ISSN:2249-5789
Dr.D B Singh et al , International Journal of Computer Science & Communication Networks,Vol 2(5), 618-622

Runtime Checking using Static Analysis

Mohd. Ishrat1, Manish Saxena2 and Dr. D.B. Singh3

1 Research Scholar, Singhania University,
Pacheri Bari, Jhujhunu, Rajasthan, India. Pin - 333515
ishratgzp@gmail.com

2 Asst. Professor, MCA Department, FGIET,
Raebareli, UP, India. Pin - 229001
manish.mohan.saxena@gmail.com, URL : www.manishsaxena.in

3 Professor & Dean, CSE Department, Lucknow Modern Institute of Technology & Management,
Lucknow, UP, India. Pin - 226001
dbsinghbbk@gmail.com

Abstract - In this paper we discuss possible uses of static analysis to facilitate runtime checking. In particular, we focus on two categories of uses: static analysis for helping with runtime bounds checking and the general case of using static analysis for helping with code instrumentation.

Keywords : Run Time checking in C, Bound Checking, Static analysis for Instrumentation of Code.

1. Introduction

Runtime checking is a mechanism of a programming language to check for errors at runtime, e.g. arithmetic overflows or invalid type casts. Most times an exception is thrown and/or the program is terminated, instead of ignoring this failure as it is done in C, C++ etc.
Runtime checking is often criticized for slowing down the resulting program. However this ignores the fact that most compilers for languages with runtime checking allow one to switch the checks off when performance is more important. While on the other hand, adding runtime checking to a language that does not support it normally is close to impossible, since those languages don't have the needed language constructs.
Manual checking of code is not enough. For one, it is both error-prone and time consuming, and secondly, humans easily get overwhelmed by complexity. This motivates the need for automation of the checking process. Moreover, the wide variety of bugs found in these systems call for innovative techniques for automated bug detection. Existing techniques can be classified into static methods, model checking, and runtime checking. In this report we study applications of static techniques for automated debugging and analysis of software. In particular, we consider two different ways to apply static analysis to software. First, we present a survey of static checking techniques for automated debugging of OS kernels. Second, we study the use of static analysis and instrumentation for runtime checking of OS kernels.
Static analysis covers all code uniformly; unlike runtime checking techniques, bugs on rarely executed code paths also get discovered. Although model checking techniques share some of these merits over runtime checking, they are rendered effectively intractable for large software systems due to scalability problems and the lack of a formal specification or model of the system.

2. How to do run time checknig in C

To use runtime checking, enable the type of checking you want to use before you run the program.

1. Turning On Memory Use and Memory Leak Checking -
   To turn on memory use and memory leak checking, type:
   (dbx) check – mem use
   When memory use checking or memory leak checking is turned on, the show block command shows the details about the heap block at a given address. The details include the location of the

block's allocation and its size. For more information, see show block Command.

2. Turning On Memory Access Checking-
   To turn on memory access checking only, type:
   (dbx) check -access
3. Turning On All Runtime Checking
   To turn on memory leak, memory use, and memory access checking, type:
   (dbx) check -all
4. Turning Off Runtime Checking
   To turn off runtime checking entirely, type:
   (dbx) uncheck -all

3. Running an program in C

After turning on the types of runtime checking you want, run the program being tested, with or without breakpoints. The program runs normally, but slowly because each memory access is checked for validity just before it occurs. Below is a simple example showing how to turn on memory access and memory use checking for a program called `hello.c`.

```
1.  % cat -n hello.c
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.   #include <string.h>
5.   char *hello1, *hello2;
6.   void
7.   memory_use()
8.   {
9.       hello1 = (char *)malloc(32);
10.      strcpy(hello1, "hello world");
11.      hello2 = (char *)malloc(strlen(hello1)+1);
12.      strcpy(hello2, hello1);
13.  }
14.  void
15.  memory_leak()
16.  {
17.      char *local;
18.      local = (char *)malloc(32);
19.      strcpy(local, "hello world");
20.  }
21.  void
22.  access_error()
23.  {
24.      int i,j;
25.      i = j;
26.  }
27.  int
28.  main()
29.  {
30.      memory_use();
31.      access_error();
32.      memory_leak();
33.      printf("%s\n", hello2);
34.      return 0;
35.  }
36.  % cc -g -o hello hello.c
37.  % dbx -C hello
38.  Reading ld.so.1
39.  Reading librtc.so
40.  Reading libc.so.1
41.  Reading libdl.so.1
42.  (dbx) check -access
43.  access checking - ON
44.  (dbx) check -memuse
45.  memuse checking - ON
46.  (dbx) run Running: hello
47.  (process id 18306)
48.  Enabling Error Checking... done
49.  Read from uninitialized (rui):
50.  Attempting to read 4 bytes at address 0xeffff068
51.  which is 96 bytes above the current stack pointer
52.  Variable is 'j'
53.  Current function is access_error
54.      i = j;
55.  (dbx) cont
```

4. Static Analysis for Bounds Checking

Static analysis has been used for improving bounds checking system in several ways. The general problem of bounds checking can be divided into two subproblems: (1) identifying a given memory reference's referent object and thus its bounds (*bound lookup problem*), and (2) comparing the reference's address with the referent's bounds and raising an exception if the bound is violated (*bound comparison problem*). One of the common uses of static analysis has been used in solving the bound comparison subproblem. The general approach is to eliminate unnecessary checks, so that the number of checks is reduced. These techniques may not be valid in the presence of concurrency.

Gupta [1] used flow analysis techniques to avoid redundant bound checks. Although this technique guarantees to identify any array bound violation, it does not necessarily detect them at the time of the error.

This was done to reduce the overhead associated with bounds checking within a loop. Such bound-checks are moved *outside* the loop so that checking them guarantees prevention of bound overflow errors within the loop body.

ABCD [2] is a light-weight algorithm for eliminating Array Bounds Checks on Demand. Despite its simplicity, ABCD has proved to be quite effective. BOON [3] checks memory errors in using standard string library functions. It formulates memory buffer overruns as integer range analysis problem. Its approach consists of two logical steps: (1) infer possible ranges for array indices, and (2) for each possible input, check using range analysis if any memory access falls beyond an array's limits. Larson and Austin present an interesting extension to this idea. Their method is identical to BOON's approach in step (1). However, for detecting possible inputs in step (2), they argue against using static analysis. In general, any static analysis cannot always determine exactly which paths will be taken at runtime. Hence, BOON's method (which relies on symbolic execution) may produce false positives. Larson and Austin's approach is to dynamically monitor variable values at runtime possibly inferring all possible ranges in which they can fall. Using this information in the range analysis, one can potentially perform well-informed bounds checks.

5. Static Analysis for Instrumentation of code

Static analysis of code can also be useful in the more general case of instrumenting code for tracing support, checking, etc. In this we first discuss *aspect oriented programming*, a generic approach to instrumenting arbitrary code. We then present the compile-time instrumentation framework of **Aristotle**, a runtime verification system we developed.

5.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is an upcoming programming paradigm which allows crosscutting concerns to be modularized as aspects that are separated from objects. Aspects can specify synchronization policies, resource sharing and performance optimizations over objects. A compiler called *weaver*, weaves aspects and objects together into a program.

The AOP paradigm was inspired by the difficulty programmers often face in understanding how the code works because it is often poorly localized, and its relation to the execution flow of the main code is hard to follow. Often, implementation for aspects of the program, like the scheduling policy is spread throughout the entire source code. As a result, understanding the code and maintaining it becomes difficult. Often, changing only a particular aspect of the code requires changing the source code in many different places.

Aspect-oriented approaches attempt to address these problems by modularizing the scattered pieces of source code that implement a single concept.

Aspects are described as specifications in a *spec language*. The language comprises of a series of *patternaction* pairs. The weaver searches for each occurrence of the pattern in the source code, and performs the corresponding action on it. For example, the action could be to insert a function call, or signal an error.

AspectJ is an aspect-oriented framework for Java. It was an offshoot of the MOPS project for separating higher level protocol specifications from source code. The system comprises of an aspect oriented language (AspectJ) for Java and a weaver. Some other frameworks currently under development are AspectC++ [1] and AspectC.

All these approaches use simple pattern matching when searching for the pattern in the target code (C or Java). The only form of flow-sensitive pattern searching supported is using the *cflow* keyword. However, the weaver as such does not perform any flow-sensitive analysis. Any flow-sensitive conditions in the pattern code are evaluated at *runtime*. This can significantly slow down runtime performance for frequently executed code.

The Bossa system [6] is an example of an AOP system that allows the programmer to use flow-sensitive static analysis for specifying patterns. The Bossa spec language supports temporal logic based flow-sensitive pattern specifications. Bossa has been used with the Linux kernel for separating the scheduling framework from the mainstream kernel.

Several other tools exist for code instrumentation. CTool [7] is a simple source transformation tool. ASTlog [8] is a tool for simple pattern matching in parse trees. Tools such as ATOM [9] provide support for binary level translation. Purify [10] performs binary level translation of generic programs for inserting bound checking.

5.2 Source Code Instrumentation in Aristotle

Aristotle is a runtime verification system for the Linux kernel currently under development in our research group. Aristotle continually monitors kernel execution at runtime, checking it for conformance to pre-defined properties. Kernel monitoring is facilitated by tracing events generated by an instrumented kernel. The tracing events are typically function calls made to Aristotle's runtime checking engine as shown in Figure 1.

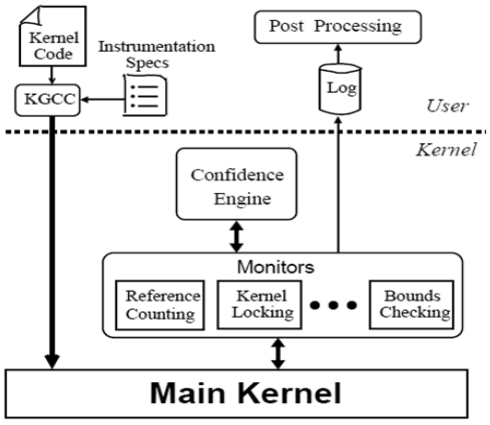

Figure 1: Architectural overview of the Aristotle system. KGCC instruments kernel code with monitoring calls.

During the execution of the instrumented kernel, the monitoring calls invoke individual monitors to dynamically verify system properties.

KGCC is a special compiler used by Aristotle for instrumenting the Linux kernel for runtime monitoring.

KGCC is an extension to GCC.

KGCC is a general source code instrumentation framework based on GCC.

Aristotle inserts custom monitoring code into the Linux kernel using KGCC. KGCC scans the parse tree generated during compilation and instruments it with the monitoring code. Using a compiler for statically instrumenting kernel code offers several advantages. Compile-time instrumentation can benefit from higher level semantic information about the code, e.g. user-defined data types that are typically not available to binary rewriters. Moreover, static analysis can be used to guide the placement of the monitoring code. In this context, Aristotle uses the compiler both as a provider of parse trees, type information, etc., and as a useful tool for instrumenting programs at the parse-tree level, code optimization, etc. KGCC also allows us to separate the monitoring code from the source code, making monitoring code easily portable to different kernel versions in the spirit of aspect-oriented programming.

GCC is a heavy-weight C compiler with little documentation of its internals. Static checking and meta compilation [11] projects in the past have preferred light-weight compilers with well-documented internals over GCC [12], since they are much easier to understand and modify. The Linux kernel, however, is designed to compile exclusively with GCC, forcing Aristotle to execute code compiled with KGCC rather than just use the compiler for its parsing capabilities. These considerations obviate our choice of a GCC-based approach. KGCC was originally based on GCC-3.2 and later ported to GCC-4.0 (beta). GCC-4.0 offers several advantages: a cleaner interface to manipulate parse trees, better documentation which significantly reduced development times, and advanced optimizations.

Like AOP, KGCC's instrumentations are also guided by *pattern–action* pairs. Patterns identify source code constructs of interest, and actions identify the corresponding actions to be performed on the source code. For each such pair, GCC scans the parse tree for every occurrence of *pattern*, and inserts the corresponding *action* in the appropriate parse-tree node. Patterns can be specified using any information typically available in the parse tree: types, identifiers, containing functions, file names, etc. Note that in our current implementation, we do not yet support a generic language for specifying patterns and actions. KGCC currently hard-codes these into the compiler itself. However, our design is easily amenable to an AOP .weaver. like setup.

KGCC actions can be logically divided into two categories: *pre-checks* and *post-ops*. Pre-checks are to be performed before executing the operation identified in the pattern. For example, in the case of memory bounds checking , a pre-check would verify pointer sanity before each pointer dereference.

A post-op is inserted after the specified operation and can be an arbitrary piece of C code (as is the case for bounds checking) or simply a call to the event dispatcher. In the latter case, the operation (event) will be dispatched to the confidence engine and may be logged for post-processing purposes. For example, when reference-counting monitoring is enabled, an operation that manipulates an object's reference count would be dispatched to the reference-counting monitor to check that the operation does not violate a system correctness property.

6. Conclusion

The contributions of this paper are that it has presented a survey of available techniques for run time checking. It have been performed a comprehensive survey of existing run time bound checking techniques and also demonstrated run time checking in C program. It has seen that static checking of C programs is limited in its scope due to computational limits. Although static checkers have been designed for detecting bugs in software, these checkers are often unsound and can miss bugs. Attempts to achieve complete soundness result in loss of precision in the form of a large number of false alarms. These false alarms are a significant practical drawback. However, runtime checking techniques

themselves are not acceptable in production systems due to performance reasons. The central of this paper is that, where correctness guarantees are desired, static analysis should be combined with runtime checking.

REFERENCES

[1] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Language and Systems*, 2(1-4):135.150, 1993.
[2] R. Bodik, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321.333, New York, NY, USA, 2000. ACM Press.
[3] D.Wagner and J. S. Foster and E. A. Brewer and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, CA, February 2000.
[4] G. C. Necula, S. McPeak, and W. Weimer. Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128.139, Portland, OR, January 2002.
[5] R. Jones and P. Kelly. Bounds Checking for C. Technical report. www.ala.doc.ic.ac.uk/phjk/BoundsChecking.html
[6] The Bossa Project. www.emn.fr/x- info/bossa/.
[7] The GNU CTool Project. http://ctool.sourceforge.net.
[8] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, October 1997.
[9] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528.539, 2004.
[10] Rational Software. Rational PurifyPlus. www.rational.com/products/pqc/index.jsp , May 2001.
[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69-82, Berlin, Germany, June 2002.
[12] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley