Software Assurance Using Specialized Compiler Technology

Gregory Pope

LLNL 9/30/2019

V1.4

LLNL-BR-753679

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

Software Assurance Using Specialized Compiler Technology

1.0 Purpose:

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, or the frustration of an application that hangs or reboots. 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 two example open source specialized compiler infrastructures (ROSE and LLVM), compare both, and describe how they help assure that the software we use is safe, secure, and high quality.

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.

2.0 Compiler Phases and Components:

If you are familiar with the phases and components of compilers you can skip to section 3. If not, read on.

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[5]. Figure 1, Typical Compiler Components, contains a diagram of typical compiler phases.

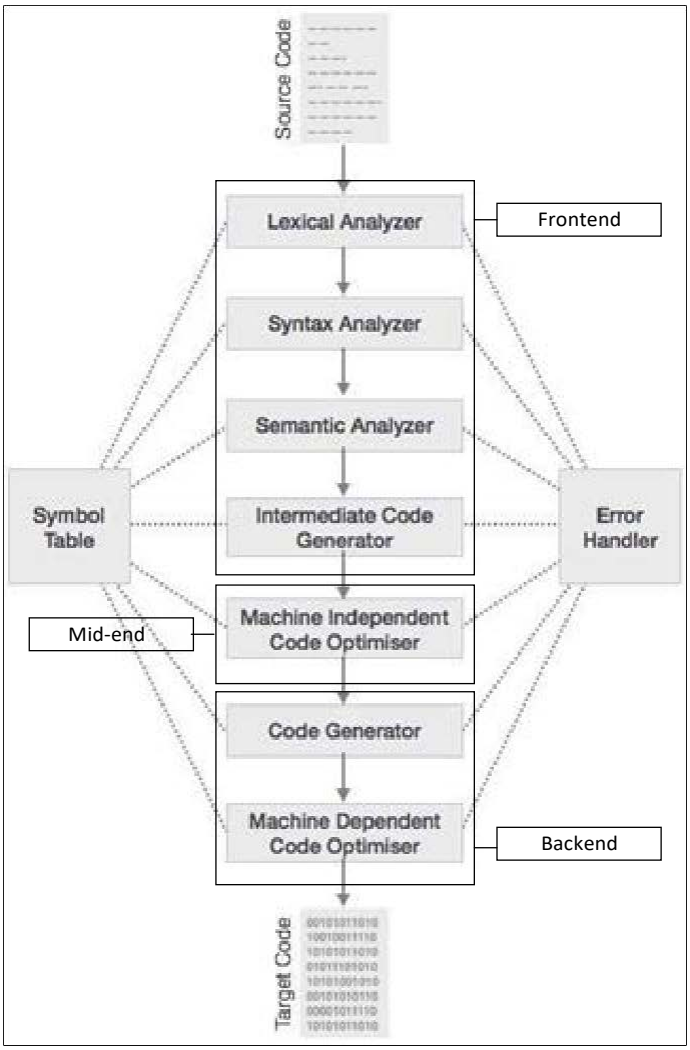

Figure 1 - Typical Compiler Components

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[6].

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[7].

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 1 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**. LLVM, which is a **Mid-end** tool, depends on various Frontends such as Clang or Flang for C, C++ and Fortran. LLVM can use multiple Backends to create machine code for several targets. The same is true for ROSE, which can use multiple Frontends. However, this paper will show that LLVM and ROSE differ most in the Mid-end and Backend components.

3.0 Specialized Compiler Infrastructures:

Specialized compiler infrastructures are built for specific purposes, usually research, and add additional capabilities to the traditional role of a compiler. ROSE and LLVM are two such open source specialized compiler infrastructures. Basically, these specialized compilers allow users to access intermediate phases of the compiler process to query, modify, or report on code of interest. Accessing intermediate phases facilitates building tools, for example: static analysis of code or instrumenting of code for dynamic analysis. In the case of ROSE and LLVM, each has some unique capabilities in addition to some overlapping ones.

4.0 Potential Applications of Specialized Compiler Infrastructures:

Specialized compiler infrastructures offer 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, 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 - Specialized Compiler Infrastructure Tools

5.0 What is the ROSE Compiler Infrastructure?

The ROSE compiler infrastructure project was initially started in 1993 by Dan Quinlan[8] at the Lawrence Livermore National Laboratory (LLNL)[9] 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[10]) Specification to C verification (such as Code Thorn[11]), Debugging aids (such as Backstroke[12]), 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 2, ROSE Components, ROSE consists of a Frontend, a Mid-end operating on its internal intermediate representation (IR), and Backend.

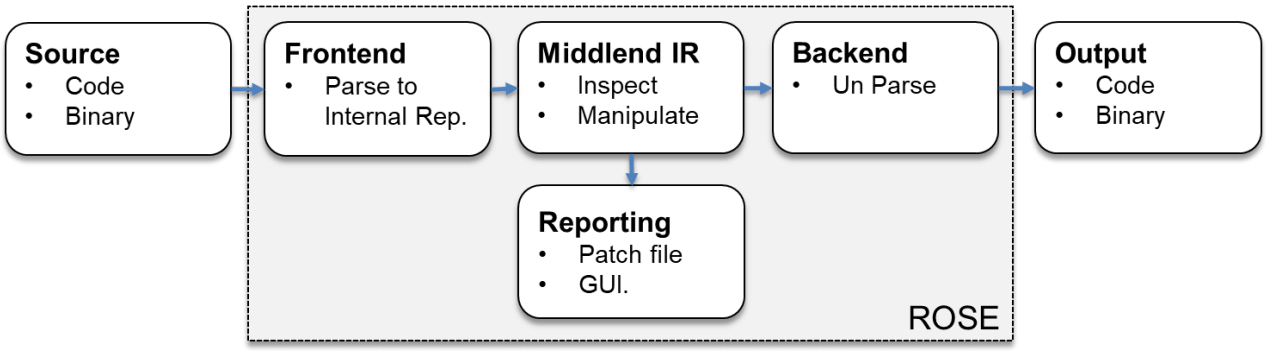

Figure 2 - ROSE Components

1. The ROSE Frontend utilizes Edison Design Group's (EDG)[13] for C and C++ and the Open Fortran Parser (OFP)[14] developed at Los Alamos National Laboratory[15] 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 Mid-end 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.

6.0 ROSE Summary:

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

1 Outputs transformed source code dynamically generated based on analysis of its high-level IR;

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.

7.0 What is the LLVM Compiler Infrastructure?

The LLVM Compiler infrastructure was originally developed and released in 2003 under the direction of Vikram Adve[16] and Chris Lattner[17] as a research project at the University of Illinois Urbana-Champlain[18]. The goal was to provide a modern SSA-based[19] compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. LLVM is a Mid-end compiler component, however it has also become a collection of modular and toolchain technologies used to develop compiler Frontends and Backends. LLVM is used in a variety of open source and commercial projects and academic research. LLVM consists of 2.5 million lines of C++ code and is licensed under UIUC[20], a permissive license like the MIT or BSD licenses. Figure 3, LLVM Components, shows the three-phase design of LLVM.

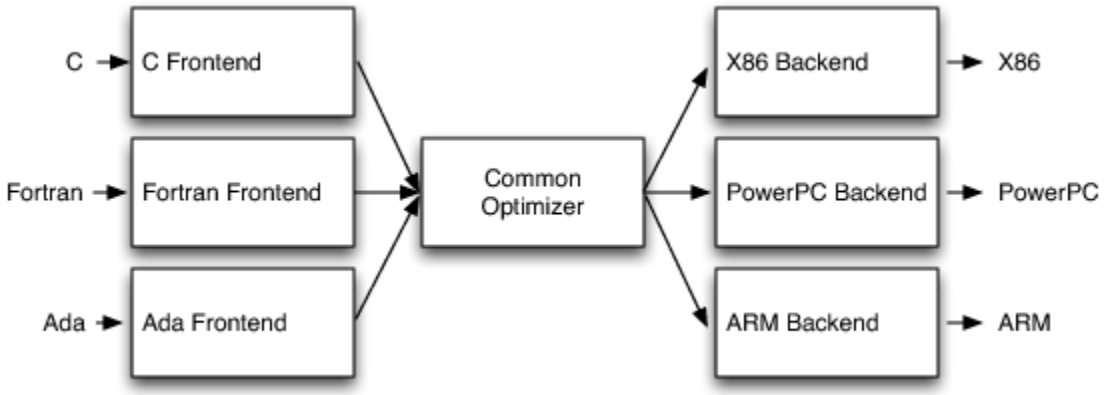

Figure 3 - LLVM Components

1 LLVM uses several Frontends. The most widely used Frontend is Clang[21] which supports, C, C++, and Objective C and C++. Clang parses the languages and creates an immutable high-level AST. Clang also consists of several reusable libraries that can be used with Clang to provide static analysis, dynamic analysis, debugging, address, thread, and memory sanitizers. Other Frontends allow parsing of ActionScript, Ada, C#, Common Lisp, Crystal, CUDA, D, Delphi, Fortran (2003), Halide, Haskell, Java bytecode, Julia, Kotlin, Lua, Objective-C, OpenGL, Open MP, Shading Language, Pony, Python, R, Ruby, Rust, Scala, Swift, and Xojo.

2 The LLVM Mid-end (Optimizer) provides a source and target independent optimizer. It does this by creating an LLVM IR which is a low-level assembler-like language to create executables for specific target machines, but not allowing reconstruction of source code.

3 The LLVM Backend provides an executable code generator for x86, x86_64, z/Architecture, ARM, and PowerPC, with at least partial support for Qualcom Hexagon, MIPS, Nvidia Parallel Thread Execution, AMD TeraScale, AMD Graphics Core Next, SPARC, and Xcore.

8.0 LLVM Summary:

The main features that make LLVM (Low Level Virtual Machine) unique as a compiler infrastructure are that it:

1 Has very fast compile times and works with several Frontends, the most common is Clang for C and C++, and Objective C and C++;
2 Creates optimized executable code for most popular CPU architectures. Limited string based rewriting capability useful for instrumentation but not general rewriting; and
3 Widely used and under the LLVM umbrella, there are several Frontends, tools, and Backends available. LLVM is a good tool to use for building a new language compiler.

9.0 Comparing Compiler Infrastructures - ROSE versus LLVM:

Both ROSE and LLVM can be viewed as consisting of three major components (this is a simplification for the purposes of this paper, since ROSE and LLVM can also refer to tools, libraries, or tool chains built using them).

Both use Frontends that are source-language dependent and parse input creating an internal representation (IR or high-level AST) for use by the Mid-end, and Backends. Both tool Frontends can parse C (89,98), C++ (98,03,11) Cuda, Fortran 2003, Java, Python, Open MP, Open GL, and PHP. ROSE alone parses Fortran (77,95) and UPC, while LLVM alone can parse C++ (14,17), D, Haskell, Lua, Objective C, Objective C++, Pure, and Ruby.

Both use a Mid-end; the ROSE and LLVM Mid-end differences are purpose driven. ROSE transforms the Frontend high-level AST into a mutable Mid-end high-level IR AST that includes sufficient information to reproduce the source code exactly. The LLVM Mid-end creates a low-level assembler- like language as an IR and performs optimizations on the IR to generate machine code for various CPU's. The LLVM IR precludes certain kinds of analysis because information needed (such as classes, loops, templates, etc.) is not retained by the LLVM low level IR.

Backends of ROSE and LLVM are also purpose driven. The ROSE Backend is an un-parser to recreate new source code from the Mid-end mutable high-level AST, while LLVM creates executables from its assembler-like low-level IR. Different LLVM Backends support different machine CPU architectures.

ROSE can directly input binary files as input for analysis. LLVM must use external programs to upload disassembled executables into its low-level IR.

ROSE can also use Clang as a Frontend and output LLVM low level IR to take advantage of LLVM Backend executable generators. Thus, combining the unique capabilities of both research compilers.

10.0 Comparison Summary:

ROSE offers unique advantages as the compiler framework to use for source to source transformations (source rewriting), including source to a different source translation (transpiling). ROSE has a self-contained binary to IR capability, whereas, LLVM requires a user supplied translator from binary to LLVM IR. LLVM with Clang (or other Frontend) is the compiler to use for fast compilation and optimized executables for a variety of CPU architectures. Both tools can accomplish source code analysis but do so using different Mid-end IR representations. ROSE is geared more for general purpose and high-level source code analysis and optimizations, LLVM is better suited for deep machine code analysis and optimizations. Many tools have been written for LLVM, including KLEE for symbolic execution and Cling an interactive C++ interpreter. As a traditional compiler LLVM is quite attractive and used by several commercial companies. Table 2, Compiler Infrastructure Comparison, summarizes the overlap and unique capabilities.

		ROSE	LLVM	Comment
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.	X	X	LLVM limited to IR analysis only.
2	Instrument source code or target code to identify problems that may only occur at run time as a debugging aid.	X	X	LLVM limited to IR modifications only.
3	Dynamic or static symbolic, concolic, or concrete analysis of source of binary codes to locate erroneous code.	X	X	LLVM limited to IR analysis only.
4	Automate repair of source or binary code when an undesired issue is detected and the fix can be specified.	X	X	LLVM limited to IR only.
5	Optimization of source code or target code for differing machine architectures.	X	X	LLVM target code only
6	Translation from one source language into another source language, for instance an obsolete language into a contemporary one.	X		ROSE has a mutable AST, LLVM is immutable.
7	Verification checking of auto generated source code from a model, specification, other another language.	X		
8	Seeding known faults into source or binary code to evaluated test case error detection and coverage.	X	X	LLVM limited to binaries only.
9	Automated or semi-automated creation of unit test cases to supplement manually created test cases.	X		
10	Model checking source or binary codes against examples of potentially vulnerable coding practices or undesired code.	X	X	LLVM limited to binaries only.

Table 2 - Compiler Infrastructure Comparison

11.0 Summary

Both LLVM and ROSE are examples of open source compiler infrastructures that are facilitating a new generation of powerful tools that leverage tried and true compiler technology developed decades ago to solve present and future software assurance issues. While some of the capabilities of LLVM and ROSE overlap, each 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 and LLVM are a essential mitigation against currently known and future unknown software challenges.

12.0 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
KLEE	symbolic virtual machine built on top of the LLVM
LLNL	Lawrence Livermore National Laboratory
LTL	Linear Temporal Logic
LLVM	Low Level Virtual Machine original meaning, but now refers to optimizer.
MIT	Massachusetts Institute of Technology (MIT). A permissive OSS license
OFP	Open Fortran Parser
OSS	Open Source Software
PhD	Doctor of Philosophy
ROSE	Name of a Compiler Infrastructure, not an acronym.
SSA	Static Single Assignment
Transpile	Translate source code from one language into another language
UIUC	University of Illinois at Urbana

13.0 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] ibid
[6] https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm
[7] One special case example the >> token. See https://stackoverflow.com/questions/19106622/compiler-limitation-of-lexical-analysis
[8] https://people.llnl.gov/quinlan1
[9] https://www.llnl.gov/
[10] https://en.wikipedia.org/wiki/Linear_temporal_logic
[11] https://github.com/rose-compiler/rose/tree/master/projects/CodeThorn
[12] https://github.com/LLNL/backstroke
[13] https://www.edg.com/
[14] http://fortran-parser.sourceforge.net/
[15] http://www.lanl.gov/
[16] https://en.wikipedia.org/wiki/Vikram_Adve
[17] https://en.wikipedia.org/wiki/Chris_Lattner
[18] http://illinois.edu/
[19] https://en.wikipedia.org/wiki/Static_single_assignment_form
[20] https://opensource.org/licenses/NCSA
[21] https://clang.llvm.org/