

# A Survey of Static and Dynamic Analyzer Tools

Paul DiPalma and  
Elise Hewett  
CSci 780 –  
Advanced Software  
Engineering

12/15/2003

1

## Outline

- Why are software analyzer tools not always used?
- Background
  - RBD basics
  - C Bugs
- Our approach
- The tools
- Comparing the tools
- Critique of the experiment
- Conclusions

12/15/2003

2

## The Problem

- Programs are buggy
- Manual inspection, "print" statements, debuggers are not always effective and are time consuming
- There is a lack of information on which software analyzer tools are the most helpful

12/15/2003

3

## RBD Basics

- Reliability Block Diagrams:
  - Can be represented as a directed, acyclic graph
  - The RBD components are the vertices in the graph
  - Each component has a reliability from 0.0-1.0 which is the probability of that component not failing

12/15/2003

4

## Approach

- Apply four analyzer tools to RBD solver programs written in C and C++
- Compare several aspects of the tools including:
  - Ease of installation
  - Ease of use
  - Error reporting
  - Extras

12/15/2003

5

## The Test Programs

- 5 Reliability Block Diagram (RBD) solvers coded in C and C++

Program ID	LOAC	Language
Student 1	318	C++
Student 2	722	C++
Student 3	318	C
Student 4	552	C
Formal RBD	823	C++

12/15/2003

6

## The Tools

Tool	Static/ Dynamic	Languages Supported	Freeware/ Commercial
Purify	Dynamic	C/C++	Commercial
Valgrind	Dynamic	C/C++	Freeware
Efence/ MemWatch	Dynamic	C	Freeware
Splint	Static	C	Freeware

12/15/2003

7

## Static vs. Dynamic

- **Static Tools:**  
analyze programs without running them
  - Analysis is possible before a program is compilable
  - No test suite is necessary
  - May not find every memory error
- **Dynamic Tools:**  
analyze the properties of a running program
  - Must be compiled before analysis
  - Usually dependent upon a test suite
  - Finding every memory error dependent upon test suite

12/15/2003

8

## Easy Installation

- Valgrind: quick and easy installation in 6 steps
  - Helpful manual
- Splint: not difficult installation—download, unzip, un-tar, configure, and make
  - Very good instructions in the manual

12/15/2003

9

## Not So Easy Installation

- Purify: straightforward but time consuming
  - Must navigate menus to ensure correct version is downloaded to be installed

12/15/2003

10

## Difficult Installation

- ElectricFence and MemWatch: a pain to install
  - We gave up and used our version already installed on our network
  - must download, unzip, and un-tar both the ElectricFence and MemWatch files
  - The efence library must be compiled and linked to the C compiler
  - Poor instructions

12/15/2003

11

## Easy to Use

- Valgrind: very easy to use
  - No linking to object files
  - `host% valgrind executable`
- Purify: fairly easy to use
  - Must be linked to every object file that is compiled and analyzed
  - `host% purify [compiler] -g [obj file]`
  - `host% ./executable`
  - GUI allows easy organization of errors

12/15/2003

12

# Purify GUI

When you run your programs, click the New Leaks tool to generate a new leaks summary while the program is running

The memory-leaked summary reports 12 bytes of leaked memory

The call chain shows how the leaked memory was allocated

Memory analysis by category

```

Purify: a.out
-----
Finished a.out ( 1 error, 12 leaked bytes)
Purify instrumented a.out (pid 870) at Fri Aug 9 16:22:57 2003
ABC: error bounds read
Current file descriptor in use: 9
Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
HEK: 12 bytes leaked at 0x44200
This memory was allocated from:
malloc (rtlib.o)
start (core0.o)
Purify Heap Analysis (Condensing suppressed and unexpressed blocks)
-----
Potentially Leaked 0 0
Leaked 12 0
InUse 0 0
-----
Total Allocated 1 12
Program exited with status code 1.
    
```

# Easy to Use

- Splint: very easy to use
  - Knowledge of annotations is not compulsory
  - Knowledge of annotations allows full functionality
  - Good documentation
  - `host% splint filename.c`

# Not so Easy to Use

- ElectricFence/MemWatch: not user friendly
  - Copy the MemWatch files into the directory where program is compiled
  - Add the "-lefence" flag to the compile line
  - Either:
    - Post-mortem debug the core dump file created when the program is run
    - Or use a debugger on the executable

# Good Error Reporting: Purify

- \*\* the program crashed before any results given
- # types of errors / # total errors
- # leaked bytes

	Purify input 2
Student 1	2/2 0
Student 2	8/2285 384
Student 3	** **
Student 4	36/148 461
Formal	0/0 0

# Purify Example

- **Purify's Message:**

```

UMR: Uninitialized memory read
* This is occurring while in:
strlen [rtlib.o]
vfprintf [libc.so.6]
get_line [zhili-rbd.c:41]
main [zhili-rbd.c:89]
    
```
  - **The C code where the error occurred:**

```

int size=0;
char *str=malloc(size);
while(ch != '\n') {
    str=realloc(str, size*sizeof(char)+1);
    sprintf(str, "%s%c", str, ch);
    ch=getchar();
} // end of code segment
    
```
- The error occurs in sprintf(), "str"

# Good Error Reporting: Valgrind

- \*\* the program crashed before any results given
- # types of errors / # total errors
- # definitely lost bytes / # unfree'd bytes

	Valgrind: input 2
Student 1	2/2 0/640
Student 2	4/2020 384/4960
Student 3	** **
Student 4	35/148 0/148
Formal	0/0 0/48K

## Valgrind Example

### Valgrind's Message:

```
== Conditional jump depends on uninitialized value(s)
== at 0x804A522: main (rbd.cc:579)
== by 0x4032CA46: __libc_start_main (in /lib/libc-2.3.2.so)
== by 0x80489D0: ??? (start.S:81)
```

### Code where error occurred

```
streamsize n=256;
cin.getline(aline, n);
for(j=0; j<n && !pfound; j++){
    if(aline[j]!='#'){
        . . .
    }
}
The possibly uninitialized value is aline[j].
```

12/15/2003

19

## Good Error Reporting: Splint

# code warnings /

# code warnings after suggested flags and annotations added

# actual warnings which are errors /

# non-redundant actual warnings

### Splint Example

	Splint
Student 3	174/172
318 LOAC	152/61
Student 4	242/194
552 LOAC	179/117

12/15/2003

20

## Splint Example

### Splint's Message:

```
rbd.c:378:60: Fresh storage str not
released before return
rbd.c:376:5: Fresh storage str allocated
```

### The C code where the error occurred:

```
str=realloc(str, size+1);
str=get_line();
if (str == ``EOF'') {
    printf(``Error\n'');
    return(1);
}
```

12/15/2003

21

## Error Reporting ElectricFence/MemWatch

- Segmentation fault occurs on the first error
- If this error is a false positive, code around it
- Unable to fix enough of the errors in the programs to get meaningful results

### False-Positive

```
int size = 0;
char * str = malloc(size);
str = realloc(str, size+1);
```

12/15/2003

22

## Extras

- Splint: static analyzer—no test suite necessary
- Languages supported:
  - C and C++: Valgrind and Purify
- Purify
  - GUI
  - Collapsible error trees

12/15/2003

23

## Critique

- All test programs were small (< 1000 LOAC)
  - Much of our criticism is still applicable to larger programs
- Only five programs used to test
- RBD solvers were a good choice of test program

12/15/2003

24

## Critique

- But...
  - Few of the programs use complex data structures
  - Student programs all contained in one file
- ElectricFence may be fair tool for more advanced users of debuggers
- Note: we are not system administrators

12/15/2003

25

## Conclusions

- Purify was a helpful dynamic tool
- Splint was a helpful static tool
  - Can be used during development and debugging

12/15/2003

26

## References

- [1] "Technical articles and tips: Secure c programming," uRL: <http://developers.sun.com/solaris/articles/secure.html>.
- [2] "Relex\_rbd glossary," uRL: [url=http://www.relexaustralia.com.au/pages/relex\\_rbd\\_glossary.htm](http://www.relexaustralia.com.au/pages/relex_rbd_glossary.htm).
- [3] "Valgrind, an open-source memory debugger for x86-gnu/linux," uRL: <http://developer.kde.org/~sewardj/>.
- [4] "Rational purify: Fast detection of memory leaks and access errors," 2003, uRL: <http://www-140.ibm.com/developerworks/rational/library/811.html>.
- [5] "Splint: Annotation-assisted lightweight static checking secure programming group," uRL: <http://www.splint.org>.
- [6] "Efence(3) - linux man page," uRL: <http://www.die.net/doc/linux/man/man3/efence.3.html>.

12/15/2003

27

## References

- [7] C. Cowan, "Software security for open-source systems," Wirex Communications. IEEE Computer Society, 2003.
- [8] "Mastering linux debugging techniques-key strategies to locate and stomp bugs on linux," uRL: <http://www-106.ibm.com/developerworks/linux/library/l-debug/>.
- [9] D. Coppit, R. R. Painter, and K. J. Sullivan, "Toward a unified semantics and implementation for computational engineering," in *Proceedings of the International Symposium on Software Reliability Engineering*. Denver, Colorado: IEEE, 17–20 Nov. 2003, to appear.
- [10] —, "Shared semantic domains for computational reliability engineering," in *Proceedings of the International Symposium on Software Reliability Engineering*. Denver, CO: IEEE, 17–20 Nov. 2003, pp. 168–180.

12/15/2003

28

Thank you

Questions?

12/15/2003

29