WALA Notes

Vaibhav Surendra Singh, Krishna Chaitanya Sripada

April 13, 2015

1 Setting up WALA

To start using wala we need to build it locally first. Wala source code can be obtained from here - https://github.com/wala/WALA

This code can be built using either eclipse or maven. But before building it there are some properties that need to be added. The Instructions for editing these can be found in the walawiki getting started page. http://wala.sourceforge.net/wiki/index.php/UserGuide:Getting_Started

The wala libs we will be needing for transformation are:

1. com.ibm.wala.core - core WALA framework support
2. com.ibm.wala.shrike - Shrike bytecode manipulation library
3. com.ibm.wala.util - general utilities

NOTE: When using Windows make sure to use '\\' for file seperators as the paths are read by wala as string and '\' is an escape character.
Once these libraries are built we can add these projects as dependency to other eclipse project or we could also create a jar file with these three libs and use it as a dependency.

2 Android Call Graphs and Pointer Analysis

For 'MUSE' project we are using **droidel** to create Android App call graphs from jar files. 'Droidel' is a tool created by Samuel Blackshear for running Program Analysis on Android Apps using wala. Droidel handles it is by extracting classes.dex from apk file. 'classes.dex' is the dexler byte code used by the Dalvik VM on the Android phones. Wala is unable to read this format of byte code directly so the dexler bytecode is converted to JVM byte code using dex2jar tool. Android tool chain converts the java source files first to JVM byte code class files and then to classes.dex and packs it in jar using the dex2jar tool. In our case we do the reverse i.e. convert dex byte code to JVM compatible byte code using dex2jar tool.

The wala libraries can now read the byte code easily and create the call graphs depending on the Activity and Activity Lifecycle methods.

There are other wala libraries available that can create call graphs too. Here we will explore how to build call graphs without using any library other that com.ibm.wala.core.

2.1 Step 1: Analysis Scope

We first need wala to read our activity class file and create an analysis scope. This is done by following two lines of code:

```
// initialize scope
AnalysisScope scope = AnalysisScopeReader.makeJavaBinaryAnalysisScope("MainActivity.class
    ", new File("excludes.txt"));
scope.addToScope(ClassLoaderReference.Primordial, new JarFile(new File("android.jar")));
}
```

Ideally what we would like to do is find out which android API the app uses from the Android_MANIFEST.xml and from this information we can now find the appropriate jar from the Android SDK and use it in the above code. The Manifest file will also tell us the Activities present in the App and which of these is the main or the starting activity.

(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:AnalysisScope for more information on Analysis Scope)

2.2 Step 2: Class Hierarchy and Adding Entry Points

Once we have created the AnalysisScope we now need to create entry points for the program. This is done in the code using wala as follows:

```
//make class hierarchy from scope
ClassHierarchy cha = ClassHierarchy.make(scope);
// find entry point onCreate in MainActivity
IClass c = cha.lookupClass(TypeReference.findOrCreate(ClassLoaderReference.Application,
                                          "Lcom/example/settagtest/MainActivity"));
List<Entrypoint> entries = new ArrayList<Entrypoint>();
Atom atom = Atom.findOrCreateUnicodeAtom("onCreate");
Collection<IMethod> allMethods = c.getAllMethods();
for (IMethod m : allMethods) {
       //if method==onCreate && ClassLoader==Application
       if (m.getName().equals(atom) &&
          (m.getDeclaringClass().getClassLoader().getReference().getName().equals(Atom.
             findOrCreateUnicodeAtom("Application")))) {
              //add the entry point
              entries.add(new DefaultEntrypoint(m, cha));
       }
}
```

(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:ClassHierarchy for more Information on Class Hierarchy)

2.3 Step 3: Creating Call Graphs and Points to Analysis

Now once we have added the entry point we can go ahead and create the call graph using wala as below:

```
AnalysisOptions options = new AnalysisOptions(scope, entries);
// build the call graph
com.ibm.wala.ipa.callgraph.CallGraphBuilder builder = Util.makeZeroCFABuilder(options,
   new AnalysisCache(), cha, scope, null, null);
CallGraph cg = builder.makeCallGraph(options, null);
//Print Call Graph statistics
System.out.println(CallGraphStats.getStats(cg));
```

(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:CallGraph for more information on Call Graphs)
(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis for more information on Pointer Analysis)

2.3.1 CG Node Analysis using IR

The CGNode class has a method getIR(). This method returns the WALA IR (Intermediate Representation) of the Instructions in the method that defines that node. Also we can get other information like return type, parameter Types, method name, declaring class etc from the CGNode for each Node. From IR we can even get Source Code Line Number Information.

WALA is not a compiler; it does not have a code generating back end and the IR is not designed for transformations. We do not know of any implementation that generates bytecode directly from the WALA SSA IR. A plausible option is to do the analysis on the IR, map the analysis results back to the Shrike representation, and do bytecode transformations in Shrike.

(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:IR for more information on IR)

3 Mapping IR to byte and source code

Given the index i of some SSAInstruction in an IR is produced from bytecode, you can get the corresponding bytecode index and source line number as follows:

```
IBytecodeMethod method = (IBytecodeMethod)ir.getMethod();
int bytecodeIndex = method.getBytecodeIndex(i);
int sourceLineNum = method.getLineNumber(bytecodeIndex);
```

To get name of Source File for a IMethod m find the declaring IClass klass. This will give the name of the source file.

```
IClass klass = ir.getMethod.getDeclaringClass();
String SrcFileName = klass.getSourceFileName();
```

For getting names of variables from Value Numbers use

```
IR.getLocalNames()
```

(Visit http://wala.sourceforge.net/wiki/index.php/UserGuide:MappingToSourceCode for more information on Source Information)

4 Transforming Byte Code

Shrike is the WALA Library used for instrumention Java Byte Code. In earlier sections we have seen how to create CFGs and Pointer Analysis from Java Byte Code. Using the analysis we can come up with the transformations that are needed to change an app. To do so we will need to transform byte Code. As discusseded earlier, one of the ways for analysing code is to change it to IR which makes analysis extremely easy. But the problem with IR is that it is immutable and there is no implementation provided by wala to modify IR and translate those to byte code. This is where Shrike comes in. We will need to map the IR to bytecode and then use Shrike to do the transformation. As mentioned earlier WALA can read JVM byte code only and hence it needs to be converted from dexler byte code using dex2jar tool or by using class files generated in build folder by IDEs like Eclipse or Android studio while creating the apk files. As it only reads JVM byte code it naturally can generate the output files in JVM byte code format only. Shrike has two main sub libraries - ShrikeCT is used to read and write class files whereas classes from ShrikeBT are used for Transforming Byte Code.

Key “ShrikeCT” classes for reading and writing .class files:
ClassReader: provides an immutable view of .class file data. Reads the data lazily.
ClassWriter: generates the JVM representation of a class.

Key ”ShrikeBT” classes for manipulating bytecodes:
MethodData: information about a method
MethodEditor: core class for transforming bytecodes via patches

ClassInstrumenter: convenient way to instrument an existing class. You can grab methods via a ClassReader from getReader(), instrument them, and then call emitClass() to obtain a ClassWriter populated with the modified class.
CTCompiler: for compiling a ShrikeBT method into bytecodes. Useful if you’re creating your own method from scratch. Also see CTUtils.compileAndAddMethodToClassWriter() and MethodData.makeWithDefaultHandlersAndInstToBytecodes().

4.1 Step 1: Reading class files

Once we've read it we give it to the ClassInstrumenter. The ClassInstrumenter is used to pull out data from methods and then we modify this data by applying patches and output the modified class. So we initialise the Instrumenter and visit the method to be modified. Let the number of the method that is being modified be saved in i.

```
int i = ...;
//visit ith method to instrument it
ClassInstrumenter instrumenter = new ClassInstrumenter(methodName, reader, null, true);
MethodData data = instrumenter.visitMethod(i); // this returns i'th method's data
```

4.2 Step 3: Edit method using MethodEditor

Now we will try to edit this method. This is done using a Method Editor as below:

```
//Now that we found the method and are visiting it
//We will instrument the code here to fix the setTag invokation
MethodEditor editor = new MethodEditor(data);
editor.beginPass();
```

4.3 Step 4: Create Patches

Once we begin editing the method using beginPass, we will have to create patches and add them somewhere in the method. MethodEditor allows us to add the patches at the Start, at the end, before or after an index or replace an Index.

```
//add patches here
editor.replaceWith(offset, new SetTagPatch());
```

The Patch can be created by implementing the MethodEditor.Patch class. A simple implementation is shown below:

```
//Creating a Patch
class SetTagPatch extends MethodEditor.Patch{
       @Override
       public void emitTo(Output w) {
              //emit the new setTag Instruction
              w.emit(LoadInstruction.make(Constants.TYPE_Object, 2));
              w.emit(LoadInstruction.make(Constants.TYPE_Object, 3));
              w.emit(Util.makeInvoke(TextView.class, "setTag", new Class[] {java.lang.
                  Object.class} ));
       }
}
```

Here we use the MethodEditor.Output class to emit ByteCode Instructions. There are three Instructions being emitted here. First one loads the object from stack on which setTag method is called. The second loads the parameter for the method. Third once invokes the setTag method using the objects loaded in earlier Instructions. If we want to remove a line then, just like here we use the replaceWith method but do not emit anything using the Output class. This will remove the bytecode instruction and it is replaced with nothing.

4.4 Step 5: Apply and verify Patches

Once several patches are created and mentioned where they must be applied, we apply the patches and stop the editor from editing.

```
//apply the patches here
editor.applyPatches();
editor.endPass();
```

The following code snippet is used to verify the changed we made to the method data.

```
//verify the changes
try {
       (new Verifier(data)).verify();
} catch (Verifier.FailureException ex) {
       System.out.println("Verification failed at instruction "+ ex.getOffset() + ": " +
           ex.getReason());
}
```

4.5 Step 6: Create the modified class file

Once we have verified the changes we must convert the changes to bytes and write them to a file. From Instrumenter we first create a ClassWriter and then convert the class data to bytes using makeBytes method.

```
//Now that the patch is applied, write the modified byte code to a file
ClassWriter writer = instrumenter.emitClass();
modifiedBytes = writer.makeBytes();
```

We now write the modified bytes to a class file as follows:

```
//write the modified bytes to a class file
Path path = Paths.get("output\\" + filename);
Files.write(path, modifiedBytes);
```

(Visit http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview for more information on Shrike)

4.6 Instruction Creation

The types of Instructions that can be used to create Patches are:

```
ArrayLengthInstruction, ArrayLoadInstruction, ArrayStoreInstruction, BinaryOpInstruction,
    CheckCastInstruction, ComparisonInstruction, ConditionalBranchInstruction,
   ConstantInstruction, ConversionInstruction, DupInstruction, GetInstruction,
   GotoInstruction, InstanceOfInstruction, InvokeInstruction, LoadInstruction,
   MonitorInstruction, NewInstruction, PopInstruction, PutInstruction, ReturnInstruction
   , ShiftInstruction, StoreInstruction, SwapInstruction, SwitchInstruction,
   ThrowInstruction, UnaryOpInstruction
```

Using XxxInstruction.make() a new byte code instruction can be made. Any one of the 200+ instructions supported by JVM can be created using the above classes.

According to wala

> Instructions are immutable objects. It is always legal to take a reference to an instruction and use it in any other context. You never need to copy instructions. It is often a good idea to keep references to frequently used instructions cached in static fields. To generate an Instruction, locate the class for the instruction you want to generate, and invoke the appropriate "make" static method on that class. The Util class has convenience methods for creating some of the more complex instructions using reflection to fill in some of the needed information (e.g., makeGet, makePut, makeInvoke). We simplify the bytecode instruction set a bit using some preprocessing and postprocessing: There is no 'iinc' instruction. 'iinc' instructions are expanded to 'iload; bipush; iadd; istore' during decoding and reassembled during compilation. There are no 'jsr' or 'ret' instructions. Bytecode subroutines are expanded inline during decoding. There are no 'ifeq', 'ifne', 'ifgt', 'ifge', 'iflt', 'ifle' instructions. These

instructions are expanded to 'bipush 0; if_icmp' during decoding and reassembled during compilation. There are no 'ifnull' or 'ifnonnull' instructions. These instructions are expanded to 'aconst_null; if_acmp' during decoding and reassembled during compilation. All Java values, including longs and doubles, occupy just one word on the stack. Places where the JVM assumes differently are fixed up during compilation. However, longs and double still take up two local variable slots (this usually doesn't matter to instrumentation). Control transfer instructions refer to target instructions using integers. These integers are usually indices into an array of instructions.

5 SetTag Example

We will now need to consider an example to showcase the bug detection and the subsequent transformation done to fix the bug.

5.1 The Bug

This bug is basically a memory leak bug. It happens due to calling the setTag function from class View or from the class that extends it. The example of incorrect use tag is shown as below:

```
ViewObj.setTag(R.id.abc, ViewObj);
```

Here we observe that the argument and the object calling the setTag method have a direct link. This creates a memory leak issue due to the way the setTag method with two parameters was written. To fix this issue the setTag call to single parameter must be made as below or alternately the link between object and argument must be broken which is not possible in the above example.

```
ArrayList list = new ArrayList();
x=list.add(tv);
// call the setTag
ViewObj.setTag(list);
```

In the above code the memory leak will not occur. We now need to use wala to identify and fix this bug.

5.2 Identifying the bug

To identify if an app has this bug we will need to find out if the app uses a call to setTag method with two parameters. This can be done by creating call graph and looking for a node that issues a call to setTag method with two parameters. Once such a call is found we then check to see if there is a link between the object calling the method and the second argument. It is exists then we would have safely detected the bug. So we need just the call graph to detect the bug.

5.3 Information needed for transformation

To fix the bug we need

1. The method calling the setTag method incorrectly
2. Name of the class file and
3. The offset of setTag call in the IR of the method using the incorrect setTag call.

Once we get the call site (parent CG Node) for method from that we can find the name of the method and the declaring class. From declaring class we can get the name of the java file and using that guess the class file name.
Also from the parent CG Node we can get the IR of that method and from that find the IR offset of setTag method in the parent method. From this offset we can get the bytecode offset and java source line number. The bytecode offset will be needed for applying the patches.

5.4 Fixing the bug

We can now use class reader to read class file we identified for transformation and then visit the method that needs to be fixed. Here we have the offset for setTag, the patch needs to be applied here. The code for correct SetTag call has been shown above while discussing patches. Apart from that we need to remove the 3 byte code lines preceding the setTag call. This can be done as follows.

```
//find setTag offset where fix is needed
int offset = this.findSetTagOffset(data);

//add patches here
// blank out the instructions that load the objects for setTag call
for(int j = 1; j <= 3; j++){
       editor.replaceWith(offset-j, new MethodEditor.Patch(){
              @Override
              public void emitTo(Output w) {
                     //replace the Instruction with Nothing
              }
       });
}
//add the new SetTag call
// this patch loads and calls the setTag method
// in a way that memory won't leak
editor.replaceWith(offset, new SetTagPatch());
```

Prior to applying the patch the bytecode for setTag invocation will look like below.

28 aload_2 [tv]
29 ldc <Integer 2131296319> [30]
31 aload_2 [tv]
32 invoke virtual android.widget.TextView.setTag(int, java.lang.Object) : void [31]

After applying the above patch it will be as below.

28 aload_2 [tv]
29 aload_3 [x]
30 invoke virtual android.widget.TextView.setTag(java.lang.Object) : void [50]

5.5 Testing the Fix

5.6 Information needed from the Code for this example

The following information is needed to be logged from the code to identify and fix the setTag memory leak bug:

5.6.1 Points to call graph

We need to save the information of successor node and predecessor node. We can query for setTag call nodes with two parameters and for the link between object and argument from call graph. So we will need to save the parameter count and types for each node where methods are invoked.
This will be used to identify the setTag bug and to pin point it's location.

5.6.2 Class name and source file name

We can then query for the methods that invoke setTag incorrectly from call graph information. While saving information we need to also save the declaring class where methods are invoked. Also from Declaring class we can get the source file name This will be used to generate name of class file to be transformed.
This will be used to read the class file and visit the method to be transformed.

5.6.3 Bytecode and IR offset

From Parent CG Node we can get IR of the method. From IR we can get offset for method invocations and from that we can get byte code offsets for each invocation. These will be used to find the instruction where we need to apply patches.
This gives bytecode offsets where patches are to be applied.

5.6.4 Argument and Object position in stack

As shown above before invocation of a method, we need to first load the object which calls the method from stack followed by similar loading of parameters from the stack. For applying patches we will need the positions of these in the stack to load them before the invocation.
This information is used for creating the patches.