Compass — A Defect Detection Tool for Large-Scale Software Applications*

Thomas Panas, Chunhua Liao,
Daniel Quinlan, Andreas Saebjornsen,
Jeremiah Willcock
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{panas2,liao6,quinlan1,
saebjornsen1,willcock2}@llnl.gov

Richard Vuduc
Georgia Institute of Technology
Atlanta, GA
richie@cc.gatech.edu

ABSTRACT

In this paper we present Compass, a software defect detection tool for source code (C and C++) and object code. Compass is an open source project designed for easy extensibility to allow users to deploy custom defect detectors. We have applied Compass to three major programs. We note that providing and applying all available defect detectors results in reports that overwhelm users; hence tool-configuration is essential. Also, the time spent analyzing a particular application for defects is not solely dependent on the size of the application, but also on the complexity of the defect detection algorithms and the complexity of the application investigated. Finally, we have realized that defect analyses can no longer run on single processors in the future; therefore we believe it to be important to build on a scalable infrastructure. Compass is built on Rose, which is just such a scalable compiler infrastructure, allowing for distributed memory parallel processing of applications.

1. INTRODUCTION

Software defects are faults that are introduced unintentionally into computer programs, preventing those programs from behaving correctly. They commonly arise through bad design or bad implementation practices. Defects lead either to the malfunction of a software system—meaning the program behaves unpredictably at run-time—or to security flaws which might later be exploited. We refer to the former as a software bug and the latter as a software vulnerability. Unfortunately, software defects are omnipresent in almost all software developed today, and even worse, they can lead to the unavailability of the most critical systems. For instance the U.S. Navy's Aegis cruiser Yorktown was "dead in the water" for several hours due to a basic programming problem [?].

Even if a software system is bug free, i.e. it behaves according to its specifications at run-time, it may still be considered to be defect if it contains exploitable vulnerabilities. Those vulnerabilities occur again through bad design and bad implementation practices. The effects of a successful vulnerability attack might include serious consequences for major economic and industrial sectors, threats to infrastructure elements such as electric power, and disruption of the response and communications capabilities of first responders. For instance, a Los Angeles Times article reported that the Red Team hackers hit the jackpot when they broke into networks that direct the 911 emergency response systems [?].

Above all, software defects are extremely expensive. According to a 2002 National Institute of Standards and Technology (NIST) study, software errors cost the U.S. economy an estimated 59.5 USD billion annually [?]. Statistics indicate that over the last five years at least 25 percent of economic growth is attributed to information technologies [?].

In this paper, we present Compass, a tool to detect defects in large-scale software applications developed in C, C++, and soon also in Fortran. It is ongoing work that also allows for programming-language-independent vulnerability analysis of software binaries. Compass is an open source tool that is built atop the Rose open source compiler infrastructure [?], a project to define a new type of compiler technology that allows non-expert users to build custom program analysis and translation tools that operate on large-scale source code written in several languages. Rose uses the Edison Design Group C++ front-end [?] to parse C and C++ programs and the Open Fortran Parser (from Los Alamos National Laboratory) to support Fortran. Rose converts the intermediate representation (IR) produced by its front-ends into the ROSE abstract syntax tree (AST) format, which is an intuitive, object-oriented IR with several levels of interfaces for building custom source-to-source analyzers and translators. Rose permits an attribute-grammar-based traversal of the AST with attribute evaluation which supports a simple programming model for users to specify software defects. For more information about Rose, please see [?].

*This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

DEFECTS '08 Seattle, WA USA

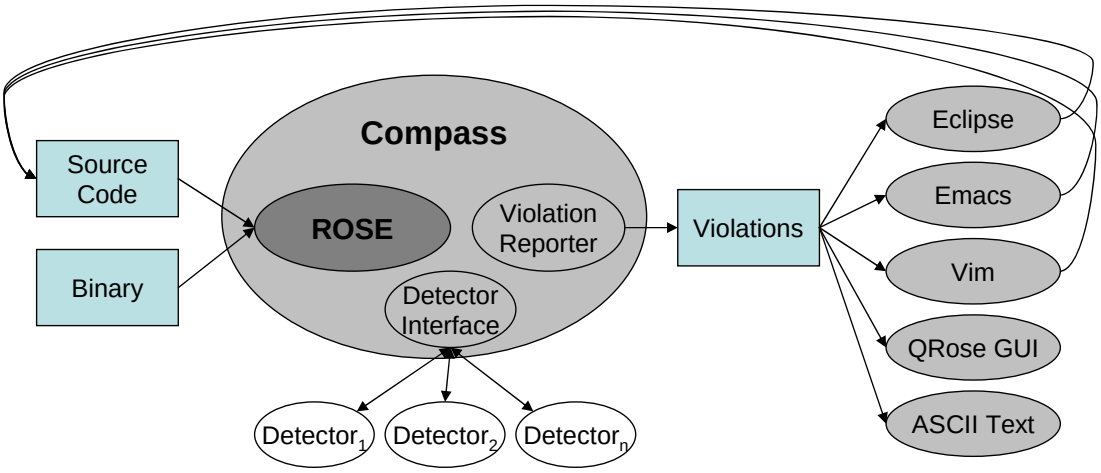

Figure 1: Compass architecture: source and binary code are processed by ROSE and analyzed by user-defined detectors. Unveiled software defects are then reported to users through several external tools.

2. COMPASS

Compass is a tool that allows users to implement detectors to locate and report software defects. Documentation of various kinds of software defects can be found in sources such as the Common Weakness Enumeration [?]. Our focus is not to define new software defects but rather to provide a platform that allows the easy implementation of defect detectors. Compass has been designed to be easy to extend, allowing users to implement their custom detectors (custom source code analyses for identifying defects), as shown in Figure ??. Compass supports the implementation of both simple as well as more advanced defect detectors. For the latter, Compass utilizes the Rose infrastructure to perform a wide range of general purpose program analyses, such as control flow analysis, data flow analysis, program slicing, etc.

Compass is designed in a way that allows users who do not necessarily have compiler backgrounds to utilize the Rose infrastructure to build their own analysis tools. Compass is foremost an extensible open source infrastructure for the development of large collections of rules. Our current implementation supports automatic defect checking, programming language restriction, and malware detection in C, C++, and object code. Support for Fortran is a new addition to ROSE and will be supported in Compass in the near future.

2.1 Source Code Analyzer

Detector rules for source code analysis can simply be specified as programs operating on the Rose AST. Each rule is specified separately, and thus the rules can be evaluated independently. More complex rules may be defined on either the control flow graph (CFG), system dependence graph (SDG), call graph, class hierarchy graph, or combinations of those.

Compass currently contains approximately 90 source code bug detectors; many of our detectors are specified and classified according to the defect severities from CERT [?]. Some are more trivial and, for instance, traverse the AST to check whether the McCabe's cyclomatic complexity of a function definition [?] exceeds a specific threshold, or whether each function is preceded by a comment in the source code. Other analyses such as a null pointer dereference analysis are more complex and do not run in linear time.

As an example, consider the following implementation of a simple Compass detector:

```
void
visit(SgNode* node) {
  SgCastExp* cast = isSgCastExp(node);
  if (cast && cast->get_cast_type() ==
            SgCastExp::e_C_style_cast)
    output->addOutput(new DetectorOutput(node));
}
```

The above code shows our implementation of the CERT [?] Secure Coding Standard specification referred to as EXP00-A. Do not use C-style casts. Although C++ allows C-style casts, from a security perspective it is safer to use C++-style casts that allow for more compiler checking. This defect has CERT severity level 3 (high) because unchecked type errors could allow attackers to exploit the code. Our implementation of this detector resides in the visit function, representing a visitor pattern: this method is called on each node of the program's AST. If the node is a cast and if it is of type e_C_style_cast, a defect is detected and reported. Because ROSE has type information available in the AST, both implicit and explicit casts are detected. Full type information is available even for sophisticated uses of C++ templates.

2.2 Binary Analyzer

Defect analysis should not be performed on source code alone, especially when source code is not available or if compilers are not trusted. Therefore, we have recently extended the Rose IR structure and Compass to support x86 and ARM binaries.

Our approach for software binary analysis is to parse a binary file and represent it as an AST. Our current options to parse binaries are our own disassembler and IDA Pro [?], a disassembler that supports many different platforms including Linux and Windows and a wide variety of processor

architectures. Our approach of representing a binary as an AST allows us to perform defect analysis on that AST, and allows us to reuse much of the Rose's infrastructure for binary analysis. In addition, our AST representation supports the easy creation of control-flow and data-flow graphs, allowing the development of more sophisticated defect analyses.

With this extended Rose infrastructure at hand, we have developed Compass detectors that traverse ASTs representing binaries to check for defects. Defects may again result from bad design and bad implementation practices, and may either lead to unpredictable behavior of software systems (bugs) or security vulnerabilities. An example of a bug would be a null pointer dereference and an example of a vulnerability could be a buffer overflow exploit.

Our Compass implementation for object code currently contains only a few detectors, such as an analysis to statically determine the register values (without emulation) at a particular interrupt call (int 0x80). This interrupt is used for system calls in Linux on x86 processors; the eax register contains the system call number. Our analysis uses data-flow analysis to determine all possible system calls that a particular interrupt could be triggering. If, for instance, the specification of a program states that the program should never write to files or streams, then an int 0x80 call with eax equal to 4 (the sys_write call) would be considered a defect. The precision of this analysis depends on the quality of the data-flow analysis algorithm, including its capabilities in resolving pointers and memory values. However, discussion of the algorithms used in Compass and their efficiency is out of the scope of this paper. Another detector currently implemented is a buffer overflow analysis that locates all memory allocations (calls to malloc) in a program[1]. Any uses of the allocated pointers to that memory are then checked to determine if they are within the allocated buffer size.

Although the examples stated above represent more advanced analyses, simple analyses such as counting the number of instructions in a function or determining the cyclomatic complexity of a function can still be performed on binaries. In future work, we will evaluate the applicability and suitability of various source code analyses to binary software.

2.3 User Interfaces

Compass currently supports five types of defect reporting (shown in Figure ??). As we observed, analysis results from all detectors can be overwhelming; thus, we allow users to select a customized set of detectors for a particular defect analysis run. Besides textual output, we have connected Compass to Emacs, Vim, Eclipse, and the QRose GUI (QRose is a Qt-based GUI library for Rose developed in collaboration with Gabriel Coutinho of Imperial College London). These tools are able to read Compass analysis results and display them in their own GUIs.

In particular, Emacs 22's Flymake can run Compass in the background while a user is editing a file, allowing a real-time display of Compass analysis results. However, because C++ is a very complex language, Compass may take several minutes to process large input files. For Vim, a compiler plugin has been provided for users to work with Vim's QuickFix commands to highlight source code lines containing rule violations. Our Eclipse plug-in for Compass is based on the C++ Rose implementation, connected to Java using SWIG [?]. Figure ?? shows a screen shot of software defect analysis using Compass with Eclipse. Defect analysis results are represented in the problem tab.

[1]This information may be hard to retrieve if a binary is stripped. Nevertheless, even for stripped binaries IDA Pro offers capabilities to detect functions based on their signatures.

3. APPLICATION OF COMPASS

We have initially applied Compass with all its detectors on three large programs: Cxx_Grammar.C, SMG2000, and IRS. In this section, we present an overview of our findings and experiences of defect detection on large C/C++ software applications.

Briefly, Cxx_Grammar.C is a file that is generated by Rose. It represents the implementations of most of the member functions of Rose's AST classes. SMG2000 is a semi-coarsening multigrid solver benchmark that is based on the hypre library [?]. Finally, IRS (implicit radiation solver) is a benchmark that solves diffusion equations on three-dimensional, block structured meshes[2]. Table ?? shows various statistics about the selected programs.

Metric	Cxx_Grammar.C	SMG2000	IRS
Programming Language	C++	C	C
Number of Files	1	71	313
Lines of Code	213,784	28,301	45,565
Number of Function Definitions	17,967	547	1525
Number of AST Nodes	548,774	137,343	314,991
Run-Time in min (default)	7.7	0.1	0.65
Run-Time in min (null-deref)	8.2	35.7	8.3
Run-Time in min (def-use)	148	1231	1877
Memory in MB (default)	78	7	33
Memory in MB (null-deref)	84	113	71
Memory in MB (def-use)	355	3596	3600

Table 1: Statistics of test programs.

The above table shows the different program sizes and the times and memory requirements to run all of our detectors on these programs. Some detectors are much more expensive than the rest; therefore, we have created separate entries for them in the table. In particular, the entry null-deref represents our null pointer dereference analysis and def-use our data-flow analysis computing an intra-procedural definition-usage chain of the program. The entry default represents all detectors except null-deref and def-use. The null-deref and def-use entries include all detectors in the default entry in their runs.

The table indicates that the execution time on a workstation for Compass to check a particular program for defects depends on three factors: the program size, the program complexity, and the complexity of the analysis. For instance, although Cxx_Grammar.C consists of the largest number of nodes, its runtime using either null-deref or def-use is smaller than the runtimes of the other programs. However, in the default setting, Cxx_Grammar.C runs the longest; it contains a large number of small, simple functions. SMG2000 is an example of a complex application containing many pointer operations. Although its program size is smaller than the other program size, its run-time is considerably longer for null-deref and def-use.

[2]Initial work has processed 2/3 of the IRS application.

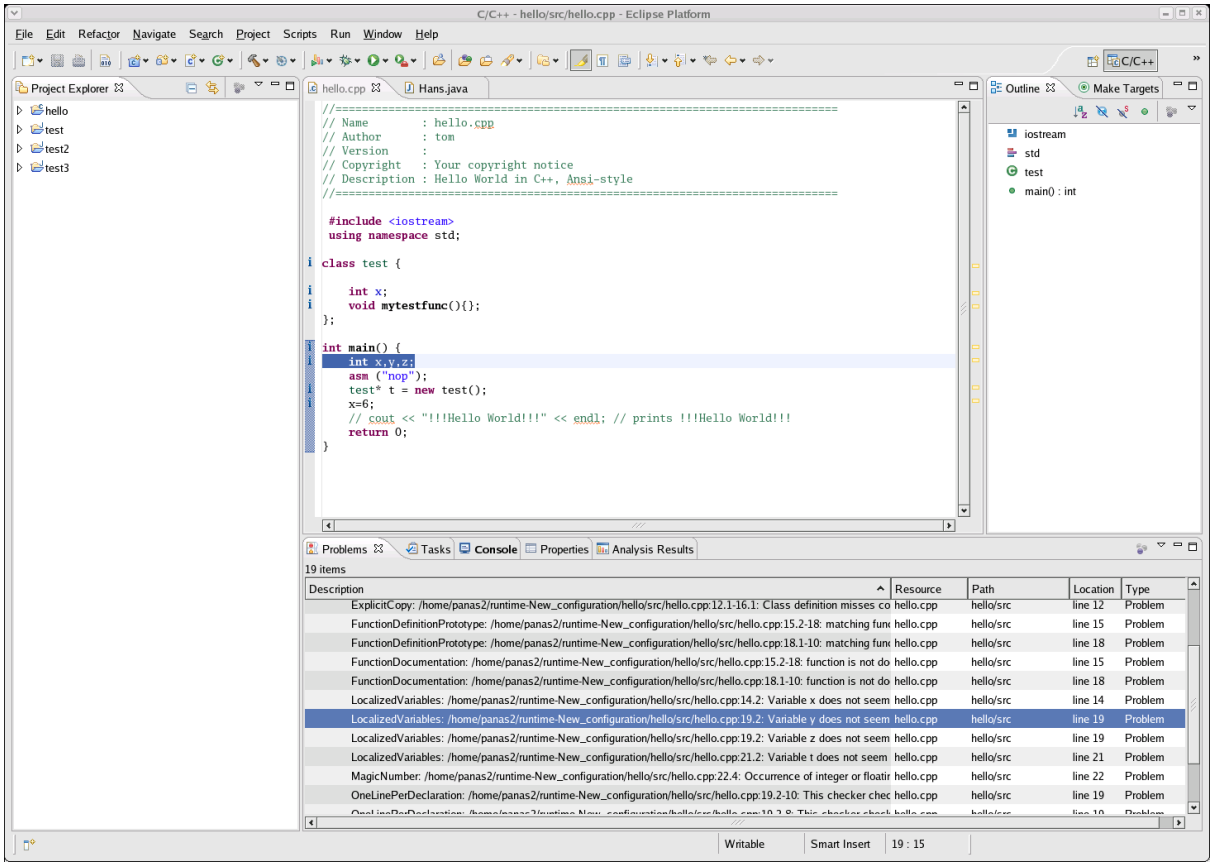

Figure 2: Compass as an Eclipse Plug-in.

Table ?? presents our defect analysis results for some detectors (from CERT [?] and other literature listed in the Compass Tutorial [?]) applied to the programs, in relation to their defect severity and likelihood of appearance (if specified by CERT). The detectors not applicable to C programs have results marked as N/A. Besides the absolute numbers of violations, the ratios of violations per 1000 lines of code (per 1000 functions for CyclomaticComplexity) are given.

Basically, ConstCast is a detector to detect casts from const types to non-const types. ConstStringLiterals are string literals that are constant and should consequently be protected by the const qualification. CyclomaticComplexity reports functions with a complexity value higher than 20. NoGoto checks for goto statements in the program. DoNotDeleteThis is a detector detecting the deletion of the this pointer in C++. Deleting this leaves a dangling pointer, which leads to undefined behavior if accessed. DoNotUseCStyleCast was explained in Section ??. NullDeref is a null pointer dereference analysis. UninitializedDefinition detects variable definitions that are not initialized. Finally, VoidStar reports locations in the source code that cast to void*.

Table ?? shows that Cxx_Grammar.C has many violations that could easily be fixed since the file is generated. SMG2000 and IRS have many uninitialized definitions and other minor problems. However, the number of detected defects is surprisingly large. This confirms our assumption that tools like Compass should be applied to software applications more often, preferably in an interactive environment such as Emacs or Eclipse. We believe our results to be valid as we have cross checked some of the reported defects. The false positive rate for most of the presented detectors should be zero. This is because Rose can precisely detect whether a particular node in the AST is, e.g., a goto statement or not. Nevertheless, some analyses are more tricky, such as the null pointer dereference analysis, as they require pointer analyses for the most precise analysis results. It is therefore crucial to build defect detectors on a well designed and functional infrastructure, such as Rose, that transparently provides the user with fundamental information about a program.

4. PARALLEL DEFECT DETECTION

One problem encountered is that defect analyses are very time consuming, especially for larger programs. The parallel execution of our Compass detectors is our way to decrease the amount of time needed to perform defect analyses. We have enhanced Rose to support parallel AST attribute evaluation which in turn enables Compass to run its detectors on distributed memory platforms using MPI. This means that a program's AST is replicated onto all processors, and each function's analysis is assigned to a processor. Each CPU runs all Compass detectors on its assigned functions; this is possible because all detectors are intraprocedural. The defect detection results from each CPU are then gathered and combined, then reported to the user [?].

Applying parallel Compass execution to our example ap-

Detector	Severity	Likelihood	Cxx_grammar.C		SMG2000		IRS	
			number	ratio	number	ratio	number	ratio
ConstCast	low	probable	1,092	5.11	170	6.01	358	7.86
ConstStringLiterals	low	likely	147	0.69	87	3.07	1373	30.13
CyclomaticComplexity	-	-	4	0.22/kfunc	19	34.73/kfunc	56	36.72/kfunc
NoGoto	-	-	12	0.056	0	0	1	0.022
DoNotDeleteThis	high	unlikely	2	0.009	N/A	N/A	N/A	N/A
DoNotUseCStyleCast	high	probable	41,705	195.08	N/A	N/A	N/A	N/A
NullDeref	high	likely	219	1.02	0	0	368	8.08
UninitializedDefinition	-	-	156	0.73	2,209	78.05	4,482	98.36
VoidStar	-	-	766	3.58	0	0	0	0

Table 2: Compass defect results for our three programs under study.

plications results in load balancing challenges. Our system distributes whole functions in the input program to each processor; thus, programs with complex functions can cause load imbalances. This especially applies to more expensive analyses such as null-deref and def-use. Thus, SMG2000 does not scale well beyond 8 processors currently. The other two applications do scale much better as load balancing becomes less of an issue. Using 8 processors on a supercomputer, Cxx_Grammar.C (using the def-use algorithm) results in a run of only 815 seconds; using 16 processors results in 180 seconds. However, using 31 processors, the time is reduced to 151 seconds for the slowest processor and 56 seconds for the fastest, indicating another load balancing issue. We have noted these issues only recently and will adapt our algorithms accordingly in the near future.

5. CONCLUSION AND FUTURE WORK

We have presented Compass, an open source Rose-based tool which allows users, either with or without compiler backgrounds, to easily add custom defect detectors. Currently, approximately 90 Compass detectors exist for C, C++, and object code; Fortran support is forthcoming. Because the program analyses used by the detectors are very expensive for large code bases, we have also developed a way to conduct defect analysis in parallel.

Our preliminary results are encouraging and have motivated us to improve on our prototype implementation. Future work includes the exploration of high level defect analysis specifications, such as Datalog (a subset of Prolog), more defect detectors for software binaries, a classification of severity for our detectors, and more case studies with a variety of source and binary applications.

6. ACKNOWLEDGMENTS

We would like to thank all the students who helped to implement Compass detectors: Gergo Barany, Michael Byrd, Valentin David, Han Kim, Robert Preissl, Gary Yuan, and Ramakrishna Upadrasta.[3]