

ECE 653 Project Report

Group 63

Student Name: Hai Jiang	Student No.: 20597659	Email: h57jiang@uwaterloo.ca
Student Name: Chang Liu	Student No.: 20632304	Email: c392liu@uwaterloo.ca
Student Name: Xiyue Zhang	Student No.: 20601564	Email: x562zhan@uwaterloo.ca

Part (1) Building an Automated Bug Detection Tool

(a) Inferring Likely Invariants for Bug Detection

In this part, we generate a hashmap (we call it “location” in our script) according to the call graph. The key of the hashmap is the callee function, and the value of the hashmap location is the caller function. In order to exclude the redundant and/or repetitive data, both the keys and values are of the SET type.

```
void scope1() {  
    A(); B(); C(); D();  
}  
void scope2() {  
    A(); C(); D();  
}
```

For example, if the code segment is shown as above, our hashmap can be listed as below:

Set(A)	Set(scope1, scope2)
Set(B)	Set(scope2)
Set(C)	Set(scope1, scope2)
Set(D)	Set(scope1, scope2)
Set(A,B)	Set(scope1)
Set(A,C)	Set(scope1, scope2)
Set(A,D)	Set(scope1, scope2)
Set(B,C)	Set(scope1)
Set(B,D)	Set(scope1)
Set(C,D)	Set(scope1, scope2)

It should be noted that the above example is only for illustrative purpose, it does not reflect the actual result of our project code.

Then, by looking up the hashmap, we can obtain the number of caller functions corresponding to a callee function or a pair of callee functions, thereby calculating the support and the confidence. As a result, we can generate the bug according to the rules given in the part1(a).

(b) Finding and Explaining False Positives

Reasons for the occurrence of false positives could be:

1. Although two methods have to be paired to realize a target function, one of them may be included in a specific method function that only gives a broader definition associated with this method. In such a case, this specific method function does not need to include the other method, since it is not for the target purpose.
2. Although two methods are paired to realize a target function in some circumstances, one of them can be paired with another method to achieve similar functions. In such a case, a false positive is generated if the new pair is regarded as a bug.

I use the code and hash map data structure in (a) to count and list the bug numbers for all the pairs. Then I figure out which 2 pairs can form 10 locations. As seen below, I list the 2 pairs I found and trace back to the source code in /opt/testing/apache_src to check if the bug is a false positive.

1. Pair (apr_thread_mutex_lock, apr_thread_mutex_unlock) as shown below, which has totally 6 bug locations.

bug: apr_thread_mutex_lock in apr_dbd_mutex_lock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 95.56%

bug: apr_thread_mutex_lock in apu_dso_mutex_lock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 95.56%

bug: apr_thread_mutex_unlock in apr_dbd_mutex_unlock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 91.49%

bug: apr_thread_mutex_unlock in apr_global_mutex_trylock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 91.49%

bug: apr_thread_mutex_unlock in apr_global_mutex_unlock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 91.49%

bug: apr_thread_mutex_unlock in apu_dso_mutex_unlock, pair: (apr_thread_mutex_lock, apr_thread_mutex_unlock), support: 43, confidence: 91.49%

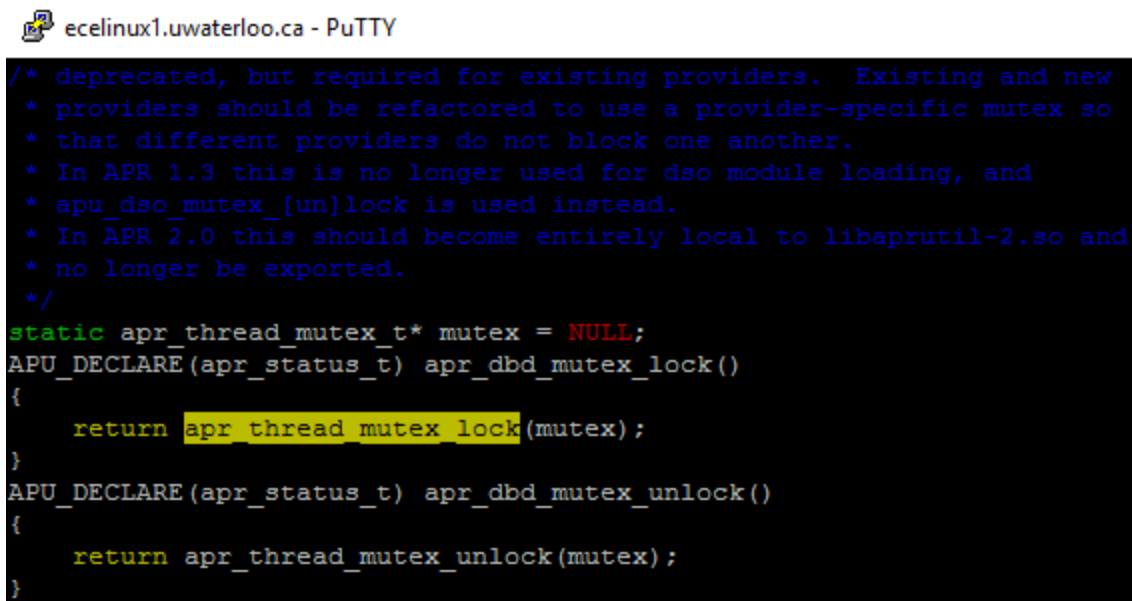
Then I use the linux instruction “grep -r” to find out which file contains the corresponding method (please refer to Fig. 2.1). As a result, I check the source code of several locations and I find out some false positives.

As shown in the Fig. 2.2, I read the source code of apr_dbd_mutex_lock(). Apparently, this method apr_dbd_mutex_lock() is used to define mutex_lock, by using the apr_thread_mutex_lock. There is no need for apr_dbd_mutex_lock() to include an unlock function. In practical use, apr_dbd_mutex_lock() is called by the other functions to perform a lock operation, and apr_dbd_mutex_unlock() is called to perform an unlock operation. In such case, apr_dbd_mutex_lock() itself does not need an unlock, and similarly

apr_dbd_mutex_unlock() does not need a lock. As such, the associated bugs shown in the report from part (a) is false positives.

```
[h57jiang@ecelinux1 httpd-2.2.21]$ grep -r "apr_dbd_mutex_lock" .
./src/lib/apr-util/dbd/apr_dbd.c:APU_DECLARE(apr_status_t) apr_dbd_mutex_lock()
./src/lib/apr-util/dbd/apr_dbd.c:APU_DECLARE(apr_status_t) apr_dbd_mutex_lock() {
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/apr_dbd_sqlite3.c:    apr_dbd_mutex_lock();
./src/lib/apr-util/dbd/NWGNUdbdsql3:    apr_dbd_mutex_lock \
./src/lib/apr-util/include/private/apr_dbd_internal.h:APU_DECLARE(apr_status_t) apr_dbd_mutex_lock(void);
```

Fig.2.1 use “grep” to search for corresponding pair



```
ecelinux1.uwaterloo.ca - PuTTY
/* deprecated, but required for existing providers. Existing and new
 * providers should be refactored to use a provider-specific mutex so
 * that different providers do not block one another.
 * In APR 1.3 this is no longer used for dso module loading, and
 * apu_dso_mutex_[un]lock is used instead.
 * In APR 2.0 this should become entirely local to libaprutil-2.so and
 * no longer be exported.
 */
static apr_thread_mutex_t* mutex = NULL;
APU_DECLARE(apr_status_t) apr_dbd_mutex_lock()
{
    return apr_thread_mutex_lock(mutex);
}
APU_DECLARE(apr_status_t) apr_dbd_mutex_unlock()
{
    return apr_thread_mutex_unlock(mutex);
}
```

Fig.2.2 the functions apr_dbd_mutex_lock() and apr_dbd_mutex_unlock()

2. Pair (apr_table_setn, apr_table_unset) as shown below, which has totally 4 bug locations.
 - bug: apr_table_unset in add_env_module_vars_unset, pair: (apr_table_setn, apr_table_unset), support: 8, confidence: 66.67%
 - bug: apr_table_unset in ap_read_request, pair: (apr_table_setn, apr_table_unset), support: 8, confidence: 66.67%
 - bug: apr_table_unset in basic_http_header, pair: (apr_table_setn, apr_table_unset), support: 8, confidence: 66.67%
 - bug: apr_table_unset in strip_headers_request_body, pair: (apr_table_setn, apr_table_unset), support: 8, confidence: 66.67%

```

static const char *add_env_module_vars_unset(cmd_parms *cmd, void *sconf_,
                                              const char *arg)
{
    env_dir_config_rec *sconf = sconf_;

    /* Always UnsetEnv FOO in the same context as {Set,Pass}Env FOO
     * only if this UnsetEnv follows the {Set,Pass}Env. The merge
     * will only apply unsetenv to the parent env (main server).
     */
    apr_table_set(sconf->unsetenv, arg, NULL);
    apr_table_unset(sconf->vars, arg);

    return NULL;
}

```

Fig.2.3

I use the same method as mentioned above to check the pair (apr_table_setn, apr_table_unset). As shown in Fig. 2.3, I read the source code of add_env_module_vars_unset. In this function, apr_table_unset() follows apr_table_set() instead of apr_table_setn(). As apr_table_set() has already performed a set function, I believe that it is reasonable to have apr_table_unset() without a apr_table_setn() here. Thus, the bug report for this function is a false positive.

(c) Inter-Procedural Analysis

We implement two different solutions for this problem. If you don't have enough time to review both, please review the solution 1, which works better to reduce false positive.

Solution 1 (please refer to 'pipair java.java')

1) Idea

The main idea of this approach is to directly examine the bug discovered by the method of part 1(a) before the bug is printed out. If the bug is turned out to be a false positive, then it is not printed, otherwise it is printed out.

For each bug (e.g., Bug1()) as discovered in part 1(a) and indicated as in a caller function (e.g., Caller1()), Bug1() needs to pair with Pair1() while Caller1() does not contain Pair1(). To seek for Pair1(), we expand one or more functions in Caller1() (we call them neighbour functions of Bug1(), e.g., neighbour1(), neighbour2(), etc.). If we can find Pair1() in at least one of the neighbour functions, then we label this bug as a false positive without printing this bug. Otherwise, the bug is regarded as a true bug, and thus is printed out.

An expansion level is set to control the maximum number of levels by which each neighbour function would expand into. We use DFS to traverse the neighbour functions. The expansion and search for one particular function terminate either when the pair function Pair1() is found or the predetermined expansion level of all the neighbour functions is reached. Please note that the terms "Bug1()", "Caller1()", "neighbour1()", "Pair1()" in this description are for illustrative purpose only, and do not reflect the actual implementations.

In our code implementation, we have created another HashMap (we call it 'callerCallee' in our script). CallerCallee stores information about all the caller functions, which is different from the HashMap 'location' as mentioned in Part1(a). For example, the key of CallerCallee is caller, and the value is all the callees. The value of CallerCallee are of set type.

More detailed steps are summarized as below:

Step 1: Traverse the call graphs to generate HashMap 'callerCallee' which lists caller as key and all the callees as value. The callees are included in a set.

Step 2: Call the HashMap 'location' to calculate support and confidence, and determine if a callee function is a bug in a caller function.

Step 3: If a bug is found, then implement DFS to expand the neighbour functions in order to search for the required pair function.

Step 4: If at step 3 the pair function is found in one of the neighbour functions, ignore the bug, otherwise print that bug.

2) Test:

We use test 3 in part 1(a) to test our code for solution 1 in part 1(c). The expansion level is set to 1, 2 and 3 in our test experiment. The test result is summarized in the following table.

Expansion level	<Support, Confidence> = <3, 0.65>		<Support, Confidence> = <10, 0.8>	
	Part 1(C)	Part 1(A)	Part 1(C)	Part 1(A)
Level=1	174	205	30	34
Level=2	166		30	
Level=3	163		30	

As is shown in the table, the number of bugs is reduced compared to the implementation in part 1(A), which means that false positives are identified and removed from the printed bug lines. More specifically, under the standard <Support, Confidence> = <3, 0.65>, part 1(a) found 205 bugs. In comparison, the bugs found by our implementation of part 1(c) are reduced to 174, 166 and 163 when the expansion level is set to 1, 2 and 3 respectively. Likewise, under the standard <Support, Confidence> = <10, 0.8>, part 1(a) found 34 bugs. Among them, 4 false positives are found by our implementation in part1(c). Thus, only 30 bug lines remain.

```
bug: apr_array_make in ap_init_virtual_host, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in ap_make_method_list, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in create_core_dir_config, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in create_core_server_config, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in apr_xml_parser_create, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in prep_walk_cache, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
```

To give a specific example in the above figure, the bug line highlighted in red can be found in the report of part 1(a). However, in the solution 1 in part 1(c), this bug line is removed because it is identified as a false positive.

3) More Discussion:

One of the advantages of this approach is that we don't add more bugs by expanding the functions. Because we aim at reducing the false positives, we only check if the identified bug is the real bug. Compared to those expanding all the sub-functions, our approach reduces the complexity of programming and is time efficient.

Moreover, we only need to check if an expansion contains the required paired function, no need to have storage space to store the expansion result. Therefore, the storage space is saved.

Solution 2 (please refer to ‘pipair java2.java’):

1) Idea

In this solution, we create HashMaps ‘callerCallee’, which works the same as solution 1. We further build up another Hashmap ‘CallerCalleeExtend’. We expand all the possible functions and store the result after expansion in ‘CallerCalleeExtend’. Then we generate the HashMap ‘location’ according to the Hashmap ‘CallerCalleeExtend’. Then the method of searching for bugs based on ‘location’, which is the same as in Part 1(a), is employed to search for the bug again.

More detailed steps are listed as below:

Step1: Traverse the call graphs to generate HashMap ‘callerCallee’.

Step2: Expand all the functions according to HashMap ‘callerCallee’ to Create HashMap ‘callerCalleeExtend’. The set of callee functions in HashMap ‘callerCalleeExtend’ is set of expanded callee functions from HashMap ‘callerCallee’, which is generated by recursion function called getCalleeExtend().

We use recursion in getCalleeExtend() to expand a caller function into a predetermined level. For example, if the recursion level is set to 2, we will expand a caller function into first level functions and then expand the first level functions one by one into multiple two level functions. The return value is a set that stores expanded callee functions, which are further stored in the HashMap ‘callerCalleeExtend’.

Step3: Update HashMap ‘location’ according to HashMap ‘callerCalleeExtend’. Then search and print bugs using the same solution of part 1(a).

2) Test:

According to the table below, this solution actually add a lot new bugs to the report. The main reason is that after the expansion, the confidences of a lot more pairs reach the threshold.

Expansion level	<Support, Confidence> = <3, 0.65>		<Support, Confidence> = <10, 0.8>	
	Part 1(C)	Part 1(A)	Part 1(C)	Part 1(A)
Level=1	1630	205	155	34
Level=2	5792		724	
Level=3	13556		2093	

However, we can see that this approach is effective to reduce false positive. Refer to the figure below.


```

bug: apr_array_make in ap_init_virtual_host, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in ap_make_method_list, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in create_core_dir_config, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in create_core_server_config, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in apr_xml_parser_create, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%
bug: apr_array_make in prep_walk_cache, pair: (apr_array_make, apr_array_push), support: 40, confidence: 86.96%

```

Similar to solution 1, we don't find the bug line as highlighted in the above figure after implementing the solution 2.

3) More Discussion:

Compared to solution 1, this approach needs more memory space for HashMap 'callerCalleeExtend' to store the function relation information after expansion. After implementation, more bugs jump out, which buries the performance of reducing the false positives, although it does.

Another question is whether to expand main function. The debate is that because main function includes all functions, any two of callee functions will turn to be paired if the expansion level is high. However the pipair in main shows some functions should appear together, for example, if there are lock and unlock functions in main, the support of this pipair will add one, and this support will influence the MAY belief that the lock and unlock functions should appear together. So we prefer to expand main function.

The final question is whether to delete caller function after expanding that function. If we do not delete caller function, there may be more pipairs including caller function, but useless to analyze false positive. And the number of bug increases rapidly with raising expansion level, we choose to delete caller function to reduce meaningless pipair.

(d)Improving the Solutions

1) Literature Review

We reviewed two papers as listed at the end of part 1(d).

Paper [1] discusses violations of temporal rules, where sequences of actions need to be considered. Some examples of temporal rules are "no < a>after < b>" (freed memory cannot be used), "< b> must follow < a>" (unlock must follow lock), and contextual rules such as "in context < b>, do after < a>" (on error paths, reverse side-effects). While there are a small number of such templates, there are many different specific operations that can fit in them.

Paper [2] discusses the control flow graph path, which required functions in a kind of orders. The null pointer analysis is a forward intra-procedural dataflow analysis performed on a control-flow graph representation of a Java method. The dataflow values are Java stack frames containing "slots" representing method parameters, local variables, and stack operands. Each slot contains a single symbolic value indicating whether the value contained in the slot is definitely null, definitely not null, or possibly null. So Infeasible control paths are a common source of inaccuracy in dataflow analysis, and avoiding them is an important challenge in the design of an analysis to find null pointer bugs.

2) Idea:

The main idea of this approach is to consider the order for pipair when performing the method of part 1(A) to discover bug. In other words, if (A, B) is required to be together and A needs to be executed before B, then (B, A) in a caller function is identified as a bug.

In our code implementation, we have changed structure of HashMap 'location' from Map<Set,Set> into Map<List,Set>. In Map<Set,Set>, the key of Set is disordered, and conversely, the key of List is ordered in Map<List,Set>. For example, the key of this HashMap will store a list of [A, B] rather than a set of (A, B). In this way, we can distinguish the list of [A, B] from the list of [B, A].

More detailed steps are summarized as below:

Step 1: Traverse the call graphs to generate HashMap 'location'.

Step 2: Call the HashMap 'location' to calculate support and confidence, and determine if a callee function is a bug in a caller function.

Step 3: If a bug is found, print that bug.

3) Test:

First, run the command as below, and change command argument according to Part1(A).

```
[x562zhan@eceLinux1 proj-skeleton]$ make
javac pipair_java.java
Picked up JAVA_TOOL_OPTIONS: -Xmx64m
[x562zhan@eceLinux1 proj-skeleton]$ cd test3
[x562zhan@eceLinux1 test3]$ ../pipair test3.bc 3 65
```

Second, replace binary file name for different test case, and see the result from Table 1.

	Part1(D)	Part1(A)
test2_3_65	2	4
test3_3_65	218	205
test3_10_80	34	34

Table 1 the number of bug line for test case

4) Analysis:

As shown in Table 1, we can see that Part 1(D) has 2 bugs for test2_3_65.out, while Part1(A) has 4 bugs ,which means we find false positives and reduce them from original bugs in Part1(A), so the number of bugs in Part1(D) is smaller.

Besides, Part1(D) has 218 bugs for test3_3_65.out in total. These bugs include the original bugs identified by Part1(A) (the false positive has already been removed herein), and also include new bugs, which will be discussed in the section “**ii>Find more bug**”. The number of bugs for test3_10_10.out in Part1(D) are the same with Part1(A), except that the pipairs which take place of the bug are in proper order in Part1(D), e.g, (A,B) is not the same as (B, A) which will be discussed in the section “**ii>Find more bug**”.

i>Reduce false positive:

The goal of our approach is to extract beliefs from code and to check for violated beliefs [1].

We should distinguish ordered pipair from disordered pipair. Given the pipair has no order in Part1(A), there will be more pipair in this part since (A,B) and (B,A) are computed as two types, which means the support of pipair will be larger. It decreases the probability to satisfy threshold support and confidence for generating May Belief. Moreover, In Part1(D), the support of ordered pipair is smaller and there will be less May Belief to check bug. As a result, May Belief in Part1(A) may not be a May Belief in Part(D), and false positive will be reduced in Part1(D). For example, the support of (A,B) is 3 in Part1(A) test2_3_65.out(see Fig.4.1). But the support of [A,B] is 2 and support of [B,A] is 1 in Part1(D) test2_3_65.out, so the support of [A,B] or [B,A] in Part1(D) does not meet threshold support, so Part1(D) only print 2 bug (see Fig.4.2). In this way the false positive is reduced.

```
bug: A in scope2, pair: (A, B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A, D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%
```

Fig.4.1Part1(A) test2_3_65.out

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

Fig.4.2 Part1(D) test2_3_65.out

ii>Find more bug:

There will be three kinds of new bug generated in Part1(D) due to ordered pipair.

First, this bug is that when the single function is required to appear more than once to achieve a sub-function, those appearing only once is regarded as a bug. For example, if A() has to be performed twice, e.g., A(), A(), that A() is called only once in a caller function is a bug. In this part, we will compute the number of its occurrences. In the test3_3_65.out the support of (ms_bad_conn, ms_bad_conn) is 6 (see Fig.4.3), which means that 'ms_bad_conn' should appear more than once, instead of computing its occurrences as one in Part1(A) (see Fig.4.4). Thus, a new bug ms_bad_conn in mc_version_ping is found as shown in Fig. 4.3.

```
bug: ms_bad_conn in mc_version_ping, pair: (ms_bad_conn, ms_bad_conn), support: 6, confidence: 75.00%
```

Fig.4.3The bug output for 'ms_bad_conn'

```
Call graph node for function: 'mc_version_ping'<<0x5a3d9c0>>
CS<0x5d2f0f8> calls function 'ms_find_conn'
CS<0x5d2ffd8> calls function 'apr_socket_sendv'
CS<0x5d303d8> calls function 'ms_bad_conn'
CS<0x5d30680> calls function 'get_server_line'
CS<0x5d308c8> calls function 'ms_release_conn'
```

Fig.4.4The callgraph for 'ms_bad_conn' should appear more than once

Second, this bug is that the function is required to be called in proper order, and thus that appearing in reverse order are regarded as bug. For example, in test3_3_65.out the support of (get_server_line, ms_bad_conn) is 6 (see in Fig.4.5), which means that (get_server_line, ms_bad_conn) should be the right order. So Even though it has pipair (ms_bad_conn, get_server_line) (see in Fig.4.6), it breaks the rule to be a bug since it is not in the right order.

```
bug: ms bad conn in mc version ping, pair: (get server line, ms bad conn), support: 6, confidence: 75.00%
```

Fig.4.5The bug output for 'ms_bad_conn'

```
Call graph node for function: 'mc_version_ping'<<0x5a3d9c0>>
CS<0x5d2f0f8> calls function 'ms_find_conn'
CS<0x5d2ffd8> calls function 'apr_socket_sendv'
CS<0x5d303d8> calls function 'ms_bad_conn'
CS<0x5d30680> calls function 'get_server_line'
CS<0x5d308c8> calls function 'ms_release_conn'
```

Fig.4.6 The call graph for wrong order

Reference

- [1] Engler, Dawson, et al. *Bugs as deviant behavior: A general approach to inferring errors in systems code*. Vol. 35. No. 5. ACM, 2001.
- [2] Hovemeyer, David, Jaime Spacco, and William Pugh. "Evaluating and tuning a static analysis to find null pointer bugs." *ACM SIGSOFT Software Engineering Notes*. Vol. 31. No. 1. ACM, 2005.

Part (2) Using a Static Bug Detection Tool

(a) Resolving Bugs in Apache Commons

1. CID 10065 (Missing break in switch (MISSING_BREAK))

Classification: False Positive

Explanation: In “case 3”(BooleanUtils.java from line 628 to line 640), if “ ch == ‘y’/’Y’ ”, it will return the corresponding Boolean value after checking whether it equals to “yes” or “YES”. Due to the lack of “break”, it will execution “case4” if it does not satisfy any case in “case 3”, and it will also does not any case of “case 4” since the unequal length. Finally, it will return false, which does not modify the output.

2. CID 10066(CN: Bad implementation of cloneable idiom(FB.CN-IDIOM))

Classification: False Positive

Explanation: n this class, it implements Cloneable interface, but have clone() method is not called in it. It will waste the space. However, it will not cause the problem.

3. CID 10067(Dm: Dubious method used (FB.DM_DEFAULT_ENCODING))

Classification: Intentional

Original code:(NestableDelegate.java line292)

```
PrintWriter pw = new PrintWriter(out, false);
```

Re-factory:

```
PrintWriter pw = new PrintWriter((new OutputStreamWriter(out, "UTF-8"), false);
```

Explanation: Different environments may have different default character encodings. you can control the character encoding. the characters should be written in such that they won't eventually end up as gibberish.

4. CID 10068(Dm: Dubious method used (FB.DM_NEXTINT_VIA_NEXTDOUBLE))

Classification: Intentional

Original code: 110 **return** (int)(Math.random() * n);

Re-factory: 110 **return** (int)(Random.nextInt(n));

Explanation: Math.random() requires about twice the processing to generate an integer and is subject to synchronization. Random.nextInt(n) uses Random.next() less than twice on average- it uses it once, and if the value obtained is above the highest multiple of n below MAX_INT it tries again, otherwise it returns the value modulo n (this prevents the values above the highest multiple of n below MAX_INT skewing the distribution), so returning a value which is uniformly distributed in the range 0 to n-1. Therefore Random.nextInt(n) is more efficient and less biased.

5. CID 10069

Eq: Problems with implementation of equals() (FB.EQ_COMPARING_CLASS_NAMES)

Classification: Bug

Faulty lines:(Enum.java line 552)

```
if (other.getClass().getName().equals(this.getClass().getName()) == false)
```

In this code, it wants to determine whether the object other is the same class as this object. However, the equal method just compare these two object if have the same name. Therefore, it can not achieve the goal of the code and is just a bug.

Bug fix: (To compare the class of an object to the intended class, the getClass() method and the comparison operator "==" should be used.)

if (other.getClass() == this.getClass())== false)

we just remove getName() method and use comparison operator "==".

6. CID 10070

Eq: Problems with implementation of equals() (FB.EQ_COMPARING_CLASS_NAMES)

Classification: Bug

Faulty lines:(Enum.java line 598)

if (other.getClass().getName().equals(this.getClass().getName()) == false)

In this code, it wants to determine whether the object other is the same class as this object. However, the equal method just compare these two object if have the same name. Therefore, it can not achieve the goal of the code and is just a bug.

Bug fix: (To compare the class of an object to the intended class, the getClass() method and the comparison operator "==" should be used.)

if (other.getClass() == this.getClass())== false)

we just remove getName() method and use comparison operator "==".

7. CID 10071(ES: Checking String equality using ==

or !=)(FB.ES_COMPARING_PARAMETER_STRING_WITH_EQ)

Classification: Intentional

Original code:(BooleanUnits.java line 614) 614 **if (str == "true")**

Re-factory : 614 **if (str.equals("true"))**

Explanation: According to the comments above the method and above this code line, the parameter of method toBoolean(String str) is interned strings, so using “==” to compare str with “true” would not cause problem. But to achieve scalability, if the parameter str is not a String constant or interned string, the result of this statement would not be functionally right. Using “equals” is a better way to promise the program function.

8. CID 10072(ES: Checking String equality using ==

or !=)(FB.ES_COMPARING_PARAMETER_STRING_WITH_EQ)

Classification:Intentional

Original code :(StringUtils.java line 4865) 4865 **if (str1 == str2)**

Re-factory : 4865 **if (str1.equals("str2"))**

Explanation: According to the comments above the method and above this code line, the parameter of method indexOfDifference(String str1, String str2) is interned strings, so using “==” to compare str1 with “str2” would not cause problem. But to achieve scalability, if the parameter str1 or str2 is not a String constant or interned string, the result of this statement would not be functionally right. Using “equals” is a better way to promise the program function.

9. **CID 10073**(ES: Checking String equality using == or !=)(FB.ES_COMPARING_PARAMETER_STRINGS_WITH_EQ)

Classification: False Positive

Original code: In DurationFormatUnits.java

```
1) 409      else if (value == S)
2) 405      else if (value == s)
3) 401      else if (value == m)
4) 397      else if (value == H)
5) 393      else if (value == d)
6) 389      else if (value == M)
7) 385      else if (value == y)
```

Explanation: 380 Object value = token.getValue(); “value” is a constant which is defined in line 380. And for else if statement, both the right hand and the left hand are constant strings, so they can be compared by “==”

10. **CID 10074**(ES: Checking String equality using == or !=)(FB.ES_COMPARING_PARAMETER_STRING_WITH_EQ)

Classification: False Positive

Explanation: previous.getValue() is constant in the source file, variable value just be assigned to a String value, but not instantiate a new object. So they can be compared using “==”.

11. **CID 10075**(IM: Questionable integer math)(FB.IM_AVERAGE_COMPUTATION_COULD_OVERFLOW)

Classification: Intentional

Original code : In Entities.java(line 649) `int mid = (low + high) >> 1;`

Re-factory: `int mid = (low + high) >>> 1;`

Explanation: If the result of (low + high) is zero or positive, it will not cause the problem. While the result is negative, “>>” will cause the wrong answer. The >> operator shifts a 1 bit into the most significant bit if it was a 1, and the >>> shifts in a 0 regardless. (low + high) >> 1 keeps the sign bit of the original, so a negative value for a gives a negative result. (low + high) >>> 1 works by introducing a zero sign bit, so the result cannot be negative for any (low + high).

12. **CID 10076**(NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL))

Classification: Intentional

Solution: left as-is

Explanation: In general, it is not reasonable to return null, which may lead to null-pointer exceptions. While in the above comments, it is said that:

61 * @return the negated Boolean, or <code>null</code> if <code>null</code> input
Therefore, This method allows to return null, So, it’s an intentional warning. According to the function of the method, it should be left as-is.

13. **CID 10077**(NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL))

Classification: Intentional

Solution: left as-is

Explanation:In general, it is not reasonable to return `null`, which may lead to null-pointer exceptions. While in the above comments, it is said

that:305 * *@return Boolean.TRUE, Boolean.FALSE, or `<code>null</code>`*

Therefore, This method allows to return `null`, So, it's an intentional warning. According to the function of the method, it should be left as-is.

14. CID 10078(NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL))

Classification:*Intentional* **Solution:**left as-is

Explanation:In general, it is not reasonable to return `null`, which may lead to null-pointer exceptions. While in the above comments, it is said that:

221 * *@return Boolean.TRUE if non-zero, Boolean.FALSE if zero,*

222 * *`<code>null</code>`* if *`<code>null</code>`* input

Therefore ,This method allows to return `null`, So, it's an intentional warning. According to the function of the method, it should be left as-is.

15. CID 10079(NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL))

Classification:*Intentional* **Solution:**left as-is

Explanation:In general, it is not reasonable to return `null`, which may lead to null-pointer exceptions. While in the above comments, it is said that:

336 * *@return Boolean.TRUE, Boolean.FALSE, or `<code>null</code>`*

Therefore, this method allows to return `null`, So, it's an intentional warning. According to the function of the method, it should be left as-is.

16. CID 10080(NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL))

Classification:*Intentional* **Solution:**left as-is

Explanation:In general, it is not reasonable to return `null`, which may lead to null-pointer exceptions. While in the above comments, it is said that:

502 * *`<p><code>'true'</code>, <code>'on'</code> or <code>'yes'</code>`*

503 * *(case insensitive) will return `<code>true</code>`*.

504 * *`<code>'false'</code>, <code>'off'</code> or <code>'no'</code>`*

505 * *(case insensitive) will return `<code>false</code>`*.

506 * *Otherwise, `<code>null</code>`* is returned.</p>

Therefore , this method allows to return `null`, So, it's an intentional warning. According to the function of the method, it should be left as-is.

17. CID 10081

NP: Null pointer dereference (FB.NP_BOOLEAN_RETURN_NULL)

Classification:*Intentional* **Solution:**left as-is

Explanation:In general, it is not reasonable to return `null`, which may lead to null-pointer exceptions. While in the above comments, it is said that:

557 * *@return the Boolean value of the string,*

558 * *`<code>null</code>`* if no match or *`<code>null</code>`* input

Therefore ,This method allows to return `null`, So, it's an intentional warning. According to the function of the method, it should be left as-is.

18. CID 10082(REC: RuntimeException capture (FB.REC_CATCH_EXCEPTION))

Classification: **Intentional**

Original code : In ExceptionUnits.java (line 97)97 **catch** (Exception e)

Re-factory: 97 **catch** (RuntimeException e)

Explanation: In Java, there are two types of exceptions: checked exceptions and un-checked exceptions. A checked exception must be handled explicitly by the code, whereas, an un-checked exception does not need to be explicitly handled.

Any exception that derives from "Exception" is a checked exception.

Any exception that derives from "RuntimeException" is an un-checked exceptions.

Generally, throwing a checked exception will not cause the problem. While the method could not be normally handle an un-checked exceptions . RuntimeExceptions do not need to be explicitly handled by the calling code.

19. CID 10083(Se: Incorrect definition of Serializable class (FB.SE_BAD_FIELD))

Classification: **Intentional**

Original code:In FastDateFormat.java (line 137) 137 **private** Rule[] mRules;

Re-factory:137 **private transient** Rule[] mRules;

Explanation: By default, all of object's variables get converted into a persistent state. In some cases, you may want to avoid persisting some variables because you don't have the need to persist those variables. So you can declare those variables as transient. If the variable is declared as transient, then it will not be persisted.

20. CID 10084(UrF: Unread field (FB.URF_UNREAD_FIELD))

Classification: **False Positive**

Explanation:85 **this**.key = key;the key has never been read,and it seems always same with hash and is redundant .But it will not cause the problem.

(b) Analyzing Your Own Code

1. After running Coverity on my own code for part1 (a), 1 error is detected as following:

```
FindBugs Checkers: 1 errors
FB.DM_DEFAULT_ENCODING 1
```

1) CID 10090

Dm: Dubious method used(FB.DM_DEFAULT_ENCODING)

Original Code: line 34 in pipair_java.java

```
34 Scanner scanner=new Scanner(System.in);
```

Fixed Code:

```
34 Scanner scanner=new Scanner(System.in,"UTF-8");
```

Explanation: Different environments may have different default character encodings. This would guarantee that, given an input file that uses encoding "UTF-8", it will be parsed in the same way regardless of what machine you are executing your program in.

2. run Coverity on fixed code, 0 error is detected.

```
FindBugs Checkers: 0 errors
```

3. the reasons for only finding one error:

1) Maybe there are some errors not documented in the Coverity, and these errors covered in our code can not be detected.

2) As the Coverity is a static analysis tool, some errors, such as memory leaks and time dependencies, can only be detected by the dynamic tool

Appendix :

the output from Coverity:

Error#1	
Meta Variable	
Checker	FB.DM_DEFAULT_ENCODING
File	/home/c392liu/sa_output/pipair_java.java
Function	pipair_java.main(java.lang.String[])
Ordered	true
Event	
Variable	
main	true
tag	defect
description	{CovLStr{v1}}{Found reliance on default encoding: {0}.}{new java.util.Scanner(InputStream)}
line	34