Workshop on Dependability Benchmarking, DSN-2002. June 23rd-26th, 2002, Washington, D.C., USA.

The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines

Michael W. Bigrigg
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh PA 15217
bigrigg@ices.cmu.edu

Jacob J. Vos
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh PA 15217
jvos@ece.cmu.edu

Abstract

A methodology is presented that will detect robustness failures in source code where I/O errors could occur and where there is no mechanism in place to handle the error. The details of the methodology are described showing how traditional compiler data flow analysis can be augmented to find structurally, within the application, code that can be used to perform error checking. In addition we describe how this code can be used to ensure the correctness of the I/O error checking.

1. Introduction

File systems routinely make extraordinary attempts on behalf of the application to provide data whenever possible to the user. Yet, problems such as network congestion or outages and heavily loaded systems can lead to failure-like situations, making it impossible for the file system to complete the entire requested operation. These situations are usually only transient and still enable the file system to provide a partial result. An example of a true failure condition for a file system is a request for data where no data is available, i.e., a read past the end of a file. This is different from a situation in which data is available, but is currently not accessible, such as when using a disconnected mobile device. When applications not intended for an unreliable environment are ported from a desktop environment to a wireless system, the application programmer must account for all such unpredictable behavior. As programmers, we often overlook error checking when we are overwhelmed with the task of identifying all possible error situations, or neglect checking in the belief that some errors are inconceivable.

Local file system interfaces are typically identical to those of a distributed file system, though the potential for failure at each is greatly different. In a local file system, failures are catastrophic. If the hard drive or other local storage device fails, it often signals the end of the device's usable lifetime. Failures in distributed file systems are more common and it is possible to recover from them. They are usually the result of unreachable remote storage devices or device overloading due to network partitioning, poor load balancing, or denial of service attacks. Users have fundamental, but not often expressed assumptions about the reliability of the system an application is built for. Yet unhandled error conditions lead to potential software failures when the underlying system cannot satisfy our requests and the application was built assuming that it can.

We present a methodology based upon program static analysis to track the propagation of error reporting in order to determine the assumptions used when the software was created.

2. Software Fault Detection Related Work

There are many approaches to using program analysis for the detection of software faults. These systems are typically aimed at providing information to the programmer in order that the program source code may be modified to eliminate software faults. In identifying code errors, there are strict guidelines regarding right and wrong within an application, i.e. dealing with the disabling and re-enabling interrupts, or the assumptions about integer size. Our method does not establish the correctness of code, instead establishing the existence of code that will ensure correctness.

Errors in an I/O system can only be identified at runtime and only after checking the status of the I/O call. One type of fault analysis techniques will run the entire program or a subset of the program to observe its behavior. As well, controlled errors can be introduced to examine how the software behaves by passing external faults into the application that cause it to fail. Such approaches include fault injection through random memory corruption or corruption of the storage system (FlakyIO) [1], passing values typically known to cause exceptions into an individual software module through its software interface (Ballista) [5], and the creation and use of a comprehensive test suite. In particular, this type of approach makes it possible to identify the type of input or condition that has led to the fault, but does not identify any remedial action that should be taken by the application.

Compile-time analysis attempts to identify program features that would cause a program to behave improperly. The analysis focuses on a particular characteristic that is typically the base cause of faults such as portability problems (i.e., when moving an application from one machine architecture to another (lint) [4]). We, however, are attempting to find portability problems, not between architectures, but between systems that have different reliability guarantees on their file system. Other approaches, such as LcLint [3] and mc [2] use programmer-defined rules that specify acceptable behavior to drive the analysis. These two systems are the most closely related projects to ours in method, but their purpose is to capture the assumptions about a program in order to establish correctness, while our focus is to uncover the original assumptions made about a program.

We present a methodology that will detect robustness failures in source code where I/O errors could occur and where there is no mechanism in place to handle the error. Many programmers fail to incorporate error checking in specific classes of I/O operations and rely on certain assumptions such as "file output is always guaranteed" to ensure correct application operation. It is this absence of error checking that we intend to detect with our methodology. Our approach to uncovering these situations combines an augmented data flow analysis with the semantics of the I/O error reporting.

We describe the details of the methodology showing how traditional compiler data flow analysis can be used to find structurally, within the application, code that can be used to perform error checking. In addition we describe how this code can be used to ensure the correctness of the error checking.

3. Error Reporting in C I/O Routines

Errors are reported in C I/O routines using out-of-range values. The return values of these routines are either a useful result (upon successful completion of the call) or an indication of the error that occurred (upon an unsuccessful completion). For instance, the successful return of the `fopen` call is a handle to a file. The range of values for a file handle is an unsigned integer greater than zero. A zero, also referred to as NULL, is then used to report that the file system was unable to open the file. The return of a `fread` call uses out-of-range values to transmit not only an error condition, but also specifies an end of file condition as well. The return identifies the number of bytes that have actually been read. The fread call, like all data buffer operations, will read up to but no more than the number of bytes that have been requested. A return of zero does not signify an error condition, just that no data is currently accessible such as at the end of a file. It is a negative return value that signifies an error condition. Since a single value can potentially be both an error condition and also a valid result, it is not until tested that we know. Just like Schroedinger's cat, we cannot tell what the value is until it is examined. When writing a program, we have to assume that both outcomes are likely and cannot assume one or the other.

The values that specify an error condition are based on the I/O routine itself. An examination of the C standard I/O library [8] shows the behavior of I/O function calls upon an error condition:

- Functions that return pointers use a NULL to designate an error condition: `fmpfile`, `fopen`, `freopen`, `fgets`.
- Functions that use EOF as an error condition: `fclose`, `fgetc`, `getchar`, `putchar`, `puts`, `ungetc`.
- Functions that use a non-zero for an error condition: `remove`, `rename`.
- Functions that use a negative number for an error condition: `fputs`, `fgetpos`, `fseek`, `fsetpos`, `fprintf`, `fscanf`, `print`, `sprintf`, `sscanf`, `vfprintf`, `vsprintf`, `fputc`, `fputs`, `gets`, `putc`, `fread`, `fwrite`.
- Functions that use a -1 for an error condition: `ftell`.

Only the data buffer operations (`fprintf`, `fscanf`, `print`, `sprintf`, `sscanf`, `vfprintf`, `vsprintf`, `fputc`, `fputs`, `gets`, `putc`, `fread`, and `fwrite`) overload the return with three potential values.

In addition, we must identify the result value. The result is the value achieved upon successful completion of the call and may be passed through a return or through an argument. The buffer operations have not only the result in the return but also an argument (a buffer), which is also a result. Not only is it important to distinguish the error from the result in the return, but also it is important to acknowledge the error before using the buffer contents. Therefore, error checking must occur before the use of any result values.

4. Identification of Error Checking

Correct error checking associated with an I/O routine must occur between the *set* (called a definition or simply *def*) of the potential error value and *use* of a result value or values along all possible paths of execution. For instance, the C code example in Figure 1 could lead to a program crash, while the code example in Figure 2 uses program logic to safeguard against a possible error condition.

```
fin = fopen("foo","r");
fread (buf, sizeof(int), 10, fin);
```

Figure 1. Code that may lead to a failure

```
fin = fopen("foo","r");
if (fin != NULL) {
    fread(fin, sizeof(int), 10, buf);
}
```

Figure 2. Program logic guards against a possible unsuccessful result

We augment traditional data flow analysis to identify missing error checking. Data flow analysis is a traditional technique used by compilers during the optimization phase as a tool to guarantee the correctness of program transformations. Value chains, called *def-use chains*, are identified between the definition of a value and the places the value is used. Data analysis is performed on values and not on variables. Figure 3 shows how a value chain is formed, dependent on the instance of a value in a variable, rather than on the name of the variable.

```
a =3; /* def of a_1 */
b =a +5; /* use of a_1 , def of b_1 */
a =8; /* def of a_2 */
```

Figure 3. Formation of a value chain

We augment the def-use chains to additionally include the *check* of a value. We define a *check* as a *use* of a value that additionally falls within the expression of a conditional statement. There is already a large body of work on the mechanisms for computing def-use chains [7]. The conditional is a guard against incorrect usage of the result value. The conditional expression that acts as an error guard may be part of any conditional structure including *if* and *if-else* statements as well as *while* and *repeat* loops as shown in Figure 4.

```
n = fread (fin, sizeof(int), 1, buf);
while (n > 0) {
    k += buf[0];
    n = fread (buf, sizeof(int), 1, fin);
}
```

Figure 4. Formation of a value chain

While *set-check-use* is a straightforward approach, there are a few issues to incorporate into our methodology. Error values and the result values are not bound to a specific variable as shown in Figure 5. These values can be assigned to other variables or even modified. In these cases, we need to track the values to make sure that the *use* of the result values does not occur before the *check* of the error values.

```
a = fopen ("foo.txt","r");
b = a;
if (b != NULL) {
    n = fread(buf, sizeof(int), 1, a);
}
```

Figure 5. Analysis based on values not variables.lue chain

It is also important to note that the use of the result value need not exist only within the body of the conditional, and that the conditional may be used to reset the result variable as shown in Figure 6. Again this involves tracking the values through all execution paths. Once the tracked value is overridden with another value, the tracking of the previous value stops along that path of execution.

```
a = fopen ("foo.txt", "r");
if (a == NULL) {
    a = stdin;
}
```

Figure 6. Resetting a value for protection

We know that there is a check, but that does not mean that the expression will accurately identify an error situation.

Finally, in order to determine if the check is valid we must examine the conditional expression. This will be explained in the remainder of this section.

Another aspect to detecting missing robustness checks is the use of error information from the language, as outlined in the previous section, to guide the *set-check-use* approach by determining which value identifies the error. The error propagation information provides a heuristic approach similar to error classification schemes [6]. An example is given to show how semantic information would drive the *def-check-use* analysis. In the case of file opening as shown in Figure 7, the def, check, and use locations use the same value for the analysis. Between the *def* and *use* of *a*, there should be a *check* of *a*.

```
a = fopen("foo","r"); /* def of a */
if (a != NULL) /* check of a */
  fread (buf, sizeof(int), 10, fin); /*use of a */
```

Figure 7. *Def-Check-Use* of the same value

In an `fopen` call, the return holds the error value. A NULL return value designates an error condition. The *def*, *check*, and *use* is to use the same value, *a*, which is the value returned from the `fopen` call. We acknowledge that it may not be possible to statistically determine the validity of the check.

5. Analysis of a Simple Program

The *wc* program is part of the GNU textutils collection of programs. Its purpose is to count the number of lines, words, and characters in a file or files identified on the command line. The v2.0 program consists of 371 text lines with 118 lines of code. It can be identified to have been in use for the past 16 years (from 1985 to 2001). It was written in C and consists of four functions. There are no instances of control flow issues where error checking only exists in a subset of execution paths. A hand analysis of the main source program was performed using the methodology presented to detect failure and to check for error conditions. The results are summarized in Table 1.

Table 1. Hand Analysis of the *wc* program

Routine	Total	Checked	Unchecked
fprintf	1	0	1
Printf	7	0	7
Puts	1	0	2
putchar	1	0	2
fstat	1	1	0
lseek	2	2	0
read	3	3	0
open	1	1	0
close	2	2	0
setlocale	1	0	1
SUMMARY	**20**	**9**	**11**

A simple calculation of the number of checks that should be performed against the number of checks actually produced results in a 45% reliability rating. The usefulness of this rating is not promoted as it does not reflect the frequency of each call, but can be included as a guide to understand the program behavior.

The major assumption that was made by the *wc* program is that all output is guaranteed to succeed.

6. Future Work

The Set-Check-Use Methodology (SCUM) work is part of the PARIS project at CMU, which is attempting to use program analysis techniques to analyze the reliability of programs. We can see the strength of this methodology for determining the reliability of an I/O program, and are in the process of implementing it in a tool that will report on the presence and absence of error checking in programs to construct a rating of reliability. At the same time, we are attempting to classify the missing error checks, and so become able to specify the types of assumptions that are made about the operating environment for I/O applications automatically.

7. Acknowledgements

This work supported by the Pennsylvania Infrastructure Technology Alliance and also as part of the PASIS project supported by DARPA/ISO's Intrusion Tolerant Systems program (Air Force contract number F30602-99-2-0539-AFRL). We would also like to thank Joan Digney for her help in preparing this paper.

8. References

[1] Michael W. Bigrigg and Joseph Slember. Testing the Portability of Desktop Applications to a Networked Embedded System. *Workshop on Reliable Embedded Systems*, Oct. 2001.

[2] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. *Proceedings of the 2000 OSDI Conference*, Oct. 2000.

[3] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. SIGSOFT *Symposium on the Foundations of Software Engineering*, Dec. 1994.

[4] S.C. Johnson. lint, a C Program Checker, *Computer Science Technical Report*, Number 65, 1978.

[5] Philip Koopman. Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. *Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98)*, Nov. 1998.

[6] Roy A. Maxion and Robert T. Olszewski. Improving Software Robustness with Dependability Cases. *28th International Symposium on Fault Tolerant Computing*, June 1998.

[7] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Pub. 1997.

[8] P.J. Plauger. *The Standard C Library*. Prentice Hall, 1992.