ROSE Specialized Compiler Infrastructure

Gregory Pope

LLNL 10/1/2019

V1.1

LLNL-BR-753916

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Table of Contents

Tables

Figures

ROSE Specialized Compiler Infrastructure

1.0 Purpose:

The purpose of this paper is to describe the ROSE specialized compiler infrastructure and its uses.

Hardly a day goes by without multiple news articles re-counting the consequences of failed software. A tragic autonomous vehicle accident, theft of personal information from a "secure" data base, a smart TV that invades privacy, or just the frustration of an application that hangs or reboots at the worst possible moment. The good news is, one of the oldest and most trusted software technologies can solve these present and future software assurance (safety, security, and quality) issues. This paper will focus on the ROSE open source specialized compiler infrastructure and describe how it helps assure that the software we use is safe, secure, and high quality.

2.0 Specialized Compiler Infrastructures:

Compilers are widely used for software development. In fact, every line of code ever written must be compiled to be useful[1] Compilers have been around for over 65 years, the first of which was written in 1951 by Corrado Bohm (University of Rome) for his PhD thesis[2]. The name "compiler" was later coined by Grace Hopper[3] in 1953[4]. They have been such a steady and foundational component of development for so long, we now take compilers for granted and, over time, have become less familiar with their inner workings and their vast potential to meet contemporary and future software challenges.

Specialized compiler infrastructures are built for specific purposes, usually research, and add additional capabilities to the traditional role of a compiler. ROSE is an example of an open source specialized compiler infrastructure. Basically, ROSE allows users to access intermediate phases (see appendix A) of the compiler process to query, modify, or report on code of interest. Accessing intermediate phases within a compiler facilitates building tools, for example: static analysis of code or instrumenting of code for dynamic analysis. ROSE has some unique capabilities in addition to some overlapping ones when compared to other specialized compiler infrastructures[5].

3.0 Potential Applications of the ROSE Specialized Compiler Infrastructures:

The ROSE specialized compiler infrastructure offers the ability for users to develop tools to leverage the power of the compiler in additional ways to address contemporary computer industry challenges. Some examples are given in Table 1, ROSE Specialized Compiler Infrastructure Tools.

1	Automated static analysis (ASA), the ability of the compiler infrastructure-built tool to enforce safety constraints, find potential security vulnerabilities, locate structural code issues in source or binary code without needing to execute it.
2	Instrument source code or target code to identify problems that may occur only at run time as a debugging aid.

3	Dynamic or static symbolic, concolic, or concrete analysis of source or binary codes to locate erroneous code.
4	Automate repair of source or binary code when an undesired issue is detected, and the fix can be specified.
5	Optimization of source code or target code for differing machine architectures, allows porting to be automated.
6	Translation from one source language into another source language, for instance an obsolete language into a contemporary one.
7	Verification checking of auto generated code from a model, specification, or another language.
8	Seeding known faults into source or binary code to evaluated test case error detection and coverage.
9	Automated or semi-automated creation of unit test cases to supplement manually created test cases.
10	Model checking source or binary codes against examples of potentially vulnerable coding practices or undesired code.

Table 1 - ROSE Specialized Compiler Infrastructure Tools

4.0 What is the ROSE Compiler Infrastructure?

The ROSE compiler infrastructure project was initially started in 1993 by Dan Quinlan[6] at the Lawrence Livermore National Laboratory (LLNL)[7] as a research compiler to perform source code analysis and generate transformed source code (rewritten source code in the same language) and translated source code from one language to another (transpiled source code). ROSE also performs binary code analysis and binary to binary code rewriting. Tools are built using C++ and the Sage API enabling users to generate their own custom code analysis and code translations. Several tools have been developed under ROSE projects for HPC (High Performance Computing) optimizations and porting, as well as LTL (Linear Temporal Logic[8]) Specification to C verification (such as Code Thorn[9]), Debugging aids (such as Backstroke[10]), and several static and binary analysis tools. ROSE is an open source code distributed under a permissive BSD license and consists of 3.5 million lines of C++ code.

As shown in Figure 1, ROSE Components, ROSE consists of a Frontend, a Mid-end operating on its internal intermediate representation (IR), and Backend.

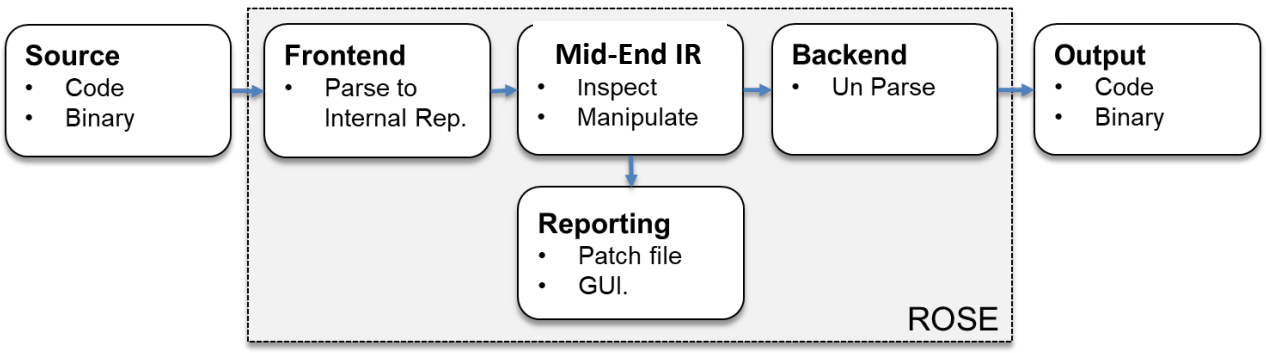

Figure 1 - ROSE Components

1. The ROSE Frontend utilizes Edison Design Group's (EDG)[11] for C and C++ and the Open Fortran Parser (OFP)[12] developed at Los Alamos National Laboratory[13] for Fortran. The ROSE Frontend can also use Open MP, Open CL, Cuda, Cuda/Fortran, Php, Java, and Clang compilers. ROSE is extensible to other source Frontends such as C#, Ada, and Jovial. ROSE also contains a binary Frontend capable of handling input from various executable formats and disassembly into Intel x86, Intel/AMD x86_64, Motorola, ARM, Power/PC, and MIPS architectures.

2. The Middlend IR consists of a mutable abstract syntax tree (AST), symbol tables, control flow graph, etc. It is an object-oriented IR with several levels of interfaces for quickly building source-to-source translators. All information from the input source code is carefully preserved in the ROSE IR, including C preprocessor control structure, source comments, source position information, and C++ template information, e.g., template arguments. The Mid-end IR is also used for binary analysis. For binary analysis the Mid-end and Frontend work closely together, the Mid-end performs analysis on incomplete data and then informs the Frontend so it can feed more complete information back to the Mid-end.

3. The Backend regenerates (un-parses) source code from the IR. If desired, this source code can be compiled with conventional compilers into the final executables. The Backend is also capable of rewriting binaries that have been input and analyzed. ROSE can also transform source into LLVM IR to take advantage of LLVM tools for low level optimization and directly creating executables.

5.0 ROSE Summary:

The main features that make ROSE unique as a specialized compiler infrastructure are that it:

1 Outputs transformed source code dynamically generated based on analysis of its high-level IR (source to source compiler);
2 Inputs binary executable formats disassembles them, and performs various types of analysis; and
3 Can translate between source languages (transpile) and check the resulting translation for equivalence.
4 Can perform analysis on a binary code and the source code that generated it.

6.0 ROSE User Categories

ROSE users fall into three primary categories:

1 **Compiler Researchers**: A relatively small but important group of users is comprised of compiler researchers and academics with advanced degrees who pursue leading edge concepts for optimizing compilers and code for present and future computer architectures, program analysis, and software assurance. They may make changes to the open source ROSE code to add new capabilities. Typically, they are not under strict time constraints.

Their work may result in technical publications for journals, presentations, and prototype code.

2 **Tool Builders:** Users with a software development backgrounds who use the ROSE specialized compiler infrastructure to write analysis or transformation tools in C++ which interrogates the ROSE internal representations using the API to support program analysis, optimizations, and software assurance. They typically do not alter the open source ROSE source code. These users may be working to deadlines imposed by development projects or driven by changes or trends in the computer science field.

3 **Tool Users:** Users who deploy ROSE based tools developed by tool developers in various software development environments. They typically do not alter the open source ROSE source code or C++ code used to build the tool. They are challenged by successfully using the ROSE built tools on various platforms, compilers, libraries, and versions of each. They would analyze and interpret ROSE based tool results These users typically have a development, IT, DevOps, or Software Testing backgrounds. They may be on daily or weekly time schedules.

7.0 ROSE Delivery Methods

ROSE is available via the ROSE homepage[14]. There are four formats available for obtaining the ROSE compiler infrastructure:

1 Binary files for use with the ROSE binary installer. This is the quickest way to install ROSE for users who do not need to alter the source code and are using supported platforms. Tool Developers and Tool Users would be the most likely to utilize this delivery method.
2 ROSE source code can be obtained in a Docker Container.
3 ROSE source code can be obtained using SPACK to simplify dependencies.
4 ROSE source code can be obtained directly.

8.0 ROSE Binary Analysis Tools

Several Binary Analysis Tools have been built or in the process of being built with the ROSE Compiler Framework for cyber security applications. These tools must be able to format a variety of file formats as well as target architectures and then perform automated analysis. Automated binary analysis allows cyber security analysts to examine a large variety of device type's firmware in a relatively short amount of time when compared to manual techniques. It also allows the examination of large qualities of families of devices (are only a few infected or is the malware wide spread). To date the types of binary analysis implemented are:

9.0 Summary

ROSE is an example of an open source compiler infrastructure that facilitates building powerful tools that leverage tried and true compiler technology developed decades ago to solve present and future software assurance issues. ROSE has unique attributes that position it to be part of the software assurance solution. The creativity of tool developers leveraging compiler infrastructures such as ROSE are an essential mitigation against currently known and future unknown software challenges.

10.0 Appendix A Compiler Phases and Components:

Most developers in the software industry use compilers on a regular basis and are familiar with what compilers do - i.e., transform source code written in a programming or computer language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code* or *machine code*). The most common reason for transforming source code is to create an executable program[15]. Figure 2, Typical Compiler Components, contains a diagram of typical compiler phases.

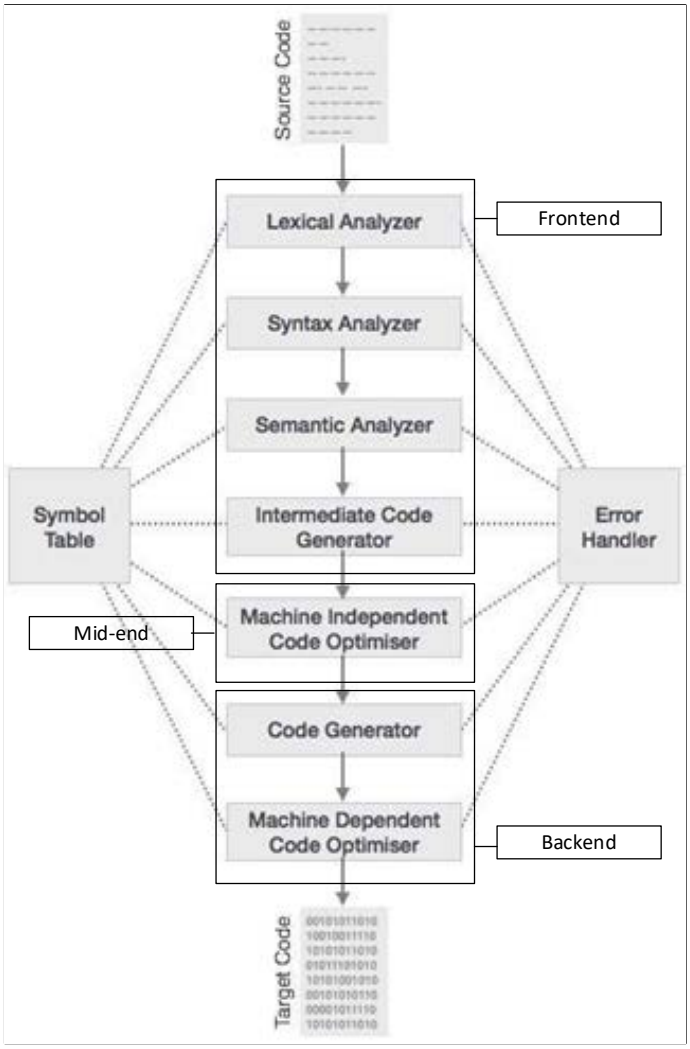

Figure 2 - Typical Compiler Components

Each phase of the compiler takes input from a previous phase, creates its own representation of the source program, and provides its output to the next phase. A brief description of each phase follows. For a more thorough treatment of the compiler phases see end note number 6[16].

The **Lexical Analyzer** scans the characters of the source code and outputs tokens to the Syntax Analyzer. Example tokens are a language keyword, number (int/float/complex/…), symbol (variable/function/… id), or operator.

The **Syntax Analyzer** uses the tokens as input to create a parse tree to check that the expression made by the tokens is syntactically correct. The parse tree may take the form of an Abstract Syntax Tree to eliminate unnecessary details.

The Semantic Analysis receives the syntax tree as input and checks that it follows the rules of the language. For example, the source statement `int foo = “bar”;` is syntactically correct but does not follow the language assignment rules and would be flagged as an error. The Semantic Analyzer

outputs an annotated syntax tree. For some languages such as C++ the Lexical, Syntax, and Semantic analysis must work together in an iterative fashion because of special cases that arise[17].

The Intermediate Code Generator inputs the syntax tree and constructs an intermediate code representation for an abstract target machine. The intermediate representation is then used as input for the Code Optimization phase.

The **Code Optimization** phase removes unnecessary code and optimizes for speed and resources. The intermediate representation allows all the phases of the compiler so far discussed to be used for multiple target machines.

The **Code Generation** phase takes the optimized intermediate code output and maps it to relocatable machine code for a specific target machine.

The **Machine Dependent Code Optimizer** inputs the relocatable machine code and performs optimizations that are unique to the target machine

The **Symbol Table** is a data structure that is maintained throughout all phases of the compilation process. Identifier names and types are stored in it. The Symbol Table is also used for scope management. This table speeds up the compilers ability to search and retrieve identifier records.

The **Error Handler** is also maintained throughout all the phases of compilation and handles errors detected in each phase by determining error severity and notifying users about the nature of the detected error.

As shown in figure 2 the seven phases of compilers can be grouped into three components of one or more phases for simplicity (the Frontend, Backend, and Mid-end). This allows compilers to be used for different source languages by having different **Frontends** and different machine targets by having different **Backends**.

11.0 Acronyms and Terms

Acronym or Term	Definition
API	Application Program Interface
ASA	Automated Static Analysis
AST	Abstract Syntax Tree
BSD	Berkeley Software Distribution. A permissive OSS license
CPU	Central Processing Unit
EDG	Edison Design Group
HPC	High Performance Computing
id	Identifier
int	integer
IR	Intermediate representation

LLNL	Lawrence Livermore National Laboratory
LTL	Linear Temporal Logic
OFP	Open Fortran Parser
ROSE	Name of a Compiler Infrastructure, not an acronym.

12.0 End Notes:

[1] Well maybe excluding early codes written in machine language or assemble.
[2] https://en.wikipedia.org/wiki/History_of_compiler_construction#cite_note-wikles1968-1
[3] Who also coined the phrase software "bug".
[4] https://en.wikipedia.org/wiki/History_of_compiler_construction#cite_note-wikles1968-1
[5] Software Assurance Using Specialized Compiler Technology, Gregory Pope, LLNL 6/24/2018, V1.3
[6] https://people.llnl.gov/quinlan1
[7] https://www.llnl.gov/
[8] https://en.wikipedia.org/wiki/Linear_temporal_logic
[9] https://github.com/rose-compiler/rose/tree/master/projects/CodeThorn
[10] https://github.com/LLNL/backstroke
[11] https://www.edg.com/
[12] http://fortran-parser.sourceforge.net/
[13] http://www.lanl.gov/
[14] http://rosecompiler.org/
[15] ibid
[16] https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm
[17] One special case example the >> token. See https://stackoverflow.com/questions/19106622/compiler-limitation-of-lexical-analysis