

# A Survey of Static and Dynamic Analyzer Tools

Elise Hewett  
Dept. of Computer Science  
The College of William & Mary  
Williamsburg, VA 23185  
enhewe@cs.wm.edu

Paul DiPalma  
Dept. of Computer Science  
The College of William & Mary  
Williamsburg, VA 23185  
rainman@cs.wm.edu

**Abstract**—Traditional methods of debugging programs, such as manual inspection and testing, have limited effectiveness in preventing memory leaks, null pointers, dangerous aliasing, and the use of constructs that may introduce security weaknesses. Software automated analysis and debugging tools have been developed which hope to address this problem; however, the effectiveness of using these tools in practice, and their relative benefits and costs are not widely known. In this research, we quantitatively compared the analysis tools’ abilities to either statically or dynamically find bugs by examining how many bugs each tool finds. In addition, qualitative analysis was performed on several aspects of each tool. We compared the difficulty in learning to use the tool, the ease of use of the user interface, the helpfulness of the tools’ error messages, and whether the tool could be applied to more than one programming language.

## I. INTRODUCTION

Debugging a complex program is a nontrivial task, particularly when errors result from the incorrect management of dynamically allocated memory and the use of complicated data structures. The most commonly used methods of debugging programs are manual inspection and testing. These methods are very time consuming and often do not detect all of the errors such as memory leaks, null pointers, dangerous aliasing, and the use of constructs that may introduce security weaknesses that they intend to find. We turn to automated software analysis and debugging tools to make this process more effective and efficient. The problem is that the better automated analyzer tools are not used as much as they might be if programmers knew that using them would be a more efficient use of time and that more bugs would be found than through the aforementioned traditional methods. This paper describes our comparison of four static and dynamic software analyzer tools which specialize in finding memory errors in C and/or C++ programs after they were each applied to a few Reliability Block Diagram (RBD) solver programs. Since an RBD solver relies on complex data structures and, depending on the input diagram, is forced to manipulate the data in complex ways, it will make a suitable test case for the tools at hand. A helpful tool should find bugs that would inhibit the RBD solver from correctly performing its analysis.

Software static and dynamic analyzer tools are programs that are applied to software systems to look for errors or security vulnerabilities in the code. The use of such software analyzer tools is not frequently encouraged in educational settings when programmers are beginning to develop their

programming habits. System designers in industrial settings may be searching for an analyzer tool that is a better fit for the types of code they develop.

Simpler forms of testing and debugging continue to be the most widely used methods of validating programs. The problem with testing lies in the fact that few large programs can truly be adequately tested. An unanticipated path of execution not included in the testing suite could cause the system to fail. Manually checking programs for errors is tedious and time-consuming. Additionally, because programmers have poor visibility into the execution of their program, errors often go unnoticed. Particularly in the case of an extremely large system, a brute force manual search for all bugs is simply impractical, if not impossible.

In this research, we have compiled a list of tools that effectively find bugs and security problems that exist in C/C++ programs, including those which may never be found through manual debugging techniques. The four tools that we surveyed are Purify, Splint, Valgrind, and MemWatch/ElectricFence. Splint was found to be a helpful static analyzer tool and Purify was an effective dynamic analyzer tool. The contribution of this study is to educate programmers of the cost/time benefit of using the tools in the debugging phase of software development.

The rest of the paper is structured as follows. Section II gives the reader some background on problems inherent in traditional forms of debugging, as well as a well-known memory security vulnerability found in C programming. The tools are introduced as well as the fundamental concepts of the RBD solvers projects. Section III outlines our research and describes the methods used in analyzing the tools for purposes of comparison. Section IV contains the results from the experiments. Our analysis of the tools and approach to the problem follow in Section V. The paper concludes in Section VI with our final remarks on the results of our research.

## II. BACKGROUND

### A. Debugging Techniques

The most primitive, yet sometimes effective, way that programmers debug their software is through simple “print” statements. The programmer can insert these output statements, by hand, at trigger points. These points are places where he/she believes that there may be an error. Although extremely time

consuming, inexperienced programmers may think this is the best or only method available.

A step up from manual output statement analysis are simple debuggers, such as “gdb” or “ddd.” In its basic functionality, gdb allows the user to see certain elements (the values that variables have been assigned, print statements, memory locations of variables, etc.) of the program while it is running. A programmer can also use ddd’s graphical interface, which uses the same debugging methods and provides the same debugging information as gdb. This creates a step-by-step process that is slowed down to an analytical pace so that a programmer can view possible problems and find the exact line of code where segmentation faults occur. This method replaces the tedious placement of output statements as described above, but still requires the user to do most of the analysis of where a problem or bug might originate. Additionally, the use of gdb and ddd requires dynamic analysis which means that the programmer must already have code that compiles, as well as, a good test suite to find any problems.

### B. Secure Programming in C

A harmful bug, seen in the C programming language, is a buffer flow error. This can happen if a programmer is not careful to check the inputs and outputs, as well as the memory allocation routines of the program. A buffer overflow occurs when programs try to store more data in a variable than it has been allocated space for. A simple example is the use of the C function `gets(var)` instead of `fgets(var, size, stdin)`. The first use of the `gets` function allows for a malicious user to input a lengthy string, longer than `size` characters, that could build on the stack the instructions needed to start a shell and gain root command of the machine [1].

We have analyzed several debugging tools that look for the problem described above, as well as other memory problems. These bugs must be found and fixed at all costs, but this cost can be reduced with the use of software analysis tools.

### C. Basic Concepts of RBDs

A Reliability Block Diagram (RBD) is a visual representation of a system that contains redundancy in its elements. Redundancy is used for mission critical functions where a single-point failure is not acceptable and reliability needs to be improved [2]. The basic elements of an RBD are parts of the system in parallel, series, series-parallel, and even a special case where *k-out-of-N* elements must work.

Simple RBDs are generally represented in a directed, acyclic graph and their reliabilities can be calculated using analytical solutions. Results may include reliability, availability, failure rate, and mean time before failure. The goal for an RBD is to improve the field reliability of the system. An RBD solver will simulate the graphical representation of the system and calculate the reliability,  $R$ , of the given system, among other statistics. The solver acts on a set of rules and formulas that are inherent in calculating reliability.

The rules for calculating the reliability are as follows:



Fig. 1. An example of a series reliability block system

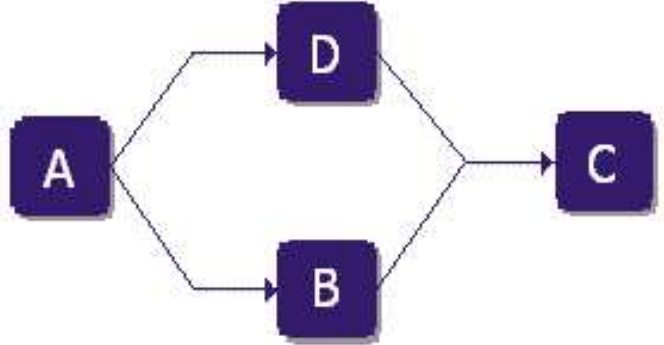


Fig. 2. An example of a parallel reliability block system

- 1) Series Components: If the parts are in a series configuration, the reliability of this series of parts is simply the multiplication of the reliability of each component in the series, so

$$R = \prod_{i=1}^n (R_i)$$

where  $R_i, i = 1, \dots, n$  is the reliability of each of  $n$  individual components in the system.

- 2) Parallel Components: If the parts are in a parallel configuration, the reliability of this parallel grouping would be equal to one minus the multiplication of one minus the reliability of each component, so,

$$R = 1 - \prod_{i=1}^n (1 - R_i)$$

where  $R_i, i = 1, \dots, n$  is the reliability of each of  $n$  individual components in the system.

These two rules form the basis for calculating the reliability of systems where series and parallel elements are intertwined. One further extension of an RBD takes into account a *k-out-of-N* reliability factor in which  $k$  out of the  $N$  elements in parallel must work at the same time for the system to work.

Since the rules for calculating the reliability for an RBD are mathematically based on probabilities, we can see how formal methods for creating a solver can be instituted. We will take a brief look at formal methods in the next section.

### D. Formal Programming Methods

Formal methods are defined as: “Mathematically based techniques for the specification, development and verification of software and hardware systems. They are “mathematically

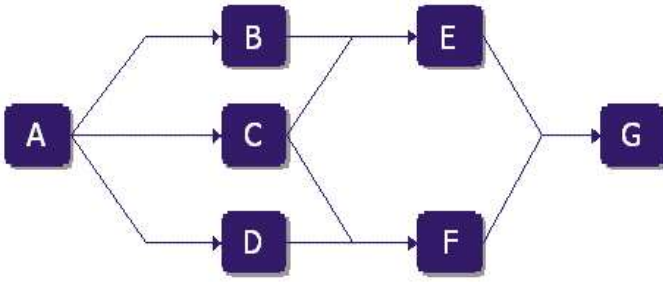


Fig. 3. An example of a series-parallel reliability block system

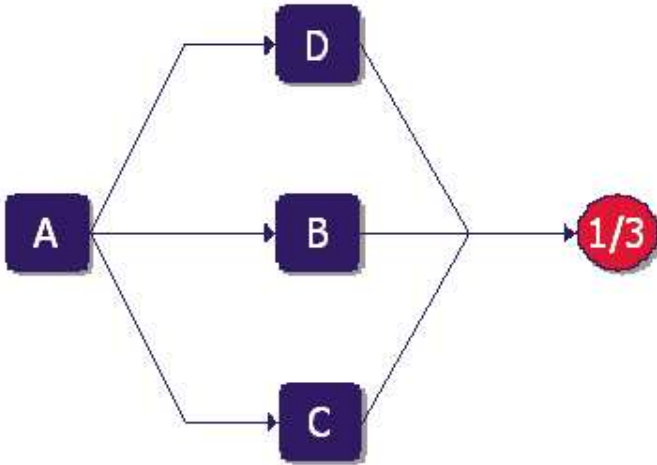


Fig. 4. An example of a  $k$ -out-of- $N$  reliability block system

precise notations, tools, and techniques used for the development of software.”

Formal methods are not usually followed by programmers working under a deadline and this can often cause the introduction of unexpected bugs which eventually leads to increased debugging time and effort. The developers of the “control program” rely on formal methods to develop a bug-free version of an RBD solver.

Now that we have taken a brief look at the RBD model, formal methods, secure programming, and debugging techniques, we will present the tools we used to attempt to solve the problem of finding bugs and leaks in both the informally and formally developed RBD solvers.

### E. The Tools

Several freely available tools have been chosen for experimentation and evaluation: Valgrind, Purify, Splint, and ElectricFence/MemWatch. These debugging tools aim to find the common bugs associated with memory leaks and security issues that take precious time and effort from a C/C++ programmer on a daily basis. As an introduction to the tools that we used, a short summary describing each tool and the bugs that it specializes in finding are listed below. The descriptions

are derived directly from the tools’ manuals. In rare cases (in this study only ElectricFence did so) do the manuals point out a tool’s shortcomings or situations where another tool would be more helpful.

1) *Valgrind*: Valgrind is a tool that helps find memory-management problems in C/C++ programs. When a program is run using Valgrind, all reads and writes of memory are checked, and calls to `malloc/new/free/delete` are intercepted. Valgrind detects the use of uninitialized memory, reading/writing memory that has been free’d, reading/writing off the end of `malloc’d` blocks, reading/writing inappropriate areas on the stack, memory leaks, passing of uninitialized and/or unaddressable memory to system calls, mismatched uses of `malloc/new/new[]` versus `free/delete/delete[]`, and some misuses of the POSIX pthreads API [3].

2) *Purify*: Purify is a software testing tool engineered by the IBM Rational software development team. Purify is just one part of a larger software suite called the PurifyPlus Family. Purify can be used on both Unix and Windows operating systems and supports the C and C++ languages as well as Java on Sun Solaris systems only. Purify is the only commercial tool included in our study. Purify detects memory leaks and invalid memory accesses to uninitialized memory, reading/writing memory that has been free’d, reading/writing off the end of `malloc’d` blocks, and reading/writing inappropriate areas on the stack. The Purify executables of the program being debugged are still fast enough to use throughout the development and testing of a product [4].

Purify links itself to the object file(s) of the program being tested. This allows it to insert additional checking instructions directly into the object code produced by the compiler. These instructions check every memory read and write performed by the program and detect several types of access errors, such as reading uninitialized memory or writing to free’d memory. In addition, Purify tracks memory usage and identifies individual memory leaks using an array of garbage collection techniques. But this memory access checking slows the target program down typically by “less than a factor of three” [4]. Purify’s visual components, along with its ability to track down memory related errors, are very helpful in sorting through numerous errors after each run.

3) *Splint*: Splint, an extended version of Lint, is a tool for statically checking C programs for security vulnerabilities and programming mistakes. Some of these mistakes were memory related problems, which we found useful in our survey. Some of these checks include: unused declarations; variable use before definition; unreachable code; ignored return values; execution paths with no return; likely infinite loops; fall through cases; dereferencing a possibly null pointer; using possibly undefined storage; type mismatches; violations of information hiding; memory management errors including uses of dangling references and memory leaks; dangerous aliasing; modifications and global variable uses that are inconsistent with specified interfaces; and buffer overflow vulnerabilities [5].

More powerful checks are made possible by additional information given in source code annotations. Annotations are

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

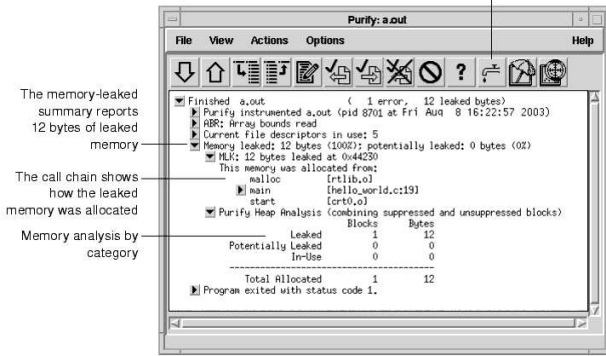


Fig. 5. Purify screen shot - Memory Leaks

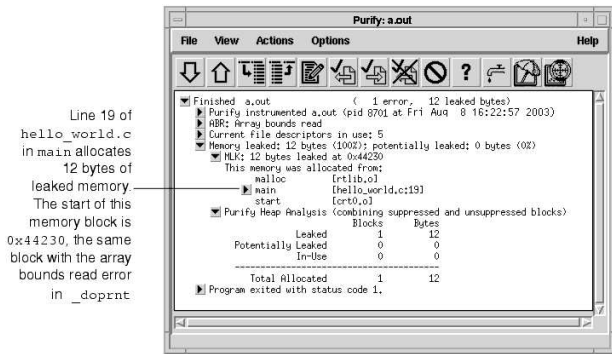


Fig. 6. Purify screen shot - correcting MLK error

stylized comments that document assumptions about functions, variables, parameters and types [5]. As more effort is put into annotating a program, Splint is able to perform better error checking. Splint is designed to be flexible and allow programmers to select appropriate points on the effort-benefit curve for particular projects. As different checks are turned on and more information is given in code annotations, the number and types of bugs that can be detected increase dramatically [5].

4) *ElectricFence and MemWatch*: ElectricFence and MemWatch, which are compiled alongside the program being debugged, help detect two common programming bugs: software that overruns the boundaries of a `malloc()` memory allocation, and software that touches a memory allocation that has been released by `free()`. The MemWatch files are used by ElectricFence to create a core dump whenever the program

being debugged is run. By adding the MemWatch files to the directory containing the source code being debugged and by adding the ElectricFence flag “-lefence” to the gcc compile statement, a programmer can track memory leaks and corruptions in a C program.

ElectricFence detects both read and write accesses and pinpoints the exact instruction that causes an error. It is most easily used with a debugger (e.g. gdb or ddd) to detect where such memory violations occur [6], [7]. Although this tool gives good feedback by reporting the line number where it detects an error, it halts after the first suspected error. Efforts to code around ElectricFence’s error reporting may not be worth the programmer’s efforts since a particular reported error may in fact be a false-positive.

ElectricFence’s and MemWatch’s strong points are that they provide a “results” log and detect memory errors including double-frees, erroneous frees, un-free’d memory, overflow and underflow, and others [7], [8]. It is interesting to note that the ElectricFence manual pages recommend using Purify since Purify is more thorough in its checking for memory errors and has a much smaller memory overhead than ElectricFence.

### III. DESCRIPTION AND EXPERIMENTATION PLAN

Devising a comprehensive list of development tools is aided by two crucial comparisons. First, the differences should be noted in the quality of a formally developed relational block diagram solver versus the quality of the informally developed RBD solver. Second, we should note the performance of the tools on the formally developed RBD solver versus the informally developed versions. The testing of a formal RBD solver will provide a good basis for comparison among the different tools.

#### A. Formally Developed Versus Informally Developed RBDs

We expected the RBD solver with a formal structure to be a more solid program than that of the inexperienced students; however, we took into account what these differences were and how they affected the programs’ performance and reliability.

The structure of the formal method for developing the Reliability Block Diagram solver implemented by two engineers, whose program we will be using in our test plan, can be found in [9]. This version of the RBD solver will act as the “control” program against which to compare the relative quality and structure of the informally developed student versions of the RBD solver. The definitions and concepts of formal methods were summarized in the background section of this paper.

Since we anticipated that the formally developed RBD was more carefully designed and tested, we expected an effective tool to reflect this. A particular tool should have found fewer errors in the formally structured than in the informally structured RBD solvers. If more errors are found in the informal RBD solver, this would indicate that the tool is effective. In a relative comparison between tools, another tool may even have found a few errors in the formal structure that were not found by the previous tool. As a result, this tool was rated higher.

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

TABLE I  
RBD SOLVER PROGRAM SUMMARY

### B. Pretesting Milestones

Several milestones were completed in leading up to the actual testing phase of our research endeavor. First, we acquired several test subjects on which to run the analyzer tools. These RBD solvers served as a basis for our comparisons of the tools. Second, in analyzing the various tools, we acquired, installed, and learned to use as many features of the tools as possible within a reasonable amount of time before applying them to the different RBD solvers that we gathered.

1) *Acquisition of Test Programs:* Since the comparison of analyzer tools depends on sample programs, our first step involved acquiring the RBD solver test subjects. Six students from the Fall 2003 Stochastic Modeling class volunteered their RBD solvers as test subjects. In the case of these RBD solvers, time was a factor and the primary goal of the project was to have a working version, or close to working version by a specified date. The formally developed RBD solver [10] was used for comparison purposes. For this version of an RBD solver, more time allowed for a formal specification to be developed and eventually led to a solid implementation of a RBD solver.

Refer to Table 1 for a summary of the test programs. In this table, “LOAC” is the “lines of actual code”, which is the number of lines of code excluding blank lines and comment lines. Of the six student projects, only four were found to be suitable for testing. Each of the four student projects contains a single file with three-hundred to eight-hundred lines of code, with two written in C and two written in the C++ language. The formally developed RBD solver is written in C++. One student program was removed due to the language barrier, as it was written in Java. Since Java has garbage collection, this language would not be suitable for our memory leak analysis. Also, since this was the only program that was written in Java, there is little means for comparison among tools on programs of similar types. The other program that was not used was removed due to critical errors in the program itself, in which the immediate core dump errors would not allow our dynamic tools to give a complete analysis.

Since these various tools are language specific and can only evaluate a program written in a certain language, we focused on testing RBD solvers that were written in the C or C++ language. Even though these RBD solver programs cover two languages, the effectiveness of the tools provided comparable results.

2) *Acquisition and Installation of Analyzer Tools:* Next we found several tools and, if necessary, installed them on

our system. We acquired Purify, Valgrind, ITS4, Splint, and MemWatch and ElectricFence. With the exception of Purify, these tools are all freeware that claim to find memory related bugs. We obtained a trial version of the commercial tool, Purify, for this research.

We found that ITS4 is not a suitable tool to be used in our analysis, because its goal is analysis of security issues in C and C++ programs, which are not frequent problems in the RBD solvers. We focused on tools that find memory errors instead of security weaknesses.

3) *Learning to Use the Analyzer Tools:* A good deal of the startup cost involved learning how to use the tools. We read manuals, manual pages, and/or README files about the tools. Each tool has several options that can be changed when analyzing a program. Deciphering which options would be useful, or if none were useful at all, took a considerable amount of time. For example, Splint’s manual is over eighty pages long, and there are many references on how to write good annotations in code that is to be analyzed. For ElectricFence and MemWatch, it is necessary to be skilled in the use of a debugger which required reading documentation on and learning how to use ddd and gdb. Additionally, the documentation for ElectricFence and MemWatch was vague in its instructions for how to apply the tool to a test program.

4) *Start-up Costs of Pretesting Milestones:* There were significant startup factors that we became aware of when installing and learning to use the tools. Some tools were very easy to install or were already installed for us through our network. Others, due mainly to poor documentation, were more difficult to install and use.

Splint was already installed on our network so setup time was minimal. The majority of the startup time for Splint was taken up by reading through the documentation for Splint. However, the manual and manual pages for Splint made learning to use the tool and applying to the RBD programs very easy.

Valgrind is freely available and was quickly found and downloaded from the developer’s website. Installation was fairly straight forward as we were guided by its README file. Valgrind is an extensive tool, so again, sifting through the documentation took more time than the installation.

We had some trouble finding and installing Purify, as it is only available on partial license through IBM and had to create a temporary user account through IBM in order to download it. The trial period ends in fifteen days.

ElectricFence and MemWatch are used together to dynamically analyze programs. The ElectricFence library is installed and linked through our network but the MemWatch files (which you must download yourself) were not. MemWatch is a set of only three compilable C files that is freely available through the developer’s site. Neither MemWatch nor ElectricFence require installation as the ElectricFence library is linked at compile time with any given program to be analyzed. Setup time for MemWatch and ElectricFence was higher due to the significant confusion that resulted from the hard to find manual pages and unclear documentation. For use with

ElectricFence and MemWatch, the gdb and ddd debuggers were already available on our network. Since ElectricFence only outputs a core dump file when a program that it has been compiled with is run, the use and knowledge of a debugger is necessary. Learning some debugger basics took a good deal of reading and time.

### C. Evaluation Plan

We evaluated the tools with respect to several attributes including ease of installation, ease of use, quality of the error messages, and extras. We tested four tools that are found on the open market including Valgrind, Purify, Splint, and ElectricFence and MemWatch. The tools are designed specifically to debug software and promise to find various bugs in any given software project, usually with specific languages in mind. To validate and rate the effectiveness of a given tool, we have applied them to Reliability Block Diagram (RBD) solvers.

Included in the ease of installation area, we considered factors such as disk space, operating system, development language, and budget. Some tools do not work on all operating systems, or across all programming languages. In the area of ease of use, we considered the learning curve associated with each tool. An easy to read manual along with an easy to learn tool is optimal. Once the tool was learned, we considered how well would it work in everyday use. The most important aspect of an analyzer tool is how effective it is at identifying where the errors are and how to fix them. The quality of the error messages played a large role in our rating of the effectiveness of each tool. In the extra features of a tool, we included the programming languages supported and other additional features such as a graphical user interface.

## IV. EXPERIMENTATION AND DATA

After completing the installation and learning phases for the four analyzer tools, we applied the tools to the RBD solver test subjects. The analyzer tools ElectricFence/MemWatch and Splint can only analyze C programs while Purify and Valgrind can analyze both C and C++ programs.

Two of the four student programs were written in C, and could be tested by all four analysis tools. The remaining two student programs and the formally developed RBD solver program, written in C++, could only be tested by Purify and Valgrind. Another variance that we had to be aware of is that the only static tool that we used was Splint, while the other four tools are dynamic. Dynamic tools link to an object file, so it requires that a program be compiled before any analysis can be done. Table 2 contains a summary of the variations of each tool mentioned above.

We must keep in mind that the effectiveness of the dynamic tools depends on the test cases used. A different test case results in different actions being performed by the RBD solver. Bugs are only found if the test cases provoke them.

For each dynamic tool, a small test suite was used to try to provoke different outputs from the analyzer tool used. This test suite consisted of four different Reliability Block Diagrams,

Tool	Static/Dynamic	Languages
Purify	Dynamic	C/C++
Valgrind	Dynamic	C/C++
E-Fence/MemWatch	Dynamic	C
Splint	Static	C

TABLE II  
ANALYZER TOOL SUMMARY

---

```
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'
```

---

Listing 1. Example of an error found by Purify for Student 3

two basic and two more complex diagrams. No test suite was necessary for the only static tool, Splint.

### A. Results

The output from each tool gives us a good idea of its overall usefulness. In most cases, there is a clear distinction in the quality of the outputs. In the listings in this section, we show a sample of one of the types of errors found by Purify, Valgrind, and Splint in some of the test subjects. These examples show the format of the output given by each tool. Some tools give several lines of output pointing the user to line numbers in the code and supplementing with suggested corrections for each error, while others do not give enough information to be helpful. These examples gave us a basis for comparing the helpfulness of each tool in finding where the problem could be.

As another method of comparison, we kept track of some statistics for each program. Table III gives a relative comparison of each tool's analysis for each test subject. The column content of the chart is as follows. For each box, the top line gives a count of how many types of errors followed by how many total errors there were. For example, when Purify was run on Student 2's program using the first test case, there were eight distinct errors with 2787 total occurrences of these eight

---

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

---

Listing 2. Example of an error found by Valgrind for Student 2

---

```
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=get_line();
if (str == '\EOF') {
    printf('\Error: enter valid node(s) amount!\n');
    return(1);
}
```

---

Listing 3. Example of an error found by Splint for Student 4

types. In the second line of data, we report how many leaked bytes were seen, and in the case of the Valgrind output, how many bytes are still reachable, i.e. not free'd. Since Splint is the only static analyzer tool, different statistics were kept. For the Splint output, the first number is the number of reported code warnings and the second number is how many code warnings were generated after the flags were added for the initially reported warnings that were not actual errors. On the second line of the Splint output, the first number is the how many of actual errors that were reported. The second number is how many of the actual errors are not redundant. We define a code warning to be *redundant* when the same error has also been reported by a previous code warning. For example, Splint reports a warning every time that a variable is used which has been allocated dynamic memory that never gets free'd.

For any data that was unavailable or could not be produced, we use a sequence of stars. The star sequence is as follows: one star for when the tool could not analyze the program due to a language barrier and two stars when the program crashed on a particular test case before any output could be seen.

Although we used four test cases for each program in the dynamic analysis, here we will only provide the statistics from the first two test cases. To explain the column headers in Table III, the column header "Pur1" refers to the first test case used with Purify, "Val1" to the first test case used with Valgrind, etc. The rest of the column headers are self-explanatory.

Since ElectricFence and MemWatch stop on every error, they are not included in this table. We were unable to fix enough of the errors in the programs being tested with ElectricFence (while continuing to ensure that the program still computed the correct reliability ) to produce meaningful quantitative results.

## V. EVALUATION/DISCUSSION

In comparing the four analyzer tools, first, we briefly discuss the pros and cons of each tool, which will highlight some of the things that we noticed while installing, learning, and using them.

### A. Ease of Installation

1) *Purify*: An important detail about Purify is that, at 60 MB, it is a large program. The installation of Purify is a frustrating process for those who only want to use the trial version. A user must navigate through several menus in order to get the right version installed and then must make sure that

errs leak	Pur1	Pur2	Val1	Val2	Splint
Sdnt 1	1/1 0	2/2 0	1/1 0/640	2/2 0/640	* *
Sdnt 2	8/2787 96	8/2285 384	4/2522 96/5360	4/2020 384/4960	* *
Sdnt 3	19/119 148	** **	18/104 123	** **	174/172 152/61
Sdnt 4	35/173 484	36/148 461	34/173 171	35/148 148	242/194 179/117
Formal	0/0 0	0/0 0	0/0 0/21K	0/0 0/48K	* *

TABLE III  
DATA

the trial license is copied to the correct directory before using Purify. These are just some little nuances that trial users should be aware of.

2) *Valgrind*: The process of installing Valgrind is spelled out nicely in the manual. It takes five or six short steps to get Valgrind up and running, with most of the time being taken up by making sure that environment variables are set up correctly. Valgrind comes packages in a tar.bzzip file, which after being decompressed contains the standard linux install file. These files total a trivial 710kb, and make for a quick install.

3) *Splint*: Installing Splint was straightforward and the developer's webpage has detailed instructions on each step. A C compiler and the Unix tools gzip and tar are required. This was a very fast installation.

4) *ElectricFence and MemWatch*: Installing ElectricFence and MemWatch was a tedious task which we eventually gave-up on and used the version that was already installed on our network. A C compiler and the Unix tools gzip and tar are required. Both the ElectricFence and MemWatch files had to be downloaded. The installation instructions were not clear. The MemWatch files can either be copied into any directory where a user wants to run ElectricFence or linked through PATH variables and were easy to set-up. ElectricFence was the bigger problem in that the efence library needs to be compiled and linked through the user's C compiler.

### B. Ease of Use

1) *Purify*: With respect to ease of use, Purify would have scored higher if it did not require the user to manually link it to every object file in the program being tested. An extra compilation step is required, as object files must first be created. Then Purify must be called, along with the compiler being used, for the final compilation step. This may require some manipulation of a make file. But aside from some confusion associated with multiple object files, Purify is very easy to use once it is linked. The GUI makes for a simple way to sort multiple errors of the same type. Information about where the error has occurred can be shown and hidden, on demand, due to the collapsible tree structure that is used.

2) *Valgrind*: Valgrind does not require any linking to the object files. This is a definite plus for the freeware Valgrind.

All that is required to use this tool is to put "valgrind", plus any options that you would like use, in front of any executable. This allows it to work for both C and C++ files. In order to take full advantage of Valgrind, the program to be tested should be compiled with the -g option.

3) *Splint*: Though we initially suspected that Splint would be difficult to use because of the need to learn the proper annotations to insert and command line flags to set, Splint was, by far, the easiest tool to use. Clear manual pages and documentation facilitated the ease of use. Splint can even be used while code is being developed since it is not necessary for code to compile to be able to apply Splint to it. Once this tool is installed it works just like a compiler. It parses any C file and it reports any warnings that it finds in the code.

4) *ElectricFence* and *MemWatch*: ElectricFence and MemWatch proved difficult to use and do not seem to be worth the time you must invest to figure out how to use them and how to modify the code to the satisfaction of the tool. The manual pages and documentation do not provide clear instructions on how to use this tool. Compiling with ElectricFence is quite straightforward. The user simply copies the three MemWatch files into the directory where the program is being compiled and compiles with:

```
gcc program.c -lefence
```

Linking ElectricFence can cause a program's running time to significantly increase since for each call to malloc or free, the cache or translation buffer entries are flushed [6]. Running an executable which contains errors will cause a segmentation fault and produce a huge core dump at the first error that ElectricFence finds. This will be a problem for users who have limited disk space and it is difficult and time consuming to debug it post-mortem.

### C. Error Reporting

1) *Purify*: After Purify is linked to the object file(s) of a program, a Purify window pops up when the program is run. Screenshots of the Purify pop-up window are seen in Figures 5 and 6. As the program runs, Purify keeps track of the memory used and reports any errors found until your program completes. Inside the window are collapsible trees, with each one storing every detail about the errors found. Others are keeping track of the memory used, as well as one that keeps track of the memory leak summary. This collapsible tree format proved to be helpful in sorting the many errors that were found in some of the test subject RBD solvers. When the program completes, a user can manipulate the Purify window and expand the tree with a particular error in it. The tree contains the type of error, the line number, and function where the error has occurred.

Although, Purify works very well with detecting and pointing out errors, there are a few drawbacks. Since Purify needs to be linked to every object file during compilation time, it requires an extra step in compiling a given program. The object file has to be created from the compiler, then linked to Purify for final compilation. This linking step greatly increases the

---

#### Valgrind's Message:

```
"More than 30000 total errors detected. I'm not
reporting any more.
==2270== Final error counts will be inaccurate.
==2270== Go fix your program!
==2270== Rerun with --error-limit=no to disable
==2270== this cutoff. Note that errors may occur
==2270== in your program without prior warning from
==2270== Valgrind, because errors are no longer
==2270== being displayed.
Segmentation fault (core dumped)".
```

---

Listing 4. Valgrind example: Too many errors

compilation time. If a Makefile is used in compiling a program with multiple files, careful manipulation is required to make sure that an object file is created and then followed by a Purify call for each object. In the formally developed RBD solver, the compilation time doubled as there were six additional Purify calls added to the make file.

2) *Valgrind*: Valgrind finds many of the same errors that Purify does, but it does not sort each error in a collapsible format. The output is simply sent to the terminal window as the program runs. Unfortunately, since Valgrind does not link to the object files it does not include any line numbers in the output where a particular error has occurred. If the program being analyzed is not compile with the "-g" option, Valgrind only provides the name of the function where the error has occurred, which function in Valgrind caused the error (i.e. malloc()), and which type of error has occurred. One unavoidable problem is found when the program being analyzed gets caught in an infinite loop. Valgrind will continue to run along side of the program collecting data until it runs out of memory. There is a built-in cutoff of thirty-thousand errors, at which point an interesting error message is sent to the screen. This example is found in Listing 6.

3) *Splint*: Splint actually outputs suggested annotations and command line flags whenever it reports a code warning. Code warnings are reported with line numbers, character positions, and the actual code that caused the warning to be generated. This makes it simple for the user to find exactly where the error might be, right down to the exact variable, in cases where there are multiple variables or commands on one line. Besides just pointing to the errors, it outputs a message stating what the error actually is, or it tries to. A negative aspect of Splint is that it reports a massive number of code warning messages. Some of these code warnings can be easily suppressed by flags and annotations. Splint gives the user the suggested annotations or flags to use when it reports the code warning. It is not necessary to learn to write annotations in order to use Splint. Unfortunately, some of the code warning messages are confusing and do not give the user any idea on how to fix the error that was reported. In some cases the warning that was reported is not actually an error. Listing 4 contains an example of a false-positive code warning.

Many of the warnings that Splint reported were redundant warnings. For example, if malloc'd memory is not free'd, then each time the variable that dynamic memory has been malloc'd

---

```
Splint's Message:
  rbd.c:282:16: Operands of <$ have incompatible types
    (int, long int): i <$ size exp

The C code where the error occurred:
  long int size_exp=1;
  int i;
  for(i=0;i<size_exp;i++){
    ...
  }
```

---

Listing 5. Splint example: a false-positive code warning found for Student 3

---

```
Splint's Message:
rbd.c:41:21: Possibly null storage str passed as non-null
  param: \ sprintf (... , str)
  A possibly null pointer is passed as a parameter
  corresponding to a formal parameter with no /*@null@*/
  annotation.  IF NULL may be used for this parameter,
  add a /*@null@*/ annotation to the function parameter
  declaration. (Use -nullpass to inhibit warning)
  rbd.c:40:6: Storage str may become null
```

---

Listing 6. Splint example: a suggested annotation for Student 3

for is used, Splint reports a code warning. Refer to Listing 3 for an example of an actual error that Splint found. In one of our test programs we found that 16 identical warnings were reported for a single case where malloc'd memory had not been free'd. Additionally, Splint sometimes reports several different warnings for the same line of code. Splint identifies the presence of an error but is unable to clearly define the cause of the problem. In this case, Splint reports several warnings in hope that one will lead the user to the actual error.

While error checking would be greatly improved by inserting better annotations, Splint still performed adequately without this extra effort. Keeping in mind that we have only taken advantage of Splint's most basic functionalities, users with harsher penalties for buggy code do need to know how to write annotations in order to use Splint's full error checking capabilities. An example of a simple annotation which was suggested by Splint is found in Listing 5.

4) *ElectricFence and MemWatch*: Although running a program that has been compiled with ElectricFence will create a huge core dump at the first error that is found by ElectricFence, the ElectricFence manuals suggest the use of a debugger directly on the compiled executable. A debugger can be used to quickly find the line where the segmentation fault originated. It is then left up to the user to determine exactly what cause the error in that line. In order to use ElectricFence and MemWatch, a user must be quite familiar with a debugger or be an apt post-mortem debugger. In order to determine why ElectricFence causes the segmentation faults to occur, the user must be intimately familiar with how C handles dynamic memory and the malloc and free functions. No suggestions for how to handle problems are given to the user. Additionally, ElectricFence halts after the first error that it finds. In order to find subsequent errors, the previous ones must be fixed. One other point to note is that some of the errors that ElectricFence reports are not actual errors. The user must

---

```
ElectricFence's Message:
  int size = 0;
  char *str=malloc(size);
  while(1)
  str=realloc(str, size*sizeof(char)+1);
```

*ElectricFence does not allow malloc'ing 0 bytes even when this does not cause an error.*

---

Listing 7. ElectricFence example: a false-positive error found for Student 4

then manipulate the code to satisfy ElectricFence so that the debugging can continue. These manipulations can obfuscate previously elegant code. See Listing 7 for an example of an ElectricFence report of a false-positive error.

Although it is clear from the example in Listing 7 that str will not ever have a size of 0 bytes after it is realloc'd, ElectricFence caused a segmentation fault. Many errors of this nature being reported caused us to abandon our efforts to "correct" the code. Therefore, actual counts of the number of errors that were reported are not available for the two projects to which we applied ElectricFence/MemWatch. Although testing time for ElectricFence and MemWatch was far higher than any of the other analyzer tools, far less debugging was accomplished.

#### D. Extras

1) *Purify*: Purify supports both C and C++ on Unix machines and Java on Sun's Sparc machines. Purify also provides an easy to use graphical user interface for debugging. The errors are reported in collapsible trees which allows for easy viewing and organization of the errors. One significant negative aspect for Purify is that since it is a dynamic analyzer tool, errors are found based on the program's execution path. Therefore, if a program does not execute a particular piece of code during a run, some errors may never be found. Finding every memory leak error may require an extensive test suite in order to execute every possible path through the program's code.

2) *Valgrind*: Both C and C++ are supported by Valgrind on Unix machines. Like Purify, Valgrind is a dynamic analyzer tool and also has the negative aspect that an extensive test suite is needed to find every possible memory error. In order to find every error in the program, the test suite must execute every possible path of execution. Creating a test suite that does this would explode testing time for large programs and in most cases is simply impractical.

3) *Splint*: Splint would be invaluable if it supported multiple languages. Splint only supports programs written in C and almost seems more like an extension to a C compiler. Since we had three RBDs that were written in C++, Splint could not be used to analyze any of these programs. Because Splint is a static analyzer tool, it is not necessary for code to compile to be able analyze it with Splint. Splint evaluates every line of C code in a program that is analyzes. Splint also performs compiler-like activities and halts when it finds a basic compiler error.

4) *ElectricFence and MemWatch*: ElectricFence is a dynamic analyzer tool and code must be compiled before it can be analyzed with ElectricFence and MemWatch. However, because the executable file that has been compiled with ElectricFence can be debugged with a debugger, no extensive test suite is necessary. Additionally, ElectricFence halts after every error. If a false-positive is reported, the programmer must find a way to modify the code and force ElectricFence to continue in order to analyze the rest of the program.

#### E. Overall Evaluation

Overall, however, we found Purify to be a more useful product than Valgrind and ElectricFence and MemWatch. Purify may not be the best product for everyone, but its usefulness in finding memory related errors and helping a programmer correct them is by far superior to Valgrind and ElectricFence and MemWatch. Splint provided very good error checking and suggestions for how to correct the errors.

#### F. Criticism of the Experiment

Our experiment shows four tools' most basic functionality when applied to five fairly small programs. Although the RBD solvers serving as test programs were small, many aspects of our criticisms of the tools are scalable to larger programs (e.g. the fact that Valgrind just dumps its output to the terminal window or that Splint produces a large number of errors with respect to the size of the program). The RBD solver programs were a good choice for test cases since they all used a good amount of dynamically allocated memory and our analyzer tools specialized in finding memory errors. Since the RBD solver programs have similar functionality and uses of memory resources, we have only been able to apply the analyzer programs to a very small subset of potential errors in code. For example, only one of the C++ student programs and the formally developed RBD solver actually used any kind of data structure. This may be significant if tools do not extend well to follow the line of execution over class boundaries. We were unable to learn enough about how to use each tool and then become skilled in using it to overcome the huge learning curve that needed to take place before the use became really efficient. Additionally, a deeper understanding of C and C++ functionality is required in order to correct some errors or to know if they are errors in the first place. Because extensive knowledge of a debugger is necessary to easily use ElectricFence, we have categorized ElectricFence as "inefficient" and "hard to use." For a programmer who is very knowledgeable about memory errors and debuggers, ElectricFence may be a good choice.

We note that neither researcher is a system administrator so we have limited experience with installation. The more limited installation information provided by some tools made the installation process very difficult.

## VI. CONCLUSION

In search of an analyzer tool that will help you bullet-proof your software, finding a tool, or set of tools, tailored to the

specific needs of the project is likely to make debugging tasks much easier. In the tools that we have surveyed throughout this paper, we have described and tested static and dynamic tools that span different languages, as well as one tool that analyzes software in the compilation phase, rather than in the runtime phase. Each developer may have different interests to consider in completing their projects. One should consider factors such as disk space, operating system, development language, and budget.

We recommend the use of a static analyzer tool along with a dynamic analyzer tool. We found Purify to be the best dynamic analyzer tool, if the budget allows, while the static analyzer tool, Splint, was also very helpful. These two tools used together, along with an extensive test suite for Purify, will surely benefit any software developer, and will decrease the time of the debugging phase.

## ACKNOWLEDGMENTS

We would like to express our appreciation to the people at IBM, the Rational group, for allowing us to use their Purify trial version for our research. We would also like to acknowledge the developers of the freeware that has been made for the good of the software community for debugging error-ridden programs. The freeware, as well as the developers are: Valgrind, authored by Julian Seward; Splint, lead by David Evans and appended by David Larochelle and others; MemWatch, authored by Johan Lindh; and ElectricFence, developed and maintained by Bruce Perens. Finally, we would like to thank those who have volunteered their RBD solvers to be used as test subjects for evaluating the analyzer tools. The formally developed RBD solver was provided by Robert Painter and David Coppit.

## 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>.
- [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.