ECE653
Group 17

Matinkhoo, Sadaf (20588163), smatinkh@uwaterloo.ca
Mazahir, Filza (20295951), fmazahir@uwaterloo.ca
Medghalchi, Nazli (20548504), medghalchi@uwaterloo.ca

April 6, 2015

Part (I)

(I)-a Inferring Likely Variants for Bug Detection

The codes for this part can be found in the *pi* directory. The algorithm is explained in detail inside the source code and in part (I)-c.

(I)-b Finding and Explaining False Positives

These are the reasons our code is finding false positives:

1. reason 1
2. reason 2

These are the two pairs of functions that are false positive for `httpd`:

1. `(apr_array_make, apr_array_push)`

   The function `apr_array_make` in the above pair has a bug on 6 locations, with a support of 40 and a confidence of 86.96%. The function `apr_array_push`, on the other hand, has a bug on 10 locations, with a support of 40 and a confidence of 80.0%. This means that over 80% of the time, the `apr_array_make` and `apr_array_push` are called together.

   Upon looking at the source code, it can be seen that the function `apr_array_make` is used to create an array, whereas the function `apr_array_push` is used to increase the size of the array. The bug being examined here is:

   bug: apr_array_make in ap_init_virtual_host, pair: (apr_array_make, apr_array_push), support:40, confidence: 86.96%

   In the `ap_init_virtual_host` function, `apr_array_make` is being called twice to create an array of size 4. The decision of whether `apr_array_push` is called to increase the array size depends on the use case as the functionalities of both the functions are independent of one another. Using `apr_array_push` does not seem to be required in this particular function.

2. `(apr_table_setn, apr_table_unset)`
   The function `apr_table_unset` in this pair has a bug in 4 locations with support of 8 and confidence of 66.67%. The bug being examined here is:

   bug: apr_table_unset in add_env_module_vars_unset, pair: (apr_table_setn, apr_table_unset), support: 8, confidence: 66.67%

   In the `add_env_modules_vars_unset` function, `apr_table_set` is used instead of `apr_table_setn`. The functionalities of `set`, `setn`, and `unset` functions are described on Apache as following:

   - **set:** adds a key/value pair to a table. If another element already exists with the same key, this will overwrite the old data. When adding data, this function makes a copy of both the key and the value.
   - **setn:** adds a key/value pair to a table. If another element already exists with the same key, this will overwrite the old data. When adding data, this function does not make a copy of the key or the value, so care should be taken to ensure that the values will not change after they have been added.
   - **unset:** Remove data from the table.

   The only difference between `set` and `setn` is that one keeps a copy of the data and the other does not. The decision between the two depends on the use case. In any case, the functionalities of `set/setn` is independent from `unset`, and using `set` instead of `setn` with `unset` is not a bug.

(I)-c Inter-Procedural Analysis

We implemented one general algorithm to do both intra- and inter-procedural analysis. This is how the algorithm works:

First, the algorithm parses the arguments that are passed to it. It needs at least one argument which is the name of the callgraph file. Other than that, it can accept up to three additional arguments for support threshold, confidence threshold, and the desired level of expansion for inter-procedural analysis. The latter has a default value of zero, which means no expansion happens i.e. intra-procedural analysis is performed.

Then, the callgraph gets parsed and the algorithm creates a hashtable with node names as keys and functions that are called in each node as values. This is just the raw initial parsing with no level of expansion. For simplicity, we will call this hashtable RawInfo in this report.

To create the desired data structure that is used to find bugs, the algorithm uses a recursive function to retrieve the function calls in each node to the desired level of expansion. Following are the pseduo codes for two algorithms that are used together in this step.

Algorithm 1 CreateDataStructure
```
1: for each node $n$ in RawInfo do
2:     functions ← RawInfo[$n$]
3:     for each function $f$ in functions do
4:         expand $f$ to level $l$
5:     end for
6: end for
```

Algorithm 2 GetFunctionCalls(Node *node*, Level *level*, Path *path*, Functions $functionCalls$)

```
1:  if $node$ in $path$ then
2:      return {avoid loops}
3:  end if
4:  if RawInfo[$node$] is empty or $level < 0$ then
5:      Add $node$ to $functionCalls$
6:      return
7:  end if
8:  Add $node$ to $path$
9:  for each function $f$ in RawInfo[$node$] do
10:     GetFunctionCalls($f$, $level - 1$)
11: end for
12: Remove $node$ from $path$
```

Our algorithm uses two other hashtables in the data structure. The first one has function names as keys and nodes in which each function is called as values. The second hashtable has function pairs as keys, and nodes where each pair is called as values. The rest of the algorithm is described in detail inside the source code. After the above step is performed on each node in RawInfo, the functions and function pairs in that node are added to these two hashtables, and once all iterations are finished, our data structure is complete. Note that since we used hash sets and hashtables, no string comparison is performed in this algorithm; thus, it is very fast.

To run our code in inter-procedural mode, assuming the source code has already been compiled and you are currently in the root of the skeleton folder, you would do any of the two following lines to analyze the bitcode file in test2 folder as an example:

```
./pipair test2/main.bc 10 80 1
./piapir test2/main.bc 1
```

Note: To avoid loops in function calls (that would result in oscilation in the number of bugs), the program keeps a record of parent functions for each function and stops if it reaches any of the parents when expanding a scope.

(I)-d Improving the Solutions

A fifth argument was added to the program to implement an algorithm to improve the solutions. The fifth argument is expected to be 1 for reducing false positives, and 2 for finding more bugs. To run the program in `pi/partD/BugDetectionD.java`, run `make` and then
./pipair ¡bitcode.bc¿ ¡support¿ ¡confidence¿ ¡0, 1 or 2 for level¿ ¡1 or 2 for algorithm no.¿
These are the approaches we propose to improve the solutions:

Reducing false positives ('1' input for 5th argument):
One way to reduce false positives is to have a sliding confidence threshold based on the support of each function. If support of a function is high, the threshold confidence should increase accordingly. We propose this relationship to be that of a linear function, such as:
$confidenceThreshold = \frac{(99-minConfidence)*(pairSupport-minSupport)}{100-minSupport} + minConfidence.$

An example is shown below to illustrate:
Input: T_SUPPORT = 10, T_CONFIDENCE = 80%
If support {A,B} is 10, then minimum confidence of {A,B}, {A} is 80%.
If support {A,B} is 50, then minimum confidence of {A,B}, {A} is 88%.
If support {A,B} is 90, then minimum confidence of {A,B}, {A} is 97%.

The rationale behind the sliding threshold of confidence is because in the current algorithm, the confidence is merely a percentage and does not take the absolute number of times the function is being called into account. The purpose of finding likely invariants is to find the outliers, therefore, the actual support of the function is important to reduce false positives. If a function is being called separately once with a total support of 10, it is more likely to be an outlier compared to if that function is being called separately 10 times with a total support of 100. A bigger support, meaning a bigger data set, should therefore mean a bigger confidence level. For a simpler algorithm, it is assumed that the minimum threshold of support will not be higher than 99.

Finding more bugs ('2' input for 5th argument):
For the purpose of this project and according to project descriptions, "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, we can take the order into account to find more bugs. Following is an example of how this strategy would work:

```
Scope1 {                  Scope2 {                  Scope3 {                  Scope4{
    A;                        B;                        A;                        A;
    B;                        A;                        B;                    }
}                         }                         }
```

In the default case, and with support and confidence threshold of 2 and 65%, the program would detect only the following bug:

```
bug: A in Scope4, pair: (A,B), support: 3, confidence: 75%
```

With differentiating `(A,B)` from `(B,A)`, the program will detect two other bugs as well:

```
bug: A in Scope2, pair: (A,B), support: 2, confidence: 66.67%
bug: B in Scope2, pair: (A,B), support: 2, confidence: 66.67%
```

Part (II)

(II)-a Resolving Bugs in Apache Commons

10022 - False Positive

The super class has actually been called but in the `KeySetView()` method instead of `KeySet()`.

10023 and 10024 - False Positive

Below, is the implementation of the super class. As you can see, `map` is a protected object meaning it is accessible by all of its subclasses. Thus, subclasses have the option to call the super class (like in the 6 instances that it is done) or use `map` directly, as in these two cases.

```
Super Class - AbstractMapDecorator:
45     protected transient Map map;

102    public Set keySet() {
103        return map.keySet();
104    }
```

10025 - False Positive
The erroneous line (147) is only implemented in case `currentIterator=Null, root!=N̄ull, transformer!̄=Null`. So, `findNext(transformer.transform(root))` can be called without any problem and passes a non-Null value to `findNext()`. In case that `findNext()` calls `FindNextByIterator()`, it passes a non-Null `Iterator` object to it since the value passed to `findNext()` was not Null. Hence, the `iterator` in `findNextByIterator()` is not Null while `currentIterator` is Null (one of the conditions this path is taken). This means that `currentIterator!=iterator` and the code enters the first `if` statement in `findNext-ByIterator()` and sets `currentIterator = iterator` before dereferencing `current-Iterator` in the `while` loop.

10026 - False Positive
This line is contained in a private method. It can only be accessed by other public methods in this class. Hence, it does not need to hold the lock itself; the lock only needs to be held by the public methods calling `get()`. In this case, after going through all uses of `get()`, it seems that this private method is always called through synchronized context, so synchronizing it would be redundant.

10027 - False Positive
The erroneous line (883) is only reached when `deleteNode.getLeft(index)` and `delete-Node.getRight(index)` are not `Null`. This implies that `deleteNode` itself is not `Null` either. Then, `deleteNode` gets passed to `nextGreater()` where line 541 in `else if` statements is implemented. Considering the condition i `else if`, the `node.getRight(index)` that gets passed to `leastNode()` is not `Null` and `leastNode()` does not return `Null`. Thus, this is a false positive. However, another bug exist here: if `delereNode` at the beginning of `doRedBlackDelete()` is `Null`, an exception will occur at line 881. To fix this bug, `deleteNode` should be checked before reaching the `if` statement.

10028 - False Positive

10029 - Bug
Lines 656 - 664 of FastHashMap.java
Bug occurs when the code is run multi-threaded. At line 656, the threads read the value of `lastReturned`, but does not lock it until line 660. This can cause a bug because if there are two threads, one thread would change the value of `lastReturned` to `null` at line 665. When the lock is released and the second thread is acquired, it would not have read the new value of `lastReturned` as `null`, and will try to execute line 664, resulting in `lastReturned.getKey()` causing a `NullPointerException`. A proposed bug fix would be to lock `FastHashMap.this` before line 656, hence the `synchronized (FastHashMap.this)` block should start before line

656 instead of at line 660.

10030 - False Positive
Lines 547 & Line 650 of FastHashMap.java
The warning occurs because the analytical checker sees a potential deadlock. At line 547, `map` is locked if condition fast is false, and then is trying to lock `SynchronizedCollection.lock`. In line 650, condition `fast` is `true` and `SynchronizedCollection.lock` is being acquired here. However, the order of locking is the same in both of these, and in line 650 when `SynchronizedCollection.` is acquired, it is not trying to acquire lock on `FastHashMap.map`, so there is no way for a potential deadlock.

10031 - False Positive
Line 656 of TreeList.java
The warnings shows that 10 out of 14 times `getRightSubTree()` returns `null` so it could possibly cause an error on line 656. However, on checking the uses of the `rotateLeft()` method, it can be seen that this has already checked for. In line 596, the method `rotateLeft()` is being called on the left node when `heightRightMinusLeft() > 0` which ensures that a right sub tree exists, hence making sure that `getRightSubTree()` will not result in null. The second time `rotateLeft()` is being called is in line 603 when `heightRightMinusLeft()` is 2, which means that since its a positive number, the right sub tree exists, and `getRightSubTree()` cannot be returned null.

10032 - False Positive
Line 504 - 514 of StaticBucketMap.java
This method is meant to check if there are any more nodes to process in the bucket map. If there are two threads that add nodes to the `current` array list, it simply increases the size of `current` array list. This does not cause a bug since the return value of `hasNext()` only depends on the `current` array list not being empty, so having an increased size does not cause any issue.

10033 - False Positive
Line 1603 of TreeBidiMap.java.
The warning shows the case where the arguments are swapped, that is, `lastReturnedNode.getValue()` is passed to key, whereas `lastReturnedNode.getKey()` is passed to value of the `UnmodifiableMapEnt` method. It is not an error because this method is being called in the case `INVERSEMAPENTRY`, where the map and key are meant to be reversed.

10034 - False Positive
Line 513-526 of StaticBucketMap.java
This method is meant to check if there are any more nodes to process in the bucket map. If there are two threads that add nodes to the `current` array list, it simply increases the size of `current` array list. This does not cause a bug since the return value of `hasNext()` only depends on the

`current` array list not being empty, so having an increased size does not cause any issue.

10035 - False Positive
Line 1221 of FastArrayList.java
The warnings shows that 9 out of 12 times, `last` was accessed after locking `FastArrayList.this`, but in this case its being modified without the lock. However, it was noticed that every time `last` was modified in the 9 examples where `FastArrayList.this`, it was really because `list` was being modified, and `last` just happened to be there in the sequence of the code. In the lines below, `list` is not being modified, so the lock is not required.

10036 - Intentional
Line 768 of FastTreeMap.java
This warning is Intentional and could be fixed by checking thread-shared field inside locked and guarded region. Because, based on Coverity results, the warning is caused under if statement `if(fast)` at line 762, which is throwing an exception. Also it could be fixed as it does have high impact. The fix is `synchronized` the `if` statement. The method `remove()` supposed to remove thread whichever is under condition `fast` and then using `synchronized(FastTreeMap.this)`. This makes it locked region. However locking the condition makes no data race issue.

10037 - Intentional
Line 673 TreeList.java
This intentional warning could be left as-is. At line 672 left child is set as `newTop`. At line 673 is checking for `leftIsPrevious`. Therefore theres a chance that `getLeftSubTree()` could return null value without checking. Which means null return. And this is Intentional warning. Although it happens 9 out of 13 times (
%70), it could be left as-is. However thep possible solution is checking null value instead of returning null value explicitly.

10038 - False positive
Line 1136 FastArrayList.java
This warning is for resource deadlock that may occur. Based on Coverity results two thread at line 1134 acquiring lock. Hence if each holds the lock that the other is waiting for, the result will be potential thread deadlock
(`collections.FastArrayList$SubList.indexOf(java.lang.object)`).

10039 - Intentional
Line 1019 TreeBidiap.java
This warning is for Difference null return value. This intentional warning should be left as-is. Because the `swapPosition()` method is called after checking `if` statement which checks for not being null value. Therefore, if either conditions are not satisfied, the `if` statement will not run and so the `swapPsition()` will not run. So this intentional warning should be left as-is.

10040 - False Positive
Line 653 FastTreeMap.java
This warning is regarding the case where two threads trying to acquire locks which one holds the

other thread's needed lock. Based on Coverity results in `org.apache.commons.collections.FastTreeMap$CollectionView.isEmpty()` threads try to acquier lockk in different order. And the case is when a thread acquiring `SynchronizedCollection.` while holding `FastTreeMap.map`. However, the virtual call resolves to `org.apache.commons.collection.AbstracCollectionDecorator.isEmpty`.

10041 - Intentional
Line 555 ReferenceMap.java
This warning regarding `ReferenceMap.remove` with high impact is intentional which could be solved by following potential solution. Based on Coverity results, depending on threads execution, as there's no key held on `modCount`, there might be a chance for volatile updating non-atomicity. As Coverity warning, any intervening update in another thread is overwritten. Potential fix for this warning is to synchronized and make it atomic updating `modCount`.

10042 - Intentional
Line 582 ReferenceMap.java
This intentional warning is concerning non atomic updating `modCount`. There's a chance of interleaving of threads execution as neither the `while()` loop, `if` statement are atomic. Therefor, at the time reading the `modCount` value in one thread, the other thread could write the `modCount` value. It is a hazard occurring because of non-atomic updating. The potential fix for this warning is to make `if` statement as locked region. Therefore there will be no volatile atomicity.

(II)-b Analyzing Code through Coverity

These are the six warnings that Coverity detected in our code (BugDetection.java):

1. CID 10272 - Resource leak. Bug
   In BugDetection.java.parseCallGraph() there is this Coverity warning regarding leak of system resource. Actually, based on CWE-404, it means that either the code is not releasing the resource or it does incorrectly release resource before actually could be made available for re-use. The problem starting from line 152 where new resource is created and assigned but is not closed or saved in `readline`. Then at line 153 if it takes false branch, it runs line 186. Which then goes out of scope and leaks. This is bad practice and true positive (Bug). Therefore it should be fixed. A fix for this bug could be somehow save the `readline` values. Or `catch(Exception ex)` at very end of code.

2. CID 10273 - Dm: Dubious method used. Bug (True positive)
   This warning is pointing out that there is a conversion of byte to string or string to byte and this will result in assuming that default platform encoding is suitable. It is reported based on `FB.DM_DEFAULT_ENCODING/ FindBugs: Internationalization - Dm: Dubious method used`. In order to fix this bug, `FileReader(CallGraph)` should be done first then it could be passed to `BufferReader()`.

3. CID 10274 - Dm: Dubious method used. Bug (True positive)
   This warning at line 135 in BugDetection.java is in BugDetection.parseArgs(java.lang.String[]) which is invoking system exit shut down the entire Java Virtual Machine. in this case it could be considered as intentional as developer tried to exit code in case that there are wrong number of input arguments. However it is bad practice. Either Intentional or Bug (True positive) it

should be fixed in order to prevent shutting down entire Java Virtual Machine. According to `FB.DM_EXIT/ FindBugs: Bad practice- Dm: Dubious method used` This should be only done when it is really appropriate. A potential fix for this bug is to use `System.exit(0)` and then from main method exit entirely from Java Virtual Machine.

4. CID 10275 - Dm: Dubious method used. Bug
   This warning is pointing to `system.exit(-1)` at line 189. In `BugDetection.parseCallGraph(`
   This line of code tries to exit from Java Virtual Machine entirely if earlier line at 186 (`catch(Exception ex)`) runs. A fix for this bug is to use `System.exit(0)` and then shut down Java Virtual Machine from main method. The other possible solution could be as this bug i repeating, there could be exit class defined (or exit method) that will handle all exit and JVM shutting down issues.

5. CID 10276 - Inner class should be made static. Intentional
   This warning is regarding the inner `Pair` class which Coverity mentions that could be defined as static class. This means there could be only one instance of that inner `Pair` class. This is actually pointing out for each outer class, `Pair`, there could be just one left and one right. This warning does have low impact and making inner class `Pair` to static would make wrong results. However it could be fixed by adding `Static` keyword to inner class, `Pair`. But it is better to be left as-is.

6. CID 10277 Format string problem - Intentional
   This warning at line 353 is saying instead of `%n` for string format,
   `n` is used which is for newline format. This warning is Intentional as developer tried to print out each bug report on newline. Also developer tried to print out percent sign right after the value of the variable. This warning is resulted because there is `%%` which first one has to switched to
   in order to print that second `%`. This has to be fixed, although it does have low impact based on Coverity results.