Evaluation of Rational Purify

Mike Hagen, Zhen Zhang
Master of Software Engineering Program
School of Computer Science
Carnegie Mellon University
{ mhagen, zhenz }@andrew.cmu.edu

1 Abstract

Our team evaluated IBM's Rational Purify, a testing application that checks for memory leaks and memory violation errors at runtime. This easy to use product automatically instruments an executable and reports any memory errors in real-time, thus simplifying the process of debugging hard-to-find memory leaks. In order to evaluate this product, we selected several evaluation criteria, such as ease of installation, usability, and accuracy of results. We then used Purify to test two programs: a "Hello World" program injected with memory violations, and an open source program, Putty, a secure shell client based on OpenSSH.

2 Problem Description

Checking a system for memory leaks and memory violations is a difficult task. Problems are often intermittent, difficult to test, and hard to pinpoint. In the instance of array access errors, the system may work correctly on the development system because the uninitialized memory has been set to zero, whereas on the deployment system, it may be set to another value, leading to aberrant behavior. For memory leaks, it is difficult to locate the source of the problem.

One solution is to avoid the problem altogether by using a garbage collected language such as Java, Visual Basic, or C#. Such languages track all memory references and detect when a block of memory no longer has any references and can be garbage collected. This unshackles the developer from the tedious task of allocating and deallocating memory. However, it introduces a similar set of problems. If a block of memory is referenced when it is no longer needed, it won't be garbage collected. Other applications will be prevented from using it, thereby leading to resource problems again.

Another solution is to implement code instrumentation. This serves to log the details of memory allocations, which will simplify the process of debugging memory leaks. However, this approach still doesn't address the problem of array access errors. Some integrated development environments (IDEs) attempt to track this in debug builds by setting uninitialized memory to a semaphore value. At runtime, the system checks for the value and displays a warning that uninitialized memory has been accessed. However, this approach doesn't work when using release builds of applications or libraries.

3 Description of Purify

3.1 What does it do?

Rational Purify is a tool used to dynamically check for memory leaks and a wide range of memory violations, such as: array bound errors, uninitialized memory errors, and invalid write errors, to name a few. There are versions of Purify available for Windows and Linux. It supports Java, C/C++, and .Net managed code. In the Windows version, there is integrated support for the Visual Studio 6.0, .Net, and Eclipse IDEs.

3.2 How is it used?

After an executable has been compiled, Purify is launched in one of two manners: either directly within the IDE, or as a standalone application. The application is then launched by invoking the "Run…" command, which runs the application normally. As memory errors are encountered, they are displayed in the main window of Purify, along with the details of the error. Upon termination of the application, errors are displayed for any memory that has been leaked.

3.3 How does it work?

Rational Purify addresses the problem of detecting memory violation errors in two manners. First, it instruments the code using its Object Code Insertion (OCI) technology. This performs automated code weaving when the executable is first loaded, and requires no changes to be made to the source code. Another key feature of OCI is that the source code isn't required, which is useful for third-party libraries. However, for maximum benefit, it should be used with debug builds, as this allows it to display the offending source code and the stack trace leading to the error.

The second approach used by Purify for detecting memory violations is to use a 2-bit state machine for each byte of memory that has been allocated. One bit is used to represent whether the memory has been allocated, while the second bit is used to record whether or not the memory has been initialized. This allows Purify to track the state of every memory location and display warnings appropriately. For instance, it is valid to write to uninitialized memory, but not to read from it.

3.4 How sound is it?

Rational Purify is an unsound analysis tool. It cannot prove that your code is free of defects, since this information is only available at runtime. However, it can guarantee that if a control flow path is taken, Purify will report the correct results. Its analysis is both safe and precise for edges of the control flow graph that have bee traversed. In order to cover the testing gap left by this unsound analysis method, a code coverage tool should also be used, such as Rational PureCoverage. This would allow the testing engineer to report that a certain percentage of the code has been tested, and that subset of code is free of memory violations.

4 Experiment Plan

We chose to evaluate the Windows version of Rational Purify. In order to evaluate the product, we first decided upon our evaluation criteria. We decided to capture qualitative data on the ease of setup, and the ease of use. We also decided to capture quantitative data associated with the number of memory defects found, as well as the test coverage. Finally, we also had to choose the products on which to test Purify. We chose a "Hello World" example, to make sure we had the tool installed correctly and that we knew how to use it. We also decided to test Putty, a widely used open-source SSH client for Windows, based on the OpenSSH library.

5 Results

After executing our experiment plan, these are the results that we recorded:

5.1 Ease of Setup

The product was difficult to set up and install in two regards. First, we initially planned to install the Linux version of the product and use it to test a third-party code base we intended to incorporate into another project. However, we spent a significant amount of time trying to get the product to work on Linux, approximately 8 hours. The problem we encountered occurred when we tried to run Purify on our "Hello World" example. The application crashed and displayed an obscure error code along with a stack trace and error report. Please refer to the appendix for a detailed error report.

```
Warning: Non-ABI conforming
section found in object file…
```

However, after searching the product documentation and performing a search on Google, we were unable to find any resolution to the error. We consulted with a Linux enthusiast, who suggested it was a versioning problem. The version of Linux we are using is Red Hat Linux 3.3.3-7, and our version of gcc is 3.3.3. This is much newer that the versions supported by Purify. However, we didn't have the time or resources to find an older version of the environment, so we opted to revise our experiment plan.

After shifting our focus to the Windows version of Purify, we ran into another speed bump. We installed the tool and imported the downloaded license file through the license manager. However, when we attempted to launch Purify from the IDE, the system just hung for 15-30 seconds, then did nothing. The application wasn't launched, and no errors were displayed. We then tried to launch the standalone version of the application. This is when we discovered the source of the problem. On application startup, Purify uses a helper application to validate the license file, and then finishes the startup process. In our case, Purify managed to find another license server on the network. However, Purify was unable to communicate with it due to a problem with a non-standard port number. After timing out, it failed to next try the valid license file that was given to it.

Despite the initial setup problems, there were no problems integrating the product into the IDE. When the IDE was launched, there were new sets of buttons to provide the functionality provided by Purify. This tight level of integration is crucial for developer adoption. It allows a software engineer to test and validate code without having to incorporate many extra steps in the development process.

5.2 Ease of Use

After we installed the tool, we found the interface to be very intuitive. Checking a program for memory violations was as simple as browsing for the executable and running it. This was important, as it doesn't require any modifications to be performed to the source code in order to see results. The benefits were immediately available.

The results were relatively easy to comprehend. Please refer to figure 1 below. One potential sticking point is that the memory errors are given a 3 letter code, such as ABR, IPW, or UMR. However, the description of the code is always displayed next to it. An explanation of the class of error is also contained in the help file.

The error summary can also be expanded to reveal details about the execution trace that lead to the error, as can be seen in figure 2. The address and thread information is displayed, as well as the stack trace. If source code is available, the stack trace can be expanded to reveal the exact location of the memory violation.

We encountered one major usability problem when using the product. We encountered a bug when executing the following steps: enable "Engage Purify Integration" in the IDE, run Purify within the IDE, attempt to shut down Visual Studio. A window was displayed with the following error:

> Microsoft Development Environment cannot shut down because a modal dialog is active. Close the active dialog and try again.

However, there was no dialog box to be seen. The only recourse was to kill the process in the Windows task manager.

5.3 Defects Found

The following are the quantitative results uncovered by our experiment plan:

5.3.1 Hello world

In order to validate our understanding of how Purify is set up and how to interpret the results, we created a "Hello World" application into which we injected several memory defects. Please refer to appendix 11.2 for the source code.

Purify was able to detect all five errors that we injected into the code. Please refer to figures 1 and 2 for screenshots of the results:

- 2 memory leaks
- 1 array bounds read (ABR) error
- 1 invalid pointer write (IPW) error
- 1 uninitialized memory read (UMR) error

Since there was only one control flow path, the line coverage was 100%.

As an additional qualitative measurement, we had initially planned on capturing the amount of time required to fix the defects in the source code. However, there would be too much variance in this measurement, since the amount of time to fix a defect depends on the severity of the defect. The figure would be further skewed if the system had to be redesigned to avoid the problem.

5.3.2 Putty

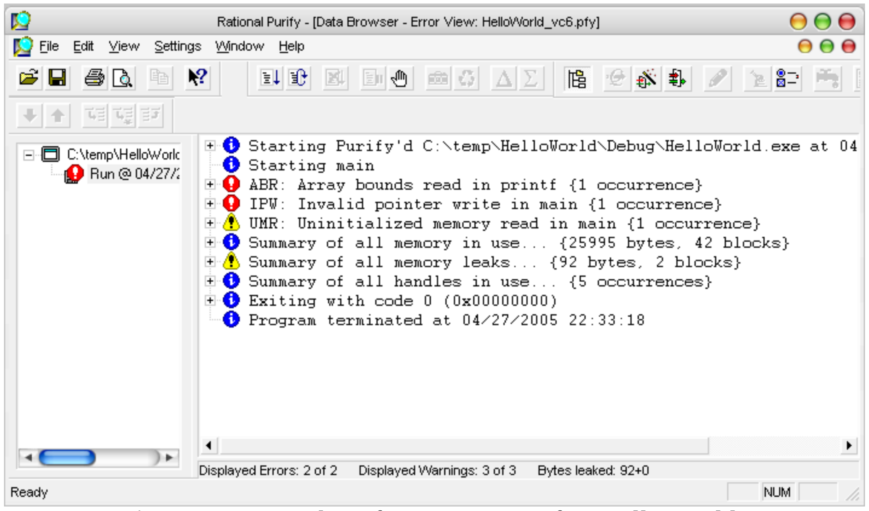
The main test of our experiment was to use Purify to analyze an open source communications application on the Windows platform. We chose Putty, a secure shell client based on OpenSSH, due to its widespread usage.

The sequence of events we chose to test was opening up an SSH connection to a server, perform some operations on the server, and then log out. Figure 3 shows the results captured by Purify. As we can see, there are three main memory leaks that were detected, and no array violations, with a total of 32% line coverage.

We should also mention that although there are more memory leaks than the three listed, those are leaks in dynamically linked libraries against which Putty is linked. These need to be tracked down, but this is difficult, as we don't have the source code for the third-party libraries. This might also be an indication of misused library interfaces.

After delving into the source code, we tracked down the three memory leaks. We should first mention that Putty wraps up the memory management functions malloc, realloc, and free with the functions safemalloc, saferealloc, and sfree. This is for the purposes of being able to switch the memory management routines with a compile-time switch. However, this adds another level of indirection for memory allocations and potentially obscures the source of errors.

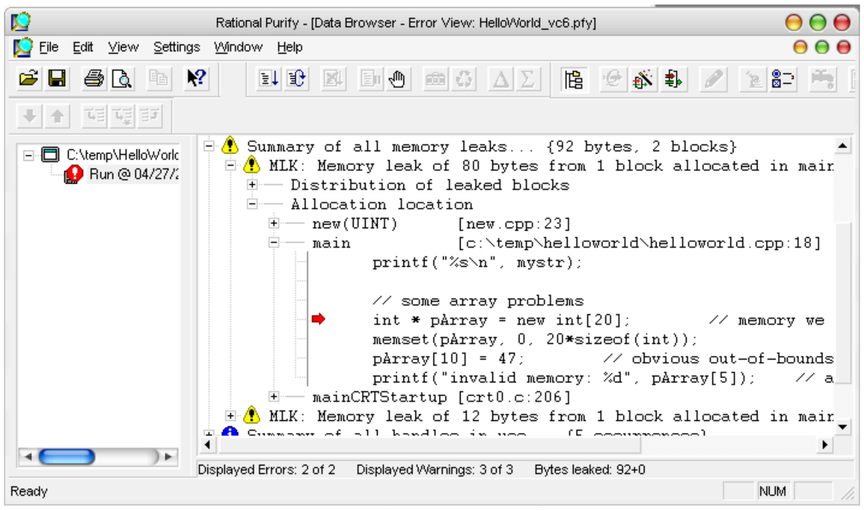
In all three memory leaks, the wrapper functions allocated memory on the behalf of another function, which then initialized a field of a data structure, which was in turn copied and returned from another method. It becomes very unclear as to who is responsible for freeing the memory. There are several references to the same data structure, and if it is deleted too soon, other references then become invalid. This is a poor design, and is a good candidate to be reengineered in C++, where it is easier to encapsulate data and avoid this problem.


Rational Purify - [Data Browser - Error View: HelloWorld_vc6.pfy]
File Edit View Settings Window Help
C:\temp\HelloWorlc
Run @ 04/27/:
Ready
NUM

```
Starting Purify'd C:\temp\HelloWorld\Debug\HelloWorld.exe at 04
Starting main
ABR: Array bounds read in printf {1 occurrence}
IPW: Invalid pointer write in main {1 occurrence}
UMR: Uninitialized memory read in main {1 occurrence}
Summary of all memory in use... {25995 bytes, 42 blocks}
Summary of all memory leaks... {92 bytes, 2 blocks}
Summary of all handles in use... {5 occurrences}
Exiting with code 0 (0x00000000)
Program terminated at 04/27/2005 22:33:18
```

Displayed Errors: 2 of 2   Displayed Warnings: 3 of 3   Bytes leaked: 92+0

Figure 1: Screenshot of error summary for “Hello World”

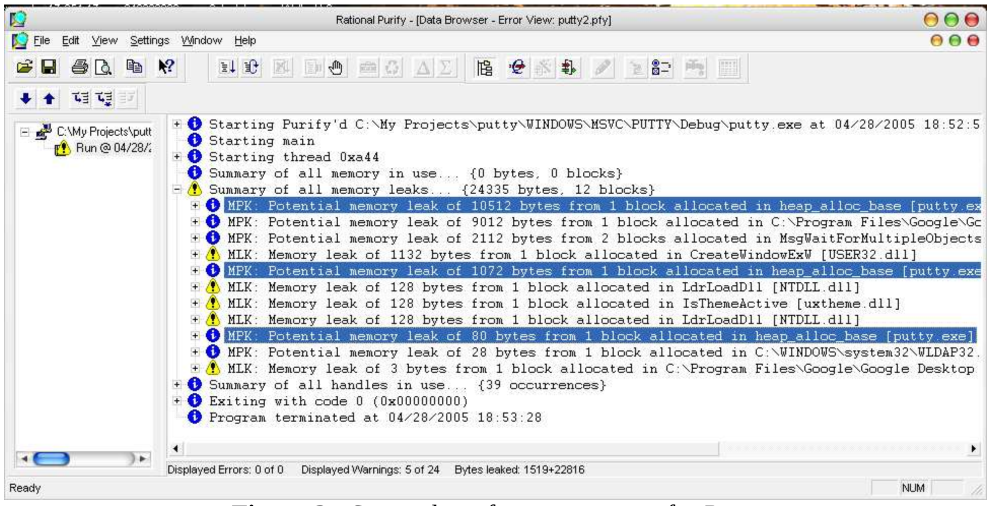

Rational Purify - [Data Browser - Error View: HelloWorld_vc6.pfy]
File Edit View Settings Window Help
C:\temp\HelloWorlc
Run @ 04/27/:
Ready
NUM

```
Summary of all memory leaks... {92 bytes, 2 blocks}
  MLK: Memory leak of 80 bytes from 1 block allocated in mair
    Distribution of leaked blocks
    Allocation location
      new(UINT)      [new.cpp:23]
      main           [c:\temp\helloworld\helloworld.cpp:18]
              printf("%s\n", mystr);

              // some array problems
              int * pArray = new int[20];        // memory we
              memset(pArray, 0, 20*sizeof(int));
              pArray[10] = 47;          // obvious out-of-bounds
              printf("invalid memory: %d", pArray[5]);     // a
      mainCRTStartup [crt0.c:206]
  MLK: Memory leak of 12 bytes from 1 block allocated in mair
```

Displayed Errors: 2 of 2   Displayed Warnings: 3 of 3   Bytes leaked: 92+0

Figure 2: Screenshot of error details for “Hello World”


Rational Purify - [Data Browser - Error View: putty2.pfy]
File Edit View Settings Window Help
C:\My Projects\putt
Run @ 04/28/

```
Starting Purify'd C:\My Projects\putty\WINDOWS\MSVC\PUTTY\Debug\putty.exe at 04/28/2005 18:52:5
Starting main
Starting thread 0xa44
Summary of all memory in use... {0 bytes, 0 blocks}
Summary of all memory leaks... {24335 bytes, 12 blocks}
  MPK: Potential memory leak of 10512 bytes from 1 block allocated in heap_alloc_base [putty.ex
  MPK: Potential memory leak of 9012 bytes from 1 block allocated in C:\Program Files\Google\Gc
  MPK: Potential memory leak of 2112 bytes from 2 blocks allocated in MsgWaitForMultipleObjects
  MLK: Memory leak of 1132 bytes from 1 block allocated in CreateWindowExW [USER32.dll]
  MPK: Potential memory leak of 1072 bytes from 1 block allocated in heap_alloc_base [putty.exe
  MLK: Memory leak of 128 bytes from 1 block allocated in LdrLoadDll [NTDLL.dll]
  MLK: Memory leak of 128 bytes from 1 block allocated in IsThemeActive [uxtheme.dll]
  MLK: Memory leak of 128 bytes from 1 block allocated in LdrLoadDll [NTDLL.dll]
  MPK: Potential memory leak of 80 bytes from 1 block allocated in heap_alloc_base [putty.exe]
  MPK: Potential memory leak of 28 bytes from 1 block allocated in C:\WINDOWS\system32\WLDAP32.
  MLK: Memory leak of 3 bytes from 1 block allocated in C:\Program Files\Google\Google Desktop
Summary of all handles in use... {39 occurrences}
Exiting with code 0 (0x00000000)
Program terminated at 04/28/2005 18:53:28
```

```
Displayed Errors: 0 of 0    Displayed Warnings: 5 of 24    Bytes leaked: 1519+22816
Ready                                                                              NUM
```

Figure 3: Screenshot of error summary for Putty

6 Difficulties

6.1 Difficulties with Linux

As was mentioned above, the difficulties we encountered with the Linux version caused us to abandon our original experiment plan.

6.2 Outdated support information

In an effort to save our initial Linux experiment plan, we attempted to contact Rational technical support via an email address listed in the product documentation. However, the email address was outdated, presumably on account of IBM's purchase of Rational. Additionally, our efforts to contact IBM's technical support were in vain. It was also difficult to find any updates or support information on IBM's website.

6.3 Problem with Visual Studio .Net

After shifting our experiment plan to test the Windows version, we compiled the "Hello World" example with Visual Studio .Net, Visual C++ 7.0. However, when we attempted to test the executable with Purify, it failed to find any of the memory violations we inserted into the code. We failed to determine the root cause of the problem, but our hypothesis was that VC7 was performing some of its own optimization or instrumentation, which got in the way of Purify's OCI process. This caused us to revert to Visual C++ 6.0, which worked fine.

6.4 Problem Building Putty Source

We finally got the "Hello World" example to run and we were able to find the inserted errors. We then proceeded to test the Putty application. However, we ran into a linker problem and were initially unable to build the source code. We tracked the problem down to an incompatibility with the IPv6 library that was installed with VC7 and the library that was expected by Putty. We were able to link to the correct version after using a compile time #define in the project settings.

6.5 Problem with Incremental Builds

We finally got Purify to run on the Putty source code. However, Purify was displaying some strange errors in addition to the detected memory leaks:

```
Pure: Trap bits found in live
chunk at 0x27ab28b,
flags=0x2000, handle=0x0
```

Of course, this made absolutely no sense to us at first. In the help file, it listed this as an internal exception thrown from within Purify. There were also several possible causes listed, one of which was a problem with incremental builds causing the debug information to get out of date. After turning off incremental builds, everything worked fine.

6.6 Obscure errors

The multiple errors we encountered while installing the Linux version, setting up the license server, and testing incremental builds could have been resolved much more easily had the system displayed more intuitive errors. One important usability principle is to give users enough information to fix problems themselves. However, Purify fails to address this concern in the manners described above. This definitely leaves some room for improvement.

6.7 Lack of documentation

Although there was ample documentation for general usage of Purify, we found a lack of support documentation in the application or on the Internet for tracking down obscure errors.

7 Strengths and Weaknesses

In summary, here are the strengths and weakness of Purify that we determined:

7.1 Strengths:

- Integration into IDE
- Doesn't require modifications to source code
- Source code isn't required
- Results are easy to interpret
- Offending source code is easy to find
- Works well with code coverage tool
- If errors are displayed, they are guaranteed to be valid errors.

7.2 Weaknesses:

- Poor product support
- Obscure errors with no description or reasoning behind them.
- Versioning support on Linux
- There are several critical software defects lending to usability problems. However, workarounds were found.
- Unsound analysis. However, this can be augmented with a code coverage tool for more accurate results.

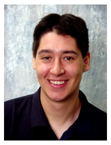
8 Conclusion

Although we had some difficulties with setting up Rational Purify, once it was running the tool provided us with immediate results without having to modify the source code. The user interface displayed the exact error and location of the memory violations, which made it easy to track the problems down. Although it uses unsound analysis methods, it can be coupled with a code coverage tool like Rational PureCoverage for more comprehensive test results. Purify should be a part of any test suite in which memory leaks could potentially be a problem.

9 References

[1] "Getting Started With Rational Purify Plus", Rational Software Corporation, 2002.

10 Biography



Mike Hagen has worked as a software developer for Day-Timer Technologies and as a programmer/ analyst for the State of Alaska. His technical skills include programming in C/C++ and C#/.Net, developing web solutions, interfacing with SQL Server databases; he has experience working on both Windows and Unix platforms. Mike graduated from Stanford University with a Bachelor of Science degree in Electrical Engineering with an emphasis in computer hardware design.

Zhen Zhang has been previously employed at Unisys and Motorola Research and Design Center in China. His main expertise is in C on the SCO Unix platform. Zhen earned his Bachelor of Science degree in Computer Science at the Beijing University of Aeronautics and Astronautics.

11 Appendix

11.1 Error report for Linux version

Runtime Error of checking "Hello_world":

```
[zhangz@msepc6 example]$ purify -g cc hello_world.o

Purify or PureCoverage engine: Warning: Unrecognized option '-g'
ignored.
Purify 2003a.06.13 Linux (32-bit) (c) Copyright IBM Corp. 1992, 2004
All rights reserved.
Instrumenting: hello_world.o libgcc_s.so.1
Purify engine: While processing file /lib/libgcc_s.so.1:
Warning: Non-ABI conforming section found in object file
/lib/libgcc_s.so.1.
Type code is 0x6ffffff7.  Treating section as uninterpreted data.
collect2: parse.c:4342: ProcessSpecialSymbols: Assertion `offset <
((((((ofile->dyn_got_block_info)->block))->block_ofile-
>elf_sec_headers+(((ofile->dyn_got_block_info)->block))-
>elf_block_index)->sh_size))' failed.

Purify engine 2003a.06.13  got signal 6 (SIGABRT - Aborted)
 sigcode=-6; pc=0x5ef7a2; sp=0xbfff4f18; addr=0x32a4
Please contact IBM Support at www.ibm.com/software/support.
```

11.2 "Hello World" source code

```
#include <stdio.h>
#include <string.h>

static char *helloWorld = "Hello, World";

int main(int argc, char* argv[])
{
      char *mystr = (char *) new(char[strlen(helloWorld)]);

      strncpy(mystr, helloWorld, 12);
      printf("%s\n", mystr);

      int * pArray = new int[20];
      pArray[27] = 42;
      printf("memory: %d", pArray[5]);

      return 0;
}
```

11.3 Putty source code

11.3.1 First memory leak:

[file: C:\My Projects\putty\ssh.c, line: 7632, function: ssh_init]

```
    const char *p;
    Ssh ssh;

->  ssh = snew(struct ssh_tag);
```

```
    ssh->cfg = *cfg;                 /* STRUCTURE COPY */
    ssh->version = 0;                /* when not ready yet */
    ssh->s = NULL;
```

11.3.2 Second memory leak:

[file: c:\my projects\putty\windows\winucs.c, line: 541, function: init_ucs]

```
            continue;
          if (!ucsdata->uni_tbl) {
->          ucsdata->uni_tbl = snewn(256, char *);
            memset(ucsdata->uni_tbl, 0, 256 * sizeof(char *));
          }
```

11.3.3 Third memory leak:

[file: c:\my projects\putty\settings.c, line: 778, function: get_sesslist]

```
                if (bufsize < buflen + len) {
                  bufsize = buflen + len + 2048;
->                list->buffer = sresize(list->buffer, bufsize, char);
                }
                strcpy(list->buffer + buflen, otherbuf);
```