Extending the LLFI Tool to Support Compilation of Project Files

Submitted in partial fulfillment of the requirements for the undergraduate thesis, EECE 496

Name: Abraham Chan
Supervisor: Karthik Pattabiraman
Date Submitted: December 1, 2014
Acknowledgements

I would like to thank my research supervisor Professor Karthik Pattabiraman for providing the initial concept and offering continuous guidance over the course of this project. I would also like to express my gratitude to graduate students, Qining Lu and Nithya Narayanan-murthy, for assisting me in the development of the command line and GUI components respectively. This project would not have been accomplished without their corresponding contributions.
Abstract

LLFI is a LLVM based fault injection tool that performs hardware fault injections at the intermediate representation (IR) level. To perform a fault-injection, the source files must first be compiled into IR, followed by the generation of executables, and the profiling of the faults. The pre-existing tool is designed for single C/C++ source files. With increased interest from industry to understand their application’s resilience to faults, the LLFI tool must be extended to operate on whole projects.

Multiple alternate solutions were proposed to implement this extension. The preferred solution comprises of a script to generate a Makefile that takes into account of user options, and subsequently builds the Makefile. This solution is incorporated into both the command line and GUI tools.

The usability of this design is accessed through the deployment of several industry benchmark applications. The design is evaluated by its efficiency, responsiveness and integration to the overall workflow. The key challenge of the analysis is on apprehending an existing user’s workflow in the tool.
# Contents

Acknowledgements .......................... i

Abstract .................................. ii

List of Definitions ......................... iv

1 Introduction .............................. 1
   1.1 Background ........................... 1
   1.2 Purpose and Scope ................... 1
   1.3 Objectives ........................... 1

2 System Overview ......................... 2

3 Methodology .............................. 3
   3.1 Proposed Solutions .................... 3
      3.1.1 Proposed Solution 1 ............... 3
      3.1.2 Proposed Solution 2 ............... 3
      3.1.3 Proposed Solution 3 ............... 4
      3.1.4 Preferred Solution ................. 4
   3.2 Solution Design ....................... 4
      3.2.1 Command Line Tool ................. 4
      3.2.2 GUI ................................ 6
   3.3 Design Assumptions ................... 9

4 Analysis of Usability .................... 10
   4.1 Benchmark Applications ............... 10
      4.1.1 Parboil ........................... 10
      4.1.2 SPEC CPU™ 2006 ................. 11

5 Results and Discussion .................. 11

6 Conclusion ................................ 12

7 Future work ............................. 12

References ................................ 13

Appendix A ................................. 14

Appendix B ................................. 18
List of Definitions

Clang
C front-end for the LLVM Compiler

Clang++
C++ front-end for the LLVM Compiler

CMake
Cross-platform Make tool that automatically creates native Makefiles

Instrumentation
The process of creating LLFI executables from an IR file

IR
Intermediate representation form, a low-level instruction set close to assembly

LLFI
Intermediate Representation Fault Injector Tool

LLVM
Back-end compiler package, utilized by LLFI

Make
Tool to manage the formation of executables and other generated files from source code

Makefile
File that contains information about source files, rules, variables and directives to organize the compilation and linking process

Profiling
User defined selection of injectable faults

Usability
A qualitative measure of user comfortability in using a software application
1 Introduction

1.1 Background

LLFI is a fault-injection tool that injects hardware faults at the LLVM IR level [1]. To perform fault-injections, LLFI must first compile source code files into its intermediate representation form. LLFI currently supports the compilation of single C/C++ source code files. With a growing industry demand to understand applications' fault tolerance, particularly in traditionally unsafe languages, LLFI must be expanded to perform fault-injections on projects files. Typical projects consist of an assortment of source files, header files and object files. Presently, a workaround that exists for the compilation of multiple code files involves the manual modification of the Makefile. However, this workaround must be applied through the command line application of LLFI, without any support from the LLFI-GUI.

1.2 Purpose and Scope

This report investigates the extension of the LLFI tool to formally enable compilation of C/C++ project files. The scope of this report will cover alternative proposed solutions, followed by the selection of the preferred solution, and concluded with an evaluation of usability on the selected solution. Usability of the implemented solution will be assessed using industry benchmark applications.

1.3 Objectives

The objective of this project is to introduce the notion of project file compilation of LLFI. The implemented solution will be applied to both the LLFI command line tool and the LLFI-GUI application. An integrated command line and GUI application will facilitate the user's experience of the tool, as well as, provide consistency over supported features.


## 2 System Overview

The typical workflow, shown in Figure 1, for a fault-injection on a single source file consists of the following order of steps:

1. Compilation
2. Instrumentation
3. Profiling
4. Fault-injection

![Figure 1: High Level Design of Pre-existing System](image)

This report focuses on modifying the compilation process to accommodate project files. The compilation process involves a LLVM-based compiler, which converts the source files into LLVM IR format. Clang is a C language front-end for the LLVM compiler [2]. Source files in C are compiled with Clang while source files in C++ are compiled with Clang++.
3 Methodology

3.1 Proposed Solutions

Three alternative solutions were proposed for LLFI to enable project file compilation:

1. Modification of the CompileToIR script to parse project files
2. Creation of a LLFI custom Makefile
3. Creation of a LLFI custom CMakeLists.txt file

All of the solutions above are compatible with both the command line tool and the GUI tool. The benefits and drawbacks of each of these proposed solutions are examined below.

3.1.1 Proposed Solution 1

The first proposed solution is to modify the existing CompileToIR script to permit parsing of the project files. The CompileToIR script is implemented as a python executable command, which accepts source files, an output filename, and compiler options as input parameters and generates a LLVM IR file. This solution would consist of modifying the input for source files to accept a project folder instead. All valid C/C++ source files within the project folder would be parsed and aggregated into a list. CompileToIR would execute with this list of source files, along with additional compiler options as indicated by the user. This approach would pose minimal impact to users as the python script internally discerns project files from individual source files. However, there are two main drawbacks to this solution. Firstly, this solution does not facilitate the input of custom compiler flags. As larger and specialized projects are run with the LLFI tool, users may seek to configure custom compiler flags. Secondly, this solution is not scalable for large projects. This solution would involve parsing and recompiling the project file each time a single code file is modified.

3.1.2 Proposed Solution 2

The second proposed solution is to create a custom LLFI Makefile. In this solution, a Makefile is generated with a list of source files and their corresponding list of IR files. Each source file is compiled individually with Clang or Clang++ to produce corresponding IR files. Afterwards, the IR files are linked together using the LLVM linker to produce a single IR file. The implementation of this solution consists of a script to generate the Makefile, followed by a Make call to build the Makefile. A custom Makefile would easily enable users to set custom compiler flags and compiler options. Makefiles provide another advantage; they allow partial compilation of project files such that only the modified files are built each time. Despite these advantages, Makefiles remain platform-dependent, which requires environment settings to be set explicitly.
3.1.3 Proposed Solution 3

The third proposed solution is to create a custom LLFI CMakeLists.txt file. CMake is a cross-platform Make system that reduces overhead associated with directive and environment settings in Makefiles [3]. This solution involves three steps for the compilation process: construction of the CMakeLists.txt, generation of the appropriate Makefile and building the Makefile. The automated construction of the CMakeLists.txt could be implemented as a project file parsing script, similarly described in Proposed Solution 2. CMake is able to discern factors such as users' platforms and source file types to call the applicable compiler along with native settings. CMake offers two main advantages: support for cross-compilation and automatic detection of project compilation flags. However, CMake does not offer a forthright method to input custom compiler flags.

3.1.4 Preferred Solution

The proposed solutions are evaluated with respect to scope, purpose, requirements. The first solution poses a significant drawback in efficiency when encountering large project files. While the third solution offers cross-platform support, LLFI is designed to operate on Linux systems. Instead, the most desirable solution is one that best integrates with the existing system with the flexibility of accommodating user options. As a result, Solution 2 is the preferred solution.

3.2 Solution Design

The following section describes the implementation of the preferred solution in both the command line tool and the GUI.

3.2.1 Command Line Tool

Preceding the development of the GUI enhancement for project compilation, a command line tool was first established to generate the Makefile for an input project file. The Makefile would be run with “Make” to directly generate the IR file. Figure 2 compares the proposed system architecture with the pre-existing system.

To implement the selected solution, a new python executable script named “GenerateMakeFile.py” was developed under the $LLFI_HOME/llfi/tools directory. The executable python script is included under Appendix A. The GenerateMakeFile script is based closely on the original CompileToIR script, with the key difference that the former produces a Makefile for projects and the latter generates an IR file.

As shown in Figure 3, GenerateMakeFile accepts the following parameters as input options: readable, verbose, debug, all, flags, output file name. Examples of use are provided below.
GenerateMakefile takes project source files as input and generates a single Makefile

Usage: GenerateMakefile [OPTIONS] <source files>

List of options:
- -o <output file>: Intermediate representation (IR) output file
- --readable: Generate human-readable output file
- --verbose: Show verbose information
- --debug: Enable debugging symbols
- --all: Compile all source code files in the directory.
- --flags: Specify additional compiler flags
- --help(-h): Show help information

Figure 2: New System Design of the Command Line Tool

Figure 3: Command Line Options
This command creates a Makefile, which compiles all valid C/C++ source files in the current folder directory into a readable IR file named "output.ll".

$ GenerateMakeFile --readable --all -o output.ll

This command creates a Makefile, which compiles the two indicated source files in the current folder directory into a readable IR file named "output.ll".

$ GenerateMakeFile --readable -o output.ll source1.c source2.c

This command creates a Makefile, which compiles all valid C/C++ source files in the current folder directory into a LLVM bytecode file named "output.bc".

$ GenerateMakeFile --debug --all -o output.bc

This command creates a Makefile, which compiles all valid C/C++ source files in the current folder directory into a readable IR file named "output.ll" along with two additional compiler options to generate complete debug information and optimize the build for speed.

$ GenerateMakeFile --debug --all --flags gOt -o output.ll

### 3.2.2 GUI

The GUI portion of the solution is designed as a natural extension of the command line tool. With the overarching objective of closer integration between the two interfaces, the command line scripts have been designed to allow the GUI to directly execute them. With this infrastructure in place, development on the GUI is focused instead on UI design to maximize the user experience.

To preserve users’ workflow on the GUI, the implemented interface extension was designed to minimize user impact. First, the menu tab under “File” was extended to add the option of “Import Project Folder”. When the option is selected, an open project file dialog, shown in Figure 4, prompts the user to select a project folder. Once a project folder is selected and opened, the GenerateMakeFile script is automatically called to create the Makefile. By default, the Makefile is generated under the selected project folder. This feature is implemented under the onClickOpenProject() event in the GUI Controller class, presented in Appendix B.
The generated Makefile is opened automatically by the LLFI GUI editor. The editor permits users to make manual adjustments to the Makefile, which is particularly useful for configuring compiler flags. To successfully build the project into IR code, the Makefile must be explicitly selected on the left side bar under the current list of files before pressing the “Compile to IR” button. Figure 5 demonstrates the selection of the Makefile. The “Compile to IR” button will call Make to build the project file if the Makefile is selected. Otherwise, it will build the source files separately. This feature is implemented under the onClickCompileToIr() event in the GUI Controller class, exhibited in Appendix B.
Figure 5: Makefile Selected Prior to Project Compilation

Upon a successful build to IR format, the corresponding IR file will be generated in the same project folder. This new IR file must be manually imported into the LLFI GUI workspace. Once the IR file is imported, it must be explicitly selected before the instrument button is clicked. Figure 6 illustrates the GUI display upon loading the IR file. The instrumentation process produces LLFI executables, which are ready to be profiled for fault-injection.
3.3 Design Assumptions

There were several design assumptions made in the development of the project compilation feature.

(i) Existing Makefiles are replaced in the project directory.

(ii) Project folders have a flat hierarchy of source files.

The GenerateMakeFile script creates a Makefile that builds source files into LLVM IR. The Makefile is generated in the selected project directory, replacing any existing Makefile in the project. Makefiles do not generally adhere to a standard format, which makes them difficult to parse. Instead, a new custom Makefile is generated by the script.

The GenerateMakeFile script assumes that all source files reside directly under the project folder. As a result, any source files organized into subdirectories are ignored during the creation of the Makefile.
4 Analysis of Usability

Software usability is defined as a qualitative measure at which users feel comfortable with the workflow of the application [4]. As this feature is integrated into an existing tool, usability of this application is a paramount factor in evaluating the viability of the design. The subsequent factors will be used in assessing usability: efficiency, responsiveness and integration with the existing workflow. Efficiency will be measured through the CPU execution time of an operation. Responsiveness will be qualitatively evaluated by the ability of the application to respond to a user request during the execution of a process. Integration with the existing workflow will also be evaluated qualitatively.

4.1 Benchmark Applications

Two industry benchmarks were used to evaluate the usability of the implemented solution. These benchmarks were Parboil and SPEC CPU$^\text{TM}$ 2006. While Parboil benchmark applications are considerably lightweight, SPEC CPU$^\text{TM}$ 2006 benchmark applications are specifically CPU intensive. The analysis below is performed by running the LLFI GUI on a 64-bit Ubuntu Linux machine.

4.1.1 Parboil

The Parboil benchmark applications are an assortment of computing applications used to study compilers and computing architecture [5]. They include a range of typical scientific and commercial applications. Each benchmark application comes in several implementation forms: base, Cuda-base, Ocl-base. Base implementations are written for regular computing architectures, while Cuda-base and Ocl-base work with specific CPU/GPU architectures. The Parboil benchmark applications are provided by the Impact Research Group at the University of Illinois at Urbana-Champaign. For the purpose of analyzing the performance of the implemented solution in the LLFI tool, the Parboil benchmark was chosen due to its relatively lightweight applications.

The ensuing analysis is based on the Sum of Absolute Differences (SAD) benchmark application in its base implementation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Execution Time</th>
<th>Responsiveness</th>
<th>Integration with Existing Workflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateMakefile</td>
<td>10 seconds</td>
<td>Fast</td>
<td>Select project folder instead of individual files</td>
</tr>
<tr>
<td>CompileToIR</td>
<td>3 seconds</td>
<td>Very fast</td>
<td>No change</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>20 seconds</td>
<td>Moderate</td>
<td>No change</td>
</tr>
<tr>
<td>Profiling</td>
<td>10 seconds</td>
<td>Fast</td>
<td>No change</td>
</tr>
</tbody>
</table>
In order to compile the SAD benchmark application, several modifications were made to the original source files. Particularly, the benchmark source files referenced several local header files as system header files. These instances were changed to local header file references instead. Another required modification was the manual addition of the Parboil system header and source files into the project folder directory.

### 4.1.2 SPEC CPU\textsuperscript{TM} 2006

SPEC CPU\textsuperscript{TM} 2006 benchmark applications are a set of standard industry computing applications, designed for stress testing computing architectures [6]. These processing intensive applications represent larger industry projects, which tests the scalability of project compilation in LLFI. The SPEC CPU\textsuperscript{TM} 2006 benchmark is a proprietary tool provided by the Standard Performance Evaluation Corporation. The analysis below will use the bzip2 benchmark application, which is an implementation of the Burrows–Wheeler data compression algorithm.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Execution Time</th>
<th>Responsiveness</th>
<th>Integration with Existing Workflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateMakefile</td>
<td>10 seconds</td>
<td>Fast</td>
<td>Select project folder instead of individual files</td>
</tr>
<tr>
<td>CompileToIR</td>
<td>5 seconds</td>
<td>Very fast</td>
<td>No change</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>5 minutes</td>
<td>Slow</td>
<td>No change, but speed is significantly reduced.</td>
</tr>
<tr>
<td>Profiling</td>
<td>30 seconds</td>
<td>Moderate</td>
<td>No change</td>
</tr>
</tbody>
</table>

### 5 Results and Discussion

The objective of this project is to extend the LLFI tool to permit compilation of entire projects. This objective has been achieved and verified through the use of benchmark applications.

The results of the usability analysis show that users can comfortably use existing operations in the LLFI tool on moderately sized projects, with minimal impact to their workflow. However, the instrumentation step continues to be a bottleneck in the fault-injection process. As seen in Table 2, the instrumentation process for large projects such as SPEC CPU\textsuperscript{TM} 2006 is slow and the responsiveness is poor as the screen is often locked during its execution.

While benchmark applications were suitable for verifying technical aspects of the solution, the principal goal of analyzing usability of the tool remains a challenge. To apprehend the
overall usability of the tool, the complete workflow of a regular user must be fully understood. The implemented solution aims to integrate effortlessly with this workflow.

6 Conclusion

The LLFI fault-injection tool has been extended for project compilation through both the command line interface and the GUI. This extension is important as many industry applications are organized as project files rather than singular code files. This enhancement provides the opportunity for industry users to inject faults into industry typical applications.

Several alternative solutions were proposed and evaluated with usability in mind. The solution to create a custom LLFI Makefile was chosen due to its intuitive integration with the pre-existing design as well as its flexibility in incorporating user specific options. The implemented solution consists of a script to generate a custom Makefile, followed by a direct call to build the Makefile. This script is either directly executed through the command line or through the project import menu in the GUI.

Usability of the implemented design was evaluated with industry standard benchmark applications, focusing on the factors of runtime duration, responsiveness and workflow integration. Understanding the user’s workflow remains a major challenge in building an intrinsic design.

7 Future work

With the LLFI tool extended to compile entire projects, two further enhancements should be considered.

- Compilation of project files with multiple directories of source files
- Selective compilation of project modules

The implemented solution operates with the assumption that the source files are organized under a flat hierarchy. However, many projects have a hierarchical structure of source files. A possible solution to this problem could involve the recursive creation of Makefiles in each of the subdirectories and the extraction of shared elements into a Makefile.common file [7].

LLFI presently requires an entire project to be compiled into IR prior to performing fault injections. With the new infrastructure for project compilation in place, featuring module based fault-injections will increase the tool’s efficiency and effectiveness for large projects.
References


Appendix A

GenerateMakefile.py

```python
#!/usr/bin/env python3

""
%(prog)s takes project source files as input and generates a single Makefile

Usage: %(prog)s [OPTIONS] <source files>

List of options:

-0 <output file>: Intermediate representation (IR) output file
--readable: Generate human-readable output file
--verbose: Show verbose information
--debug: Enable debugging symbols
--all: Compile all source code files in the directory.
--flags: Specify additional compiler flags
--help(-h): Show help information
""

import sys, os, subprocess, tempfile
script_path = os.path.realpath(os.path.dirname(__file__))
sys.path.append(os.path.join(script_path, '..', 'config'))
import llvm_paths
import re
import glob

prog = os.path.basename(sys.argv[0])

llvmlink = os.path.join(llvm_paths.LLVM_DST_ROOT, "bin/llvm-link")
clang = os.path.join(llvm_paths.LLVM_GXX_BIN_DIR, "clang")
clangxx = os.path.join(llvm_paths.LLVM_GXX_BIN_DIR, "clang++")
basedir = os.getcwd()

fname = 'Makefile'
newline = '\n\n'
indent = '\n\t'

options = {
    "o": "a.out",
    "sources": [],
    "readable": False,
    "debug": False,
    "verbose": False,
    "all": False,
    "flags": []
}
```

14
def usage(msg = None):
    retval = 0
    if msg is not None:
        retval = 1
        msg = "ERROR: " + msg
        print(msg, file=sys.stderr)
        print(__doc__ % globals(), file=sys.stderr)
        sys.exit(retval)

def verbosePrint(msg, verbose):
    if verbose:
        print(msg)

def parseArgs(args):
    global options
    argid = 0
    while argid < len(args):
        arg = args[argid]
        if arg.startswith("-"):
            if arg == "-o"
                argid += 1
                options["o"] = os.path.join(basedir, args[argid])
            elif arg == "--readable"
                options["readable"] = True
            elif arg == "--verbose"
                options["verbose"] = True
            elif arg == "--debug"
                options["debug"] = True
            elif arg == "--all"
                options["all"] = True
            elif arg == "--flags"
                argid += 1
                while argid < len(args) and not (args[argid].startswith('('-1)):
                    options["flags"].append(args[argid])
                argid -= 1
            elif arg == "--help" or arg == "-h"
                usage()
            else:
                usage("Invalid argument: " + arg)
        else:
            options["sources"].append(arg)
            argid += 1

    if len(options["sources"]) == 0 and options["all"] == False:
        usage("No input file(s) specified.")

def selectCompiler(sourceFiles):

for inputFile in sourceFiles:
    if inputFile.endswith(".cpp"):
        return clangxx
    return clang

#Build the header for the Makefile
def initializeMakefile(sourceFiles):
    with open(fname, 'w') as fout:
        fout.write('CC=' + selectCompiler(sourceFiles) + '\n')
        fout.write('LINKER=' + llvmlink + '\n')
        fout.write('OUTPUT=' + options['o'] + '\n')

        cflags = ['-w', '-emit-llvm']
        lflags = ['-o', '${OUTPUT}']

        if options['readable']:
            cflags.append('-S')
            lflags.append('-S')
        else:
            cflags.append('-c')
            lflags.append('-c')

        if options['debug']:
            cflags.append('-g')

        if options['flags']:
            additionalFlags = ['- ' + flag for flag in options['flags']]
            cflags += additionalFlags

        fout.write('CFLAGS=' + ' '.join(cflags) + '\n')
        fout.write('LINKERFLAGS=' + ' '.join(lflags) + '\n')

    #Define the body of the Makefile
def constructMakeFile(sourceFiles):
        objList = []

        if options['debug']:
            fileextension = '.bc'
        else:
            fileextension = '.ll'

        with open(fname, 'a') as fout:
            fout.write('SRCDIR_OBJS=')

            for codeFile in sourceFiles:
                objFile = re.sub('\.c$', fileextension, codeFile)
                objList.append(objFile)
                fout.write(objFile + ' ')

            fout.write(newline)
            fout.write('build:')
```python
fout.write('$(SRCDIR_OBJJS) ')

fout.write(indent)
fout.write('$(LINKER) ')
fout.write('$(LINKER_FLAGS) ')
fout.write('$(SRCDIR_OBJJS) ')

for codeFile in sourceFiles:
    objFile = re.sub(r'\.*\$', fileextension, codeFile)
    fout.write(objFile + ' : ' + codeFile)
    fout.write(indent)
fout.write('$(CC) $(CFLAGS) ' + codeFile)
    fout.write(newline)

fout.write('clean: ')
fout.write(indent)
fout.write('rm -rf *ll *bc')
fout.write(newline)

def main(args):
    parseArgs(args)

    if options["all"]:  # Read all C/C++ files in the project directory,
        filetypes = ("*.c", "*.cpp")
        sourceFiles = []
        for files in filetypes:
            sourceFiles.extend(glob.glob(files))
    else:
        sourceFiles = options["sources"]

    initializeMakefile(sourceFiles)
    constructMakeFile(sourceFiles)

if __name__=='__main__':
    main(sys.argv[1:])
```
Appendix B

Controller.java - onClickOpenProject()

```java
@FXML
private void onClickOpenProject(ActionEvent event) {
    Parent root = null;
    fileCount = 0;
    Stage stage = new Stage();
    DirectoryChooser dirChooser = new DirectoryChooser();
    dirChooser.setTitle("Open Project Folder");
    File folder = dirChooser.showDialog(stage);

    if (folder != null) {
        System.out.println(folder);
        currentFilePath = folder.getAbsolutePath();
        System.out.println("currentProgramFolder: " + currentProgramFolder);
        String command = llvmbuildPath + " tools/compileProjtoIRClang --readable --all -o " + folder + "/" + folder.getName() + ".ll";

        try {
            System.out.println(command);
            Process p = Runtime.getRuntime().exec(command, null, folder);
            p.waitFor();
            p.destroy();
        } catch (Exception e) {
            System.out.println(e);
        }

        File[] list = folder.listFiles();

        for (File file : list) {
            String ext = getFileExtension(file);
            System.out.println(file.getName());
            if (file.getName().equals("Makefile"))
                openFile(file);
            else if (ext.equals("c") || ext.equals("cpp"))
                openFile(file);
        }
    }
}
```
private void onClickCompileToIr(ActionEvent event) {
    Parent root;
    try {
        boolean flag = false;
        console = new ArrayList<String>();
        tabBottom.getSelectionModel().select(profilingTab);
        String cmd = "echo $llfibuild";
        // Delete the old .ll file
        ProcessBuilder deleteCmd = new ProcessBuilder("/bin/tcsh", "−c", "$rm " +
            currentProgramFolder+"/"+currentFileName+".ll");
        Process delete = deleteCmd.start();
        delete.waitFor();
        delete.destroy();

        if (currentFileName.equals("Makefile")) {
            String command = "make";
            File folder = currentFilePath;
            try {
                Process p = Runtime.getRuntime().exec(command, null, folder);
                BufferedReader in1 = new BufferedReader(new InputStreamReader(p.getErrorStream()));
                errorTextArea.clear();
                errorString = new ArrayList<>();
                while ((line = in1.readLine()) != null) {
                    console.add(line+"\n");
                    errorString.add(line+"\n");
                }
                in1.close();
                p.waitFor();
                p.destroy();
            }
            catch (Exception e) {
                System.out.println(e);
            }
        } else {
            String command = llfibuildPath+" tools/compileToIR —debug —readable —o "+
                currentProgramFolder+"/"+currentProgramFolder+".ll "+
                currentProgramFolder+"/"+currentFileName;
            console.add("$ "$+command+"\n");
            Process p = Runtime.getRuntime().exec(command);
            BufferedReader in1 = new BufferedReader(new InputStreamReader(p.getErrorStream()));
            errorTextArea.clear();
            errorString = new ArrayList<>();
        }
    }
}
while ((line = in1.readLine()) != null) {
    console.add(line+"\n");
    errorString.add(line+"\n");
}
in1.close();
p.waitFor();
p.destroy();
}
if (errorString.size() == 0) {
    root = FXMLLoader.load(getClass().getClassLoader().getResource("application/compileToIR.fxml"));
    Stage stage = new Stage();
    stage.setTitle("Compiling To IR Result");
    stage.setScene(new Scene(root, 500, 150));
    stage.show();
    // Clear fileContent
    fileContent = new ArrayList<>();
    String line;
    FileReader inputFile = new FileReader(currentProgramFolder+"/"+currentProgramFolder+".ll");
    BufferedReader bufferReader = new BufferedReader(inputFile);
    // Read file contents
    while ((line = bufferReader.readLine()) != null) {
        fileContent.add(line+"\n");
    }
    programInputText.clear();
    // Clear the Text area
    programTextArea.clear();
    // Write file contents to Text Area
    fileContentIndex.setText("Index");
    for(int i = 0; i < fileContent.size(); i++)
    {
        programTextArea.appendText("\t\t"+fileContent.get(i));
    }
    File file = new File(currentProgramFolder+".ll");
    Path path = file.toPath();
    String fileName = path.getFileName().toString();
    for(int n = 0; n<fileNameLists.size(); n++)
    {
        if(fileNameLists.get(n).equalsIgnoreCase(fileName))
        {
            fileNameLists.remove(n);
            fileNameLists.add(fileName);
            flag =true;
            break;
        }
    }
}
if (!flag)
{
    fileNameLists.add(fileName);
    flag = false;
}
items = FXCollections.observableArrayList(fileNameLists);
fileList.setItems(items);
fileSelectMap.put(fileName, fileContent);
instrumentButton.setDisable(false);
profilingButton.setDisable(true);
runtimeButton.setDisable(true);
injectFaultButton.setDisable(true);
tracegraphButton.setDisable(true);
showTraceOutputText.setVisible(false);
} else
{
    root = FXMLLoader.load(getClass().getClassLoader().getResource("application/ErrorDisplay.fxml"));
    Stage stage = new Stage();
    stage.setTitle("Error");
    stage.setScene(new Scene(root, 450, 100));
    stage.show();
}

} catch (IOException e)
{
    e.printStackTrace();
    System.out.println(e);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    System.out.println(e.getMessage());
}