SensA Project Tool

Ying-jie Zhang

Tsinghua University(2009-2013)

Summer program in University of Notre Dame (2012)

zhangyj1991@gmail.com

SensA, whose original name comes from "sensibility analysis", based on Soot library, is a tool that instruments and monitors the location of a prospective change in a Java program, and then reports the impacted statements and probability ranking. What's more, it is based on java byte-codes, rather than the source codes.

INTRODUCTION

These are simple instructions to use SensA for instrumenting certain statements in specified location and analyzing the probability of impacted output and statements.

To run SensA, you need the following in your Java classpath (Soot and DuaForensics are **NOT** included in this package):

--***DuaF*** (program analyzer and instrumenter)
--***LocalsBox*** and ***InstrReporters*** (the runtime components for coverage monitoring and reporting)
--all ***Soot*** components (soot 2.2.4 or 2.3.0, polyglot, jasmin -- check the Soot website: http://www.sable.mcgill.ca/soot/)

** **Note**: SensA has not been officially tried with Soot 2.4 or later versions

SensA uses Java bytecode subjects, instead of Java source files, to instrument and analyze. It analyzes the .class file obtained by compiling the .java files.

PREPARING SUBJECT

Since SensA Project is based on Dua-Forensics, you must include LocalsBox and InstrReporters in your subject's classpath and then insert calls to *BranchReporter.__link()* and *DUAReporter.__link()* in any class of your subject that will be instrumented (a good practice is the main/entry class). In addition, in order to invoke modify method to modify your subject, you should insert calls to Sensa.*Modify._link()*. And *ExecHistReporter.__link()* will help you observe the change of statement executions. The example is as follows:

```
import profile.BranchReporter;
```

```
import profile.DUAReporter;
import profile.ExecHistReporter;
import change.DynSliceReporter;
  ...
static void __link() { Sensa.Modify.__link(); ExecHistReporter.__link(); BranchReporter.__link();
DUAReporter.__link(); } // to allow Soot to instrument
```

Note: if you use other, less mature functionality included with DuaF, you might also need to link to other 'reporter' classes located in InstrReporters.

USAGE

SensA accepts the same command line options as Soot (check Soot's webpage for more info) and DuaF options (check DUAF's README document for more info), plus additional SENSA's own options which SensA takes before forwarding the rest of the command line to DuaF.

SensA could be divided into two parts. One is for instrument while the other is for subject running.

1)Instrument Part

The main class is **SensaInst** (i.e., use 'java SensaInst <command-line-options>'). The following is a typical list of SensaInst command-line options.

```
<variable_name_for_modification> <variable_location_in_certain_method>
<if_or_assignmeng_statement_to_modify> <left_or_right_reference_of_ifstmt> -w -cp
"<path_of_rt.jar>;<path_of_DUAForensics>;<path_of_LocalsBox>;<path_of_InstrReporters>
;<path_of_subject_classfiles>" -p cg verbose:true,implicit-entry:false -p cg.spark
verbose:true,on-fly-cg:true,rta:true -f c -d "<soot_output_path>" -brinstr:off
-duainstr:on -duaverbose -allowphantom -slicetxinses
-start:<line_number_of_statement_in_subject_to_be_modified> -main-class
<main_class__no_.class_extension> -entry:<main_class_no__.class_extension>
<list_class_files_to_instrument__no_.class_extension>
```

All options above are for Soot, except for -brinstr:off, -duainstr:on, -duaverbose, -allowphantom, -slicetxinses -start and -entry which are for SensA. Parts surrounded by <> have to be replaced by the corresponding elements from your own java/subject setup.

Notes:

-*<variable_name_for_modification>* is the name of variable that will be modified. The name should be the same in jimple file.

-*<variable_location_in_certain_method>* means SensA will instrument before this Jimple statement. Moreover, the line number is the relative line number of jimple file in the method remained to be modified.

-<if_or_assignment_statement_to_modify> has two options, “if” and “assign” to identify which kind of statement to be modified.

- <left_or_right_reference_of_ifstmt> is needed merely when the change statement is if statement.

- The *-cp* Soot option specifies the classpath for the subject to instrument. Surrounding with double-quotes ("") is recommended in case there is a space in some classpath.

- The separator for Java classpaths in Windows is semi-colon (;), and in Unix it is colon (:)

- *<path_of_rt.jar>* is the location of the rt.jar of the version of Java against which you compiled/linked your subject. RECOMMENDED: use the oldest possible version of JRE (i.e., 1.3 or 1.4); the older rt.jar is, the smaller it is, and the less time Soot takes to run.

- *<path_of_subject_classfiles>* points to the root package where the .class files of your subject are located

- The *main class name* (including package, but excluding the .class extension) must be the same for -main-class (Soot option) and for -entry (DuaF option).

- *<list_class_files_to_instrument__no_.class_extension>* is a space-separated list of all classes to analyze/instrument (again, without the .class extension). Alternatively, you can use -process-dir *<basepath>* to make SensA include all classes rooted at <basepath>.

-*<line_number_of_statement_in_subject_to_be_modified>* is to locate where the method to modify is in a jimple file. You could provide any line number of statements inside the method.

-<-exechist> is optional. It will be useful only if you want to observe the execution of certain subject.

2)Running Part

The main class is **SensaRun** (i.e., use 'java SensaRun <command-line-options>'). The following is a typical list of SensaRun command-line options.

```
<main_class_no__.class_extension> <basepath> <path_of_subject_classfiles>
<line_number_of_statement_in_subject_to_be_modified>
```

****IMPORTANT:**

To successfully run SensA, you should create ①a valid **Sensa.cfg** in each subject. ②testinput.txt in the “inputs” file under subject’s base directory.

In the .cfg file, you should provide the following parameters:

-**mod_alg**: default is Random algorithm.
-**runs**: means the times you want to run for each test set
-**test**: means the number of test set you want to run. Default number is -1 which means to run all test sets.
-**increment**: provides incremental value for Increment Algorithm. Default value is 1 or 1.0
-**min**: means the minimum value allowed for the modified variable
-**max**: means the maxmum value allowed for the modified variable

Note:

-D Command: Users can use "-DRunOneInMod=false" to allow subject execute modify method more than one time per run. By default, modify will be executed only once per run.

SensA will read the content per line as the input of specific subject.

RUNNING INSTRUMENTED SUBJECT

The instrumented classes go to <soot_output_path> (see options above). To run the instrumented subject, please include ExecHistReporter, DynSliceReporter, LocalsBox and InstrReporters in your classpath. <soot_output_path> will include files generated by SensA, such as 'stmtids.out'.

Besides, SensA will create two folders under subject's root directory. One is called "**runs**", which contains the results of subject for N test sets and M run times. To be specific, the value in run1.out is actually not modified in each test. The other is called "**valuetried**" which contains modified value in each execution.

ALGORITHM

SensA provides three kinds of modify algorithm, Random Algorithm, Incremental Algorithm and Observed Algorithm (default). For each algorithm, SensA supports several kinds of types, such as int, float, long, short, double, char, byte, String and etc.

The details of the three modify algorithms are as follows:

●Random Algorithm:

--For int, float, short, long, double, byte: it will provide a random number.

--For String, it will randomly choose to insert, append, delete or replace with a random letter.

--For char: it will provide a random valid ASCII character.

--For other object, it will return NULL if and only if the reference is not NULL.

●Increment Algorithm:

--For int, float, short, long, double, byte: it will provide in a sequence such as n+1, n-1, n+2, n-2 and etc

--For char: it will add 1, 2, 3 and etc. to create a new valid ASCII character.

--For String: it will repeat the previous string for 1, 2, 3 or etc. times. That is to say, it will modify "cat" to "catcat" or "catcatcat".

--For other object, it will return NULL if and only if the reference is not NULL.

●Oberseved Algorithm:

--For each test tN, to make x=Modify(x) try all values observed at that point other than the value of x for tN.

--For Object especially, it will return NULL if and only if NULL has been observed (and the current value of the reference is not NULL).

EXTENSION

You can create your own modify algorithm while SensA offers an interface called “**IModify**”. To be simplified, you can also inherit from **ModifyDefault**.java and what you need to do is merely modify specified method. The default method in this class is to return the same variable without modification.

** **Note**: Since SensA is mainly used to analyze dynamic program slicing, it will NOT allow modify algorithm to return the same value in a test set. And thus your own modification algorithm should be able to create different values for each test set.

You can also add other types of variable for SensA to support though it will be much more complicated. You can modify the source code of SensA in proper order, that is from Modify to IModify as well as related modify algorithm class. You can read the comments in “IModify.java” to add your own code. And you can get the previous value from “preObject” and you must return a modified value to “returnObject”.