Dynamic Loading And Reflection For Context Sensitive Inter-Procedural Analysis

Introduction to Dynamic loading:

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory. It is one of the 3 mechanisms by which a computer program can use some other software; the other two are static linking and dynamic linking. Unlike static linking and dynamic linking, dynamic loading allows a computer program to start up in the absence of these libraries, to discover available libraries, and to potentially gain additional functionality.

Uses:

Dynamic loading is most frequently used in implementing software plugins. For example, the Apache Web Server's *.dso "dynamic shared object" plugin files are libraries which are loaded at runtime with dynamic loading. Dynamic loading is also used in implementing computer programs where multiple different libraries may supply the requisite functionality and where the user has the option to select which library or libraries to provide.

Description:

Languages like Java allow dynamic loading of classes. It is impossible to analyze all the possible code executed by a program, and hence impossible to provide any conservative approximation of call graphs or pointer aliases statically. Static analysis can only provide an approximation based on the code analyzed. Remember that all the analyses described here can be applied at the Java byte-code level, and thus it is not necessary to examine the source code. This option is especially significant because Java programs tend to use many libraries.

Even if we assume that all the code to be executed is analyzed, there is one more complication that makes conservative analysis impossible: reflection.

Reflection allows a program to determine dynamically the types of objects to be created, the names of methods invoked, as well as the names of the fields accessed. The type, method, and field names can be computed or derived from user input, so in general the only possible approximation is to assume the universe.

The code below shows a common use of reflection:

String className = ...;
Class c = Class.forName(className); Object o = c.newInstanceO ;
T t = (T) o;

The **forName** method in the **Class** library takes a string containing the class name and returns the class. The method **newInstance** returns an instance of that class. Instead of leaving the object o with type **Object,** this object is cast to a superclass T of all the expected classes.

While many large Java applications use reflection, they tend to use common idioms. As long as the application does not redefine the class loader, we can tell the class of the object if we know the value of className. If the value of className is defined in the program, because strings are immutable in Java, knowing what className points to will provide the name of the class. This technique is another use of points-to analysis. If the value of className is based on user input, then the points-to analysis can help locate where the value is entered, and the developer may be able to limit the scope of its value.

Context-sensitive analysis:

– Re-analyzes callee for each caller.

– Also known as polyvariant analysis.

Interprocedural analysis:

– Gather information across multiple procedures (typically across the entire program).

– Can use this information to improve intraprocedural analyses and optimization (e.g., CSE).

Reference:

Bibliography:

Compilers: **Principles**, **Techniques**, and **Tools**, 2nd Edition. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.