(ECE453/CS447/ECE653/CS647/SE465)
Software Testing, Quality Assurance, and Maintenance
Project (100 Points), Version 5

Instructor: Lin Tan
Lab Instructor: Bernie Roehl
Part (I) TA in Charge: Tian Jiang
Part (II) TA in Charge: Lei Zhang
Release Date: January 28, 2013
Revision Date: March 23, 2013

Due: 5:00 PM, Thursday, April 4, 2013
Submit: An electronic copy on LEARN

Please download proj-skeleton.tar.gz from the course website to get the necessary source code and test cases needed for finishing this project.

I expect each group to work on the project independently (discussion and collaboration among group members are expected). I will follow UW's Policy 71 if I discover any cases of plagiarism.

Submission Instructions:

Please read the following instructions carefully. **If you do not follow the instructions, you may be penalized up to 5 points**.

Electronic submission: Go to "Dropbox" $-\rangle$ "Project Submission" on LEARN. Submit only one file in .tar.gz format. Please name your file

Group<GroupNumber>.tar.gz

For example, use Group1.tar.gz for submissions from Group 1. *You must include your file name as part of your submission comments.* The .tar.gz file should contain the following items:

- a single pdf file "proj_sub.pdf" for both part (I) and part (II). You must include a **cover page** that contains your team members' full names, student numbers, the class numbers (one of ECE453, CS447, ECE653, CS647, & SE465), and your uwaterloo email addresses.
- a directory "pi" that contains your code for Project part (I) (follow the instructions of part (I)). The code for for part I (c) and (d) should be in directory "pi" as well. It should not break the marking script given to you for Part I (a). **You must use C/C++ or Java for part (I).**
- a directory "pii" that contains the compressed output from Coverity for Project part (II)(b) (follow the instructions of part (II)).

You can submit multiple times. After submission, you can view your submissions to make sure you have uploaded the right files/versions.

Part (I) Building an Automated Bug Detection Tool

You will learn likely-invariants from call graphs in a way that is similar to the SOSP'01 paper discussed in class (Coverity). In particular, you will learn what pairs of functions are called together. You will then use these likely-invariants to automatically detect software bugs.

(a) Inferring Likely Invariants for Bug Detection

Take a look at the following contrived code segment.

```
void scope1() {
  A(); B(); C(); D();
}
void scope2() {
  A(); C(); D();
}
void scope3() {
  A(); B(); B();
}
void scope4() {
  B(); D(); scope1();
}
void scope5() {
  B(); D(); A();
}
void scope6() {
  B(); D();
}
```

We can learn that function `A` and function `B` are called together three times in function `scope1`, `scope3`, and `scope5`. Function `A` is called four times in function `scope1`, `scope2`, `scope3`, and `scope5`. We infer that the one time that function `A` is called without `B` in `scope2` is a bug, as function `A` and `B` are called together 3 times. Please note that we only count `B` once in `scope3` although `scope3` calls `B` two times.

We define *support* as the number of times a pair of functions appears together. Therefore, $support(\{A, B\})$ is 3. We define $confidence(\{A, B\}, \{A\})$ as $\frac{support(\{A,B\})}{support(\{A\})}$, which is $\frac{3}{4}$. We set the threshold for support and confidence to be $T\_SUPPORT$ and $T\_CONFIDENCE$, whose default values are 3 and 65%. You only print bugs with confidence $T\_CONFIDENCE$ or more and with support $T\_SUPPORT$ times or more. For example, function `B` is called five times in function `scope1`, `scope3`, `scope4`, `scope5`, and `scope6`. The two times that function `B` is called alone are not printed as a bug as the confidence is only 60% ($\frac{support(A,B)}{support(B)} = \frac{3}{5}$), lower than the $T\_THRESHOLD$, which is 65%. Please note that both $support(A, B)$ and $support(B, A)$ are the same, 3.

Perform intra-procedural analysis. For example, do not expand `scope1` in `scope4` to the four functions `A`, `B`, `C`, and `D`. Match function names only. Do not consider function parameters. For example, `scope1()` and `scope1(int)` are considered the same function.

The sample output with the default support and confidence thresholds should be:

```
bug: A in scope2, pair: (A B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B D), support: 4, confidence: 80.00%
```

Generate call graphs using LLVM. Given a bitcode file, you use LLVM *opt* to generate call graphs into plain text format, and then analyze the textual call graphs to generate function pairs and detect bugs. The *opt* tool is installed on ecelinux under /usr/local/bin. If you want to work on your own computer, you need to use LLVM 3.0 64 bits in order to avoid compatibility issues. LLVM 3.0 is available as a package in many popular Linux distros.

A sample call graph in text format is shown as follows:

```
Call graph node for function: 'ap_get_mime_headers_core'<<0x42ff770>>  #uses=4
  CS<0x4574458> calls function 'ap_rgetline_core'
  CS<0x4575958> calls external node
  CS<0x4575a50> calls external node
  CS<0x4575b20> calls function 'apr_table_setn'
  CS<0x45775a8> calls external node
  CS<0x45776a0> calls external node
  CS<0x4577770> calls function 'apr_table_setn'
  CS<0x45783b8> calls external node
  CS<0x4579dc0> calls function 'apr_table_setn'
  CS<0x4579f78> calls function 'strchr'
  CS<0x457a948> calls external node
  CS<0x457aa40> calls external node
  CS<0x457ab10> calls function 'apr_table_setn'
  CS<0x457d9d0> calls function 'apr_table_addn'
  CS<0x457e4e8> calls function 'apr_table_compress'
and
Call graph node <<null function>><<0x2411e40>>  #uses=0
```

The following explanation should help you understand the call graph:

- The above call graph represents the function `ap_get_mine_headers_core` calls functions `ap_rgetline_core`, `apr_table_setn`, etc.
- **#uses=** gives the number of times the caller is called by other functions. For example, `ap_get_mine_headers_core` is called 4 times.
- **CS** denotes call site.
- **0x?????** indicates the memory address of the data structure, i.e., call graph nodes and call sites.
- An **external node** denotes a function that is not implemented in this object file. Ignore external node functions.
- A call graph node of **null function** represents all external callers. Ignore null function nodes entirely.

Performance. We will evaluate the performance/scalability of your program on a pass/fail basis. The largest program that we test will contain up to 20k nodes and 60k edges. Each test case will be given a timeout of two minutes. A timed-out test case will receive zero points.

Hints:

- Avoid string comparison. Use some kind of ID instead.
- Avoid using strings as keys in hash tables. Use some kind of ID instead.
- Avoid using nested loops. An $O(nlgn)$ approach is MUCH more efficient than an $O(n^2)$ approach.
- In a nested loop, prune an iteration in the outer loop is more beneficial than in the inner loop.
- Profile your code if the provided test cases take more than 5 seconds to run.

Submission protocol. Please follow this protocol and read it CAREFULLY:

- Submit your source code. Please document/comment your code properly so that it is understandable. You must use the skeleton files and include a *Makefile* to compile your program from source.
- Put your *Makefile* directly under *pi* directory. To reduce the size of your submission, please remove the sample test cases and the scripts given to you from your submission.
- The script (*verify.sh*) that we plan to use to verify your output is given to you, or will be given to you soon. Please write your code and makefile to fit the script.
- **Please make sure your code runs on ecelinux machines**. We will use the provided scripts to grade your project on ecelinux machines. If your code doesn't run on ecelinux, you will receive at most 20 points out of the full 100 points. **You must use C/C++ or Java for part (I).**
- Marking will be done with strict line comparison. We will count how many pairs are missing (false negatives) and how many pairs are false positives.

- You should print one pair per line to stdout in this format:
  bug: %s in %s, pair: (%s %s), support: %d, confidence: %.2f\%\n
- If you use Java, make sure you round decimal numbers in the same way as C/C++ on Linux by setting the appropriate RoundingMode. For example:

```
NumberFormat numf = NumberFormat.getNumberInstance();
numf.setMaximumFractionDigits(2);
numf.setRoundingMode (RoundingMode.HALF_EVEN);
System.out.println("confidence:" + numf.format(confidence*100.0) + "%");
```

- The order of different pairs does not matter. We will run *sort* before *diff* as you can see from the script *verify.sh*.
- The order of the two functions in a pair does not matter in the calculations. Both (foo, bar) and (bar, foo) contribute to the count of pair (bar, foo). However, in order to have consistent output (for easier grading), sort the pair alphabetically while printing. For example, print (bar, foo) instead of (foo, bar).
- Ignore duplicate calls to the same function. For example, if we have the code segment `scope() { A(); B(); A(); B(); A(); }`, the support for the pair (A, B), $support(A, B)$, is 1.
- Name your program binary *pipair*. We should be able to run it with the following command:
  ./pipair <bitcode file> <T_SUPPORT> <T_CONFIDENCE>,
  e.g.,
  ./pipair hello.bc 10 80, to specify the support of 10, and the confidence of 80%,
  or
  ./pipair <bitcode file>,
  e.g.,
  ./pipair hello.bc, to use the default support of 3, and the default confidence of 65%.
- If running your program requires special command line format such as *java YourProgram arguments*, your *pipair* should be a wrapper, e.g.:

```
#!/bin/bash
java YourProgram $@  # $@ represents all command line arguments
```

Skeleton files and marking.

- It is recommended to develop on ecelinux or your own computer running a popular flavour of Linux. There were reports show that "sort", as an example, behaves differently in Mac than in Linux. Cygwin and MinGW were not tested.
- To generate the output for one test, run "make" in the test's directory. Your output should be identical to the "gold" file. Your output should be passed through "sort" before "diff"ing with the "gold" file, i.e. cat <your output> | sort | diff - gold_x_xx
- To run all tests together, execute "verify.sh". Logs of all output can be found in /tmp.
- clean.sh runs "make clean" in all test directories.
- For marking, we will untar your submission, copy the content of the pi directory, run make, copy over the skeleton files and the full test suite with six tests and run then verify.sh. Note that Each test is given 2 minutes. My prototype runs test3 in 2 seconds and test6 in 25 seconds, and all other tests in tens or hundreds of milliseconds.
- Since the skeleton files and tests are copied over during marking, do NOT modify any files given in proj-skeleton since they will be over-written.

Common issues.

- It says "Can't make".

  This error indicates there was a problem while verify.sh tried to run make inside the test directories. The error message is usually in the testx_x_xx.out, or sometimes in /tmp/testing-<your username>-pi-<time of log>.log. Some common errors:

  - You are running verify.sh just "out-of-box". pipair is not found because you haven't written one.
  - pipair exits with an error.

- Don't know how to get the call graph using *opt*, getting nothing or gibberish.

  You should use LLVM's *opt* tool. *opt* prints the call graph to stderr, and the content of the bitcode file to stdout if stdout is not the terminal. There are many ways of getting the call graphs:

  - Call *opt* from inside your program, and capture *opt*'s stderr. See the sample C code from Tutorial 2.
  - Use an intermediate file for the call graph.
  - Write a wrapper, and pipe the call graph to your program as if it is typed by hand to stdin, like "opt -print-callgraph 2>&1 >/dev/null | YourProgram $@".

  There are still many other ways depending on your design and your language of choice.

- Getting "Segmentation Fault" from opt.

  If it's only test3, you are using an old version of LLVM. You must use LLVM 3.0 64bit to be able to generate call graph for test3. If it's all tests, you are using a broken LLVM compilation. It is recommended to work with the LLVM on ecelinux or a package on a popular flavor of Linux.

- Java throws "NoClassDefFoundError" when running verify.sh, but it works fine running my class manually.

  The working directories while running the tests is the test directories. You need to specify the path to your .class file using "-cp" option. Your command should look similar to "java -cp .. YourClassName $@" in your pipair wrapper.

- LLVM on ecelinux machines.

  You can find LLVM binaries on ecelinux machines in directory /usr/local/bin/.

(b) Finding and Explaining False Positives

Examine the output of your code for (a) for Apache HTTPD server. Did you find any real bugs? There are some false positives. A false positive is where a line says "bug ..." in your output, but there is nothing wrong with the corresponding code. Discuss why there are false positives. Discuss at least two reasons. (10 points)

Identify at least 10 "bug ..." lines (referred to as 10 locations) for HTTPD combined related to at least 2 pairs of functions that are false positives. For example, the following sample output contains 4 locations regarding 3 pairs, i.e., (A B) (A D) and (B D).

```
bug: A in scope2, pair: (A B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B D), support: 4, confidence: 80.00%
```

The source code of HTTPD has been provided on ecelinux for your convenience. It is located at: */home/testing/apache_src*.

Each directory contains the source code, the bitcode file (.bc), the call graph file in plain text format (.txt), and the sample output (.out) of each project. At the end of the call graph, you'll find the file path where each function is defined, which will help you find the actual function bodies.

If you find new bugs (bugs that have not been found by other people yet), you may receive bonus points. Check the corresponding bug databases of Apache[1] to make sure that the bug you found has not been reported there already. Clearly explain why you think it is a bug and provide evidence that it is a new bug.

(c) Inter-Procedural Analysis.

One solution to reduce false positives is inter-procedural analysis, e.g., expanding the function `scope1` in the function `scope4` to the four functions `A`, `B`, `C`, and `D`. Implement this solution, and write a report (up to 2 pages) to describe what you did and what you found. You can play with the numbers of levels that you expand. Give an example to illustrate that your solution can do better than the default algorithm used in (a). We will read your report and code. If you submit only one of them (either report or code), you will receive at most 50% of the points for this part ((c)). Make sure your code is well documented and your report is understandable.

[1] `https://issues.apache.org/bugzilla/`

(d) Improving the Solutions. For ECE653 and CS647 students only!

Propose another solution to reduce false positives and one solution to find more bugs. Implement them and write a 2 page summary. It is OK to implement and describe an approach that has been published. If you do so, make sure you cite the papers.

Part (II) — Using a Static Bug Detection Tool

For this part of the project you will be using the static code analysis tool Coverity to find defects in source code.

(a) Resolving Bugs in Apache Tomcat

In this task, you are to inspect warning messages resulting from running Coverity static analysis on Apache Tomcat6 (a web application server written in Java). We have already compiled and analyzed a version of Tomcat6 for you. The pre-analyzed code is stored in a compressed file (`tomcat6_java_output_r729554.tar.gz`) which you can download from LEARN.

Next, you will need access to the Tomcat6 source code. Using a subversion client, check out revision **729554** from the repository at `http://svn.apache.org/repos/asf/tomcat/tc6.0.x/trunk`.

```
svn co -r 729554 http://svn.apache.org/repos/asf/tomcat/tc6.0.x/trunk
```

In the directory containing the pre-analyzed code, the results from the analysis are located in `java_output/`. There are results from six different checkers for you to look at:

- REVERSE_INULL
- NULL_RETURNS
- GUARDED_BY_VIOLATION
- FORWARD_NULL
- RESOURCE_LEAK
- CALL_SUPER

For details on each checker, read the checker documentation (see `https://ece.uwaterloo.ca/~lintan/courses/testing/coverity.html#reference_material`).

Your task is to use the warnings from the analysis to fix the problems indicated. Some results may be false positives. If this is the case, label the warning as such and give a brief explanation of why you think this is the case. To reduce the amount of work for you, we have reduced the total number of warnings to 20.

For each bug, provide a code snippet (including the file name and line numbers) along with a description of the proposed bug fix.

The report for this part should be no more than 4 pages. Be concise.

(b) Analyzing Your Own Code

Run Coverity on your own code from part (I). You learn more from having the tool find defects in code that you wrote, fixing the problem and seeing the defect go away. Discuss two of the bugs detected by Coverity using up to 1 page. If Coverity finds no bugs, what are the possible reasons?

Include a compressed file (in .tar.gz format) of the results from your analysis with your submission (put it in directory pii).

You can find instructions on how to use Coverity to build and analyze your code at `https://ece.uwaterloo.ca/~lintan/courses/testing/coverity.html`.

Quinn Hanam kindly wrote a php script to convert the Coverity xml output file to a prettier format. You can use this site (`https://ece.uwaterloo.ca/~qhanam/read-errors.php`) to make the Coverity's xml output easier to read.