Application and Evaluation of Rational Purify

Apr 24, 2007

Youngki Hong, Taeho Kong, Whanchul Kang,
Hyunwoong Park, Youngjin Ro, Hyeeun You

Contents

Deleted: 10

Deleted: 10

Deleted: 12

1 Introduction

1.1 Overview

IBM Rational® Purify is a tool for runtime analysis to help developers consider memory issues during the development phase. Purify is generally used for memory corruption detection, memory leak detection, application performance profiling, and code coverage analysis.

1.2 Working Environment

Purify provides memory leak detection for C/C++, Java, Visual C++, and all VS.NET managed languages in Windows, Linux and Unix environments.

1.3 Purpose

It is very difficult to check memory leaks and memory violations for a system since the bugs usually show their symptoms intermittently. Moreover, it is hard and time-consuming to find the source of the memory leaks for large development projects.

Purify can be used to reduce the cost for testing and debugging this problem by allowing a user to find defects quickly. It also provides detail information such as reasons why the defects occur or locations of bugs to aid identification of defects related with memory issues. The memory access defects which Purify usually finds are the followings:

- **Array Bound Checking Defects**: Purify checks defects in statistically or dynamically allocated memory for array. Array bounds read, array bounds write, and array bounds write defect message can be addressed to this type of defects.
- **Memory Usage Defects**: Purify checks defects related with memory usage such as un-initialized memory use, free memory use, and free mismatch defects.
- **Pointer Defects**: It is not allowed to use invalid or null pointers for reading, writing, or freeing. Purify checks null pointer use and invalid pointer use.
- **Stack Related Defects**: Purify shows the stack use defect such as stack overflow and stack out of bounds.
- **Memory Allocation Failure and Memory Leak**: Purify informs when memory allocation failure or memory leak occurs.

1.4 How it works

When it is linked with a program, Purify automatically inserts its verification code to the executable by parsing and adding to the object code. Purify maintains a table which is used for tracking the status of each byte of memory with two bits; the first bit recording whether it is allocated or not and the second bit recording whether it is initialized or not. Figure 1-1 shows four states which each byte of memory can have: red (unallocated and un-initialized), yellow (allocated but un-initialized), green (allocated and initialized), and blue (unallocated but initialized).

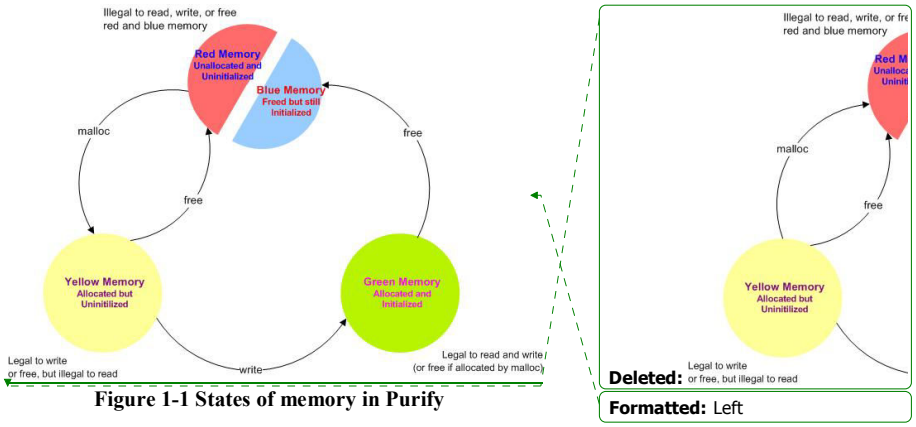


Figure 1-1 States of memory in Purify

Deleted:

Formatted: Left

2 Tool Experience and Evaluation Approach

2.1 Tool Experience (FileZilla Client and Server)

Our team chose FileZilla Client and Server to experience Purify. FileZilla Client is an open source FTP client for Windows. It supports FTP, SFTP, and FTPS (FTP over SSL/TLS). FileZilla Server is another FTP server product which FileZilla Client can communicate with. It supports FTP and FTP over SSL/TLS. [1] Both of FileZilla Client and Server were developed in C++.

Our team applied Purify to each of FileZilla Client and Server. We set up data collection option in Purify so that it can collect only memory related defect and leak data without collecting memory profiling data, since FileZilla Client and Server were written by C++ language. The results of tool experience are as follows.

Firstly, for the test of FileZilla Client, the sequence of events was opening up an FTP connection to FileZilla Server that is already running, performed some operations, such as file uploads and downloads, to FileZilla server, and then log out. Figure 2-1 shows that Purify detected three main memory leaks and no memory defects.

Secondly, for the test of FileZilla Server, the sequence of events was starting FileZilla Server before receiving FTP requests so that it can listen to FileZilla Client, performed some operations like file upload and download from FileZilla Client, and then disconnect FTP connection and shutdown FileZilla Server. Figure 2-2 shows that Purify only detected one main memory leakage.

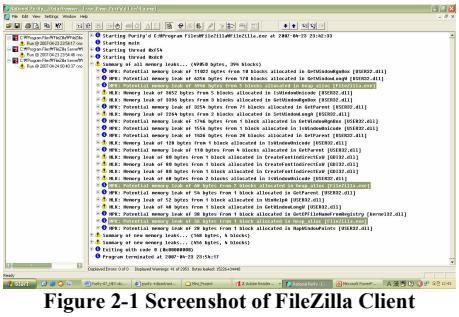


Figure 2-1 Screenshot of FileZilla Client

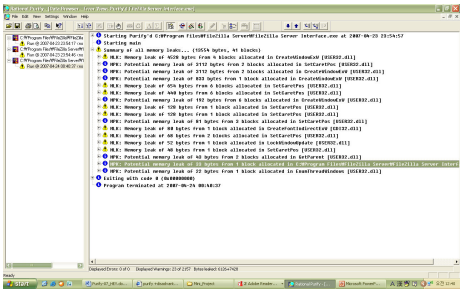


Figure 2-2 Screenshot of FileZilla Server

2.2 Evaluation Approach

From the tool experience, we found that Purify can be applied to find defect information like memory leaks and to fix the defects. However, the tool experience is not enough to evaluate Purify closely. The major reason is that only memory leakage information was detected in FileZilla open source, even though the tool can detect memory defects and memory profiling in detail. The other reason is that this tool experience was only based on the source codes implemented by C++. This can underestimate the capability of Purify that can detect memory defects and leaks related to Java programs. We derived C and Java program experiment cases with three evaluation criteria to evaluate Purify in more objective ways.

Deleted: the results of

Deleted: is

Deleted: may not be

Deleted: good

Deleted: provided

Deleted: open

Deleted: s

Deleted: For those reasons, our team

Deleted: additionally

Deleted: several

Deleted: and

Deleted: criteria

Deleted: as follows, so that we can

Deleted: much

3 Application of Purify

3.1 Purify for C

In order to validate our understanding of how Purify can find memory defects, we created a "memerrors.c" application into which we injected 8 memory defects and 2 memory leaks.

- 2 un-initialized memory defects: un-initialized memory read (UMR) and un-initialized memory copy (UMC)
- 2 invalid pointer defects: invalid pointer read (IPR) and invalid pointer write (IPW)
- 2 beyond stack defects: beyond stack read (BSR) and beyond stack write (BSW)
- 2 array bound defects: array bound read (ABR) and array bound write (ABW)
- 2 memory leaks (MLK)

For the source code with injected memory defects, refer to appendix 1.

As shown in figure 3-1, Purify detected 6 memory defects out of 8 and 2 memory leaks out of 2 that we injected into the code.

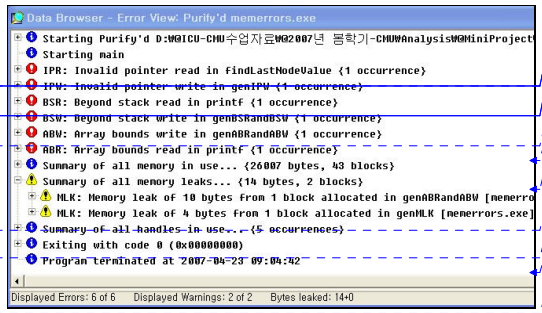


Figure 3-1 Detection Defects with Purify

Since there was only one control flow and function call, the line coverage was 100%.

2.2.1 Application cases

We applied Purify to three application cases: a simple C program, a simple Java program, and a more complicated Java program. For each experiment case, we injected several intentional defects or memory leaks.

Experiment cases		Kind of injected defects
C	Simple	• 8 kinds of defects • 2 kinds of memory leaks
Java	Simple	• Simple memory leak: n memory allocation & n-1 memory free
	More complicated	• Comparison between a method with memory leak and a method with very high memry allocation without leak.

Table 2-1 Application Cases

Defects and leakage data will be collected for applying the tool to C program, and profiling data will be collected for applying the tool to Java programs.

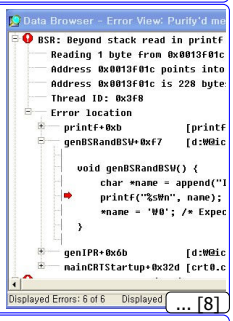
2.2.2 Evaluation Criteria

On the basis of data resulted from application cases, we will evaluate Purify with the following criteria.

Factor	Definition & Criteria
Soundness	• The extent of how many false-positives or false-negatives Purify has
Usability	• How fast Purify can be installed • How easily Purify can be used
Performance	• The extent of how fast Purify detects defects in comparison with inspection

Table 2-2 Evaluation Criteria

Deleted: Experiment
Formatted: Bullets and Numbering
Deleted: There are…three …major experimen…t… that are applied to Purify…. …would intentionally … [1]
Deleted: to
Deleted: Purify could detect, and validate whether Purify can actually detecting such defects.
Deleted: i
Deleted: Complicated memory leak: c…it [2]
Deleted: C
Deleted: Experiment
Deleted: The …d…s…data needed to evaluate Purify … applied to Purif [3]
Deleted: was able to …among…among Please refer to the following screenshots to see the detected memory defects and leaks. [4]
Formatted: Bullets and Numbering
Deleted: ¶ ¶ Ev [5]
Deleted: collected by Purify after doing experiment…would …whether …can detect all of defects, memory leaks [6]
Deleted: .
Formatted Table
Formatted: Bullets and Numbering
Deleted: Usability
Deleted: Capability related to how fast a user can install Purify and how e [7]
Formatted: Bullets and Numbering
Deleted: Soundness
Deleted: The extent of how many false-positives or false-negatives Purify has



Deleted: [8]
Deleted: Consistency [9]
Deleted: s

3.2 Purify for Java

Java is generally known as it has few memory related issue because it doesn't allow using pointers and provides strict runtime checking function with Java Virtual Machine (JVM). Furthermore, JVM uses garbage collection to solve unused memory problems. However, Java still has some memory problem issues.

Purify enable users to find out potential memory leaks by profiling the memory usage of a Java program.

3.2.1 A simple Java program

We applied Purify to a simple Java program which has a method causing intentional memory leaks to evaluate its effectiveness. The basic steps are the followings:

1) Run an analysis for memory leaks by collecting memory profiling information in a Java application.

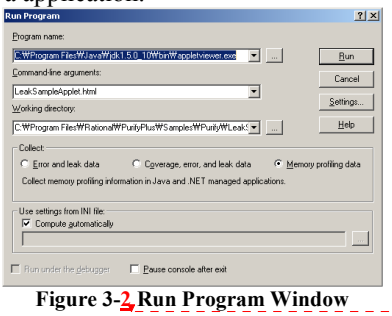


Figure 3-2 Run Program Window

2) After the analysis gets started, perform a garbage collection in Purify so that the application to analyze becomes stable, and then take a snapshot.

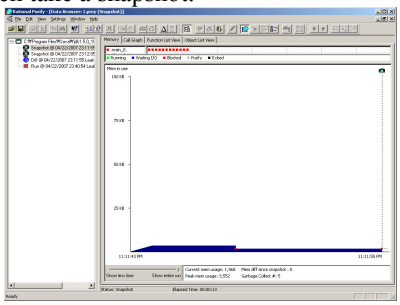


Figure 3-3 Garbage collection in Purify

3) On the application to be analyzed, generate some memory leaks by pressing the start button (This application is intentionally written to have a memory leak problem.)

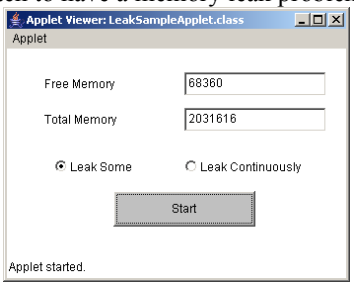


Figure 3-4 Generating memory leaks

4) As soon as the memory leak is generated, it is shown that the height of the graph (i.e., current memory usage of the application) rapidly increases. A couple of seconds later, perform one more garbage collection and then take another snapshot.

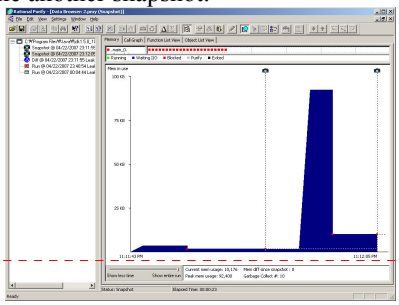


Figure 3-5 Current memory usage

5) At a glance, it is shown that the obvious difference in height between the previous and the next states of the graph. For more precise comparison, view the difference by using the Compare Runs feature with a call graph representing the amount of current memory usage of each method as the thickness of edges. It is considered that the thicker edge the more memory usage.

Deleted: 5

Deleted: 3

Deleted: 6

Deleted: 4

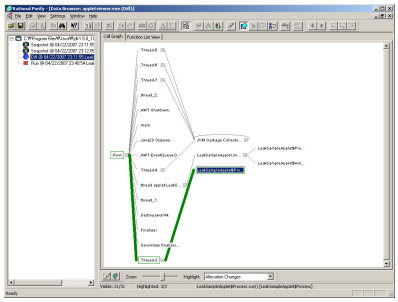


Figure 3-6 Compare Run

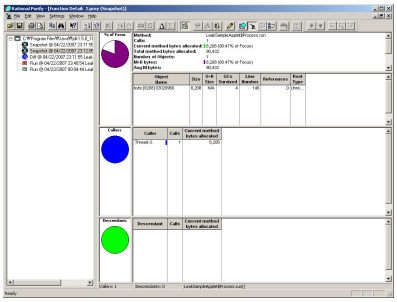


Figure 3-8 Detail view of a method

Also, it is possible to see exact difference in terms of the number of calls by using a function list view. (e.g., the difference in memory is 8,208 as below.)

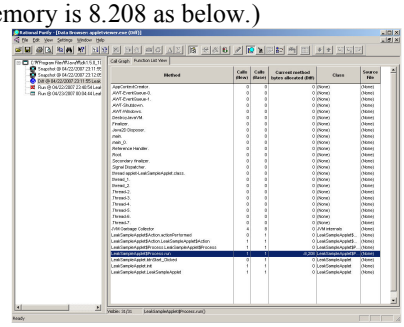


Figure 3-7 Difference in memory

6) With the results shown in the step 5, it is found that 'run' method of 'Process' class is likely to contain memory leak defects. The detail view of a method provides more precise information in terms of callers and descendants as well as the callee itself. Furthermore, Purify provides 'roughly' where each memory allocation—which can be used as a yardstick for memory leaks—is carried out in the source.

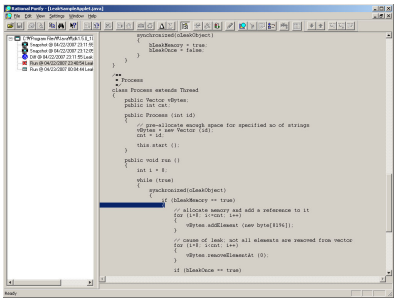


Figure 3-9 Source view of a method

7) In the source code, it is shown that the second (i.e. lower) for-statement causes a memory leak since one less element of 'VBytes' vector is removed than added by the first (i.e. upper) for-statement in 'run' method. → for (i=0; i<cnt; i++). Therefore, change 'i<cnt' to 'i<=cnt', recompile the code and re-run the analysis, and then it is shown that the memory leak disappears as below.

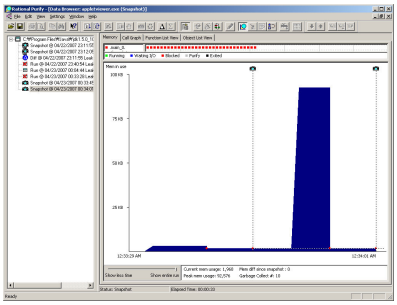


Figure 3-10 Disappearance of memory leak

Deleted: 7

Deleted: 9

Formatted: Indent: Left: 0.17"

Deleted: 10

Deleted: 8

Deleted: 11

3.2.2 A more complicated Java program

Now, we applied Purify to a more complex Java program to evaluate its soundness. The program has two methods: leakingRequestLog and noLeak. The leakingRequestLog method causes memory leak. The noLeak method causes the highest memory allocation, but no memory leak because all of the objects it used get garbage collected at the end of the method. The main method of the program iterates the two methods. For capturing exact snapshot, we made a main thread sleep to slow down leaking process.

Figure 3-12 shows snapshot memory profile data which is captured after the garbage collector is invoked.

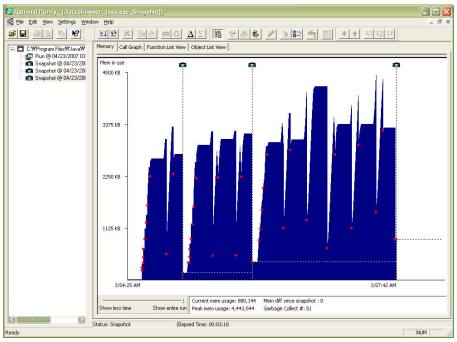


Figure 3-11 Memory in Use profile data for a more complex Java program

As we have seen in the previous simple example, the amount of memory leakage can be estimated by looking at the difference in the y-axis value between the second and third snapshot point values in Figure 3-12. However, since this program has two methods, we can't know where the leakage occurs (in other words, which method causes the leakage) from figure 3-12.

Figure 3-13 is a call graph, which shows the chain of methods called between the second and third snapshots. The call graph displays methods with significant memory usage, and the method that used most of the memory in runtime is highlighted with box and bold-colored line. The highlighted method is the first candidate for possible memory leak. We can see that both highlighted leakingRequestLog method and non-highlighted noLeak method.

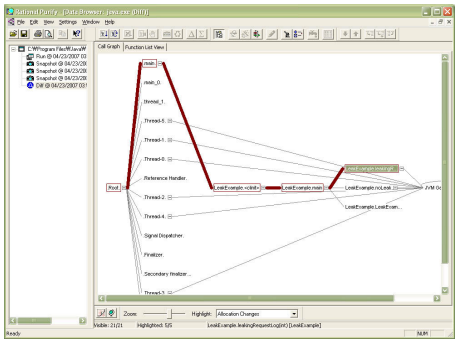


Figure 3-12 Memory Call Graph from the compared runs

Deleted: 13

Figure 3-14 is a function list view, which enables us to confirm the exact leak by comparing method list of the two snapshots. The leakingRequestLog method has 480,000 as difference of current method bytes allocated. On other hand, the noLeak method has 0 as difference of current method bytes allocated.

Method	Calls (New)	Calls (Base)	Current method bytes allocated (Diff)	Class	Source File
.Finalizer.	0	0	0	(None)	(None)
.main.	0	0	0	(None)	(None)
.main_0.	0	0	0	(None)	(None)
.Reference Handler.	0	0	0	(None)	(None)
.Root.	0	0	0	(None)	(None)
.Secondary finalizer.	0	0	0	(None)	(None)
.Signal Dispatcher.	0	0	0	(None)	(None)
.thread_1.	0	0	0	(None)	(None)
.thread_2.	0	0	0	(None)	(None)
.Thread-0.	0	0	0	(None)	(None)
.Thread-1.	0	0	0	(None)	(None)
.Thread-2.	0	0	0	(None)	(None)
.Thread-3.	0	0	0	(None)	(None)
.Thread-4.	0	0	0	(None)	(None)
.Thread-5.	0	0	0	(None)	(None)
JVM Garbage Collector	48	30	0	JVM internals	(None)
LeakExample.<clinit>	1	1	0	LeakExample	(None)
LeakExample.LeakExample	1	1	0	LeakExample	(None)
LeakExample.leakingRequestLog	11	5	480,000	LeakExample	(None)
LeakExample.main	1	1	0	LeakExample	(None)
LeakExample.noLeak	11	5	0	LeakExample	(None)

Figure 3-13 List of methods invoked in two snapshots

Deleted: 12

Deleted: 14

From these two Java examples, we can know about memory leakage of Java programs through memory profiling function of Purify. The tool also enables us review and modify the source code if we have the source code. Thus, Purify gives sound results when users can capture clear snapshots.

4 Evaluation of Purify

4.1 Soundness & Precision

Purify is a relatively sound memory defect detection tool since as described in the section 3.1 it has correctly detected 8 out of 10 injected defects (i.e., 75% detection; six memory defects in terms of Invalid Pointer defect, Beyond Stack defect and Array Bound Defect, and two memory leaks). The detection of memory defects from C/C++ applications such as Invalid Pointer defect, Beyond Stack defect, and Array Bound Defect is important to developers because the memory defects may cause the entire software system to be broken out. It is usually difficult for developers to find those defects during code inspection. We could find those defects easily by the help of Purify. Therefore, Purify can be used to effectively prevent the program corruption caused by memory defects. Discovery of two memory leaks with little time and effort is certainly valuable to decrease memory leaks before delivering an application, because an accumulated memory leak having begun with a small one may cause a significant problem if the application runs for a long time.

Purify for C/C++ did not shown any false positives which mean detecting false, imprecise defects. However, there were a couple of false negatives that Purify was not able to find uninitialized memory defects (i.e., copies and reads) which means that it missed some defects injected. The uninitialized memory defects (UMR and UMC) may be trivial or important depending on circumstances where a software system is developed. However, they would be very critical, for example, in banking systems because uninitialized memory defects may wrong change the balance of a banking account. IBM Rational which created Purify mentions that it should be able to detect uninitialized memory defects but it could not.

It is different from C/C++ that Purify provides only a memory leak analysis way for Java by collecting memory profiling information. More precisely, Purify just offers information only about memory usages and not directly points out the where and why of memory leaks at all. You may expect that some methods of the code-which occupy a great deal of (current) memory usage at the point of time of an analysis snapshot-are the most possible spots to cause memory leaks.

In short, Purify is very sound in terms of Java memory leak analysis because it just shows memory profiling information with which developers may or may not detect memory leaks. From another perspective, some may argue that Purify is unsound in a sense that it often shows the methods which had occupied the most amount of memory usage (i.e., used as an indication of possible memory leaks) but which are not actually memory leaks. However, what Purify is supposed to do is just showing memory usages, not exactly showing memory leaks. In other words, as long as you can certainly take a snapshot at the desired point of time under clear conditions, Purify always indicates actual memory leaks. For this reason, Purify is sound analysis tool for finding memory leaks for Java.

4.2 Usability

4.2.1 Ease of installation (setup)

Generally, the installation of Rational Purify Windows version is not much difficult for users who are familiar with Windows applications. It uses IBM Rational Setup Wizard which is very similar with other setup wizards such as Microsoft Visual Studio setup wizard or Microsoft Office setup wizard. Therefore, even though a user has never installed Rational Purify before, the installation does not cause serious confusing to the user. Actually, all of six group members completely installed the application within 3 minutes. In conclusion, the installation of Rational Purify is very easy and not time-consuming.

There is only one problem in terms of usability while an ordinary user tries to install Rational Purify. The application uses a special tool, IBM Rational License Key Administrator, for validating users' license key. If a user is not familiar with the way of license key validation of IBM Rational, then the user can be confused for a while. Fortunately, IBM Rational License Key Administrator looks very simple and most of users can find correct way of license key validation intuitively and quickly.

4.2.2 Ease of Application

Purify is easy to use because it supports visualization, various views, and access to source files.

First, Purify presents memory related issues in visual ways. It shows the results with different symbols and colors. Especially, when Purify is applied to C/C++ programs, each type of defects is shown in red, yellow, green, or blue. Since red and yellow color parts can be critical, users can easily recognize where critical issues occur. It also shows error detection view hierarchically. Second, Purify provides various views, which enable users to easily understand the analysis result. For example, when Purify is applied to Java programs, it shows memory profiling information in memory, call graph, function list view, and object list views. Third, Purify supports users to access to the source code where the problem occurs when the source files exist.

However, Purify has some limitations in usability. Capturing usable snapshots requires much works for users. If users are not able to capture usable snapshots, the result given by Purify can be confusing to users. Also, since it is a dynamic analysis tool, Purify can't notify exact location of defects to users.

4.3 Performance

In this section, we observe the whole examining time with Purify and the inspection. When we use Purify for detecting memory leak defects, we have to spend time to compile and execute the source code. Our team used C/C++ source code having 40 lines, so we estimated that time as about 3 minutes because we need compile time, Purify execution time and result analysis time. Also, we can expect that the entire time for analysis with Purify will not increase sharply according to the amount of LOC.

However, the general review rate for source code inspection is 200 lines of code/hour on average [5]. Based on this factor, we can estimate that we will spend about 15 minutes about inspection time of 40 lines. On the other side, the time for inspection will be increase according to the amount of LOC.

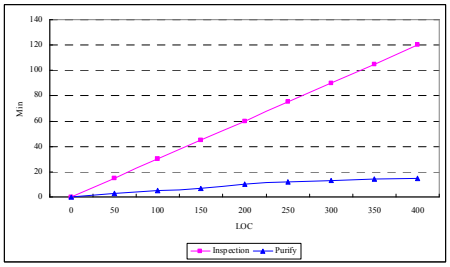


Figure 4-1 Time estimation to detect memory leak

We can detect memory leaks easily if a program is small size. However, if the program is huge size, the detecting memory leak line by line can be time consuming and cost a lot. We could find those memory leaks very easily with the memory leak detection function of Purify. Besides, all defects detected from Purify cannot be found in the inspection. It is because the defects of memory leak can be found in run-time.

Thus, in the viewpoint of performance, the examining speed with Purify can be much faster than that with code inspection for finding memory leak related defects in the code. Purify can save a time and a cost.

5 Conclusion

5.1 Strengths & Weaknesses

5.1.1 Strengths

1) Usability

- Usable visualization of analysis results (e.g., a user can easily distinguish of what type each defect found is by seeing the color of its description.)
- Ease of installation and conformance to other common applications
- Various views in terms of memory defects and leaks (e.g., memory usage graph, call graph, function list view and object list view)
- Ease of access to source code files so as to figure out where a memory defect or leak is caused.
- Running an analysis without having source code (i.e., taking advantage of dynamic analysis)
- Details in terms of memory leaks results (i.e., number of calls, current method bytes allocated, number of objects and source files)

2) Soundness

- Consistent profiling for current memory usages in Java applications
- Detection of different types of possible memory defects such as invalid pointer errors, beyond stack errors and array bound errors

3) Performance

- Higher effectiveness to find memory defects and leaks compared with code inspection

5.1.2 Weaknesses

1) Usability

- Unrealistically and artificially modified source code required to properly perform an analysis so as to figure out the two points of time between before and after a memory leak
- Purify does not exactly show in what line of code a memory defect is caused and can not indicate it at all in some cases
- High dependency on manual inputs from a user to complete an analysis (e.g., a user should manually let Purify precisely know where to take snapshots for Java application analysis)

2) Soundness

- Unable to detect uninitialized memory defects

3) Performance

- Memory profiling which usually heavily affects the performance of Java application tested

5.2 Would you continue using it?

So far, we have identified strength and weakness on the basis of results of several experiments. According to the strength and weakness, we can say that Purify analysis tool can be good enough to be applied to development project such as Studio project, even though applying it to Java based development project needs some additional efforts to determine whether memory leaks shown by Purify is actually defects or not.

Our team's Studio project is to develop DBAuditor system that supports DB performance test according to TPC (Transaction Processing Council) standard. For doing this, DBAuditor system has 7 major components, such as (1) Project management, (2) Schema Builder, (3) Data Generator, (4) Query Executor, (5) ACID test, (6) System Usage Monitoring, and (7) Report management. We will implement DBAuditor system using Java for improving platform independency. Furthermore, we will implement (6) System Usage Monitoring component using C, and will wrap it with Java using JNI (Java Native Interface). For this reason,

DBAuditor system has a characteristic of having Java as well as C program.

For this kind of characteristics of our project, Purify can be effectively applied to our Studio project. There are two major reasons as follows.

Firstly, C program for System Usage Monitoring component can be effectively and efficiently analyzed by Purify, due to its high usability, soundness for defects, and performance for detecting defects.

Secondly, Java program for most of components in DBAuditor system can be efficiently analyzed by Purify. The major reason is that Purify provides high performance for analyzing memory leaks in Java program. However, it may require more efforts to determine whether profiling data actually indicates memory leaks or not, or it may be recommended that it should be better to being used with other complementary analysis tools.

To sum up, Purify is very appropriate to be applied to projects that have a goal to develop a system using C as well as Java at the same time, due to its high usability, soundness for detecting defects, and performance for detecting easily.

6 Future Work

We analyzed C-source with three C-Compilers, Visual Studio 6.0 C-Compiler, Visual Studio 2005 C-Compiler, and Dev-C Compiler. We expected same result regardless of type of C-Compiler. However, purify produced in different results according to C-Complier. For more different defects according to the type of C-Compiler, see the Appendix 2. We were very confused why this happen. We are not sure whether problems are caused by defect detection of smart compiler or by mismatch between Purify and each C-Compiler. Therefore, Purify should consider a consistency of analysis regardless of type of Compiler.

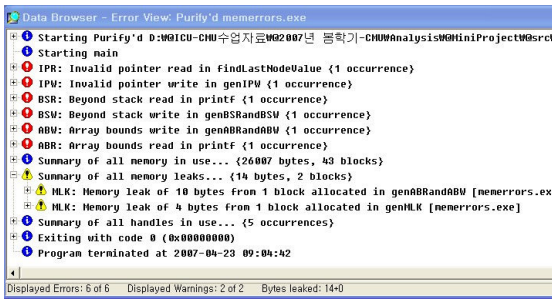
References

[1] http://en.wikipedia.org/wiki/FileZilla
[2] http://www-306.ibm.com/software/awdtools/purify/
[3] http://en.wikipedia.org/wiki/IBM_Rational_Purify
[4] Goran Begic, etc., 'An introduction to runtime analysis with Rational PurifyPlus' (http://www-128.ibm.com/developerworks/rational/library/mar07/begic_pratt/index.html), 2007
[5] Alison A. Gately, 'Design and Code Inspection Metrics', ASM 1999
[6] http://www.ing.iac.es/~docs/external/purify/purify-4_1.pdf
[7] Jim Patrick, Handling memory leaks in Java programs, IBM Pervasive Computing, 01 Feb 2001, (http://www-128.ibm.com/developerworks/java/library/j-leaks)
[8] Sanjay Gupta, Preventing Memory Leaks in a Java Application with Rational Purify: A Case Study, Wipro Technologies, 04 Dec 2003, (http://www-128.ibm.com /developerworks/rational/library/1499.html)
[9] Satish Chandra Gupta, Java memory leaks - Catch me if you can, Rational Software, IBM, 16 Aug 2005

Appendix 1 : Source with injected memory defects

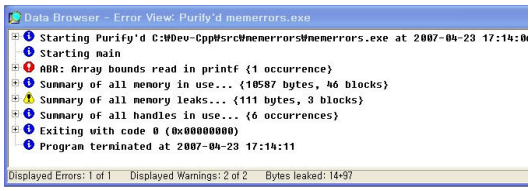
```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <fcntl.h>

int *plk = NULL;
int *mlk = NULL;

void genUMC(int *pi) {
   int j;
   *pi = j;  /* Expect UMC: j is un-initialized, copied into
*pi */
}

void genUMR() {
   int i=10, j;
   genUMC(&i);
   j = i + 2; /* Expect UMR: Using i, which is now junk
value */
}

void genIPR() {
   int *ipr = (int *) malloc(4 * sizeof(int));
   int i = *(ipr - 1000); /* Expect IPR */
   int j = *(ipr + 1000);
   free(ipr);
}

void genIPW() {
   int *ipw = (int *) malloc(5 * sizeof(int));
   *(ipw - 1000) = 0; /* Expect IPW */
   *(ipw + 1000) = 0;
   free(ipw);
}

char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;

   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
      result[i] = s1[j];
   }

   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
      result[i] = s2[j];
   }

   result[++i] = '\0';
   return result;
}

void genBSRandBSW() {
   char *name = append("IBM ", append("Rational ",
"Purify"));
   printf("%s\n", name); /* Expect BSR */
   *name = '\0'; /* Expect BSW */
}

void genABRandABW() {
   const char *name = "IBM Rational Purify";
   char *str = (char*) malloc(10);
   strncpy(str, name, 10);
   str[11] = '\0'; /* Expect ABW */
   printf("%s\n", str); /* Expect ABR */
}

void genMLK() {
   mlk = (int *) malloc(1 * sizeof(int)); /* Expect MLK */
   mlk = NULL;
}

int main() {
   printf("Hello Purify!\n");
   printf("Generating UMR & UMC:\n"); genUMR();
   printf("Generating IPR:\n"); genIPR();
   printf("Generating IPW:\n"); genIPW();
   printf("Generating BSR & BSW:\n");
genBSRandBSW();
   printf("Generating ABR & ABW:\n");
genABRandABW();
   printf("Generating MLK:\n"); genMLK();
   return 0;
}
```

Appendix 2. Different results according to different types of C-Compiler.



A2-1 Detection Defect with Visual Studio 6.0 C-Compiler

A2-2 Detection Defect with Visual Studio 2005 C-Compiler



A2-3 Detection Defect with Dev-C Compiler

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:53:00 AM**

There are

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:52:00 AM**

three

Page 5: [1] Deleted **HyeEun You** **4/24/2007 5:00:00 AM**

major experimen

Page 5: [1] Deleted **HyeEun You** **4/24/2007 5:09:00 AM**

t

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:53:00 AM**

that are applied to Purify.

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:54:00 AM**

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:54:00 AM**

would intentionally

Page 5: [1] Deleted **HyeEun You** **4/24/2007 4:55:00 AM**

that

Page 5: [2] Deleted **HyeEun You** **4/24/2007 5:03:00 AM**

Complicated memory leak: c

Page 5: [2] Deleted **HyeEun You** **4/24/2007 5:04:00 AM**

it

Page 5: [3] Deleted **HyeEun You** **4/24/2007 5:12:00 AM**

The

Page 5: [3] Deleted **HyeEun You** **4/24/2007 5:12:00 AM**

d

Page 5: [3] Deleted **HyeEun You** **4/24/2007 5:13:00 AM**

s

Page 5: [3] Deleted **HyeEun You** **4/24/2007 5:12:00 AM**

data needed to evaluate Purify

Page 5: [3] Deleted **HyeEun You** **4/24/2007 5:05:00 AM**

applied to Purify

Page 5: [4] Deleted **HyeEun You** **4/24/2007 5:38:00 AM**

was able to

Page 5: [4] Deleted **HyeEun You** **4/24/2007 5:42:00 AM**

among

Page 5: [4] Deleted **HyeEun You** **4/24/2007 5:42:00 AM**

among

Page 5: [4] Deleted **HyeEun You** **4/24/2007 5:41:00 AM**

Please refer to the following screenshots to see the detected memory defects and leaks.

Page 5: [5] Deleted **HyeEun You** **4/24/2007 4:50:00 AM**

Page 5: [5] Deleted **HyeEun You** **4/24/2007 4:50:00 AM**

Ev

Page 5: [6] Deleted **HyeEun You** **4/24/2007 5:16:00 AM**

collected by Purify after doing experiment

Page 5: [6] Deleted **HyeEun You** **4/24/2007 5:18:00 AM**

would

Page 5: [6] Deleted **HyeEun You** **4/24/2007 5:17:00 AM**

whether

Page 5: [6] Deleted **HyeEun You** **4/24/2007 5:18:00 AM**

can detect all of defects, memory leaks. For doing this, we would use four major evaluation

Page 5: [6] Deleted **HyeEun You** **4/24/2007 5:24:00 AM**

as follows

Page 5: [7] Deleted **HyeEun You** **4/24/2007 5:25:00 AM**

Capability related to how fast a user can install Purify and how easy a user can use Purify

Page 5: [8] Deleted **HyeEun You** **4/24/2007 5:45:00 AM**

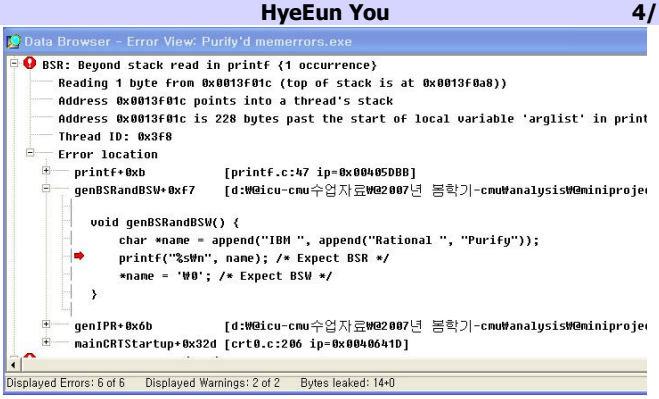


Figure 3-2 Good user view with hierarchical structure

Page 5: [9] Deleted **HyeEun You** **4/24/2007 5:25:00 AM**

Consistency	Capability related to how exactly Purify can detect defects regardless of used compilers