SE 465 - Group 47 Project

Mart van Buren | 20552849 | mvanbure@uwaterloo.ca
Peter Lakner | 20703212 | pmlakner@uwaterloo.ca
Andrew McBurney | 20578351 | armcburn@uwaterloo.ca

Part I - Building an Automated Bug Detection Tool

b) Finding and Explaining False Positives

False positives occur when the static analysis tools present an error when it does not actually exist. In general they can happen for two fundamental reasons. First, bugs can be reported based on inferred beliefs from observing behaviour. In this case there will be situations in which although a behaviour is frequently observed, a deviance from this behaviour is not necessarily a bug. For example the automated detection tool built in this project will suggest bugs for methods that just coincidentally are called together. Second, static analysis have to identify bad practices or where a bug could be introduced, and it is hard to determine whether these situations are intentional by the developer or actually a fault.

Below are two examples of false positives generated by our bug detection tool:

1. pair: (strlen, strncmp), support: 16, confidence: 66.67% - 8 bugs

This pair is a case where the pair of function calls just coincidentally occurs together frequently. This is obvious if we simply consider the functionality of each. It is easy to imagine cases when we want to compare two strings for equality, and not need a new length and vice-versa. For instance, in the source code the developer defines a global variable for a length which is used in many strcmp calls.

```
99      #define MS_END "END"
100     #define MS_END_LEN (sizeof(MS_END)-1)
...
1699    if (strncmp(MS_END, conn->buffer, MS_END_LEN) == 0)
```

2. pair: (qsort, strcmp), support: 4, confidence: 66.67% - 2 bugs

This bug is also a false positive. qsort is just a sorting function and can be applied to any array. A possibility where these may necessarily be paired is if the compare function supplied to qsort needed to compare strings. However, in the below example from the source code, the compare function sorts by number of components.

```
1699    qsort(sortbin, nelts, sizeof(*sortbin), reorder_sorter);
```

c) Inter-Procedural Analysis

Description

For our algorithm, we decided to add a fourth argument to ./pipair: k. By default k=0. Specifying a k > 0 will "expand" all functions k times. Each time a function is expanded, this means we add all calls that any subroutines make as if they were calls made directly by the parent function. In other words, for each node in the call graph, we add grandchildren of that node as regular children. One design decision we made was to not expand main(void), since this function doesn't intuitively seem like a "routine" but more of an entrypoint function.

```
$ ./pipair <BC> <T_SUPPORT> <T_CONFIDENCE> <NUM_LEVELS_TO_EXPAND>
```

Example: test2

Before	After 1 Function Expansion

`int main() {`
`  scope1();`
`  scope2();`
`  scope3();`
`  scope4();`
`  scope5();`
`  scope6();`
`}`
 	
`void scope1() {`
`  A();`
`  B();`
`  C();`
`  D();`
`}`
`void scope2() {`
`  A();`
`  C();`
`  D();`
`}`
`void scope3() {`
`  A();`
`  B();`
`}`
`void scope4() {`
`  B();`
`  D();`
`  scope1();`
`}`
`void scope5() {`
`  B();`
`  D();`
`  A();`
`}`
`void scope6() {`
`  B();`
	`int main() {`
	`  scope1();`
	`  scope2();`
	`  scope3();`
	`  scope4();`
	`  scope5();`
	`  scope6();`
	`}`
	
	`void scope1() {`
	`  A();`
	`  B();`
	`  C();`
	`  D();`
	`}`
	`void scope2() {`
	`  A();`
	`  C();`
	`  D();`
	`}`
	`void scope3() {`
	`  A();`
	`  B();`
	`}`
	`void scope4() {`
	`  B();`
	`  D();`
	`  **A();**`
	`  **C();**`
	`}`
	`void scope5() {`
	`  B();`
	`  D();`
	`  A();`
	`}`
	`void scope6() {`

```
  D();
}
```

```
  B();
  D();
}
```

After (bold is new)

```
bug: A in scope2, pair: (A, B), support: 4, confidence: 80.00%
bug: A in scope3, pair: (A, D), support: 4, confidence: 80.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: B in scope6, pair: (A, B), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope6, pair: (A, D), support: 4, confidence: 80.00%
```

We see that two new lines appear. As an example, before the expansion, B() was called without A(), so when it was called in scope6 it was not considered a bug. After the expansion, B() was only called without A() once, so it deserves to be recognized as a bug the one time it is called alone.

Stress Testing with test3

Levels of Expansion	Avg. Runtime	Bugs @ 3, 65%	Bugs @ 10, 80%
0	1.27 s	410	35
1	1.68 s	6,788	912
2	11.37 s	70,404	20,395
100	62.13 s	119,370	40,232

Conclusions

We can see that the number of bugs detected increases dramatically as we expand more levels, until eventually we reach the maximum depth and expanding any further does not result in a different call graph. The reason for this increase is that each expansion dramatically increases the support for function pairs. Essentially, each function contains strictly more or equal function calls after each expansion, so we expect support between these new function calls to increase, which leads more potential bugs to surpass <T_SUPPORT>.

This may seem useless then, but this actually gives us more data to find bugs with higher confidence, we just need to adjust our thresholds. Running with 1 function expansion, and T_SUPPORT=30, T_CONFIDENCE=95% on test3, we only get 40 bugs. These bugs, however, we can be very confident are not false positives as these sections of code are executed together so many times, it's unexpected that it makes sense to run them in isolation. Without function expansion, we would potentially never notice these connections, and in particular it would be hard to filter for CONFIDENCE >= 95%.

Part II - Using a Static Bug Detection Tool

a) Resolving Bugs in Apache Commons

1. False Positive

TiffImageParser.java | 537

CID 10001 (#1 of 1): Dereference after null check (FORWARD_NULL)
8. unbox_null: Unboxing null object `iy0`
11. unbox_null: Unboxing null object `ix0`.

This warning message is a false positive, because you will never have a scenario where you unbox `null` object `iy0` or `ix0`.

This is since, when either `iy0` or `ix0` are `null`, the string builder `sb` will append a non empty string to itself.

```
520 if (ix0 == null) {
521   sb.append(" x0,");
522 }

523 if (iy0 == null) {
524   sb.append(" y0,");
525 }
```

Hence, since the length of the string builder is > 0, the function will throw a `ImageReadException`, thus never reaching the return statement on line `537`.

```
532 if (sb.length() > 0) {
533   sb.setLength(sb.length() - 1);
534   throw new ImageReadException("Incomplete subimage parameters, missing" + sb.toString());
535 }
536
537 return new Rectangle(ix0, iy0, iwidth, iheight);
```

2. Bug

TiffImageWriterLossless.java | 306

CID 10004 (#1 of 1): Dereference null return value (NULL_RETURNS)
1. null_return: Calling `findDirectory` which might return null.
2. return_null_fn: Returning the return value of `findDirectory`, which might be null.
3. null_method_call: Calling a method on null object `rootDirectory`

The function: `public TiffOutputDirectory findDirectory(final int directoryType)` returns `null` if the condition `directory.type == directoryType`, takes the `false` branch for all iterations in the loop.

TiffOutputSet.java

5. return_null: Explicitly returning null.

```
125         return null;
```

This leads to calling a method on `rootDirectory` which could be `null` - leading to a `NullPointerException` in line `306`.

TiffImageWriterLossless.java

```
306         writeImageFileHeader(headerBinaryStream, rootDirectory.getOffset());
```

A potential solution to this problem would be to add a conditional check to see if `rootDirectory` is equal to `null` before calling a method on it.

3. Bug

DataReaderStrips.java | `203`

CID 10009 (#1 of 1): Unintentional integer overflow (OVERFLOW_BEFORE_WIDEN)
overflow_before_widen: Potentially overflowing expression `bitsPerPixel * width` with type `int` (32 bits, signed) is evaluated using 32-bit arithmetic, and then used in a context that expects an expression of type `long` (64 bits, signed).

An integer overflow could occur. As per Coverity's recommendation: To avoid overflow, cast either `bitsPerPixel` or `width` to type `long` on line `203` in DataReaderStrips.java.

4. Bug

PsdImageParser.java | `310`

CID 10010 (#1 of 1): Resource leak on an exceptional path (RESOURCE_LEAK)
18. leaked_resource: Variable `is` going out of scope leaks the resource it refers to.

Change the lines:

```
309  if (notFound && is != null) {
310      is.close();
311  }
```

to:

```
309  if (is != null) {
```

```
310      is.close();
311  }
```

This prevents the resource leak in the case where `notFound` evaluates to false, and `is != null` evaluates to `true`.

5. Bug

T4AndT6Compression.java | 456

CID 10011 (#1 of 1): Resource leak (RESOURCE_LEAK)
1. new_resource: Created a new object of type `org.apache.commons.imaging.common.itu_t4.BitArrayOutputStream`, which implements `java.io.Closeable`.
2. var_assign: Assigning: `outputStream` = resource returned from `new org.apache.commons.imaging.common.itu_t4.BitArrayOutputStream()`.
3. Condition `y < height`, taking false branch.
4. noescape: Resource `outputStream` is not closed or saved in `toByteArray`
5. leaked_resource: Variable `outputStream` going out of scope leaks the resource it refers to.

The outputStream is not closed, and goes out of scope in line 456:

```
return outputStream.toByteArray();
```

Hence, save the byte array in a separate variable, and close the stream before returning the saved byte array. A tentative solution may be found below:

```
final byte[] out = outputStream.toByteArray();     // save the byte array stream
outputStream.close();                              // close the stream
return out;                                        // return the saved byte array
```

6. Bug

ColorConversions.java | 738

CID 10016 (#1 of 1): Result is not floating-point (UNINTENDED_INTEGER_DIVISION) integer_division: Dividing integer expressions `16` and `116`, and then converting the integer quotient to type `double`. Any remainder, or fractional part of the quotient, is ignored

As per Coverity's recommendation - to fix this bug: change or cast either operand to type **`double`**. If integer division is intended, consider indicating that by casting the result to type **`int`** .

Assuming that an integer division is not wanted, cast the numerator to type `double` to return a `double` from the division.

7. Intentional

BinaryConstant.java | 31

CID 10017 (#1 of 1): CN: Bad implementation of cloneable idiom
(FB.CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE)
1. defect: org.apache.commons.imaging.common.BinaryConstant defines clone() but doesn't implement Cloneable.

The line 31 throws a `CloneNotSupportedException` exception, which leads us to believe this is intended behaviour. In this case, the writer of the code might want to control how subclasses clone themselves, which is an intended behaviour. If this is the case, we believe the code should be left as is - instead of refactoring to suppress the warning.

```
31    public BinaryConstant clone() throws CloneNotSupportedException {
32        return (BinaryConstant) super.clone();
33    }
```

Reference: http://findbugs.sourceforge.net/bugDescriptions.html#CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE

8. Bug

Declaration.java | 191 & PluginManager.java | 191

There is a case in Declaration.java on line 191 where a `null` value could be passed into a method.

Declaration.java

CID 10018 (#1 of 1): Dereference after null check (FORWARD_NULL)
7. var_deref_model: Passing null pointer `pluginClass` to `findLoader`, which dereferences it.

```
191 ruleLoader = pm.findLoader( digester, id, pluginClass, properties );
```

Consequently, the function invoked in Declaration.java line 191 calls a method on that `null` function parameter on line 191 `in` PluginManager.java.

PluginManager.java

11. method_call: Calling method on `pluginClass`.

```
190 throw new PluginException("Unable to locate plugin rules for plugin with id [" + id + "]"
191   + ", and class [" + pluginClass.getName() + "]" + ":" + e.getMessage(), e.getCause() );
```

To avoid calling a method on `null`, add conditional logic in Declaration.java such that if `pluginClass` is `null`, do not call the method `pm.findLoader` (there is a `null` check for `ruleLoader` later on in the same function which will catch account for this new case).

9. Bug

FromAnnotationsRuleModule.java | 172

CID 10021 (#1 of 1): Dereference null return value (NULL_RETURNS)
4. dereference: Dereferencing a pointer that might be null `annotatedElements` when calling `visitElements`

Line 172 potentially passes a `null` value into the function `visitElements`, which acessess the length of the parameter on line 182 in file FromAnnotationsRuleModule.java.

```
182 for ( AnnotatedElement element : annotatedElements ) {
```

A `NullPointerException` will be thrown when invoking a range based for loop on a `null` value.

A potential fix for this problem would be to have a `null` check before calling `visitElements` on line 172 of FromAnnotationsRuleModule.java - ensuring the function is never invoked on `null` elements.

10. Intentional

UndirectedMutableGraph.java | 80

CID 10028 (#1 of 1): Arguments in wrong order (SWAPPED_ARGUMENTS)
swapped_arguments: The positions of arguments in the call to `internalRemoveEdge` do not match the ordering of the parameters:

- `getVertices(e).getTail()` is passed to `head`
- `getVertices(e).getHead()` is passed to `tail`

```
130    protected void internalRemoveEdge( V head, E e, V tail )
```

1. param_names: The parameters are called: `head`, `e` and `tail`.

The arguments passed into the function `internalRemoveEdge` are perceived to be out of order. Line 130, the function signature for `internalRemoveEdge` has the `head` as the first parameter, and the `tail` as the third.

```
80         internalRemoveEdge( getVertices( e ).getTail(), e, getVertices( e ).getHead() );
```

However, while this looks incorrect semantically - it is intended behaviour. Hence, we believe the code should remain as is - rather than refactoring to suppress the warning.

11. Intentional

AbstractExporter.java | 81

CID 10029 (#1 of 1): Dm: Dubious method used (FB.DM_DEFAULT_ENCODING)1. defect: Found reliance on default encoding: new java.io.OutputStreamWriter(OutputStream).

The construction of OutputStreamWriter without an argument specifying the encoding will cause a byte to string using the default platform encoding. The developer may want it to be translated using the platform's default representation, however if they had a specific encoding that should be produced, then they should specify if explicitly (in which case this may actually be a bug). To eliminate the warning construct OutPutStreamWriter with an additional argument to specify the desired encoding.

12. Bug

DisjointSetNode.java | 117

CID 10030 (#1 of 1): Eq: Problems with implementation of equals() (FB.EQ_COMPARETO_USE_OBJECT_EQUALS)1. defect: org.apache.commons.graph.collections.DisjointSetNode defines compareTo(DisjointSetNode) and uses Object.equals().

The developer defines compareTo, but inherits its equals method from java.lang.Object. This is a bug because compareTo should only return 0 when equals returns true, however, compareTo will return 0 when their “rank” values are equal. To fix this, define an equals method which returns rank.equals(o.getRank()) to match the compareTo function.

13. Intentional

SynchronizedMutableGraph.java | 25

CID 10032 (#1 of 1): Eq: Problems with implementation of equals() (FB.EQ_DOESNT_OVERRIDE_EQUALS)1. defect: org.apache.commons.graph.SynchronizedMutableGraph doesn't override SynchronizedGraph.equals(Object).

SynchronizedMutableGraph does not override the equals method defined in its superclass, but adds fields. This is intentional as there is effectively no difference in fields other than containing a MutableGraph instead of a Graph which return true on equals for equal references. To remove the warning the developer could override the equals method and just return the result of super.equals(this).

14. Intentional

ShortestDistances.java | 34

CID 10033 (#1 of 1): Se: Incorrect definition of Serializable class (FB.SE_COMPARATOR_SHOULD_BE_SERIALIZABLE)1. defect: org.apache.commons.graph.shortestpath.ShortestDistances implements Comparator but not Serializable.

The class implements Comparator but not Serializable. Implementing Serializable is good defensive programming, because the code does prevents collections constructed from but the

it was not required in this case.To bypass this warning, implement Serializable and define a serialVersionUID field.

15. Intentional

UncoloredOrderedVertices.java | 35

CID 10035 (#1 of 1): Se: Incorrect definition of Serializable class (FB.SE_COMPARATOR_SHOULD_BE_SERIALIZABLE)1. defect: org.apache.commons.graph.coloring.UncoloredOrderedVertices implements Comparator but not Serializable.

Same case as above (14.)

16. False Positive

BlockRealMatrix.java | 307

CID 10039 (#1 of 1): Missing call to superclass (CALL_SUPER)
1. missing_super_call: Missing call to `org.apache.commons.math.linear.AbstractRealMatrix.add(org.apache.commons.math.linear.RealMatrix)` (as is done elsewhere 3 out of 4 times).

This subclass does not make a call to the superclass in overridden methods, while other subclasses do. This is a false positive because the programmer does not need to make calls to super, and in fact does not do so for many other overridden methods in the class.

17. Bug

Array2DRowFieldMatrix.java | 109

CID 10048 (#1 of 1): Dereference before null check (REVERSE_INULL)
check_after_deref: Null-checking `d` suggests that it may be null, but it has already been dereferenced on all paths leading to the check.

```
109             if (d == null) { ...
```

The code checks that the parameter d is non-null however it has already been dereferenced on all paths leading to the check. This means this check is not necessary here, but could be checked further back in paths before it is dereferenced.

18. Bug

BrentOptimizer.java | 169

CID 10059 (#1 of 1): FE: Test for floating point equality (FB.FE_FLOATING_POINT_EQUALITY)
1. defect: Test for floating point equality.

```
169             if ((fu <= fw) || (w == x)) { ...
```

The programmer checks for equality of two floating point values. Floating point values should instead be compared within some range.

19. Bug

EigenDecompositionImpl.java | 1438

CID 10061 (#1 of 1): FE: Test for floating point equality (FB.FE_FLOATING_POINT_EQUALITY)
1. defect: Test for floating point equality.

```
1438            if (dMin == dN || dMin == dN1) {
```

Same as above (18.)

20. Bug

FirstMoment.java | 157
CID 10062 (#1 of 1): SA: Useless self-operation (FB.SA_FIELD_SELF_ASSIGNMENT)
1. defect: Self assignment of field FirstMoment.nDev.

The copy function in FirstMoment.java doesn't copy over all values from the `source` parameter. Assuming that all values need to be copied over, this would be a bug.

A proposed fix would be changing line 157 as follows:

```
157        dest.nDev = dest.nDev;
```

To:

```
157        dest.nDev = source.nDev;
```

b) Analyzing Your Own Code

Warning 1

Coverity throws a warning on line 135 for tainting the argument `refFunctionId` due to using `operator >>`.

28. tainted_data_argument: Calling function `operator >>` taints argument `refFunctionId`.

```
135                ss >> refFunctionId;
```

This causes Coverity to throw a later the warning on lines 144 and 145 for passing `refFunctionId` into `cg.nodes`, because no sanity checks are done on the string `refFunctionId`.

If input weren't coming from a safe place (in this case it is, `opt`), this could lead to a buffer overflow security exploit.

```
...
142                 // Add forward & backward edge
143                 cg.createNodeIfNotExists(refFunctionId);
```

CID 10165 (#3-7 of 10): Use of untrusted scalar value (TAINTED_SCALAR)
29. tainted_data: Passing tainted variable `refFunctionId` to a tainted sink.

```
144                 cg.nodes[curFunctionId]->callsFunctions.insert(cg.nodes[refFunctionId]);
```

CID 10165 (#9-5 of 10): Use of untrusted scalar value (TAINTED_SCALAR)
29. tainted_data: Passing tainted variable `refFunctionId` to a tainted sink.

```
145                 cg.nodes[refFunctionId]->calledByFunctions.insert(cg.nodes[curFunctionId]);
```

Warning 2

The operator overload for the output stream has a warning with regards to not restoring `ostream` format. The error message from Coverity may be found below:

CID 10164 (#1 of 1): Not restoring ostream format (STREAM_FORMAT_STATE)
1. format_changed: `fixed` changes the format state of `out` for category floatfield.
2. format_changed: `setprecision` changes the format state of `out` for category precision.
3. end_of_path: Changing format state of stream `out` for categories floatfield, precision without later restoring it.

```
32  ostream& operator<<(ostream &out, const Bug &bug) {
33      out << "bug: " << bug.func << " in " << bug.parent << ", ";
34      out << "pair: " << bug.pair << ", ";
35      out << "support: " << bug.support << ", ";
36      out << "confidence: " << fixed << setprecision(2) << bug.confidence << "%";
37      return out;
38  }
```

`setprecision` changes the precision of floating point numbers. It isn't reset within the scope of the `ostream` operator overload function before returning - which is the cause for the warning.

A potential fix, which resets the default precision value before returning is:

```
32  ostream& operator<<(ostream &out, const Bug &bug) {
33      streamsize default_ss = out.precision(); // store default precision value
34      out.precision(2); // set for floating point numbers to 2 decimal places
35      out << "bug: " << bug.func << " in " << bug.parent << ", ";
36      out << "pair: " << bug.pair << ", ";
37      out << "support: " << bug.support << ", ";
38      out << "confidence: " << fixed << bug.confidence << defaultfloat << "%";
39      out.precision(default_ss);   // set precision back to default value
40      return out;
```

```
41 }
```