# Software Testing, Quality Assurance, and Maintenance Project
# Winter 2015

Group 10

Jongseok Park, 20323127, CS 447, j58park@uwaterloo.ca

Karol Tinsley, 20309119, CS 447, kntinsle@uwaterloo.ca

Sandford Zhu, 20393653, CS 447, sjzhu@uwaterloo.ca

## Part I (b) Finding and Explaining False Positives

Why are there false positives? Our analyzer simply detects common pairs of functions and assumes that if there are enough pairs it can determine with a certain confidence that the pair belongs together. One reason why this type of analysis fails and detects non-existent bugs is because sometimes a pair of functions which appear together often does not necessarily mean they absolutely need to appear together. It is possible that the functions can be used individually without any problems yet will be falsely flagged as a bug. Another reason false positives occur is sometimes the support or confidence level may be set too low. The lower threshold will create more bug outputs which are false positives because the analyzer will detect weaker function pairs in the code.

The following analysis is using a support level of 10 and confidence level of 80 to analyze httpd.

The function apr_array_push in the pair (apr_array_push, apr_array_make) appears ten times as a bug in the output log. This pair of functions appear together often because an array must be created first before pushing an item into the array. However, locations marked as a bug assume either a properly initialized array already exists or the calling function creates an array before apr_array_push is used. An example of this is in ap_location_walk where another function prep_walk_cache is used to copy or create a new array before apr_array_push is used. Since the pair (apr_array_push, apr_array_make) do not need to appear together, this is a false positive.

The function apr_hook_debug_show appears many times in the bug output and is commonly paired with either apr_array_make or apr_array_push. However, this is a debug statement which does not affect the code, therefore every bug report containing this function is a false positive.

## Part I (c) Inter-Procedural Analysis

**Implementation**
Our bug detector has 3 major component, Parser.java, Invariants.java, and lastly BugDetector.java. The data flow flows from Parser.java -> Invariants.java -> BugDetector.java.

There are 3 major data structures that is being generated by Parser.java in order to detect bugs:

1. List<Scope> mScopes; // this holds list of scopes from the callgraph
    a. Class Scopes {String name; Set<String> functions;}
2. Map<String, Integer> mIndividualHashMap; // this holds function frequency
3. Map<Pair, Integer> mFunctionPairHashMap; // this holds pair frequency
    a. Class Pair {String function1; String function2;}

In order to implement inter-procedural analysis, we added another step inside Parser.java called populateDepthScopes(mDepth) to generate list of scopes (mScopes) differently.

The steps are as follows (Parser.java: populateDepthScopes(mDepth)):

1. Parse callgraph that was generated by llvm and generate mScopes
2. Copy mScopes called tempScopes
3. Iterate through mScopes and iterate through its functions and replace each function with mScopes.functions if the function exists
    a. for(Scope *scope* : *mScope*) {

```
for(Scope scope : mScope) {
        for(String function : scope.functions) {
                if(function exists in tempScopes) {
                        - replace current function with tempScopes[function]
                        - make sure that it doesn't create a cycle
                }
        }
}
depth = depth - 1;
```

4. Loop back to step 3, if depth > 0

Implementation is located inside Parser.java, line: 103 ~ 130

populateDepthScopes(mDepth) will only run when mDepth is greater than 0. mDepth is 0 by default and it is passed by 3rd parameter in args. When each function gets expanded, it will make sure that it will not create a cycle (it will not append function).

**Analysis**

| Test File | Support | Confidence | Depth | Cycle Remove | # of Bugs |
|---|---|---|---|---|---|
| test3.bc | 10 | 80 | 0 | false | **34** |

| | | | | | |
|---|---|---|---|---|---|
| test3.bc | 10 | 80 | 1 | false | 136 |
| test3.bc | 10 | 80 | 1 | true | **156** |
| test3.bc | 10 | 80 | 100 | false | 638 |
| test3.bc | 10 | 80 | 100 | true | **614** |
| test3.bc | 10 | 80 | 250 | false | 638 |
| test3.bc | 10 | 80 | 250 | true | **614** |
| test3.bc | 10 | 80 | 500 | false | 640 |
| test3.bc | 10 | 80 | 500 | true | **270** |

In order to determine if depth reduces false positives, I will use 2 false positive bug from part 1 (b) in depth 0 for gold_10_80, *(apr_array_make, apr_array_push), (apr_array_make, apr_hook_debug_show)* and compare it with depth 1, and 500 with cycle removed on.

*Running httpd with support 10, confidence 80, depth 0, 1, 500, cycle remove on*

| **Depth Level** | **False Positive Pair** | **# of Bugs** |
|---|---|---|
| 0 | *apr_array_make, apr_array_push* | **16** |
| 1 | *apr_array_make, apr_array_push* | **0** |
| 500 | *apr_array_make, apr_array_push* | **0** |
| 0 | *apr_array_make, apr_hook_debug_show* | **4** |
| 1 | *apr_array_make, apr_hook_debug_show* | **0** |
| 500 | *apr_array_make, apr_hook_debug_show* | **0** |

As shown on the table above, both false positive pairs are not to be found (0 bugs found) from depth 1 and depth 500.  Therefore, it runs better than original algorithm by expanding the function in scope to another scope.

# Part II (a) - Resolving Bugs in Apache Commons

**CID 10022 - False Positive**

This is a false positive because creating a new KeySetView calls super in it's constructor.

**CID 10023 - False Positive**

Coverity has flagged this issue because in other instances where the function keyset() is used, it had a call to the superclass keyset(). The flagged code does the same thing functionally as calling the superclass (return map.keySet())so this is not an issue.

**CID 10024 - False Positive**

*Same warning as CID 10041*

Coverity has flagged this issue because in other instances where the function keyset() is used, it had a call to the superclass keyset(). The flagged code does the same thing functionally as calling the superclass (return map.keySet()) so this is not an issue.

**CID 10025 - False Positive**

This warning is falsely accusing currentIterator to be possibly null which then gets dereferenced. The function findNextByIterator() has a condition where if iterator doesn't equal currentIterator then currentIterator will be set to iterator and currentIterator will not be dereferenced. This branch will always be true if currentIterator is null since the iterator that is passed into findNextByIterator() has been checked to be a valid iterator, thus a null value will never be dereferenced.

**CID 10026 - Intentional**

The unguarded read was left intentionally in the code as a way to improve the speed at which the code runs in a multithreaded environment when it runs in "fast" mode. The documentation in the code states that FastArrayList.java is meant to be used in an environment which is mostly read-only to prevent errors which means the developers were aware of the situation.

**CID 10027 - False Positive**

This is a false positive because the conditional branch requires the deleted node to have both a left and a right index. This means that calling the functions nextGreater() and then leastNode() on the deleted node will never return null because deleted node is not null and has a left and right index.

**CID 10028 - False positive**

This appears to be a false positive because previously there was always a lock on FastArray.this. Before reaching the point where last is accessed, checkMod() is used to check if the list had been modified which prevents changes to last if the list was modified.

**CID 10029 - Intentional**

This warning indicates in the FastHashMap.java class there is a check of a thread-shared field (*lastReturned*) that evades lock acquisition, which could lead to multiple threads having inconsistent states. This is intentional and the intended use of the FastHashMap class. In the javadoc it states:

“[FastHashMap is a] customized implementation of java.util.HashMap designed to operate in a multithreaded environment where the large majority of method calls are read-only, instead of structural changes.  When operating in "fast" mode, read calls are non-synchronized and write calls perform the following steps: …”

Even though there may be a fault it is clearly documented that operations may fail:

**“NOTE**: *This class is not cross-platform. Using it may cause unexpected failures on some architectures*. It suffers from the same problems as the double-checked locking idiom. In particular, the instruction that clones the internal collection and the instruction that sets the internal reference to the clone can be executed or perceived out-of-order.  This means that any read operation might fail unexpectedly, as it may be reading the state of the internal collection before the internal collection is fully formed.”

Therefore the code should be left as is, and the warning should be ignored.

**CID 10030 - Intentional**

This warning is a similar warning to CID 10029, except it is located in a different part of FastHashMap.java. Like CID 10029, this is intentional as it is the intended use of the class.

**CID 10031 - Intentional**

This warning indicates that in the *rotateLeft* function, the call to *getRightSubTree()* can explicitly return a null value, thus when the line *getRightSubTree().getLeftSubTree()* is called, the program could dereference a null value. This was done intentionally by the developer, as they assumed the right sub tree will not be null if you are trying to rotate the AVL tree to the left. There really is no guarantee that the right sub tree will not be null, so to fix this warning the developers should add in a check to see if the rightSubTree is null before trying to access it.

**CID 10032 - Intentional**

For this warning Coverity indicates that in the EntryIterator for the StaticBucketMap.java class, there is a thread-shared field (*bucket*) that evades lock acquistion, which could lead to multiple threads having inconsistent states. This was done intentionally by the developers as the threads are managed by monitors. In the javadoc at the beginning of class it states:

“The iterators returned by the collection views of this class are *not* fail-fast.  They will *never* raise a java.util.ConcurrentModificationException. Keys and values added to the map after the iterator is created do not necessarily appear during iteration.  Similarly, the iterator does not necessarily fail to return keys and values that were removed after the iterator was created.”

So the code should stay as is and the warning should be ignored.

**CID 10033 - False Positive**

For this warning Coverity indicates the arguments given to a new instance of an *UnmodifiableMapEntry* are in the wrong order. In the function containing the warning, it contains a switch statement on the int *dataType*. When *dataType* is equal to INVERSEMAPENTRY (a static int set earlier in the class), the function returns a map entry where the key and value are reversed. The name INVERSEMAPENTRY suggests that it is intentionally inverting the key and value of the map, and therefore it is a false positive.

**CID 10034 - Intentional**

This is a duplicate of warning CID 10032. The StaticBucketMap.java class is duplicated in another folder.

**CID 10035 - False Positive**

For this warning Coverity is indicating that there is an unguarded write on a field *last.* In most locations, *last* is accessed while holding a lock and therefore it seems like this is a mistake. This is intentional as the first thing in the function checks to see if the given list has been modified and then throws a ConcurrentModificationException(). Therefore there is no possibility for a fault.

**CID 10036 - Intentional**

This warning indicates in the FastTreeMap.java class there is a check of a thread-shared field (*lastReturned*) that evades lock acquisition, which could lead to multiple threads having inconsistent states. This is intentional and the intended use of the FastTreeMap class. The javadoc is exact to that of FastHashMap with the class name changed:

“[FastTreeMap is a] customized implementation of java.util.TreeMap designed to operate in a multithreaded environment where the large majority of method calls are read-only, instead of structural changes.  When operating in "fast" mode, read calls are non-synchronized and write calls perform the following steps: ...”

The same note indicating that the class may contain unexpected failures is also present in FastTreeMap. Therefore the code should be left as is, and the warning should be ignored.

**CID 10037 - Intentional**

This warning is similar to CID 10031 and indicates that in the *rotateRight* function, the call to *getLeftSubTree()* can explicitly return a null value, thus when the line *getLeftSubTree().getRightSubTree()* is called, the program could dereference a null value. This was done intentionally by the developer, as they assumed the left sub tree will not be null if you are trying to rotate the AVL tree to the right. There really is no guarantee that the right sub tree will not be null, so to fix this warning the developers should add in a check to see if the getLeftSubTree is null before trying to access it.

**CID 10038 - Bug**

A potential deadlock could occur for line 1136. *return get(expected).indexOf(o)* already holds *list* lock before acquiring *lock* lock when calling indexOf(o), however, coverity detected 5 different places where it holds *lock* lock first before acquiring *list* lock, therefore, this line could potentially be a deadlock.

In order to fix this bug, make sure to acquire same order as others by acquiring *lock* lock before *list* lock.

**CID 10039 - False Positive**

*nextGreater(deletedNode, index)* will never be *null* because of the *if* check on *(deletedNode.getLeft(index) != null) && (deletedNode.getRight(index) != null)*. *Right node* is not null, and *nextGreater* will call *leastNode* on *right node* which returns a non *null* since it right node was never null to begin, thus, this is false positive. So, there is no possibility for a fault.

**CID 10040 - Bug**

A Potential deadlock could occur for line 545. *return get(map).isEmpty()* already holds *map* lock before acquiring *lock* lock when calling *isEmpty()*, however, coverity detected 5 different places where it holds *lock* lock first before acquiring *map* lock, therefore, this line could potentially be a deadlock situation.

In order to fix this bug, make sure to acquire same order as others by acquiring *lock* lock before *map* lock.

**CID 10041 - Intentional**

Coverity is indicating that there is a *"volatile not atomically updated"* warning for *modCount* variable inside *ReferenceMap.put* function. *modCount* is a *transient volatile int* variable that is expecting synchronization lock. But the lock is not neccesary here because according to Apache.org ReferenceMap documenation[1], it states that "***Note that ReferenceMap is not synchronized and is not thread-safe.*** *If you wish to use this map from multiple threads concurrently, you must use appropriate synchronization*". Therefore, this was intentional and there is no possibility for race condition because the user has to create their own synchronization around *ReferenceMap* object.

The code should stay as is and ignore the warning should be ignored.

**CID 10042 - Intentional**

This warning is the same as CID 10041. Coverity is indicating that there is a *"volatile not atomically updated"* warning for *modCount* variable inside *ReferenceMap.remove* function. *modCount* is a *transient volatile int* variable that is expecting synchronization lock. But the lock is not neccesary here because according to Apache.org ReferenceMap documenation, it states that "***Note that ReferenceMap is not synchronized and is not thread-safe.*** *If you wish to use this map from multiple threads concurrently, you must use appropriate synchronization*". Therefore, this was intentional and there is no possibility for race condition because the user has to create their own synchronization around *ReferenceMap* object.

The code should stay as is and ignore the warning should be ignored.

---

[1] **Apache.org** ReferenceMap Documentation
*https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.1/org/apache/commons/collections/map/ReferenceMap.html*

## Part II (b) - Analyzing Your Own Code

In our code Coverity found 4 defects. We will discuss the defects CID 10062 and CID 10063.

In CID 10062, Coverity is warning us that there is a deference null return in the *populateScopes()* function of the Parser class. More specifically, when trying to read in a call graph, the call to *br.readLine()* on line 34 could return a null. On line 37 the while loop condition is *!line.isEmpty()*. So if the line read on line 34 was null then we would be trying to dereference a null value.

This defect is intentional. The call to *br.readLine()* is surrounded by a try-catch, where the general *Exception* is caught. So if the program did try to dereference a null value, the exception would be caught and the error message would be printed to the console. However, this is not a best practice. We can do a few things to fix this error. The first being adding an if statement before the while loop to see if *br.readLine()* returned null, and if it did handle the exception gracefully. Or we can add a NullPointerException catch to the try-catch in addition to the parent class *Exception*. This way if there was an exception we could pinpoint where it happened more easily.

For CID 10063, Coverity is warning us of a resource leak also in the *populateScopes()* function of the Parser class. When reading in a call graph the FileReader and BufferedReader are never closed, therefore the readers remain open at the end of the scope and those resources are lost until the program is finished executing.

This is a bug. To fix this we should add a finally statement to the try-catch in the populateScopes() function. In this finally statement we need to include the lines:

*br.close();*
*fileReader.close();*

After these changes are made, the defect should be fixed.