LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-760320

Making Compiler-Based Tools Accessible Online

M. I. Abdalla, C. Liao

October 23, 2018

Disclaimer

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

Making Compiler-Based Tools Accessible Online

Manal I Abdalla
Auburn University at Montgomery
7430 East Dr., Montgomery, AL 36117
Email: mabdalla@aum.edu

Chunhua Liao
Lawrence Livermore National Laboratory
7000 East Ave., Livermore, CA 94550
Email: liao6@llnl.gov

***Abstract*—Compiler-based tools are widely used to analyze and optimize source codes for many purposes, including extracting insights about applications' properties or restructuring code for better performance. However, using compiler-based tools is often difficult due to tedious and error-prone steps to configure and install such tools on various machines. In this paper, we describe a solution to use CGI + Bash to expose compiler-based tools to a web page so users can directly use these tools without installing them.**

1. Introduction

Source code analysis and transformation tools are widely used for various purposes, such as extracting insights from software, automatically re-factoring code, or finding bugs. A popular approach to developing these tool is to leverage existing compilers such as GCC [4], Clang [2], and ROSE [3]. However, a major difficulty for users to use these tools is that their installation processes are complex, tedious and error-prone. Users often give up during the installation process before they get a chance to actually try out the tools.

In this paper, we aim to develop a solution to make compiler-based tools accessible online, without requiring users to download, install or configure these tools. Our solution is to install these tools on an Apache web server and expose them to users via Common Gateway Interface (CGI), a standard protocol for web servers to execute programs. Our paper has made the following contributions:

- We explored the challenges to make compiler-based tools accessible online.
- We made ROSE-based tools accessible online, with adaptive and easy-to-use interface to demonstrate their usage. We call this website as the demo website for ROSE tools. Users can even generate various analysis graphs using their smart phones.
- We implemented some security measures to protect the server, and put down plans for more added security in the future.

In this paper, we will explain the challenges we faced implementing this web interface, and the approach we took in providing solutions. We will also explain the reasoning behind the design based on ROSE's background. Also, we will provide a list of related work and show how our work is different. Moreover, we will discuss future work needed to give the user the full experience of using compiler-based tools.

2. Background

We briefly introduce the ROSE source-to-source compiler framework and it's tools.

Developed at Lawrence Livermore National Laboratory, ROSE [3] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. As shown in Fig 1, ROSE generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for input codes. Sophisticated compiler analyses, transformations and optimizations are developed on top of the AST and encapsulated as simple function calls, which can be readily leveraged by tool developers.

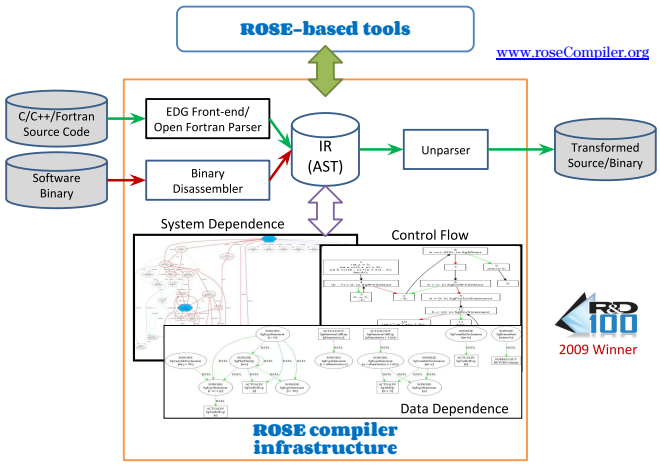

Figure 1. Overview of the ROSE compiler framework

Example program analyses available in ROSE include call graph analysis, control flow analysis, data flow analysis (live variables, def-use chain, reaching definition, alias analysis, etc.), class hierarchy analysis, data dependence

and system dependence analysis, and MPI communication pattern analysis. Representative program optimizations and translations developed with ROSE are partial redundancy elimination, constant folding, inlining, outlining (separating out a portion of code as a function), OpenMP directive lowering, and loop transformations (a loop optimizer supporting aggressive loop optimizations such as fusion, fission, interchange, unrolling, and blocking).

ROSE is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, performance analysis, and cyber-security. ROSE is released under a BSD-style license and is portable to Linux and Mac OS X on IA32 and X86_64 platforms.

2.1. ROSE-Based Tools

ROSE has many tools in various stages of development. We have made the following tools available in this paper:

- identityTranslator: this is the simplest tool built using ROSE. It takes input source files, builds AST, and then unparses the AST back to compilable source code.
- Plugin: with this feature, users can develop their ROSE-based tools as dynamically loadable plugins.
- dotGenerator: this tool generates a simple dot graph from input code.
- dotGeneratorWholeASTGraph: this tool creates a more comprehensive dot graph from an input source file.
- pdfGenerator: this is a tool to generate bookmarked pdf file for larger input code.
- callGraphGenerator: this tool generates static call graphs.
- ASTInliner: this is a tool to replace function callsites with bodies of called functions.
- ASTOutliner: a tool to extract code portions and make them into functions.
- OpenMP Lowering: a special mode of identityTranslator to translate OpenMP input code into parallel code calling OpenMP runtime functions.
- AutoPar: an automatic parallelization tool using OpenMP.
- LoopProcessor: a loop optimization tool to automatically tile, interchange, block, fuse, split loops.

In general, the process of using a ROSE-based tool involves three steps:

1) Specify the input file. That is, locating the source file on which a tool will process. For example input.c.
2) Choose the ROSE-based tool that will perform a unique analysis, translation, or optimization. The use of the tool will do the following:
    - First of all, a function provided by ROSE, named fronted() will be called to parse and generate an abstract syntax tree/ AST of the input.c.
    - Then based on the tool, other analysis/transformation/optimization functions, also provided by ROSE, will be called to transform the AST.
    - Finally, the backend() function is called to unparse the transformed AST to source code again.
3) Output file is generated from the unparsed code and named like, rose_input.c.

3. The Design

For our interface, we wanted something basic, simple and intuitive. We did not want the use of the interface to be a steep learning curve so we used self-explanatory hyperlinks to navigate from one page to another, as shown in Figure 2. At times, button functionality was added to hyperlinks to preserve the simplicity of the design.

Output is also displayed based on it's format. Text-based output is displayed in a text-area box, where as dot graph output is given as hyper-text in its corresponding PNG and PDF formats. The clickable text will open the desired file in a new tab in the browser. Furthermore, when uploading a file, the user will have the option to download the output.

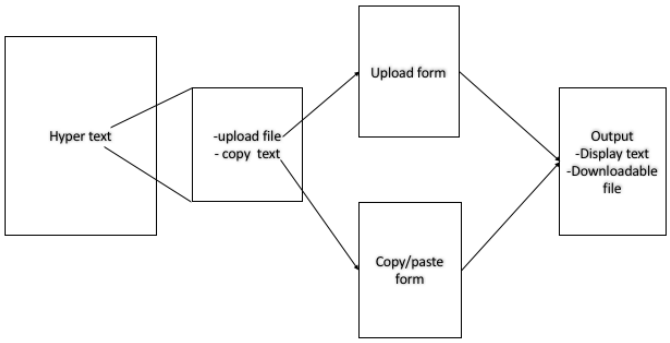

Figure 2. Work flow of the ROSE demo page

3.1. Development Tools

For our interface we used html, Javascript and CGI-Bash scripts. We don't expect complicated interactions from users, so using CGI based scripts for parsing users' data form is an optimal choice for us. The web portion of the interface is built with HTML and JavaScript functions to add functionality and ease of use, and populate forms on-click events.

For this paper, we use Ubuntu's default server, Apache2. Apache2 provides many HTTP server and proxy features that suits the purpose of demonstrating various compiler-based tools.

4. Challenges and Solutions

We faced ROSE-related challenges and also environment-related ones. One challenge was to insure the simplicity of the design and the work flow of the tool. That was solved by adding Javascript functions, which controlled the flow of the interface by submitting forms in the background.

4.1. Dynamic Interface

An important functional goal was to make the interface dynamic. We provided some examples for each tool, but we wanted the users to be able to use their own code. However, this functionality was in a way handed to us by the server-based environment. HTTP servers can handle forms and file uploads among many other features. We utilized the client-server communication channel, and engineered forms with specific data that would allow for the instantaneous processing of the personalized users' input.

4.1.1. The Core Bash Functions. We wanted to provide users with options on the choice of input methods. We implemented the toggle functionality using the showme() function shown in Figure 3.

```
    <!-- This script has two functions. showme(), is a function  to switch betwe
en the input forms, copy/paste or upload. getVars(), is to get the tool name sen
t from index.html -->
    <script>
        function showme(){
            var choice = document.getElementsByName("question");
            var vis = "none";
            var vis2 = "block";
            for(var i=0;i<choice.length;i++){
                if(choice[i].id == "upload" & choice[i].checked){
                    vis = "block";
                    vis2 = "none";
                    break;
                }
            }
            document.getElementById('form2').style.display = vis;
            document.getElementById('form1').style.display = vis2;

        }
```

Figure 3. The javascript function used to toggle between input forms

We initially wrapped each form with div tags, and we assigned an id value and display styling attribute to each div. Then, the showme() function changes the value of the display attribute onclick.

Another dynamically generated view is the help page. First we have a hidden form to submit the tool's name as shown in Figure 4.

```
<!--a hidden form that submit a value of the tool name to dynamically gener
ate help page-->
<form action="/cgi-bin/help_page.sh" method="post" id="manform" target="_bl
ank">
<input type="hidden" name="tool" id="helptool" value="">
</form>
```

Figure 4. The help hidden form

The input value for the tool name will be adjusted based on the selected tool using javascript. The form will be also submitted using javascript onclick as shown in the code snippet from Figure 5.

In order to be able to upload files, we had to specify an encoding type different than the one for the copy form. It

```
Check the <a href="#" onclick="document.getElementById('manform').submit();
">Manual</a> for usage and options.<br>
```

Figure 5. This is how we invoke the help CGI form

means that we need a different encoding method in the CGI form as well. Figure 6 shows part of the decoding process.

```
# This code will take the query string for post method and stor it in POST_DATA
if [ "$REQUEST_METHOD" = "POST" ]; then
    if [ "$CONTENT_LENGTH" -gt 0 ]; then
        read -N $CONTENT_LENGTH POST_DATA <&0
    else
        echo " POST_DATA string is empty \n " >> $LOG_FILE
    fi
else
    echo "form method must be POST \n " >> $LOG_FILE
fi
# the following will write the content of POST_DATA to tmp_input0_nosuffix
(
    echo "$POST_DATA" | while read line
    do
        echo "$line"
    done
 ) > $tmp_input0_nosuffix
# the following will extract the suffix and the options from tmp_input0_nosuffix
linecount=`cat $tmp_input0_nosuffix | wc -l $1 | cut -f 1 -d ' '`
# this will extract the suffix from the file name given in form
suff=`sed -n "2p" "$tmp_input0_nosuffix" | sed -e "s/^M/\n/g" | awk -F";" '{ print $1
suffi=${suff##*.}
suffix=`echo $suffi | cut -f1 -d"\""`
echo "suffix extracted $suffix \n" >>$LOG_FILE
```

Figure 6. Decoding the upload form

4.2. Dot Files

ROSE has many program analysis tools that would parse the given code and generate the Abstract Syntax Tree, AST. Then the ROSE-based tool will generate a dot formatted version of the tree. For this paper, we used the dot command provided by Ubuntu's Graphviz package. We started by providing a PNG and a PDF formats. We wanted to time the script from start, by calling the decoding function, to finish, after we echo the last line of the output. We recorded the times with the PDF and PNG formats, JPEG and PDF and each of the formats by itself. After our testing, we have found the following:

- About 33% of the tool's running time was spent on the dot command generating PNG and PDF formats.
- When generating only PDF format, script's time spent on reformatting was less than 10%.
- When generating only PNG format, script's time spent on reformatting was about 30%.
- The time for generating JPEG and PDF formats, in the worst running case, was a bit shy of 20% of the total script running time.

As a result we changed the available formats for the graph-generating tools to JPEG and PDF.

4.3. Plugin

ROSE developers have implemented a plugin feature in a recent release. It allows users to create their own ROSE-based translators, and then use command line options of ROSE to run it.

The form structure for the plugin feature is different from the one for the other tools, and it was challenging for the following reasons:

- The form needed to read and decode two input files, the plugin itself and the input file to be processed. While this was not a very hard task to do, we needed to make sure there was no confusion for the users.
- The plugin feature is dynamic. In web implementation, that dynamic flow means more forms and high server-client communication. As one of our goals was to maintain the simplicity of the design, we tried to get all needed information in one form.
- Unlike the other tools, the plugin tool creates an executable that runs as a tool on its own. Hence arise the security issue. The security of a web server could be a three-year research on its own. However, we tried to implement some security measures in two ways:

    1) The web server is hosted within an isolated Amazon cloud virtual machine.
    2) The user's input is directed in the background to one directory. Also, alias names are used.
    3) We are implementing a script that would restore the server daily to a specific state. In this case, we make sure we are restoring the server to a malicious-free state.

  With these precautionary security measures, the ROSE demo server is much less vulnerable to cyber threats.

5. Example: User Session

In this section we will walk through a user's session of ROSE Demo Page. The interaction starts when the user go to http://demo.rosecompiler.org.

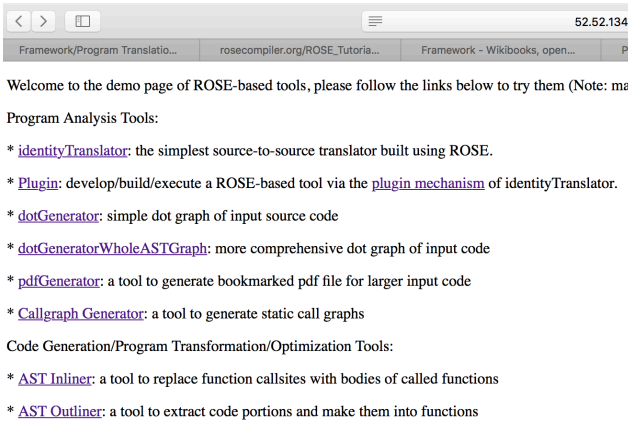

Figure 7. First page of the Rose demo web interface

As you see in Figure 7, we have the tools separated into two groups, analysis tools and optimization tools. The tool name is underlined, indicating its clickable, and has a brief explanation of what it does. Clicking on one of the tool, identityTranslator for example, the user is moved to the next page shown by Figure 8. This part will prompt the user for the information needed to run the chosen tool.

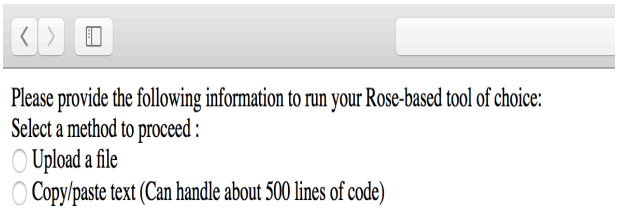

Figure 8. Users have the freedom to choose the method to provide the input file

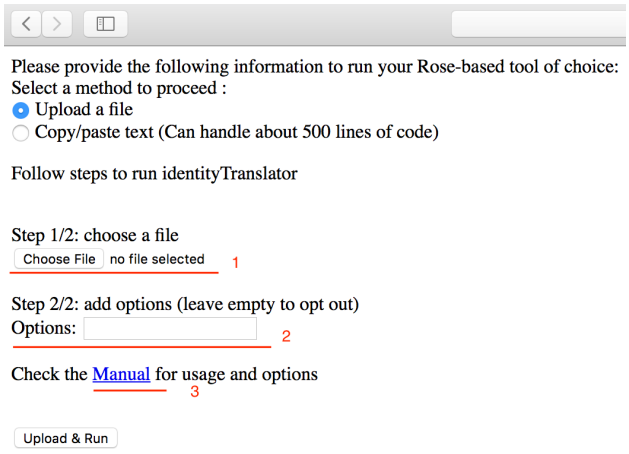

Figure 9. The upload form

Input files are uploaded from the user's local machine as shown in Figure 9. Each of ROSE's tools has its own options for finer tuning and more precise results. A user could always check the manual page to learn these options and their usage. The other choice for input data is copy/paste text. Again, Figure 10 shows that we have the tool's name on top and the added options with the manual page link at the bottom. The user could type in some code, copy/paste or choose an example from the drop down menu. There are clear instructions, that are easy to follow, on what is expected from the user in each each step. Figure 11 shows the manual help page for the selected tool, identityTranslator.

In the final result page shown in Figure 12, we start by stating the facts. That is, we display the code's language, the options added to run with the tool and the input code. After that we let the user know the result of running the command. If the compilation is successful, we show the output result code. Otherwise, the compilation message box will show the error messages for the failed command line.

The plugin tool is a little different. In the description, from the index page, there is a link to a wiki-book page that explains the specifications of the tool. Note that in Figure 14, plugin action and arguments come from the source code users have provided in step 1.

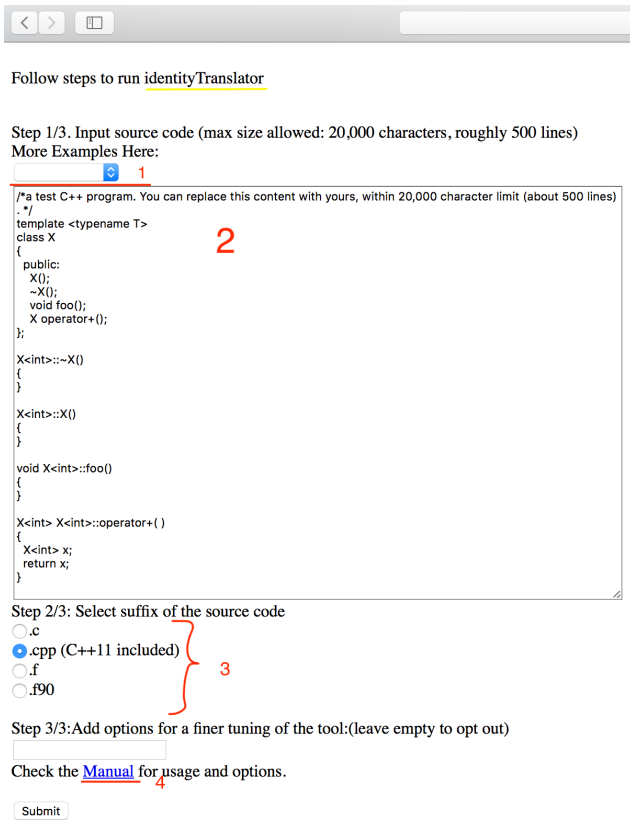

Figure 10. figures/Copy/paste form

6. Related Work

Our goal is to make compiler-based tools accessible to users. One alternative approach available is using a virtual machine image. Though getting the VM image is much easier than installing ROSE from scratch, there are some issues with the approach. The main issue would be the ROSE updates. Since ROSE is still in development mode, that means every time there are some updates we need to create a new VM image. Moreover, the user would need to download a huge VM image files of several giga bytes. Whereas, in our web interface, we have a crone job that updates the ROSE periodically. This way users are not concerned about getting the newest version, they are using the newest version all the time.

Another approach is to provide a terminal access. We surveyed a few web-terminal applications to try and implement a shell terminal within the ROSE demo web interface. The only application that insured some level of security, was Jupyter [1]. However, to implement a Jupiter notebook, most input must be predetermined, and that did not fit the purpose of our application. We wanted users to be able to analyze and optimize their code on the go.

We also checked some of the web based terminal applications like Butterfly and Shellinabox. The problem with these applications is that they offer minimal to no security at all. Basically, to open a shell terminal through the web

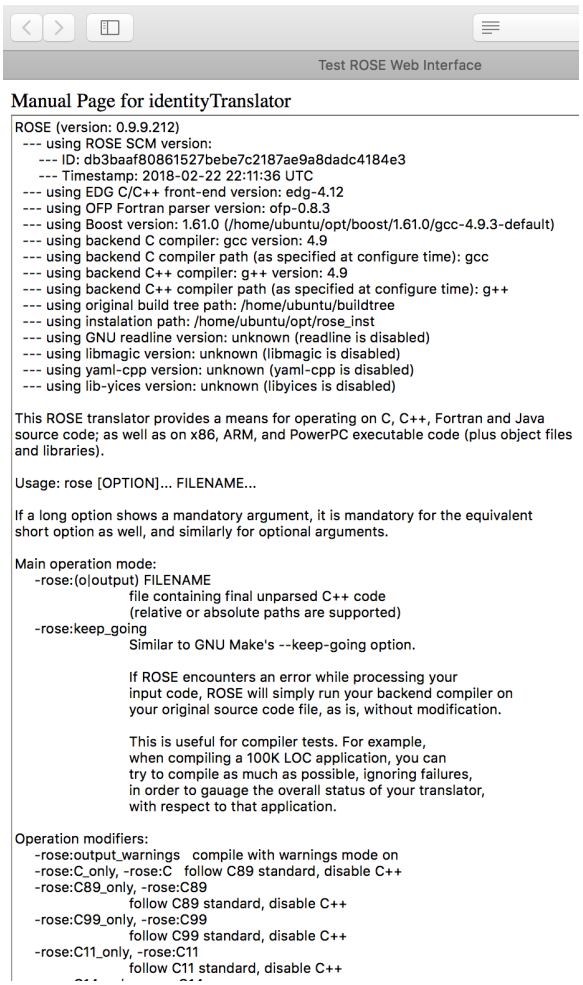

Figure 11. Help page for the identityTranslator tool

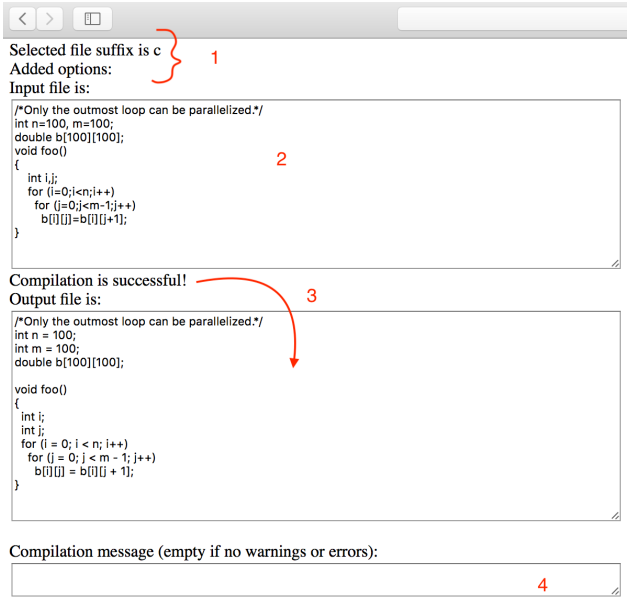

Figure 12. The result of running the Rose tool

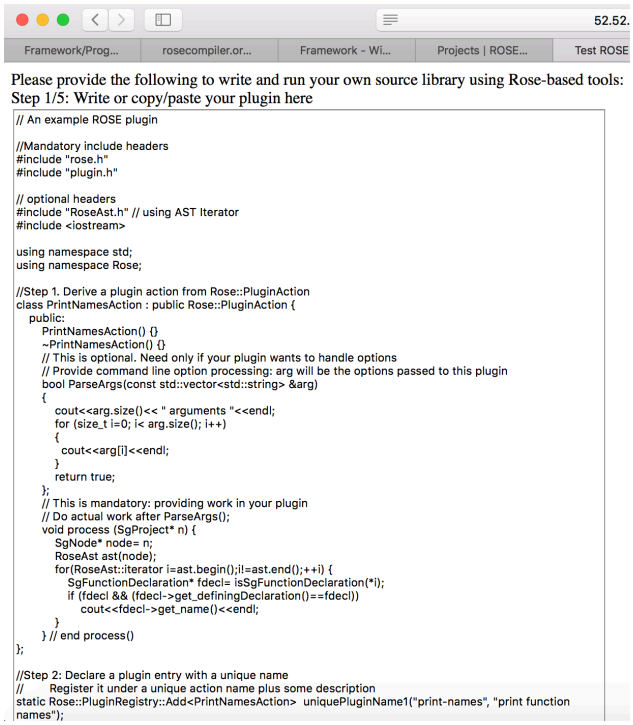

Figure 13. Step 1/5, this is where users enter the plugin implementation

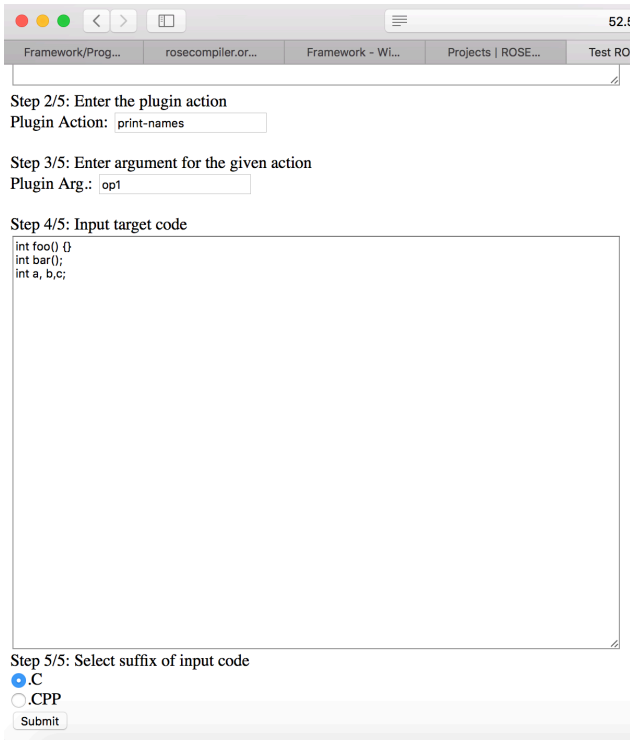

Figure 14. Continue: The plugin form

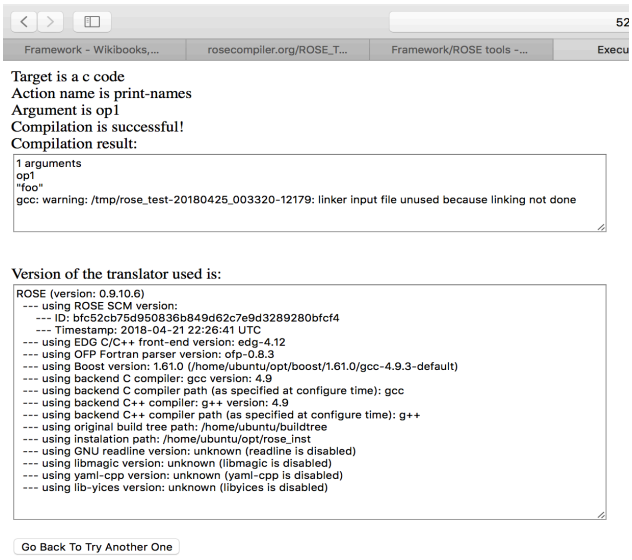

Figure 15. Result of running the plugin

interface, a daemon server connected to the terminal must be running and listening all the time.

Another open-source application we surveyed was Web-Terminal. The problem is it's still in early development stages and as a result it did not integrate well with our environment.

7. Conclusion

In conclusion, we have been able to design and implement an interactive web interface for users to easily try out compiler-based tools, without going through complex installing and configuration steps. This current demo website exposes a number of ROSE-based tools to users. It also gives users the choice to copy/past their programs or just snippets of a program. Moreover, it provides a choice of uploading large input files and provides output in various formats. The interface also supports running user-defined ROSE tools as plugins.

7.1. Future Goals

As the ROSE project is still an active one, there will always be room for improvement. The team is working on more tools and as a result the demo interface will have to grow and expand. Of course, as the user base of ROSE grows, the need for more scalable solution may arise.

7.1.1. Improve Security. We are actually allowing users to create executables and run them. Additional security measures can always added. For example, we need to understand and point out the kinds of threats we are facing. We are interested in adding a multi-layered system that will filter the input code and flag the files that deemed malicious. Finally, we need to decide the course of action. That is when a file is flagged, how should the ROSE demo application proceed.

7.1.2. Provide A Terminal Access. The best way for a user to experience an compiler-based tool in its full glory, is through the command line interface. Because of the insecure nature of the world wild web, we had to limit the functionality of some of the tools in our web interface.

One way to do that would be to implement a custom made ROSE-Shell. This would provide a way for the user to experience the CLI of ROSE without needing to go through the downloading process.

Acknowledgments

Work on the core web interface was supported by the LLNL-LDRD Program under Project No. 18-ERD-006. The additional features of supporting plugins and large file uploading were supported by the DOE Office of Sciences Science Undergraduate Laboratory Internship (SULI) program. Release number: LLNL-TR-760320.

References

[1] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[2] B. C. Lopes and R. Auler. *Getting started with LLVM core libraries*. Packt Publishing Ltd, 2014.

[3] D. Quinlan and C. Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.

[4] R. Stallman. Using and porting the gnu compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer, 2001.