Evaluation of Facebook Infer on Juliet Test Suite

The goal of this work is to evaluate the quality of Facebook Infer on Juliet test suite test cases. The main focuses were the type of detected weaknesses, and the number of true positives, false positives, and replications. Juliet Test Suite consists of 112 weaknesses groups, the number of the test cases is different for each test cases group.

We used the latest version of Infer, which is v1.1.0.

Installation and analysis technical steps:

#Apache ant

sudo apt install ant

#Java 17

sudo apt install openjdk-17-jdk

#download and extract Juliet 1.3 for Java

wget https://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.3_for_Java.zip

unzip Juliet_Test_Suite_v1.3_for_Java.zip

#OCaml opam

sudo add-apt-repository ppa:avsm/ppa

sudo apt update

sudo apt install opam

#sqlite3 dependency for models

sudo apt install sqlite3

#facebook infer

git clone https://github.com/facebook/infer.git

cd infer

./build-infer.sh java

#an example, on my machine

cd ..

cd Java/src/testcases/CWE772_Missing_Release_of_Resource~/infer/infer/bin/infer -- ant

ant clean

~/Downloads/infer/infer/bin/infer -- ant

##for showing all the analyzing results on the screen

~/infer explore -- ant

##for enabling all the checkers for C/C++ Code:

~/infer --no-filtering -- ant

However, I used it for Java test cases of Juliet, but I did not see any changes.

refer to: https://github.com/facebook/infer/issues/1114

##So, I used the following script to activate all checkers (except 2*):

~/infer --config-checks-between-markers --bufferoverrun --biabduction --annotation-reachability --config-impact-analysis --cost --fragment-retains-view --immutable-cast --impurity --inefficient-keyset-iterator --litho-required-props --loop-hoisting --printf-args --pulse --purity --quandary --racerd --starvation --topl --config-impact-analysis -- ant

*Infer crashes when I used --eradicate, this is also reported in: https://github.com/facebook/infer/issues/1338.

Infer also shows the message: "Run the command again with `--keep-going` to try and ignore this error", when using --resource-leak-lab.

This is why I removed both previous checkers while analyzing.

The Analysis Results: find the JulietAnalysisByInfer.xlsx file attached with this report.

Juliet Test Cases Detection Details:

Abbreviations: TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative, Recall = TP Rate (TP/# test cases, which differs from Juliet weakness to another)

CWE134_Uncontrolled_Format_String: Infer checker *'Checkers Printf Args '* could detect this weakness with high FP detections for this test cases group, as it always detects the line of code where there is no explicit defined string formatting regardless of the source of the used data.

Example of *uncontrolled string formatting* flaw (line 9), with a bad source of reading data (an environment variable, line 5). This code snippet represents a case where Infer successfully detect the flaw in line 9 (TP):

```
1  public void bad() throws Throwable
2    {
3        String data;
4
5        data = System.getenv("ADD");
6
7        if (data != null)
8        {
9            System.out.printf(data);
10       }
11
12   }
```

Example of possible *uncontrolled string formatting* flaw (line 9), with a good source of data used (hardcoded string, line 5). This code snippet represents a case where Infer failed to understand that there is no such flaw in line 9 (FP):

```
1  private void goodG2B() throws Throwable
2  {
3      String data;
4
5      data = "foo";
6
7     if (data != null)
8     {
9         System.out.printf(data);
10    }
11
12  }
```

Example of possible *uncontrolled string formatting* flaw (line 9), with a bad source of data used (an environment variable, line 5). This code snippet represents a case where Infer successfully understand that there is no flaw in line 9 (TN), where explicitly defined string formatting:

```
1 private void goodB2G() throws Throwable
2    {
3        String data;
4
5       data = System.getenv("ADD");
6
7       if (data != null)
8       {
9           System.out.printf("%s%n", data);
10      }
11
12   }
```

So, it correctly does not detect the case when there is explicitly defined string formatting, even if the data is coming from a bad source.

However, Infer totally missed all cases (FN), when *System.out.format(data);* is used instead of *System.out.printf(data);* so the checker was not able to detect such weakness when using *format().*

CWE191_Integer_Underflow: Infer checker *'Integer overflow'* could only detect 15% of these test cases with 0 FP.

Example of *Integer underflow* flaw (line 9), with bad source data (line 5). This code snippet represents a case where Infer successfully detect the flaw in line 9 (TP):

```
1  public void bad() throws Throwable
2    {
3       int data;
4
5       data = Integer.MIN_VALUE;
6
7       if(data < 0)
8       {
9           int result = (int)(data * 2);
10          IO.writeLine("result: " + result);
11       }
12  }
```

Example of *Integer underflow* flaw (line 55), with bad source data (line 4). This code snippet represents a case where Infer fails to detect the flaw in line 55, where this will underflow because of (data * 2) < Byte.MIN_VALUE (FN):

```
1 public void bad() throws Throwable
2    {
3       byte data;
4       data = -1;
5       BufferedReader readerBuffered = null;
6       InputStreamReader readerInputStream = null;
7       try
8       {
9           readerInputStream = new InputStreamReader(System.in, "UTF-8");
10          readerBuffered = new BufferedReader(readerInputStream);
11          String stringNumber = readerBuffered.readLine();
12          if (stringNumber != null)
13          {
14              data = Byte.parseByte(stringNumber.trim());
15          }
16      }
17       catch (IOException exceptIO)
18      {
19           IO.logger.log(Level.WARNING, "Error with stream reading", exceptIO);
20      }
21       catch (NumberFormatException exceptNumberFormat)
22      {
23           IO.logger.log(Level.WARNING, "Error with number parsing", exceptNumberFormat);
24      }
25       finally
26      {
27          try
28           {
29              if (readerBuffered != null)
30              {
31                  readerBuffered.close();
32              }
33           }
34          catch (IOException exceptIO)
35          {
36               IO.logger.log(Level.WARNING, "Error closing BufferedReader", exceptIO);
37          }
38          finally
39          {
40              try
41              {
42                  if (readerInputStream != null)
43                  {
44                       readerInputStream.close();
45                  }
46              }
47              catch (IOException exceptIO)
48              {
49                  IO.logger.log(Level.WARNING, "Error closing InputStreamReader", exceptIO);
50              }
51          }
52      }
53      if (data < 0) /* ensure we won't have an overflow */
54      {
55          byte result = (byte)(data * 2);
56          IO.writeLine("result: " + result);
57        }
58    }
```

CWE193_Off_by_One_Error: Infer checker *'Buffer Overrun'* could detect 100% of these test cases with 0 FP.

An example of the detected TP, line 8 has the specified flaw because of line 11.

```
1  public void bad() throws Throwable
2   {
3
4     int[] intArray = new int[10];
5     int i = 0;
6     do
7     {
8         IO.writeLine("intArray[" + i + "] = " + (intArray[i] = i));
9         i++;
10    }
11     while (i <= intArray.length);
12  }
```

An example of the successfully undetected good() methods (TN), as in line 11 < has been used instead of <=.

```
1  private void good1() throws Throwable
2     {
3
4       int[] intArray = new int[10];
5       int i = 0;
6       do
7       {
8           IO.writeLine("intArray[" + i + "] = " + (intArray[i] = i));
9           i++;
10      }
11      while (i < intArray.length);
12    }
```

CWE248_Uncaught_Exception: Infer checker *'Unreachable Code'* could detect the one test case of this weakness with the FP.

The detected (TP), line 3 :

```
1  public void bad()
2     {
3        throw new Error("Really bad Error");
4     }
```

The detected (FP), line 25:

```
1  private void good1()
2     {
3        try
4       {
5            throw new Error("Really bad Error");
6       }
7       catch(Error error)
8       {
9           IO.logger.log(Level.WARNING, "Caught an Error", error);
10      }
11    }
```

CWE476_NULL_Pointer_Dereference: Infer checkers of *'Null Dereference'* which consist of two checkers: NULL_DEREFERENCE and NULLPTR_DEREFERNCE, was able to detect 80% of these test cases.

Example of *Null Pointer Dereference* flaw (line 7), with bad source data (line 6). This code snippet represents a case where Infer successfully detect the flaw in line 7 (TP):

```
1 public void bad() throws Throwable
2    {
3        if (IO.staticFive == 5)
4        {
5            {
6                String myString = null;
7                if ((myString != null) & (myString.length() > 0))
8                {
9                    IO.writeLine("The string length is greater than 0");
10               }
11           }
12        }
13    }
```

An example of a successfully undetected possible *Null Pointer Dereference* flaw, in line 14 (TN).

```
1 private void goodG2B1() throws Throwable
2    {
3        String data;
4        if (IO.STATIC_FINAL_FALSE)
5       {
6            data = null;
7       }
8       else
9       {
10           data = "This is not null";
11      }
12       if (IO.STATIC_FINAL_TRUE)
13      {
14           IO.writeLine("" + data.length());
15      }
16   }
```

An example of missed *Null Pointer Dereference* flaw, line 8 (FN).

```
1 public void bad() throws Throwable
2   {
3       if (privateReturnsTrue())
4       {
5           {
6              String myString = null;
7              myString = "Hello";
8              IO.writeLine(myString.length());
9              if (myString != null)
10               {
11                 myString = "my, how I've changed";
12               }
13              IO.writeLine(myString.length());
14          }
15       }
16   }
```

Note1: The last code snippet, which has been taken from CWE476 Juliet test cases group named 'null_check_after_deref', was missed by Infer because I think it is not a flaw, cause in line 7 'myString' has been assigned to hardcoded string value, even though it was initialized to null at line 6. For verifying this I deleted line 7, then re-ran the analysis and this time Infer detect line 8 correctly, which means this Juliet test case group is not efficient.

Note2: Infer checker *'Null Dereference'* did not give any FP for this weakness type, rather it gave replications. These replications came from NULLPTR_DEREFERNCE checker.

CWE690_NULL_Deref_From_Return: Infer checkers of *'Null Dereference'*; NULLPTR-DEREFERENCE and NULL_DEREERENCE, have detected 43% of these Juliet test cases; however, these checkers again gave some replications, and also some FP.

Example of *Null Dereference From Return* OR *Unchecked Return Value to NULL Pointer Dereference* flaw (line 14), with bad source data (line 6). This code snippet represents a case where Infer successfully detect the flaw in line 14 (TP):

```
1  public void bad() throws Throwable
2    {
3        StringBuilder data;
4        if (true)
5        {
6           data = CWE690_NULL_Deref_From_Return__Class_Helper.getStringBuilderBad();
7        }
8        else
9        {
10           data = null;
11       }
12        if (true)
13       {
14          String stringTrimmed = data.toString().trim();
15          IO.writeLine(stringTrimmed);
16       }
17    }
```

Example of same possible flaw (line 14), but with a good source of data used (line 10). This code snippet represents a case where Infer failed to understand that there is no flaw in line 14 (FP):

```
1 private void goodG2B1() throws Throwable
2    {
3       StringBuilder data;
4       if (privateFalse)
5       {
6           data = null;
7       }
8       else
9       {
10         data = CWE690_NULL_Deref_From_Return__Class_Helper.getStringBuilderGood();
11      }
12       if (privateTrue)
13      {
14          String stringTrimmed = data.toString().trim();
15          IO.writeLine(stringTrimmed);
16      }
17   }
```

Example of possible flaw (line 6), because of (line 4).This code snippet represents a case where Infer failed to understand that there is a flaw in line 6 (FN):

```
1 public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable
2    {
3        String data;
4        if(IO.staticReturnsTrueOrFalse())
5       {
6            data = request.getParameter("CWE690");
7       }
8       else
9       {
10           data = "CWE690";
11      }
12      if(IO.staticReturnsTrueOrFalse())
13      {
14           if(data.equals("CWE690"))
15          {
16              IO.writeLine("data is CWE690");
17          }
18       }
19      else
20        {
21          /* FIX: call equals() on string literal (that is not null) */
22          if("CWE690".equals(data))
23          {
24                IO.writeLine("data is CWE690");
25          }
26       }
27    }
```

CWE775_Missing_Release_of_File_Descriptor_or_Handle: There are only 2 test cases in this group, however, Infer checker *'Resource Leak'* detected one of them and missed the other, with 0 FP.

The detected (TP): here Infer detected line 7, however, according to Juliet line 17 should be detected.

```
1  public void bad()
2   {
3       ZipFile zFile = null;
4     try
5     {
6         zFile = new ZipFile("C:\\file.zip");
7         IO.writeLine("File contains " + zFile.size() + " entries.");
8     }
9     catch (ZipException exceptZip)
10    {
11        IO.logger.log(Level.WARNING, "Error with ZIP format", exceptZip);
12    }
13    catch (IOException exceptIO)
14    {
15        IO.logger.log(Level.WARNING, "Error reading file", exceptIO);
16    }
17      /* FLAW: ZipFile is not closed */
18  }
```

The (FN): Infer missed the following.

```
1 public void bad()
2    {
3         BufferedReader readerBuffered = null;
4         FileReader readerFile = null;
5        try
6        {
7            File file = new File("c:\\file.txt");
8            readerFile = new FileReader(file);
9            readerBuffered = new BufferedReader(readerFile);
10           String readString = readerBuffered.readLine();
11           IO.writeLine(readString);
12       }
13       catch (IOException exceptIO)
14       {
15           IO.logger.log(Level.WARNING, "Error with stream reading", exceptIO);
16       }
17         /* FLAW: file, readerFile, and readerBuffered not closed */
18   }
```

CWE833_Deadlock: Infer checker *'Deadlock'* was able to detect 4 out of 6 test cases, but the detected line is not accurate, and it also gave 4 FP out of 6 good()s.

The detected (TP): here Infer detected line 5.

```
1  public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable
2    {
3        if(request.isRequestedSessionIdValid())
4        {
5            helperAddBad();
6        }
7        else
8        {
9            helperMultiplyBad();
10       }
11        response.getWriter().write(intBadNumber1);
12    }
```

A missed test case (FN):

```
1  public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable
2  {
3     /* Branch so that not all requests call the same method.  If a valid request and an invalid
4      * one come in at the same time, a deadlock will result */
5     if(request.isRequestedSessionIdValid())
6     {
7          objectBadFirst.helperBowBad(objectBadSecond);
8     }
9     else
10    {
11         objectBadSecond.helperBowBad(objectBadFirst);
12    }
13 }
```

Conclusion: The primary results' observations of running Facebook Infer over the Juliet Test Suite using the default configuration, show that the tool was able to detect only three weaknesses out of the total 112 weaknesses test cases. The detected weaknesses are

"CWE476_NULL_Pointer_Dereference" (Recall=0.80), "CWE833_Deadlock" (Recall=0.67) ,and "CWE775_Missing_Release_of_File_Descriptor_or_Handle" (Recall=0.50).

However, when activating most of the checkers, the tool was able to detect extra five weaknesses, without showing any enhancement in the Recall of the previously detected weaknesses.

The final list of detected weaknesses (with activating most of the checkers):

Weaknesses ID and name	**Recall**	**False-positive**	**replications**	**checker name**
CWE134_Uncontrolled_Format_String	0.59	Yes	No	Checkers Printf Args
CWE191_Integer_Underflow	0.15	No	No	Integer overflow
CWE193_Off_by_One_Error	1.0	No	No	Buffer Overrun
CWE248_Uncaught_Exception	1.0	Yes	No	Unreachable Code
CWE476_NULL_Pointer_Dereference	0.80	No	Yes	Null Dereference
CWE690_NULL_Deref_From_Return	0.49	Yes	Yes	Null Dereference
CWE775_Missing_Release_of_File_Descriptor_or_Handle	0.50	No	No	resource leak
CWE833_Deadlock	0.67	Yes	No	Deadlock