods of the outer classes (including those visible through inheritance) and any method of local classes declared in the surrounding method.
- **Call to a native method**: we view native members as external code entities and, therefore, assume that they will only access `public` methods and constructors. In addition, it is assumed that it could still call any `clinit`.

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

C. Correct Casting

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

> *Correct Casting Information Assumption* (Correct-Casting): This extension of Encap-Respect assumes that casts of reflectively instantiated objects will not cause a run-time exception.[2]

This assumption makes it possible to resolve any dynamic class loading site *DynoSite* with the following characteristics:

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

Since *DynoSite* is post-dominated by *NewSite*, and it is the only reaching definition of `x` at *NewSite*, and *DynoSite* is not a reaching definition of `x` at *DynoSite* (this can happen in loops), it must hold that every execution of *DynoSite* is followed by an execution of *NewSite*. It is also true that the class being instantiated by *NewSite* is the same that was loaded by *DynoSite*. A similar relationship is true for *NewSite* and the casting operation that post-dominates it. For dynamic class loading sites that meet these requirements, it can be inferred that the loaded classes must be of type `CastType` or a subtype of `CastType`. We call this set of types (`CastType` and its subtypes) the *resolving class set*, and we call the dynamic class loading sites *cast resolvable*. This set of characteristics is easily extended to dynamic class loading sites which are post-dominated by a *NewSite* of the form `o = con.newInstance()` where `con` is of type `Constructor`.

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

The type information from resolved dynamic class loading sites can be used to resolve reflective invocations. Consider the statement `m = c.getMethod(...)`. Assume that the only reaching definitions of `c` are a set of resolved dynamic class loading sites. Each of these resolved dynamic class loading sites has a set of resolved classes associated with it; we call the union of these sets $ResolvedDyno$. The method represented by `m` must be declared in a class, or a superclass of a class, contained in $ResolvedDyno$. Carrying this reasoning forward, if for a statement of the form `x = m.invoke(...)` all the reaching definitions of `m` come from resolved instances of `getMethod`, then it is possible, using similar logic, to determine the set of methods that could possibly be invoked; we call this set of methods the *resolving method set*.

With the capability to resolve certain dynamic features, the CHA call graph construction algorithm's treatment of these features becomes more precise.

- **Dynamic class loading features**: if the feature is cast resolvable, it is assumed that it could invoke any `clinit` in its resolving class set. Otherwise, it is assumed that it can invoke any `clinit` in $AppClasses$.
- **Reflective instantiation**: if the feature is cast resolvable, it is assumed that it could invoke any constructor in its resolving class set. Otherwise it is assumed that

[2]Livshits et al. [12] make a similar assumption so that they can resolve instances of dynamic class loading in their points-to analysis.

it has the same access as under the Encap-Respect assumption.

- **Reflective invocation**: if the feature is cast resolvable, it is assumed that it can call any encapsulation safe method in its resolving method set (e.g., it can call all `public` methods in the set). Otherwise it is assumed that it has the same access as under the Encap-Respect assumption.
- **Call to a native method**: treatment is the same as under Encap-Respect.

D. Correct String Values

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

Correct String Information (Correct-String): This extension of Correct-Casting assumes that (1) features such as reflection will not affect the string-typed formal parameters of `private` and `package-private` methods and `private` fields whose string values flow to dynamic class loading sites and (2) dynamic class loading sites will not throw a `ClassNotFoundException`.

This assumption contains all previous assumptions including that reflection will respect encapsulation. Correct-String further assumes that reflection contained in a package or class will not affect a very limited number of encapsulation safe method parameters and fields. It also assumes that all string values which flow to dynamic class loading sites designate a valid class which could be loaded.

The Correct-Strings assumption allows the CHA call graph construction algorithm to incorporate information from a string analysis to aid in the resolution of dynamic features. Specifically, we propose to use a modified version of the Java String Analyzer (JSA) [2]. This version of JSA is very conservative. It assumes that all methods in $AppClasses$ are reachable, and uses a CHA analysis (this is a separate analysis from Correct-String which is the client of JSA) to resolve only virtual calls that may affect the values of string variables. It further assumes that the only entities not affected by dynamic features are local string variables, string formal parameters of `private` and `package-private` methods, `private` fields and string variables returned by `private` and `package-private` methods, whose values flow to dynamic class loading sites. Since the assumptions JSA operates under are more conservative than the Correct-String assumption, a call graph operating under Correct-String can use the information for JSA without a loss of soundness.

The results of JSA can be integrated into the call graph algorithm to aid in the resolution of dynamic class loading sites. For example, consider a statement *loadSite* of the form `x = Class.forName(s)` where `s` is of type `String`. If JSA is able to determine a finite set $classNames$ of run-time values for `s`, it can be inferred that *loadSite* can load any valid class whose name is in $classNames$. Since Correct-String extends Correct-Casting, it assumes that type information from cast operations are correct. Therefore, if *loadSite* is cast resolvable, then string values in $classNames$ that do not specify the names of classes in the cast resolved class set of *loadSite* can be discarded. We call the set of valid classes the *string resolved class set* and we designate *loadSite* as *string resolvable*. It has been shown [14] that the type information from resolved dynamic class loading sites can be used to resolve certain instances of reflective instantiation. Instances of reflective invocation sites can also be resolved in the same manner as described above for Correct-Casting.

Under Correct-String the following assumptions are made:

- **Dynamic class loading features**: if the feature is string resolvable, it is assumed that it could invoke any `clinit` in its string resolving class set. Otherwise, it is treated as specified by Correct-Casting.
- **Reflective instantiation**: if the feature is string resolvable, it is assumed that it could invoke any constructor declared by a class, or a superclass of a class, in its string resolving class set. Otherwise, it is treated as specified by Correct-Casting.
- **Reflective invocation**: if the feature is string resolvable, it is assumed that it can call any encapsulation safe method in its string resolving method set. Otherwise, it is treated as specified by Correct-Casting.
- **Call to a native method**: treatment is the same as under Encap-Respect.

E. Dynamically Gathered Environment Information

Previous work [24], [14] has shown that the incorporation of dynamically gathered environment information can increase JSA's ability to resolve dynamic class loading and reflective instantiation sites in the Java 1.4 standard libraries. The following assumption allows this dynamically gathered information to be used to increase the precision of the CHA algorithm:

Correct Environment Assumption (Correct-Env): This extension of Correct-Strings assumes that (1) dynamic features will not affect the string-typed formal parameters of `private` and `package-private` visible methods and `private` fields whose values flow to environment variable access methods and (2) the values of environment variables which can affect

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

Table I
BENCHMARKS STATISTICS: NUMBER OF CLASSES, METHODS, JIMPLE STATEMENTS, INVOCATIONS OF DYNAMIC CLASS LOADING METHODS, `newInstance` METHODS, `Method.invoke`, AND NATIVE METHODS.

dynamic class loading sites will be the same at analysis time and at run time.

This assumption allows our version of JSA to gather string values from environment variables at analysis time. For example, custom-defined event handlers can be specified in the `sun.awt.exception.handler` environment variable. This environment variable will hold the string value of the custom event handler's full qualified name. At run time the handler will be dynamically loaded and accessed through reflection. Java has a well defined API which allows users to access environment variables (e.g., `System.getProperty`).

Just like the static version of JSA and Correct-Strings, the hybrid version of JSA[3] operates under a much more conservative assumption than Correct-Env so its information can be used by a client call graph analysis operating under Correct-Env without a loss of soundness. Making this assumption about environment variables means that the generated call graph will be tailored to the system configuration (i.e., set of environment variable values) under analysis. It will no longer be sound in a global context. Instead, it will only be sound for systems where the environment variables are the same as those observed by the call graph analysis.

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

IV. EXPERIMENTS

To evaluate how the assumptions outlined above will affect the results of a CHA analyses, we performed the following empirical study. We implemented a unique version of a CHA call graph construction algorithm for each level of the assumption hierarchy presented in Section III and applied them to 10 real-world benchmark applications. Our implementations were built upon the Soot analysis framework [25] which generates a Jimple intermediate representation of the application.

[3]The hybrid version of JSA executes very small portions of the application under analysis in order to look up referenced environment variables.

Table I presents the benchmarks that were used in this study. For each benchmark **Classes** shows the number of class files that are unique to the application; this number does not include library classes that may be referenced by the application. Column **Meths** presents the number of methods (including constructors and `clinits`) contained in the application classes and column **K Jimple** displays the number of Jimple statements they contain. **Dyno** presents the number of invocations of dynamic class loading methods present in the Jimple representation of the application's classes. Similarly, **NewInst** and **Invoke** display the number of invocations of `newInstance` (both `Class` and `Constructor`) and `Method.invoke`, respectively. Column **Native** displays the number of native methods declared in the application.

Table II presents the number of nodes and edges in the call graphs created by each implementation of the analysis. These call graphs include not only methods declared in the application but methods called in the Java 1.4 standard library. Column **Behaved-CLs** presents the results of the most conservative implementation. This version is used as the baseline to which the other implementations are compared. The graph generated by Behaved-CLs for a particular application is a supergraph of all other implementations' graphs for that same application. Row **AVE**Δ_N displays the average reduction in the number of nodes in the graphs generated by the corresponding implementation, compared to Behaved-CLs. Row **AVE**Δ_E contains similar information but with respect to the number of edges.

A. The Nodes

The nodes of the call graph represent methods that are reachable from the `main` method of an application. By assuming that dynamic features will respect encapsulation, an average of 10% of the nodes are removed from the fully conservative graph. With the addition of each technique—using (1) type information from casting operations, (2) constant string values, and (3) environment variable string

	Number of Nodes				
Apps	**Behaved-CLs**	**Encap-Respect**	**Correct-Cast**	**Correct-String**	**Correct-Env**
DB	32537	29060	29046	29045	29044
Javac	33662	30180	30167	30166	30165
JEdit	39083	35640	35625	35624	35624
JGap	32890	29404	29391	29390	29388
Jpws	34663	31162	31148	31148	31145
Mindterm	33268	29779	29764	29762	29761
Muffin	33976	30416	30400	30400	30399
SableCC	33807	30364	30349	30348	30347
VietPad	33647	30157	30144	30144	30143
Violet	33577	30143	30130	30129	30126
AVEΔ_N	–	10.2%	10.2%	10.2%	10.2%
	Number of Edges				
DB	1696889	921170	831580	779502	774273
Javac	1761197	967652	874079	820521	815220
JEdit	2626960	1513423	1401168	1075290	1069333
JGap	1792150	975737	880632	819586	814309
Jpws	1823623	1005794	907518	848529	842952
Mindterm	1743252	951832	854017	800711	795385
Muffin	1800537	985564	886112	826637	821207
SableCC	1466806	845363	749460	730955	725507
VietPad	2019676	1076309	981160	920010	914611
Violet	1834669	997052	896791	842490	836950
AVEΔ_E	–	44.8%	50.2%	54.1%	54.4%

Table II
CHA CALL GRAPH CONSTRUCTION ALGORITHM RESULTS: NUMBER OF NODES AND EDGES IN THE GRAPH CREATED BY THE CORRESPONDING VERSION. **AVE**Δ IS THE AVERAGE PERCENTAGE DECREASE WITH RESPECT TO COLUMN **Behaved-CLs**.

values—the number of nodes in the corresponding graphs are reduced, but not significantly. This trend can be seen in columns **Correct-Casting**, **Correct-String**, and **Correct-Env**. The reason that more nodes are not removed is due to the treatment of unresolved `Method.invoke` calls and calls to native methods. Starting at Correct-Env in the assumption hierarchy it is assumed that both native methods and unresolved `Method.invoke` calls could potentially call all public methods. This has the effect of making all public methods reachable. Consequently the only methods that will not be reachable (and not represented in the graph) are those not reachable from a public method. Every execution of the analysis encountered both unresolved instances of `Method.invoke` and calls to native methods either in application classes or classes in the Java 1.4 standard library.

B. The Edges

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

C. Resolution of Dynamic Features

Table III presents the average percentage of reflective sites each version was able to resolve for all benchmark applications. Row *Dyno Loading* displays the average percentage of dynamic class loading sites each version was able to resolve. Row *newInstance* shows the percentage of calls to `newInstance` (for both `Class` and `Constructor` objects) that were resolved. Row *Invoke* presents the percentage of `Method.invoke` calls that were resolved.

The use of casting information appears to be only moderately effective at resolving instances of dynamic class loading. Correct-Casting could only resolve an average of 16% of such sites. However, it is much more successful at resolving calls to `newInstance` (on average it was able to resolve 56% of such sites). One of the reasons this technique was not more successful in resolving instances of dynamic class loading is due to the fact that the post-dominance and reaching definitions analyses used by Correct-Cast were intraprocedural. In most instances, if a reflective instantiation of a class is used in a casting operation, it is instantiated and casted in the same method. However, it is not uncommon for the `Class` objects from dynamic class loading sites to flow through multiple methods before being instantiated. Additionally some of the dynamic class loading sites were not post-dominated by cast operations thus casting information could not be used.

Correct-String incorporation of a static string analysis enabled it to be much more successful at resolving dynamic class loading sites. On average, it resolved 46% of the dynamic loading sites encountered. This increased precision enabled it to resolve 58% of the `newInstance` sites it encountered. It was not able to resolve sites that depended on (1) string values that flowed from formal parameters of `public` or `protected` methods which were not used under the assumption that these values could be affected by unresolved reflective calls and native code, (2) string values passed through structures, such as `HashMaps`, which the string analysis is not powerful enough to model, or (3) string values that flowed from dynamic sources (e.g., environment variables or configuration files.)

Correct-Env extends Correct-String with a hybrid string analysis. This version performs a lookup of environment variables whose values flow to dynamic class loading sites (see [14] for a detailed description.) The addition of dynamically gathered environment information allowed Correct-Env to resolve 50% of all dynamic class loading sites and 61% of `newInstance` sites.

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

V. Related Work

The challenges caused by the use of dynamic class loading, reflection and native methods have been addressed in other static analyses with various degrees of sophistication. In this section we present a few of the most relevant approaches.

Some analyses (e.g., [26], [27], [25], [12], [28]) give their clients the option to manually specify the precise effects of certain dynamic features. This method is only as sound and precise as the information entered by the user. The identification of these features and their effects can be laborious and challenging as many of them will exist in third party libraries. Our approach is automatic and conservative.

The CHA call graph construction in the Soot analysis framework [25] allows the user to choose from several options that specify the treatment of certain dynamic features. The most conservative option provides treatment for dynamic class loading calls of the form `Class.forName(String)`, and `newInstance` calls of the form `Class.newInstance()`. It resolves `Class.forName(lit)` calls where `lit` is a string literal value. If it is unable to resolve a call to `forName`, it will add an edge to all `clinit` methods in $AppClasses$. For `Class.newInstance` calls, it adds an edge to every no argument constructor in $AppClasses$. Soot's CHA analysis does not address calls to native methods, reflective invocations, calls of the form `Constructor.newInstance(...)`, or dynamic class loading methods other than `Class.forName`. Our analysis provides a conservative treatment for all of these features. We feel that static analysis frameworks such as Soot could benefit from our hierarchical approach to assumptions about dynamic features. Our structure allows clients to precisely choose the level of risk they are willing to assume.

Our analysis uses a hybrid version of the powerful string analysis JSA created by Christensen et al. [2]. They present a small case study that investigates the ability to resolve calls to `Class.forName`. In [14], [24] we extended JSA to considers a much wider range of dynamic class loading methods and include dynamically gathered environment variable information. We showed that this information greatly increase JSA's ability to resolve instances of dynamic class loading which in turn allowed us to resolve more instances of reflective instantiation [14]. The work presented in this paper represents the first time that information from a hybrid string analysis was incorporated into a CHA call graph construction algorithm.

The work of Livshits et al. [12] proposes a tiered approach to the resolution of dynamic class loading and reflection that is similar to our approach. They present a static analysis algorithm that uses points-to information to determine the objects that could be loaded dynamically. Their algorithm tracks constant string values that flow to instances of dynamic class loading and reflection. For cases where they are unable to resolve the target string's value, they utilize casting information. If such information is not present, their approach relies on user specifications. We use similar techniques in a CHA call graph construction algorithm. Our techniques could enhance the automation and precision of their analysis. We employ a more advanced string analysis and incorporate information that currently has to be manually provided to their analysis by the user. Our encapsulation safe treatment for unresolved instances of dynamic features also provides an alternative to user

Features	Correct-Cast	Correct-String	Correct-Env
Dyno Loading	16%	46%	50%
newInstance	56%	58%	61%
Invoke	0%	6%	6%

Table III
RESOLUTIONS RESULTS: AVERAGE PERCENTAGE OF RESOLVED INSTANCE OF DYNAMIC FEATURES.

specifications.

Some existing work [29], [30], [31], [32], [33], [34] circumvents the typical shortcomings of static analyses by developing online algorithms. All of the above approaches require either (1) modifications to the JVM services that handle dynamic class loading and reflection or (2) instrumentation of application code. These alterations allow the analyses to observe the actual execution of an application, which can be used to resolve any ambiguity introduced by the use of dynamic class loading. However, as with any purely-dynamic analysis, the results are unsound and represent only properties of the observed execution, not of all possible executions. Our most relaxed implementation of CHA a has a more restricted form of unsoundness, as defined in [14].

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

VI. CONCLUSION AND FUTURE WORK

We present a hierarchy of assumptions a *Class Hierarchy Analysis* call graph construction algorithm could make about the dynamic features of Java. At the top of the hierarchy is the most conservative assumption which generates an imprecise call graph for an application making use of dynamic features. Each consecutive level of the hierarchy represents a slight relaxation of the preceding level. Consequently, a graph created under each level of the hierarchy is a subgraph of the one generated by the preceding level. These relaxations allow the algorithm to incorporate various techniques that attempt to precisely resolve instances of dynamic class loading, reflective invocation, and reflective instantiation.

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

An obvious extension of this work would be to apply similar assumptions and resolution techniques to a more precise call graph construction algorithm. A natural choice would a Rapid Type Analysis [35] call graph construction algorithm. It would also be interesting to explore new techniques for the treatment of native methods and reflective invocation as well as techniques for newer versions of Java.

VII. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful insights and comments. This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award.

REFERENCES

[1] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

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

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

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

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

[6] N. Glew and J. Palsberg, "Method inlining, dynamic class loading, and type soundness." *Journal of Object Technology*, vol. 4, no. 8, pp. 33–53, 2005.

[7] N. Glew, J. Palsberg, and C. Grothoff, "Type-safe optimisation of plugin architectures," in *Static Analysis Symposium*, 2005, pp. 135–154.

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

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

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

[11] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott, “Robustness testing of Java server applications,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 292–311, 2005.

[12] B. Livshits, J. Whaley, and M. Lam, “Reflection analysis for Java,” in *Asian Symposium on Programming Languages and Systems*, ser. LNCS 3780, 2005, pp. 139–160.

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

[14] J. Sawin and A. Rountev, “Improving static resolution of dynamic class loading in Java using dynamically gathered environment information,” *International Journal of Automated Software Engineering*, vol. 16, no. 2, pp. 357–381, Jun. 2009.

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

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

[17] M. J. Harrold and G. Rothermel, “Separate computation of alias information for reuse,” *IEEE Transactions on Software Engineering*, vol. 22, no. 7, pp. 442–460, Jul. 1996.

[18] A. Rountev, B. G. Ryder, and W. Landi, “Data-flow analysis of program fragments,” in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. LNCS 1687, 1999, pp. 235–252.

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

[20] A. Rountev, “Precise identification of side-effect-free methods in Java,” in *IEEE International Conference on Software Maintenance*, 2004, pp. 82–91.

[21] A. Rountev, A. Milanova, and B. G. Ryder, “Fragment class analysis for testing of polymorphism in Java software,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 372–387, Jun. 2004.

[22] J. Xue and P. H. Nguyen, “Completeness analysis for incomplete object-oriented programs,” in *International Conference on Compiler Construction*, 2005, pp. 271–286.

[23] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, “Practical extraction techniques for Java,” *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 6, pp. 625–666, 2002.

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

[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*, ser. LNCS 1781, 2000, pp. 18–34.

[26] 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*, 1999, pp. 292–305.

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

[28] O. Lhoták and L. Hendren, “Scaling Java points-to analysis using Spark,” in *International Conference on Compiler Construction*, ser. LNCS 2622, 2003, pp. 153–169.

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

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

[31] F. Qian and L. Hendren, “Towards dynamic interprocedural analysis in JVMs,” in *Virtual Machine Research and Technology Symposium*, 2004, pp. 139–150.

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

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

[34] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind, “Fast online pointer analysis,” *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 2, p. 11, 2007.

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