**SE465 PROJECT**

(in order)

**Members**:Xi Chen, Richard Di, Charles Qi

**Student Numbers**: 20512548, 20525661 ,20507729

**Emails**: [x335chen@uwaterloo.ca](mailto:x335chen@uwaterloo.ca),

w4di@uwaterloo.ca

**Class number**: SE465-001

**PART I**

**b)**

One reason for false positive is the lack of inter-procedral analysis. We can see in the first “bug” that inter-procedral analysis would reveal that array\_push is actualy called after array\_make. Two pairs that does not occur in the same function, when though they would in most cases, would be considered a bug. Considering the fact that the code would still be correct if the paired functions are called in different functions, this is a source for false positives.

Another reason is that two commonly used functions might be paired without any particular relevance. For example array\_make is a common function in the httpd server, if it occurs with another certain function, even if they has different purposes, they can be seen as a pair and the lack of one would be shown as a bug. Thus a false positive would emerge as these two functions do not intefere with each other.

The pairs apr\_array\_push and apr\_array\_make is shown as a bug.

bug: apr\_array\_make in create\_core\_dir\_config, pair: (apr\_array\_make, apr\_array\_push), support: 40, confidence: 86.96%

What apr\_array\_make does is that is allocates memory for a apr\_array and returns the pointer. apr\_array\_push allocate some more memory on top of the given and returns the pointer to it. As we can see we must call apr\_array\_push before apr\_array\_make, however they do not have to be done at the same function. We can call make first then pass the pointer to another function who calls push. Thus this is can be a false positive.

For example, In create\_core\_dir\_config inside core.c, we make conf->sec\_file an array, however there was no push inside this function. Nevertheless we can see other function pushing to this array in the same file. Creating an array then pushing to it later is not a bug. Thus it is a false positive.

The other pair is

bug: apr\_thread\_mutex\_unlock in apr\_global\_mutex\_unlock, pair: (apr\_thread\_mutex\_lock, apr\_thread\_mutex\_unlock), support: 43, confidence: 91.49%

The function apr\_global\_mutex\_unlock inside global\_mutex.c unlocks the specified global mutex and returns success code. It is the intended use to call apr\_thread\_mutex\_unlock without apr\_thread\_mutex\_lock since the function only releases the log. Thus this is not a bug. Perhaps it is a bug to call apr\_thread\_mutex\_unlock without apr\_thread\_mutex\_lock, however since it is not reported, we can be confident that this is a false positive.

c)

To run the program, do ./pipairPartC.sh test3/test3/bc 10 80 4

4 is the level to be expanded, 1 would return the same as partA

The algorithm is as follows. For every function in the call graph, we perform an expansion. Depending on the level of expansion specified, we push the function and its level on to a stack. Then we try to empty the stack. Initially, we push our main function, that is the function we are interested in, to the stack. For every function on the top of the stack that is popped, let's call that the current function; we check if it is already included. If not, we include that function. We also check if the maxmium level has been reached. If neither of these things happen we can start the expansion.

If we have reached the end of our expansion or there is no function called inside the current function, we keep that function in our list and move on. If we can expand, we first remove the current function from the list, then push all the function in our current function on to the stack, record their levels and move on to the next function in the stack. Eventually we will be able to empty the stack and we would have a list of the functions included for our main function, then we return it to our data structure that stores a list of functions for every function. Subsequently we use that map to analyze the support and bugs for the call graph.

There are a few details to consider, first, should we included a function that could be expanded, or should we remove it and replace it with the results of the expansion? We decided since inter procedral analysis is intended to improve the performance of the bug finding, it would be better to replace the function with sub functions. In real world applications such as the apache server, functions are often unique and uses are few in number. Therefore we can deduct the content of a function with the contents of its sub functions. Keeping the function itself would result in many more false positives.

As a our experiment, we set support to 10 and confidence to 80 and ran it on the apache httpd server code. We firstly tried level 2 expansion, which gives us 110 bugs. Level 3 gives us 601 bugs. Level 4 gives us 1758 bugs. Level 5 gives 6079 bugs. Level 8 gives 6915 bugs.

It can be deducted that the number of bugs increases in a linear fashion for the first few levels, then slows down after that. From the result of level 2 we can see that some false positives were eliminated. For example, the bug apr\_array\_make and apr\_array\_push disappeared after level2, which can be explained by the fact that the expansion eliminated these function, and the result of the expansion was adjudged to be not a bug. Other bugs are replaced with a bug that is deeper in the expansion tree, such as fputc in ap\_hook\_default\_port, pair: (apr\_palloc, fputc), which replaces the bug apr\_hook\_debug\_show in the partA.

Interestingly, some new false positives are introduced. For example bug: substring\_conf in ap\_getword\_conf\_nc, pair: (strlen, substring\_conf), support: 13, confidence: 81.25% is a new false positive.

There is no necessary relation between strlen, which is a c library function, and substring\_conf. The reason behind the increasing in bugs is by expanding the functions we are introducing much more nodes in each function of the graph. These nodes are deeper than first level, thus they will eliminat the false positives introduced at the first level of expansion, but introduced a great amount of new bugs at deeper levels. Since many new functions are found as we increase the number of the expansion, it can be expected that there will be a linear increase in the number of bugs. When we reach a certain level of expansion, we might reach the end of our expansion with some dead nodes. These nodes cannot be expanded and will stay dead even if we increase our levels further. As a result when we go beyond level 5 the increase in the number of bugs is very slow. We can deduce that perhaps 5 levels is around the same depath of function calls in one function.

In conclusion, our algorithm indeed do better than partA in that it eliminates many false positives. It also bloated the number of bugs, some valid, some not.

**PART II**

a)

**CID 10065**

Bug: Faulty lines are the missing breaks in each case (or cases from values 3 and up). Even though there are return statements for each case, some of them are wrapped in if statements, which means they may not be reached. If they are not reached, the program falls through to the next case of the switch statement, which may cause a failure (out of bounds), since characters outside the length of the string may be accessed.

Fix by adding break statements to each switch case.

**CID 10066**

Bug: The clone method is missing. The class implements Cloneable but does not define a clone method. This could create a problem when StrBuilder treated as Cloneable class.

Faulty Lines: Missing clone method

Fix: Define a clone method in StrBuilder, with appropriate implementation.

**CID 10067**

False Positive: This warning warns about using nextInt rather than nextDouble casted to int to generate random numbers. However, the warning line uses Math.Random static method, and this is the usual way of using it. Coverity may have confused the two cases.

**CID 10068**

Intentional: Application behaviour may vary between platforms, however this is the way to implement this type of method, printing to specified stream. It should be left as-is.

**CID 10069**

Intentional: The reason the developer is comparing class names instead of class objects is to differentiate between different class loaders, and this is mentioned in the comments above the if statement in question.

Additionally, we can see that in the preceding else ifs, we can see “other.getClass() == this.getClass()“, so there is no reason to compare class objects again.

It should be left as-is even though it may seem like bad practice.

**CID 10070**

Intentional: Same as CID 10069.

**CID 10071**

Bug: “==” operator compares by reference, however, the warning line compares a string to a string literal, and they can be different references (despite both string representing “true”). In the comment preconditions section of the *toBoolean* method, we can see that only string literals are passed in as parameters, this means that this method will work as intended for those inputs, but this is still bad practice and may very well result in an error if *toBoolean* is called on a string object (this could happen if the developer has not read the preconditions), therefore we classify this as **bug**.

Faulty Lines: line 614 from BooleanUtil.java

Fix: Replace faulty line with *str.equals(“true”)*

**CID 10072**

Bug: Same explanation as 10071, the only difference is that this is a different method which has 2 string inputs. This does not change the bug, both those inputs must still be string literals for this method to work properly, and that is not always the case, it is bad practice to compare strings using “==” operator.

Faulty Lines: line 4865 from StringUtils.java

Fix: Replace faulty line with *str.equals(“true”)*

**CID 10073**

Bug: Similar explanation as 10071, in this case value is compared to another string object (static final object), so they will not have the same reference and therefore the string comparison would always return false.

Faulty Lines: lines 385, 389, 393, 397, 401, 405, 409, DurationFormatUtils.java

Fix: Replace faulty line with a string value comparison, i.e. using *string.equals*, e.g. *value.equals(y)*

**CID 10074**

Bug: Similar to previous 3, the string from *previous.getValue()* is being compared using “==” to another string object. This may work depending on how the two strings being compared were initialized (as object vs literal) but is considered bad practice at the least so we label as bug.

Faulty Lines: line 485, DurationFormatUtils.java

Fix: Compare part of faulty line *previous.getValue() == value* with *previous.getValue().equals(value)*

**CID 10075**

Bug: This is bad practice, the code is attempting to compute the mid index at an iteration of binary search. It is intelligently using bitshift rather than division (which could cause overflow for very large arrays). However, the coder has used signed bitshift, which is very inefficient for this situation (there are no negative indexes), so this code can be refactored to use unsigned bitshifting, to further prevent integer overflow.

Faulty Lines: line 649, Entities.java

Fix: Change to unsigned bitshifting, i.e. *int mid = (low + high) >>> 1*

**CID 10076**

Intentional: In the comments there are preconditions which mention that a null *Boolean* *bool* parameter may be passed in to this method, in which case, the method is intended return null, therefore this was not a mistake. This may seem like bad practice at first, but it was entirely intentional since null return value was explicitly stated and explained in the comment section.

It should be left as-is.

**CID 10077**

Intentional: Similar to 10076, null is explicitly stated in the preconditions. This method is intended to return null if the value matches the nullValue parameter.

**CID 10078**

Intentional: same as 10076. From the preconditions, value parameter can be passed in as null Integer, and if it’s null, the return value should be null as well.

**CID 10079**

Intentional: Same as 10077 except this time nullValue parameter is an Integer.

**CID 10080**

Intentional: Method summary states that return value should be null if no match or null input string. It should be left as-is.

**CID 10081**

Intentional: Method converts string to Boolean based on some conditions, one of which includes the null if the string does not match any true/false value. It should be left as-is.

**CID 10082**

Intentional: This line of code will not cause problems but it is bad practice, the catch block catches all exceptions, which may mask runtime exceptions unintentionally. While there may not be runtime exceptions in this case, it is better to catch more specific exceptions or have multiple catch blocks.

Line 97 (ExceptionUtils.java) can be refactored to catching a specific exception type for good practice.

**CID 10083**

Bug: Rule is not serialization so it must be transient, but it is neither. This could cause problems during deserialization of the class if there is a non-Serializable object stored in the field.

Faulty Lines: line 137, FastDateFormat.java

Fix: add transient to faulty line, or ensure all objects in the *mRules* interface array are serializable (risky)

**CID 10084**

Intentional: A hashmap definitely requires a key as part of the key-value pair. The associated methods for hashmap happen not to make use of the key but this class may be expanded in the future and may require reading the key.

It should be left as-is.

b)

Coverity Analysis (cov-analyze):

Coverity Static Analysis version 8.0.0 on Linux 2.6.18-407.el5 x86\_64

Internal version numbers: 2c247ff7c4 p-jasper-push-24789.298

Using 8 workers as limited by CPU(s)

Looking for translation units

|0----------25-----------50----------75---------100|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[STATUS] Loading topological sort from disk (538 functions)

|0----------25-----------50----------75---------100|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[STATUS] Computing node costs

|0----------25-----------50----------75---------100|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[STATUS] Starting analysis run

|0----------25-----------50----------75---------100|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[STATUS] Starting analysis run (types/warnings pass)

|0----------25-----------50----------75---------100|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Analysis summary report:

------------------------

Files analyzed : 30

Total LoC input to cov-analyze : 38370

Functions analyzed : 538

Classes/structs analyzed : 124

Paths analyzed : 1326

Time taken by analysis : 00:00:01

Defect occurrences found : 0

Unfotunately, Coverity did not find any bugs/defects, as shown above. Please view *output* file located inside pii directory for more details.

This may be in part due to C++ being the language of choice. C++ was designed for efficient execution so there may be many operations that may be considered bad practice but are used to enhance performance. Since C++ is based on C, it is more flexible and follows fewer conventions than most other programming languages. Overall, C++ does not have as many syntactic/semantic restrictions. Naturally, Coverity may reflect the aspects of C++ and java programming languages, and may show more lenience towards C++ code. One example is the NULL return value from a Boolean function, which is detected by the java static bug detection tool, but not the C++ one.

In addition, as we are experienced developers, we are able to prevent many bad practices that newer programmers may make, which Coverity would detect.