Interactive Exploration of the Executable Linking Process

Andrei Ciprian Aprodu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Bucharest, 060042
Email: andrei.aprodu@stud.acs.upb.ro

Răzvan Deaconescu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Bucharest, 060042
Email: razvan.deaconescu@cs.pub.ro

***Abstract*—Students learn about the existence of the linking process, but few know and understand what it really does. The theoretical explanation often requires a good background in compilers and operating systems, but it can be easier to explain with a proper example that shows every change of the process. While the other available tools are not beginner friendly, ELF Detective is, and by using any binary files, even a personal ones that are well known for the user, it can thoroughly analyse and explain in an educational way the linking phase. This makes it a lot easier to understand without having a very good technical background. This is the tool to use for a more educational perspective of the linking stage.**

I. Introduction

The linking stage is one of the most important topics in computer science and represents a cornerstone for every individual training in this area. While at first is easy to feel like this process is easy to understand, one might find himself to a point where the lack of a good knowledge of this stage led to complications.

The problem is not as obvious as one not knowing what happens during the process, but the lack of a more practical experience, which is hard to gain. Ideally, we could learn about this topic in a less theoretical way, that can form a better background for an individual of any level of expertise.

Doing a manual analysis of the transformations that happen during the linking stage will only lead to a whole of machine code that gets merged. This is not very helpful since the changes are almost unrecognizable, leading to confusion for an unexperienced user.

The project that we present in this paper, ELF Detective, is an educational tool created with the goal of helping students understand the changes that occur in the linking process. It offers an interactive interface where it presents the important data as a comparison between its states in the executable and object files. To find any additional information, the user has to click on any item and it will show the information in an easy to read format that clarifies the ”how’s” and the ”what’s” regarding the changes.

A. Motivation

We base our motivation for this project on the fact that many complex topics, in this case the linking process, are a lot harder to understand without a practical example, and even in the case of one being provided, most often it’s too generic. To our knowledge, there are no educational tools allowing an easy exploration of the linking process, and any other program that might help is not beginner friendly.

B. Objective

Our main objective with this project is to help others understand the linking process better while spending a lot less time. Since there is no other tool that we know of that has this focus, the one we were to create had to be easy understand.

From an education point of view, this project must be able to explain the changes that occur in the linking stage in the easiest way possible so a student of any background can understand it well. This means that all addresses must be explained, any link between the executable and object files is well represented and the wording of the explanation is not too complex.

Our second objective for this project is to be interactive. This means that not only the program will have a good response time, but the GUI will seem straightforward to use. If the user is currently inspecting a project and require more information, he just has to click on it. This works for both symbols and code lines, and the output not only that explains the requested information as best as it can, but it does it for both the executable file and the object file it came from.

Lastly the project must be correct, have a good response time (under a second) and scale well. These topic is covered later on.

II. Related Work

There are many applications that offer a great overview of binary files, but each of them have a different purpose. To our knowledge, ELF Detective is the first of its kind because it can handle ELF files just as well as any other existing program, but its focus is to find connections between files that may

have been changed during the linking stage and to provide an educational insight of these changes. Because of this, it is more relevant that we only present tools that offer similar functionalities.

Presenting the tools of the trade shows the need for a project like the one we present in this paper. These tools are old and require human reasoning to completely understand their output, since they present a lot of information, but in a raw, terminal based format.

The most generic tools that handle ELFs are: **objdump**[1], **readelf**[2], **nm**[3], **addr2line**[4], tools provided by the same package (binutils), which rarely gets an update. These tools are revised only for bug fixes or tweakings of the display format. As a result of their age they're heavily used and well known, but due to the shortage of updates over the time they became outmoded. They have a minimum level of interactivity, which means that based on the what we ask for, they answer with some raw data and nothing more. This is not really an issue for more advanced users, but they are less appreciated by anyone that wants to get better at understanding ELFs and, even if you have a good knowledge of how they work, you still need to analyse the output and connect the dots in between to find the answer you were looking for. These tool do not allow to inspect multiple files at once and will show incomplete information when it comes to external symbols and even some function calls.

Alongside these generic tools, there are plenty more specific programs that work very well, but only implement a specific subset of functionalities. These are tools with a high level of interactivity, but they do not serve any educational purpose since they are mainly used for debugging, cracking and reverse engineering. A few of these tools are:

- Hex editor: **Bless**[5], **wx Hex Editor**[6]
- Disassembly and reverse engineering: **Radare 2**[7], **IDA**[8]

III. Requirements and Design Principles

ELF Detective is a tool that allows to interactively explore the linking process based on executables and the object files that build them. As the project name says, it currently only works with ELF files, which means it's a Linux only project. It provides a highly interactive GUI that is easy to use and understand, making the process of analysing files a lot easier. Further in this chapter we will be present how the program is implemented and the reasoning behind it.

[1]https://sourceware.org/binutils/docs/binutils/objdump.html
[2]https://sourceware.org/binutils/docs/binutils/readelf.html
[3]https://sourceware.org/binutils/docs/binutils/nm.html
[4]https://sourceware.org/binutils/docs/binutils/addr2line.html
[5]http://home.gna.org/bless/
[6]http://www.wxhexeditor.org/
[7]http://radare.org/r/
[8]https://www.hex-rays.com/products/ida/

At the moment ELF Detective is a 64-bit program only due to library dependencies, but it can work with binaries compiled for 32-bit as well. The term ELF is used in a generous manner since it can only currently work with x86 ELFs.

A. Building Blocks

In order to present how the project works and behaves as a whole there are some functionalities that need to be explained individually for the sake of clarity. In this section we will present the building blocks of the application in unique subsections in hope of a better understanding of the project implementation.

1) ELF File Representation: The inner representation of the ELF files is the core of this project. To be able to keep this files in an organized manner, the **Binary File Descriptor**[9] was used to open, parse and extract data that otherwise would be very difficult to do. The ELF file as a unit is a wrapper over the BFD format so that any required data can be stored. By caching any frequently used data, it saves a lot of calls to the **BFD** library, where most of them translate into system calls, and therefore saving resources.

2) Disassembly Module: The purpose of this module is to translate the machine code into assembly language and to offer an easy to use API for the GUI. It focuses mainly on the .text section since only functions need to be disassembled. In order to do so, this will find a disassembling function in the system with the help of **libopcodes**, that will handle the translation.

The output of this module is a complete list of functions from the requested file. Each of these functions contain a list of code lines. A code line has information about the offset, the disassembled line of code, the opcodes and the symbol it references if it does and how it does it. This vector of functions is added to the ELF file representation so that it can be easily accessed by the GUI, without interrogating this module.

3) Symbol Analysis Module: In order to have an accurate output, without any useless symbols, this module analyse all the files currently open in the project. The simplest way to describe this functionality is as a comparison between the symbols found in the object files and those in the executable file. This means that all the relevant symbols (there's no need for section names or any auxiliary compiler variables) used in the object files will be gathered and filtered for duplicates. If we find any symbol that's used in more than one file, it means that this symbol was defined in only one of those files and for the others it is unknown before link time. For this project there's need only for keeping the defined symbol, and data of where it is not known. After obtaining the symbol collection from the object files, the executable is parsed as well and it adds new data for each symbol that is already in the collection.

[9]https://sourceware.org/binutils/docs-2.26/bfd/index.htmll

The output is this collection of symbols and all the data regarding them, which means that all the changes that happened during the relocation and address binding phases can be shown. It is represented in a Hash Map so a symbol can be easily found when asked by the GUI.

4) Graphical User Interface: The GUI is what makes this project unique and gives it a purpose. It is implemented with Qt[10] (initially version 5.5, now updated to 5.6), a C++ framework, which allows easy integration of a project with a graphical interface, with easy to catch events and drag and drop interface design. This framework allowed the program to be highly interactive and easy to use. The GUI shows the executable and the object files (placed in tabs) side by side for easy comparison. Each ELF file have two accordion items, 'Data' and 'Functions', containing information from both the disassembly and the symbol analysis modules. The 'Data' page holds information about variables and symbols that are unknown at link time (e.g. a call to printf). By clicking on any item, it will select the tab of the corresponding object file, it will highlight the symbol declaration that tab and it will show every information about the symbol, from both executable and object files. The 'Functions' page shows a list of expandable function names. By expanding a function, it will show the disassembly code. The clicking behaviour is similar to the 'Data' page one. The code lines shown here can be changed into opcodes by clicking on the check box 'Hexcodes'.

IV. Implementation Details

With all the modules of the project explained, we will start describing the program as a whole. We chose C++ for implementing ELF Detective due to its compatibility with low level C libraries, as well as its offering of high level data structures and libraries. The core data structure is the **ELFFile** which is the representation of the ELF files.

Since most of the tools that inspect bianry files are open source, we decided to base our work on one of them, **objdump** which led to some of its code being used for the implementation of the disassembly module. Also, objdump's code was our main source of learning how the bfd library works, since its documentation is not very helpful.

To inspect a project with ELF Detective is pretty easy. You first have to add an executable file (CTRL + E) and the linkable files (CTRL + O), it is not required to add all of them since it can work an a subset as well. After everything the user added all the files and it initialised the internal data structures, the project can be ran (CTRL + R) and all the buttons allowing to add more files will be disabled.

Symbol analysis module is the first to run after the initialisation of these structures. It will not modify the state of the files, but merely interrogate them to gather symbol information. After its execution, it will have a **Hash Map** of **Symbols**, a class that keeps information about symbols, that will be used by the GUI functionality.

The disassembly module is the next to run. It will add to each file its corresponding vector of **Function**s that holds any needed data, including the code. The GUI expects this vector to be initalised in all the files, after the project ran.

The GUI inspect functionality expects that all the data was gathered and it can be found. For almost every task, it inspects the Hash Map containing symbols or the ELFFile's vector of functions to obtain the required information to display.

By clearing the project (CTRL + C), it will release all the stored data, free the memory and delete the specific GUI elements. The buttons that allow to add more files and the run button will be active again, allowing a new project to be inspected.

The tools used for the implementation of this project will be presented in the next section.

A. Implementation Tools

This section will cover the libraries and frameworks used for implementation. Some of them have already been mentioned while explaining the building blocks, but a better understanding of them is needed by anyone reading this paper.

1) Binary File Descriptor library: The BFD package is part of the **GNU Project**[11] and represents a mechanism that offers binary file manipulation. It supports multiple file formats on many processor architectures. Its main goal is to present a common abstract view of the binary files. For this to happen, it needs to translate from binary data to the abstract view and the other way around. It takes into consideration details like the endianness, the architecture for which the file was compiled for, address arithmetic etc. This is what they present as "BFD is split into two parts: the front end, and the back ends (one for each object file format)."[12].

For this project, the only thing of interest was the front end of BFD. This provides an API that allows easy access to symbols, sections, relocation entries and all of their characteristics. This together with **libiberty**[13], a library that will not be presented separately, allowed the clean and secure access to any properties of the binary file that were needed.

Alongside **libopcodes**, another library that will not be presented here since it usually comes with the operating system, BFD can be used to parse the disassembly data received from the opcodes library.

2) ELF Headers: We found these ELF headers in the project files of **objdump** and they represent part of the back ends of BFD. There's one header for each supported architecture the installs usually configure to use the corresponding file.

[10]https://www.qt.io/qt-framework/

[11]https://www.gnu.org/gnu/thegnuproject.html

[12]https://sourceware.org/binutils/docs/bfd/Overview.html

[13]https://gcc.gnu.org/onlinedocs/libiberty/Overview.html

For this project, we used only the most generic headers, but all the other ones are available as well. This means that the project can be extended for any architecture that is supported by BFD.

3) Qt: Qt is a cross-platform framework used for GUI implementation. Qt extends C++ with signals and slots that simplify how events are handled, making it easier to create core functionalities of the interface. By using **Qt Creator IDE**[14] the development of a GUI becomes a lot easier. It has a drag and drop interface that allows the user to add and arrange items easily, and by right clicking on any widget one gains access to the slots, descriptions, tool tips and even widget design.

V. Feedback and Evaluation

The tests were conducted using randomly generated C projects of up to 20 object files. In the next sections, these tests will be presented in a top - down manner, from the look and feel of the program to exact measurements of the functionality.

Besides our tests, some other people, colleagues and some bachelor students from the Faculty of Automatic Controls and Computers, helped us to have a beta testing phase in which they tested the project in any way they wanted to, using the test generator or running their own projects. The feedback that was obtained from them was very helpful, and so, we were able to make the program run better and feel easier to use.

A. Test Generator

In order to eliminate the human error out of the equation, it felt like having randomly generated tests that use the items of interested for this project, external symbols and functions calls to other object files, would be the best solution. This test generator is a Python[15] script that generates C files with random amount of symbols, from which a random amount are functions and the rest are variables. The variables have a random keyword between **const**, **static**, **extern** or none.

B. Look and Feel

ELF detective aims to be a highly interactive project and this requires an interface that looks friendly and easy enough to use so that it can be learned at first use. To reach this goal, each file can be seen on its own, the executable has it's own panel and the object files are separated into tabs, the data is partitioned into expandable items, the hex code check box is positioned in an easy to view spot, and the buttons show as icons that represent their functionality well. For each of those button there is a small explanation pop-up showing at hover, and all of them have keyboard shortcuts so it can be easier to use.

The output shows at the bottom of the GUI in two columns representing the executable file and the active object file. It presents each line of output in both columns showing the data from each of those files, and has just enough information to make it clear and not show too much text.

C. Correctness

The most important part of this project was to respect all the specifications. The main requirements were for the program to be interactive, to list all the symbols in the project, to allow inspection of functions bodies and to provide insight of what changes during linking for any of this information.

With regard to making sure that all of this data is correct, the best alternative was to cross reference it to what any other tool of the trade can display. By generating random tests and comparing the output of ELF Detective to what the other tools show, it confirmed that all the data shown by this project was in fact correct, and every missing data was intentional.

D. Evaluation

In order to evaluate the project in a proper manner we have used different projects for up to 20 object files and the corresponding executable. As stated earlier, all of these projects were randomly generated and had the focus mainly on the use of external variables and function calls to different files.

For a better insight, the response time of **objdump -dt**, -d disassembles the executable sections and -t shows the symbols, was tested, with and without any output. The response times of ELF Detective were obtained by using the C++ clock()[16] function which allows to compute the total time spent between 2 calls of this function. The time was tested only for the "Project Run" command, which means that the files were already added into the program. For objdump, **time**[17] command was used. The data that shows on the chart was collected by testing each command 10 times and computing the average of the real time.

The testing environment was an Ubuntu 14.04 virtual machine with 2 GB of 1600 MHz DDR3 RAM and 2 x 3.40 GHz CPU cores.

The results showed that ELF Detective is scaling well with the number of files compared to objdump. It seems that the bottleneck of objdump is presenting the output, but the functionality it's faster than ELF Detectives, which is to be expected. Considering that ELF Detective parses the raw data and shows it in an interactive GUI, having such a low response time that seems to be instant for the human eye, makes the program look as interactive as it should.

The next important thing that needed to be tested was how long it takes for each module to run. This was tested for a

[14]https://www.qt.io/ide/
[15]https://www.python.org/
[16]http://www.cplusplus.com/reference/ctime/clock/
[17]http://linux.die.net/man/1/time

20 object files and one executable file project, and it was run multiple times. Same as before, this was tested after running the project since this is the moment when all these module come in play, so 100% of the chart means the full run of the project.

Initializing the files and running a symbol analysis module takes very little time. They only initialize the internal structure of each file and parses it, finding every relevant symbol. The bottleneck here is the disassembly module. It takes more than 72% of the time, because, after receiving the raw data it has to do some string manipulations in order to offer easy access to any information and it has to create some inner connections between some code lines. The GUI take a fair amount of time as well, but this is to be expected. There is a lot of data that needs to show up in the GUI, some that it's not even available at the default screen, e.g. the instruction opcodes, and there are a lot of connections that need to be solved "behind the scenes".

The last thing that needs to be tested is the GUI response time when clicking on a symbol or a line of code. For each category, the best and the worst case scenario were found. The best case scenario is when the inspected item is in one of the first object files, this object's tab is active and is one of the first symbols. The worst case is when the item is part of the last object files, the object's tab is not active and is one of the last symbols to be defined. While inspecting the code lines, the cases are similar, only that they apply to their parent.

The best case scenarios are closed to being constant. This was to be expected since there is very little work to do and it's not affected by the number of files or symbols. The worst case scenario seem to scale well with both number of files and symbols.

E. Feedback

As stated earlier in this chapter, the project went through a beta testing phase. Some of the received feedback that was used to make the project better will be presented in this section.

It was said that it feels annoying that the user had to add the linkable files one by one, and that was true for projects with a lot of object files, and now the project can handle multiple file selecting while adding linkable files to the project. Another similar thing that felt out of place was that after adding a file, the file browser would not remember the last directory that was used. Since at that moment the user could only add one file at a time, this didn't feel as it should. Now the file browser correctly remembers the last used directory.

Another thing that was stated is the fact that due to human error an object file view can be closed, but not reopened. The suggestion was to allow adding files after the projects ran to solve this issue, but this could easily add more ways to wrongfully use the project. To fix this issue, the object file tabs are no longer closable without cleaning the project.

Overall, the feedback we received was mainly positive, the testers finding this project easy to use and understand.

VI. Conclusion and Further Work

In this work we have developed an interactive tool that can explore the depths of the linking process. Every requirement was met and surpassed, the project offering more than it was asked for. It can inspect both 32-bit and 64-bit ELFs, it can analyse the contents of these files and create links between matching data, it can disassemble every relevant function and it offers a full functional graphical interface that allows the user to interact easily with the program.

The tests we conducted varied from basic user interaction to numerical data, and all the test results led to a positive outcome. ELF Detective works correctly, provides complete data, is very interactive and scales well. Another quality that came out of the beta testing phase was that the project is eye-candy, it feels great and looks user friendly.

A. Further Work

Our next plan for this project is to extend it with support for more binary file format, more platforms and usable on more operating systems. Due to its modularity, this can be done easily by targeting specific environments.

References