CS 408 Project 2
Project Report
Troy Gayman (tgayman@purdue.edu)
Niyati Sriram (nsriram@purude.edu)

Part 1B.

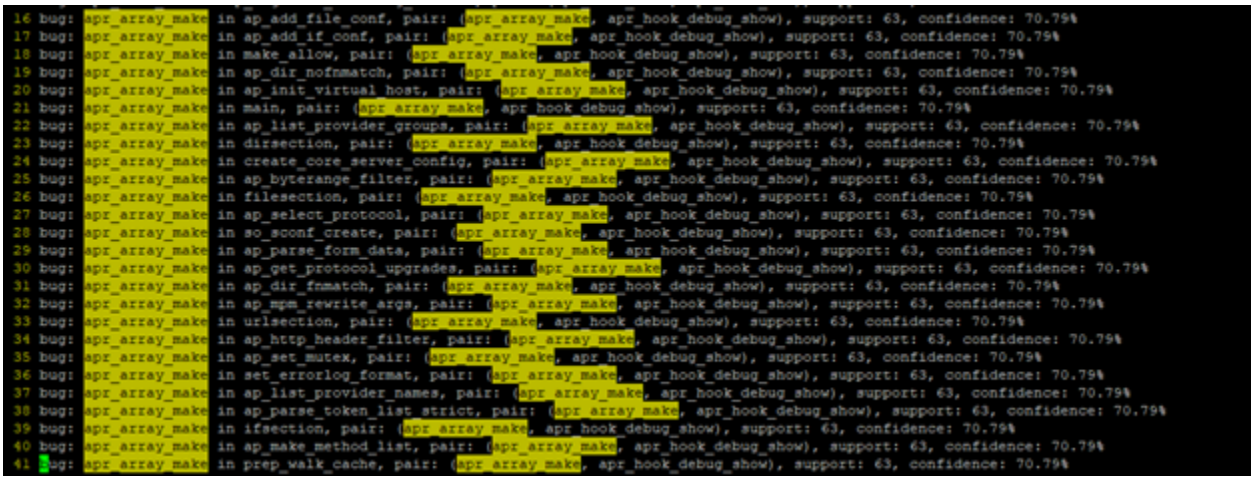
. Two fundamental reasons why a false positive could occur:

It could be that the debugging program is not aware of certain exceptions to the rule it follows to locate bugs. False positives could also occur when the thresholds used to identify bugs are incorrect or too low.

The following false positive pairs resulted from running test3_3_65 on our implementation from part A.

Pair 1: (apr_array_make, apr_hook_debug_show)

- Number of bugs: 26



- The function set_errorlog_format() in httpd-2.4.41/Server/core.c was flagged as a bug for calling apr_array_make() without calling apr_hook_debug_show(). In this function, apr_array_make() was called to set a field of a conf object.

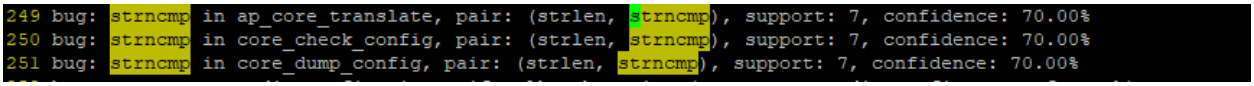
  //line from set_errorlog_format()

  ```
  conf->error_log_conn = apr_array_make(cmd->pool, 5, sizeof(apr_array_header_t
  *));
  ```

  apr_hook_debug_show() is a debugging function that is frequently called with apr_array_make() but they don't have to be called together by necessity. This false positive could have been avoided by using a higher confidence threshold.

Pair 2: (strlen, strncmp)

- Number of bugs: 3



- The function ap_core_translate() in httpd-2.4.41/Server/core.c was flagged as a bug for calling strcmp() but not strlen(). In this function strncmp was used to determine if a given URI is valid or not. strncmp() and strlen() are both commonly used for working with strings in C, and are frequently called together, but it doesn't necessarily indicate a bug when they are not called together. This false positive could have been resolved by using a higher confidence threshold.

  //line from ap_core_translate()

```
if (!r->uri || ((r->uri[0] != '/') && strcmp(r->uri, "*"))) {...
```

Part 1C.

Code Explanation:

//our code is included in pi/partC

First we check if the user has added the argument “expand” when running Pi.java. From there, our code fills the hashMaps called graphMap, and usesMap by reading in the call graph just as we do in partA. In part A we pass these hashMaps to permute.java to find the bugs, but here, if the user included the “expand” argument, we run the method fillExpandedMap() to create a hashMap with contains the expanded version of the call graph, and pass that to permute.java. Permute.java functions the same way it does in part A.

Experiment:

We will compare the output quantity of bugs from test3 when expanded and not expanded. We will use four different combinations of confidence and support thresholds.

	Not Expanded Qnty	Expanded Qnty
Test3 3 65	253	34942
Test3 3 80	58	25242
Test3 10 65	89	8900
Test3 10 80	25	7244

We can see in each case that the quantity of bugs was many times larger when the call graph was expanded. Many more of these bugs are likely false positives, but it is also the case that it detected bugs that were not otherwise detected. Expanding the call graph might be a better option when run with high confidence thresholds to reduce false positives, as well as adding other functionality to detect and remove false positives.

Part 1D.

We used a two part strategy to improve our static analyzer that reduces the number of resulting false positives. First, we implemented a feature that allows users to specify several functions they know are not involved in bugs to be ignored. Second, we implemented a feature that allows users to specify a distance parameter so that only functions within that distance of each other in a scope will be considered.

//our code is included in pi/partD

Feature 1: User specifying functions to ignore

The motivation for this feature is that developers might be certain that several functions are not involved in bugs, but are frequently called and will lead to many false positives. Now they will be able to specify what those functions are, and receive more accurate bug reports. For example, if developers have included many print statements in a code base for debugging, they would be able to tell the static analyzer to ignore bugs involving fprintf(). This feature might be more useful for small to medium sized pieces of code.

We implemented this feature by first allowing the user to add “ignore” as an argument followed by a list of function names. We read these function names if they exist, and add them to an array list of strings called functionsToIgnore in Pi.java. We pass this list as a parameter to Permute.java. Permute.java then runs as it would without this feature until it reaches the function identifyBugs(). In identifyBugs() we check that any given bug does not contain any of the functions included in the functionsToIgnore list. By default, the list is empty and our analyzer works as though this feature were not implemented.

Feature 2: User specifying distance threshold

The motivation for this feature is that if two functions are far away from each other in a scope, they might be less likely to be associated. Conversely if they are called right next to each other, they might be more likely to be connected. This feature would allow users to solely search for bugs involving functions that are called right next to each other.

We implemented this feature by first allowing the user to enter a distance parameter t_distance in Pi.java. The default value is set to Integer.maxValue(), so if the user does not enter a value, the analyzer will functionally work as though the feature had not been implemented. We also check to make sure that t_distance is a positive number, and print a usage message to the user if it isn't. If the user does enter a value, we will pass this as a parameter to Permute.java. No other changes were made within Pi.java.

The function Permute() in Permute.java finds all possible permutations of functions that we later use to check for bugs. To implement this feature we added an if statement to remove pairs of functions that are not within t_distance of each other. This feature could be useful for analyzing code bases of any size. Developers would be able to generate baseline reports without this feature, as well as more constrained reports with t_distance set to a small numbers.

Part II

Coverity Analysis, Apache Tomcat:

10689: Bug
Line 280 of DeltaSession.java performs an unguarded read, differing from 13/17 other such reads. Because of the ratio, it is most likely a bug in the code, and the method should be modified to guard the read by accessing deltaRequest with the holding lock DeltaSession.diffLock.lock.

10654: Bug
There is a dereference after null check error found in line 684 of JDBCStore.java, where db connection is closed in the catch of a try-catch block. The reason this error arose is because the code checks to see if preparedRemoveSql is null in the try (line 669), but it is referenced in the close() method. The log indicates the try block never gave preparedRemoveSql a value as intended in line 673, so close() will cause a NullPointerException. We can potentially fix this by editing close() to check if preparedRemoveSql = null, and act accordingly if this is the case.

10625: False Positive
In line 174 of FarmWarDeployer.java, the code checks if engine != null and performs an operation when true. This caused an error because engine is dereferenced earlier in the method, indicating it is never null. However, it is good practice to have the if-condition regardless so as to avoid any NullPointerExceptions, and not run the operation inside the if-statement unconditionally.

10514: Bug.
In line 76 of ChannelCoordinator.java, method head "`public void sendMessage(Member[] destination, ChannelMessage msg, InterceptorPayload payload) throws ChannelException`" does not have a reference to the superclass. We can fix this by referencing the superclass above the method head, as is seen elsewhere in the program. Once fixed, it will look like:

```
org.apache.catalina.tribes.group.ChannelInterceptorBase.sendMessage(org.apache.catalina.tribes
.Member[], org.apache.catalina.tribes.ChannelMessage,
org.apache.catalina.tribes.group.InterceptorPayload)
public void sendMessage(Member[] destination, ChannelMessage msg, InterceptorPayload payload)
throws ChannelException
```

10507: Bug.
In line 330 of MessageBytes.java, we see the if statement "`if( strValue==null && s!=null) return false;`" followed by "`return strValue.equalsIgnoreCase( s );`" in line 331. The issue here is that if strValue == null but the first if condition does not pass because s == null as well, the return statement in line 331 will throw a NullPointerException because we are running equalsIgnorease on a null variable. To fix this issue, we can alter the if statement:
`if( strValue!=null && s!=null) return strValue.equalsIgnoreCase( s );`"

And alter the second return statement to return false, because we will reach that point if either strValue, s, or both are null.

10453: Intentional.

In lines 118-120 of ChannelUn.java, we see:

```
if( apr==null || ! apr.isLoaded() ) {
119             log.debug("Apr is not available, disabling unix channel ");
120             apr=null;
```

The flagged error is in line 120, where apr is set to null even though the if-statement checks to see if apr == null. This is most likely done because the if statement is or-conditional, meaning there is a chance that apr != null but !apr.isLoaded(), so the developer sets apr = null inside the statement anyway to ensure it is null before the method returns. We can fix this bad practice by rewriting the code as follows:

```
if( apr==null) log.debug("Apr is not available, disabling unix channel ");
Else if (! apr.isLoaded() ) {
           log.debug("Apr is not available, disabling unix channel ");
           apr=null;
}
```

Or, alternatively:

```
if( apr==null || ! apr.isLoaded() ) {
119             log.debug("Apr is not available, disabling unix channel ");
120             if (apr != null) apr=null;
```

10435: Bug

This is a resource leak, as can be seen in lines 233-235 of JDTCompiler.java:

```
233 InputStream is =
234                         classLoader.getResourceAsStream(resourceName);
235                     return is == null;
```

The return statement is the last in the boolean method, but the method never closes the stream before returning. This is most likely an error made by the developer, as it can be easy to forget to close streams after use. A solution would be to use a try-with-resources block, demonstrated below:

```
233 try (InputStream is = classLoader.getResourceAsStream(resourceName)) {
234         return is == null;
235     } return false;
```

10396: Bug

The program indicates a dereference after null check in FarmWarDeployer.java, implying the variable in question may be null when a method is called on it. For clarity, the variable in question is engine, of type Engine. In line 151, we check to see if econtainer (which eventually is used to construct engine) is checked to ensure it is both not null and an instance of an Engine object. If it is null at this point, an error will be logged and the method will return. The flagged issue is in line 159, where getName() is called on engine, which the program believes may be null. However, this occurs inside of a try-catch block, so that in itself is not an issue; if engine happens to be null at that point, the block will catch and throw the exception. The issue is actually in the if statement at line 151, which as indicated by the error message in line 152- it is written incorrectly. It should instead be:

```
if(econtainer == null || !(econtainer instanceof Engine))
```
Which would give the actual desired effect; logging an error and returning is econtainer is null, OR if it is NOT of type Engine.

10395: Bug

This bug in StandardWrapper.java was flagged because it differs from 14 other similar statements in the code which access “instance” with the holding lock “Standard.this.” We can fix this in line 770 by changing the method head to be synchronous, as is seen in the other examples where the locking is done correctly.

10381: Intentional

In the method scanForTlds() (line 190-207) of VirtualDirContext.java, we take a file and check it for any Tlds. The developer did not include a check to ensure the input file is not null. This is most likely something the developer felt was not needed, but is good practice to have just in case since the method will return a NullPointer of IndexOutOfBounds exception if File dir = null. We can fix it by adding a check, ie “if (dir == null) return” in the first line of the method (191).

10351: Intentional

In lines 896 to 984 of SimpleTcpCluster.java, there is an if condition that checks if message is not null, and when true changes the flag “accepted” to true after performing some operations. Then, in 905-915, a debug log is printed out if accepted is false and the debug log is enabled. This was written to debug any issues where the listener does not accept the message. However, the debug log requires message to be not null, as it includes method calls to message (line 912-915). Entering the debug log when message = null is an edge case that was most likely overlooked by the developer; it can be fixed by changing the code to return if message is null at the top of the method (line 886).

10291: False Positive

In lines 420-421 of SSIServletExternalResponder.java we see:
```
String noContext = getPathWithoutContext(normContext.getContextPath(), normalized);
```
This was flagged because normContext.getContextPath() could return null. However, this will not cause any runtime issues because the code then checks to see if noContext = null, so if the context path also returns null this case is covered.

10231: Bug

Variable ssc is defined in line 390 of ChannelNioSocket.java by opening a new Server Socket, but in lines 411-413, if no free port is found the method returns without closing ssc, causing a resource leak. We can easily fix this by closing ssc before returning in line 413.

10176: Bug

In lines 613-616 of SimpleTcpReplicationManager.java we see:
```
if ( log.isDebugEnabled() ) {
                        log.debug("Received replicated session=" + session +
                           " isValid=" + session.isValid());
                    }
```

This will cause a NullPointerException if session is null. We can fix it by adding to the if condition:

```
if ( log.isDebugEnabled() && session != null) {
```

And possibly adding a case, if session = null, as needed depending on implementation.

10167: False Positive

Line 234 of DeltaRequest.java calls reset(), which sets all variables to null However, the developer does a second check in lines 238-241:

```
238 if (actions == null)
239             actions = new LinkedList();
240         else
241             actions.clear();
```

Although this if-else may be redundant, since it is most likely that reset() sets actions to null, it is still important to check anyway. Additionally, actions is set to a new LinkedList() here, so the if-else cannot be omitted. This code is correct and necessary for accomplishing the goals of the developer.

10148: Bug

There is a resource leak in line 455 of JkMain.java:

```
 props.load( new FileInputStream(propsF) );
```

Because props.load() does not close or save propsF. Instead of returning directly after this line, we can instead close propsF or save it after the load (whichever is most appropriate for the goal of the method) and then return.

10102: Intentional

In line 121 of SSIProcessor.java, a new String variable strCmd is initialized, and then several operations are performed with the variable. However, there is no check to ensure the variable is not null to avoid a NullPointerException. This is most likely an edge case the developer overlooked, and this can be fixed by checking if strCmd = null after it is initialized, and terminating the method or performing actions as appropriate.

10026: False Positive

Line 294 of JspDocumentParser.java checks if a variable attrs = null, and performs several operations if this is the case. The line was flagged because attrs is dereferenced earlier in the method, and is also an input to the method. However, the dereference could still work as intended if attr is null (NullPointerException is not guaranteed here), so it is good practice to check if attr = null anyway. Additionally, the if-block contains a lot of operations, so it may be essential to the program.

Coverity Analysis, pi/Part A:

10002: Dubious method used, Pi.java main(), low impact, internalization

> This error is called on line 37 of Pi.java, where we initialize a new Scanner to read the callgraph. The reason for this is that Coverity found a reliance of the default encoding of Scanner, which typically reads only the platform's default encoding. In order to better generalize our code and not limit the input type of the callgraph, we can allow for a more universal encoding, such as UTF8. To do this, we decided to use a BufferedReader instead of a Scanner, which can be made more flexible by adding the input StandardCharsets.UTF_8.

11091: Inefficient map iterator, Pi.java printGraphMap(), low impact, performance

> This error is called in line 135 of Pi.Java, where we use keySet() to loop through a hashmap in a for-each loop and then access the value of each key with get(). The reason for this is that Coverity found an inefficient use of the keySet() iterator instead of entrySet() iterator. This could potentially cause runtime issues as we are accessing the map twice in each iteration of the for-each loop, once for the keySet() and once for getting the value of each key. Instead, if we use entrySet(), we will be able to access the map only once and then get the key and value of each entry as needed, which will improve overall efficiency. This is a simple fix; we simply need to change the for-each loop to loop through the map's entrySet() and then get the key and value from each entry inside the loop, performing the same actions as necessary.

We would like to note here that Coverity found a total of 7 bugs in our codebase, six of which were keySet()/entrySet() errors like 11091, and one of which was error 10002 discussed above.