(ECE453/CS447/ECE653/CS647/SE465)
Software Testing, Quality Assurance, and Maintenance
Project, Version 2

Instructor: Lin Tan & Patrick Lam
Lab Instructor: Bernie Roehl
Release Date: January 28, 2015
Revision Date: January 29, 2015

Due: 11:59 PM, Monday, April 6, 2015
Submit: via ecgit

Please go to `http://patricklam.ca/se465-groups/1151/project/` to sign up your project group and get your repo address. The group signup also forks the skeleton repo for you.

We have also placed `proj-skeleton.tar.gz` in the course repo, in case you want to look at the files before committing to a group. But use the skeleton in your forked repo before starting to work on your project. The skeleton contains the necessary source code and test cases for the project.

You can choose your own group members (2–3 students per group). Do not mix graduate students and undergraduate students in the same group since the project is slightly different.

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%**.

Electronic submission: Push your project files to your group repo on ecgit. We expect to find:

- in the root of the repo, 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 course numbers (one of ECE453, CS447, ECE653, CS647, & SE465-001 or SE465-002), and your uwaterloo email addresses.

- a directory "pi" that contains your code for Project part (I) (follow the instructions for part (I)). The code for part I (c) and (d) should be in directory "pi" as well. It must 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 for part (II)).

The layout of your directories is important, since we use automated software to mark your submissions.

In your top-level "pi" directory there should be a `Makefile` and a `pipair` file (which may be created by your Makefile). Our marking script will copy the skeleton files into this directory before running `verify.sh`; for example, the first test will be in "pi/test1". Placing files into other folders, placing your `pipair` file in a

different location, or doing anything which prevents `verify.sh` from running will result in losing marks. Also note that you cannot modify `verify.sh`, since it will be automatically overwritten during marking.

You can push your code multiple times. After submission, **please clone your submissions to make sure you have uploaded the right files/versions**.

Part	People in Charge
(I)	Bernie
	Edmund
	Taiyue
(II)	Thibaud
	Song

Part (I) Building an Automated Bug Detection Tool

You will learn *likely-invariants* from call graphs in a way reminiscent to that in 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. 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 pair 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% $\left(\frac{support(A,B)}{support(B)} = \frac{3}{5}\right)$, lower than the $T\_THRESHOLD$, which is 65%. Please note that $support(A, B)$ and $support(B, A)$ are equal, and both 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.

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 in 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 64-bit LLVM 3.0 to avoid compatibility issues. LLVM 3.0 is available as a package in many popular Linux distributions.

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 and 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. We do not recommand using multi-threading because it only provides a linear gain in performance.

Hints:

- (Very Important!) Do not use string comparisons because it will greatly impact the performance. Use a hashing method to store and retrieve your mappings.

- Minimize the number of nested loops. Pruning an iteration in the outer loop is more beneficial than in the inner loop.

- None of the provided test cases should take more than 5 seconds to run.

Submission protocol. Please follow this protocol and read it *carefully*:

- You only have to submit three items inside the `pi` directory. Your `.gitignore` should ensure that you don't push compiled binaries or `test` subdirectories.
  - A file called `Makefile` to compile your source code; it must include 'all' and 'clean' targets.
  - A file called `pipair` as an executable script. You don't need to submit this if your `pipair` is an executable binary, which will be generated automatically when our script executes 'make' on your `Makefile`.
  - `Source code` for your program. Please document/comment your code properly so that it is understandable. Third party libraries are allowed. Make sure all your code runs on ECE Linux. We will not edit any environment variables when we run your code on our account, so make sure your program is self-contained. We will be testing the code under the bash shell.

  Place all of the above items at the root of your repository.

- We provide the marking script (*verify.sh*) that we'll use to test your pipair. Please write your code and `Makefile` to fit the script.

- **Make sure your code works 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 0 points. **You must use C/C++ or Java for part (I).**

- Marking will be done with strict line comparisons. 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
  ```

- The order of the pairs in your output does not matter. As you can see from the script *verify.sh*, we will run *sort* before *diff*.

- 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, to have consistent output (for easier grading), you must sort the pair alphabetically while printing. For example, print (bar, foo) instead of (foo, bar).

  The golden output uses natural sorting with the following API:

  http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

- 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 `pipair`. We must 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 support of 10 and confidence of 80%,
  or

./pipair <bitcode file>,

e.g.

./pipair hello.bc, to use the default support of 3 and default confidence of 65%.

- If running your program requires a special command line format such as *java YourProgram arguments,* your *pipair* should be a wrapper bash script, e.g.:

```
#!/bin/bash
java YourProgram $@
```

$@ represents all command line arguments.

- If you are using Java, you must configure the `-Xmx128M` flag to ensure Java Virtual Machine allocates enough memory during the startup. The reason is that the VM will not dynamically increase the heap size for the larger test cases.

Skeleton files and marking.

- We recommend developing on ecelinux. We have heard that the UNIX "sort" command behaves differently on Mac than on Linux.

- To test a specific test case, run "`make clean`" and then "`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. For example:

`sort testX_Y_Z.out | diff gold_Y_Z -`

- 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.

- Our marking procedure:

  1. We will clone your repo.
  2. We will copy the required files from your `pi` directory into the root of a clean skeleton folder that contains the full text suite with seven tests.
  3. We will run "`verify.sh`" to test your program, which will automatically:
     (a) Run "`make`" to compile your program.
     (b) Execute each test with a two minute limit.

- Since the skeleton files and tests are copied over during marking, do *not* modify any files given in the project skeleton, since they will be over-written.

Common issues.

- It says "error".

  This error indicates `verify.sh` encountered a problem when it tried to run `make` inside the test directories. This is likely due to a failure in `pipair`.

  Here are a few things that you can do to help troubleshoot this issue:

  – The error message is usually in the `testx_x_xx.out` file. This file contains what got written to stdout by your `pipair` executable.
  – The log output of `verify.sh` is dumped to a file at `/tmp/testing-<your username>-pi-<time of log>.log`.

- Instead of testing the whole test suite using `verify.sh`, run the individual test cases manually with "`make clean`" and then "`make`" inside each testX folder.
- Verify the correctness of your `pipair` by running it manually inside each test folder. If your `pipair` is a bash script that consists of multiple commands, manually execute the bash commands one at a time.

- How should I implement `pipair`?

  You can implement pipair as an executable binary, or you can implement pipair as a bash script. It might be easier if you do it the bash script approach. The reason is that you can call `opt` within the bash script and either

  - pipe the call graph output from `opt` into your C/Java program; or,
  - write the call graph to a file and then read the file with your C/Java program to perform the analysis.

- My program is printing garbage; how can I get rid of message X or message Y?

  Please check both stdout and stderr. You can discard those messages by dumping them to `/dev/null`.

- I don't know how to get the call graph using `opt`. I'm getting either nothing or gibberish.

  Use LLVM's `opt` tool. `opt` prints the call graph to stderr, and the contents of the bitcode file to stdout if stdout is not the terminal. There are many ways of getting the call graphs:

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

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

- I can't run a shell command with I/O redirection (e.g., "`opt -print-callgraph 2>&1 >/dev/null`").

  This is because your Linux shell does not support I/O redirection. You can check what shell you are running by executing "ps -p $$". Change your shell to `bash` by issuing "`bash`" in the terminal to fix the issue.

- I'm getting "Segmentation Fault" from `opt`.

  If it's only test3, you are using an old version of LLVM. You must use 64-bit LLVM 3.0 to generate a call graph for test3. If all tests segfaults, you are using a broken LLVM compilation. We recommend using LLVM on ecelinux.

- Java throws "NoClassDefFoundError" if I run my program within the testX directory.

  The working directory while running the tests is *inside* the testX directories. Therefore, you need to specify the path to your .class file using the "`-cp`" option in your pipair wrapper. For example:

  `java -cp [classpath here] YourClassName $@"`

(b) Finding and Explaining False Positives

Examine the output of your code for (a) on the 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. Include at least two different fundamental reasons.

Identify 2 pairs of functions that are false positives for `httpd`. The two pairs should have a combined total of at least 10 "bug ..." line locations. For example, the following sample output contains 4 locations regarding 3 pairs: (A, B) (A, D) and (B, D). You do not have to provide explanations for all 10 locations, but you will want to use at least one location to discuss each pair. Briefly explain why the pair is a false positive with reference to the source code.

```
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: */opt/testing/a-pache_src*.

This directory contains the source code, the bitcode file (`.bc`), the call graph file in plain text format (`.txt`), and the sample output (`.out`) for a run of `httpd`. 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. For example, one might expand the function `scope1` in the function `scope4` to the four functions `A`, `B`, `C`, and `D`. Implement this solution by adding an optional fourth command line parameter to `pipair`. We leave the algorithm's implementation details up to you.

Write a report (up to 2 pages) to describe your algorithm and what you found. Run an experiment on your algorithm and report the details. For example, you can vary the numbers of levels that you expand as an experiment. Provide a concrete 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 only submit a report, 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.

Test 3 from part (a) utilizes the same program (`httpd`) as part (b). Therefore, you might find it convenient to use test 3 to varify your inter-procedural experiment.

(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 in code and place them inside a folder called `pi/partD/`. Write a 2 page summary to describe the two solutions. It is okay if you only propose a single solution, provided that it can reduce false positives and find more bugs at the same time. You may 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. We have acquired a Coverity license for student use. Occasionally, we will collect groups, generate Coverity credentials, and mail you a link which allows you choose a password for your credentials.

(a) Resolving Bugs in Apache Commons

In this task, you are to triage warning messages resulting from running Coverity static analysis on Apache Commons (a library of useful Java tools). We have already compiled and analyzed a version of Apache Commons for you. The pre-analyzed code is available on Coverity's web interface at `http://ecelinux5.uwaterloo.ca:8080`. Log in using username "student" and password "Se7#p!Am". To view and triage warnings from the

---

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

analysis, select the "Outstanding Defects" item in the navigation tree on the left hand side of the page. Click on warnings to view details of the warning and the source code that triggered the warning.

The analysis results contain warnings from both Coverity and FindBugs static analysis tools.

Your task is to triage and repair these warnings. There are 21 warnings for you to triage and repair. Triage each warning by classifying it as one of the following: *False Positive*, *Intentional* or *Bug*. Below are descriptions of each triage classification:

- **False Positive**: The warning does not indicate a fault or deficiency.

- **Intentional**: You believe the warning points to code that the developer intentionally created that is incorrect (contains a fault) or considered bad practice.

- **Bug**: The warning points to a defect (a fault or bad practice) in the code.

If you classify the warning as *False Positive*, provide an explanation as to why you believe this is the case. If you classify the warning as *Intentional*, explain how the code can be re-factored to remove the warning, or explain why it should be left as-is. If you classify the warning as bug, provide the faulty lines (1-2 lines is often enough, including the file name and line numbers) and a description of a proposed bug fix.

For details on each Coverity checker, read the checker documentation found on ecelinux (see `https://ece.uwaterloo.ca/~lintan/courses/testing/coverity.html#reference_material`). Information on FindBugs checkers can be found at `http://findbugs.sourceforge.net/bugDescriptions.html`.

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) (a). See instructions on `https://ece.uwaterloo.ca/~lintan/courses/testing/coverity.html` to analyse your code with Coverity Connect interface. Once your project is uploaded on the web interface, you should **make it inaccessible for other students: connect on** `http://ecelinux5.uwaterloo.ca:8080` **with your credentials. Click on "Configuration" , "Projects & Streams" and Select your project. In the "Roles" tab, click on "add" and check "No Access" for the group "Students".**

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 the results from your analysis with your submission. Put them 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`.

We have provided 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 Coverity's xml output easier to read.