University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Interactive Exploration of the Process of Linking Executables

Scientific Adviser:
Ș.l.dr.ing. Adrian-Răzvan Deaconescu

Author:
Andrei Ciprian Aprodu

Bucharest, 2016

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.

Contents

List of Figures

Chapter 1

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 god 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.

Analysing by hand the inputs and the output of this phase will only get you a whole of machine code that gets merged. There's more to linking than that, and the changes are almost unrecognizable unless this is what you aim for, leading to confusion for an unexperienced user.

The project that we present in this paper, ELFDetective, is an educational tool with the goal of helping students understand the changes that occur in the linking process. It offers an interactive interface where it shows all 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.

1.1 Motivation

Throughout college, computer science students work with ELF files without even knowing at first. Their first 'Hello World' C programs that were written on Linux were ELFs, but they were not aware of that yet. Little after that they started using makefiles and separate the compiling from linking when building a project, which showed them the linkable files, which not only they share the same format with the executable, but they look alike as well. At this point, some may have asked "How does the linker work?", but the theoretical explanation was hard to understand without a proper background.

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 an example 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.

1.2 Objective

Our main objective with this project is to help others understand the linking process better while spending a lot less time. Since there isn't any other tool that has this focus, it had to be an improvement of all the flaws that any other tool of the trade has. These tools offer a text interface only, ELF Detective provides a simple, yet rich, GUI that will allow easy access to any information that's needed. They can only have output for a file at a time, ELF Detective works with projects, which means that by providing an executable file and any number of linkable files, it will gather data from each object that will be mashed together forming a knowledge base that will be compared with the data from the executable file, resulting in a more complex output.

From an education point of view, this project must be able to explain the changes that were found, in the easiest way possible so a student of any background can understand it well. This means that all addresses must be explained, e.g. a symbol is defined at an address because is part of section .bss at offset 0xd or an address is computed based on the instruction pointer etc., 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 he wants to know more about any available 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. We will cover all these topics in Chapter 5, where each characteristic of the project will be tested and analysed in order to make sure that this objective is reached.

Chapter 2

Background

ELF files have mostly the same structure since they were created and anything new comes by extension, leaving the format unchanged. Since they never change, it is understandable that any tools that read/inspect/analyse these files had no reason to be modernised.

There are many applications that offer a great overview of binary files, such as **IDA**[2], but each of them have a different purpose. Many of these are debuggers or useful for reverse engineering, and are not friendly enough for an unexperienced user. 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. More information on this topic will be discussed in Section 2.4.

In the following sections, we will explain the basic concepts of the project, in the interest of focusing more on the application itself later on.

2.1 Linkable Files

Linkable files, also known as object files, result from compilation. Each of these files represents the translation of a source file into machine code. Even though they are not usually executable, their importance come from their structure, which defines how the executable file will look like. The most important elements of a linkable file are the **symbols** and the **sections**. Symbols represent references to memory addresses, usually where we define variables or functions, in a program. Sections help organizing the object file by category. The most important sections are the code (.text), initialized data (.data) and uninitialized data (.bss). In object files, the address range of sections start at zero because their actual program addresses are unknown before de linking stage. The process of how object files are linked together will be covered separately.

[2]https://www.hex-rays.com/products/ida/

2.2 Executable Files

Executable files result from the linking stage. The combination of object files lead to executables, they they must be merged in such a way that all the references between them have been resolved. They have a similar structure to the linkable files, but every section's start address now points to a unique location in the program memory map and every symbol can be found at a concrete unique address. Having only one file with these characteristics and a few others, e.g. having a start address, allows these files to be executed and launch the program.

2.3 Linking Process

The most simple explanation for the linking process is as a stage that combines linkable files into one executable file, as it can be seen in Figure 2.1. As a more complex view, object files can be seen as collections of uninitialized sections that need to be grouped and initialized. This sounds easy because most of it is, but there are two cases where this gets a bit complicated. An object file can use at any point a function call that is part of another file or even reference an external variable, and the only way for either of these to work is by knowing the symbol address, which is unknown before link time.

This task got easier over the years due to virtual memory, all running programs get the same, fixed amount of memory (2^N, N being 32 or 64 for modern architectures), which allows the linker to use the same address range for each program that starts at zero.

Since it seems like the linker is in charge of two separate things, merge the object files and initialize the addresses, it was only logical that the linking process should be split in two stages, **relocation** and **address binding**, which are covered in the next subsections.

2.3.1 Relocation

Relocation is the first stage of linking and has the purpose of merging the object files by combining the sections of the same type (.data, .text, .bss etc.) to create the same section of the executable file. After all the required data has been placed in the executable file, the only thing left to do in this phase is to initialize each section. All the previously zero-based address ranges of sections get translated into real addresses in the executable file. A visual explanation of this stage can be seen in Figure 2.2.

2.3.2 Address Binding

Address binding has to deal with the complicated part of linking. As stated earlier, there are references to symbols that are unknown at this time and they need to be solved. Establishing these connections between different parts of the program is what makes the executable homogenous.

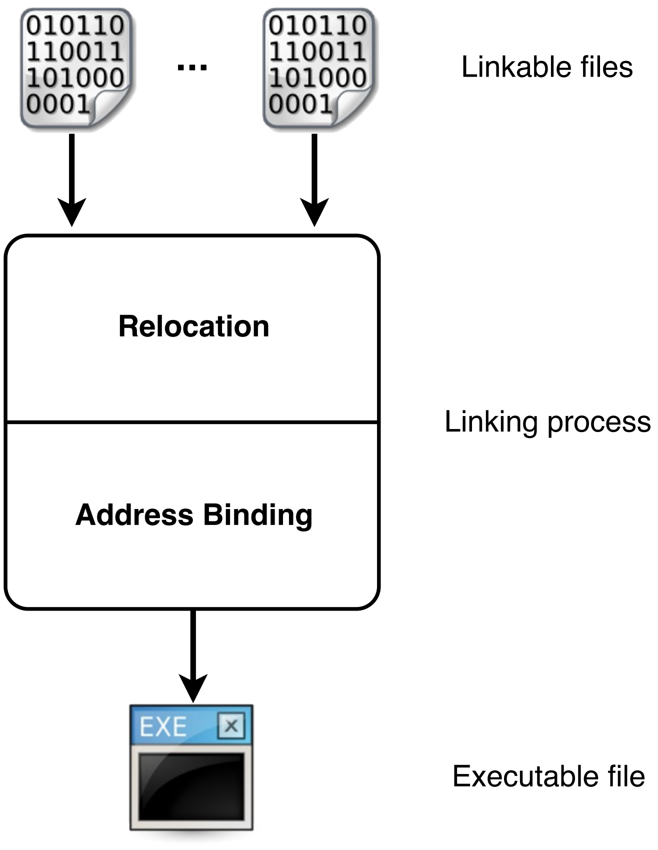


Figure 2.1: A diagram of the linking process

According to the book "C and C++ Compiling" by Milan Stevanovic[3], to solve this issues, the linker must:

- Examine the sections as resulted from relocation
- Find the code that calls outside of its original section
- Find the exact address of the referenced symbol
- Replace the default address with the actual one

2.4 Tools of the Trade

It is important to present the tools of the trade because it 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 format showed in the terminal.

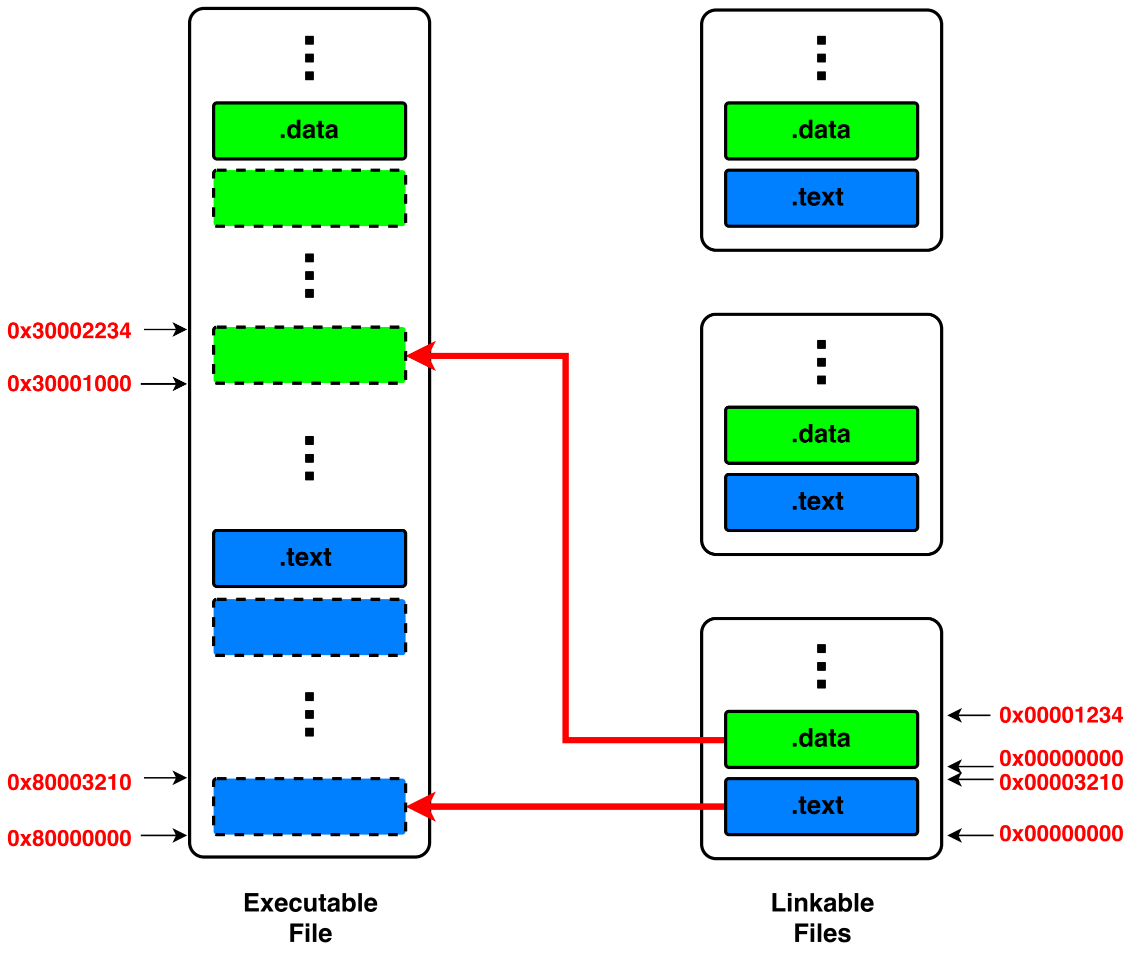


Figure 2.2: A relocation example

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.

As an example, we present the output of **nm --format sysv** in Listing 2.1, which is the prettiest output of what these tools offer. These are the symbols from an object file as seen by nm. This

[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

file has a lot of external symbols and function calls to other files, and nm, or any other tool mentioned earlier, cannot get any information about them because they do not cross reference with any other files. It can be easily seen that for most of these symbols there is nothing that can be said beside that they are part of section ***UND***, which means a symbol is undefined, that they are of **NOTYPE** and don't have any values. This clearly demonstrates that these tools are not very useful in exploring the linking process, not even in inspecting a single file and finding complete information, and they are even worse if we consider how they look to a student that tries to learn more about linking.

```
Name   Value    Class  Type    Size     Line      Section
cyktj |        |   U | NOTYPE|        |        |*UND*
ebp   |        |   U | NOTYPE|        |        |*UND*
fsjg  |        |   U | NOTYPE|        |        |*UND*
gabk  |        |   U | NOTYPE|        |        |*UND*
gjnjw |        |   U | NOTYPE|        |        |*UND*
gvd   |        |   U | NOTYPE|        |        |*UND*
hoe   |00000004|   C | OBJECT|00000004|        |*COM*
iyn   |        |   U | NOTYPE|        |        |*UND*
main  |00000000|   T |   FUNC|00000083|        |.text
mfg   |        |   U | NOTYPE|        |        |*UND*
oj    |        |   U | NOTYPE|        |        |*UND*
qeycz |        |   U | NOTYPE|        |        |*UND*
qfrta |        |   U | NOTYPE|        |        |*UND*
qsmml |00000004|   C | OBJECT|00000004|        |*COM*
rloi  |        |   U | NOTYPE|        |        |*UND*
rmzk  |        |   U | NOTYPE|        |        |*UND*
siu   |        |   U | NOTYPE|        |        |*UND*
tvyu  |00000083|   T |   FUNC|0000015f|        |.text
txc   |00000004|   C | OBJECT|00000004|        |*COM*
uocfq |        |   U | NOTYPE|        |        |*UND*
vgceb |        |   U | NOTYPE|        |        |*UND*
yb    |        |   U | NOTYPE|        |        |*UND*
ytlt  |00000000|   b | OBJECT|00000004|        |.bss
```

Listing 2.1: Output of nm --format sysv

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**[1], **wx Hex Editor**[2]

[1] http://home.gna.org/bless/
[2] http://www.wxhexeditor.org/

- Disassembly and reverse engineering: **Radare 2**[1], **IDA**

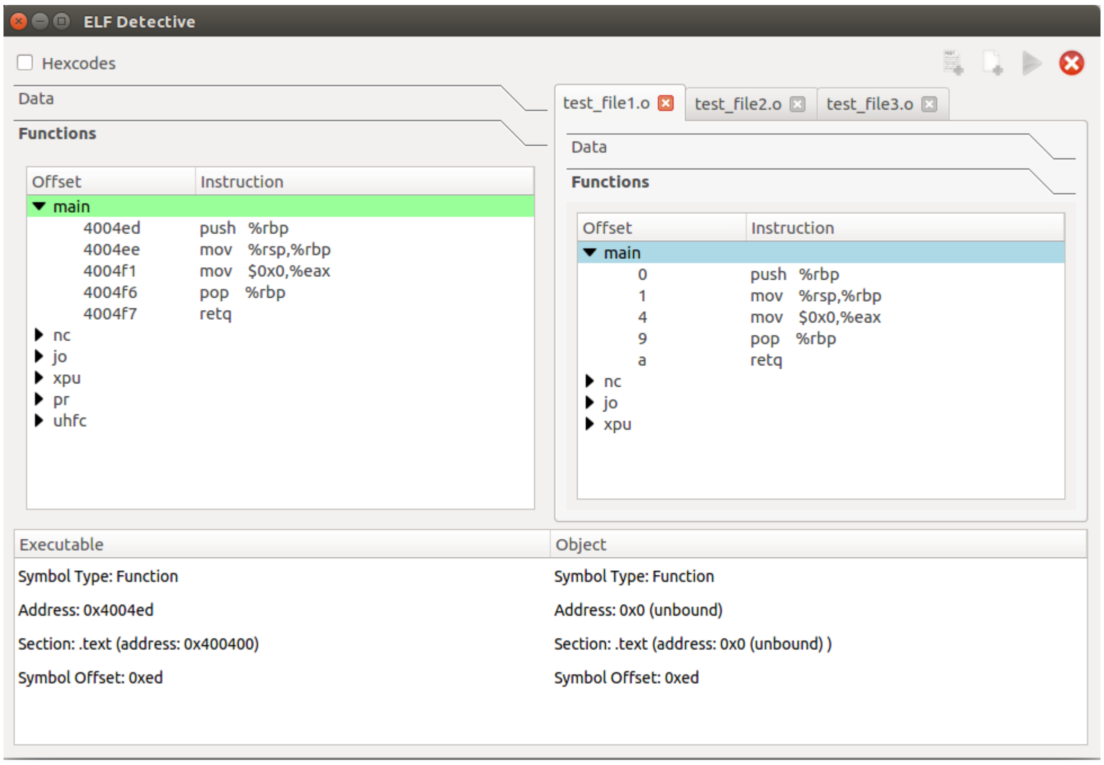
[1] http://radare.org/r/

Chapter 3

Project Overview

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.

At the moment ELF Detective is a 64-bit program only due to library dependencies, but it can work with ELFs compiled for 32-bit as well.



```
Offset        Instruction
▼ main
   4004ed     push  %rbp
   4004ee     mov   %rsp,%rbp
   4004f1     mov   $0x0,%eax
   4004f6     pop   %rbp
   4004f7     retq
▶ nc
▶ jo
▶ xpu
▶ pr
▶ uhfc
```

```
Offset        Instruction
▼ main
   0          push  %rbp
   1          mov   %rsp,%rbp
   4          mov   $0x0,%eax
   9          pop   %rbp
   a          retq
▶ nc
▶ jo
▶ xpu
```

Executable	Object
Symbol Type: Function	Symbol Type: Function
Address: 0x4004ed	Address: 0x0 (unbound)
Section: .text (address: 0x400400)	Section: .text (address: 0x0 (unbound))
Symbol Offset: 0xed	Symbol Offset: 0xed

Figure 3.1: Function view of ELF Detective

3.1 Use Cases

Writing use cases is the most simple and effective way to show the need for a program, as well as presenting how it works and how it can be used. All of the following use cases have the prerequisite that a project was loaded and ran.

Inspect executable and linkable files. This is the most generic use of the program. After the project was ran, any available data about each file will show in an easy to read format. Inspecting a subset of linkable files is also possible, and all the other symbols that have not been defined in the subset are marked as unknown in the executable panel.

Inspect variables. By clicking on a variable name in the executable file panel, it will highlight its definition in the object file and every information about the variable, from both executable and linkable files, will show.

Inspect functions and their disassembled code. Similar to inspecting variables, but this also has the option to show a function's code. Each code line is clickable as well, and if there is any relevant information, e.g. if an instruction uses an address, it will show what that address means.

Find external references that can't be solved at link time. Any external reference will show in the variables container and it will say that is not defined when clicked.

Identify changes that were made at link time. This is the main goal of the application, by inspecting the symbols and the code lines, it will show the state of that item in both the object and executable files and any difference between its properties, usually the address changes, will be visible.

3.2 Building Blocks

In order to present how the project works and behaves as a whole in Chapter 4, 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.

The most important building blocks of this tool are: the inner representation of the ELF files, the disassembly module, the symbol analysis module and the graphical user interface. The way they work and interconnect can be seen in Figure 3.2.

3.2.1 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**[1] 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

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

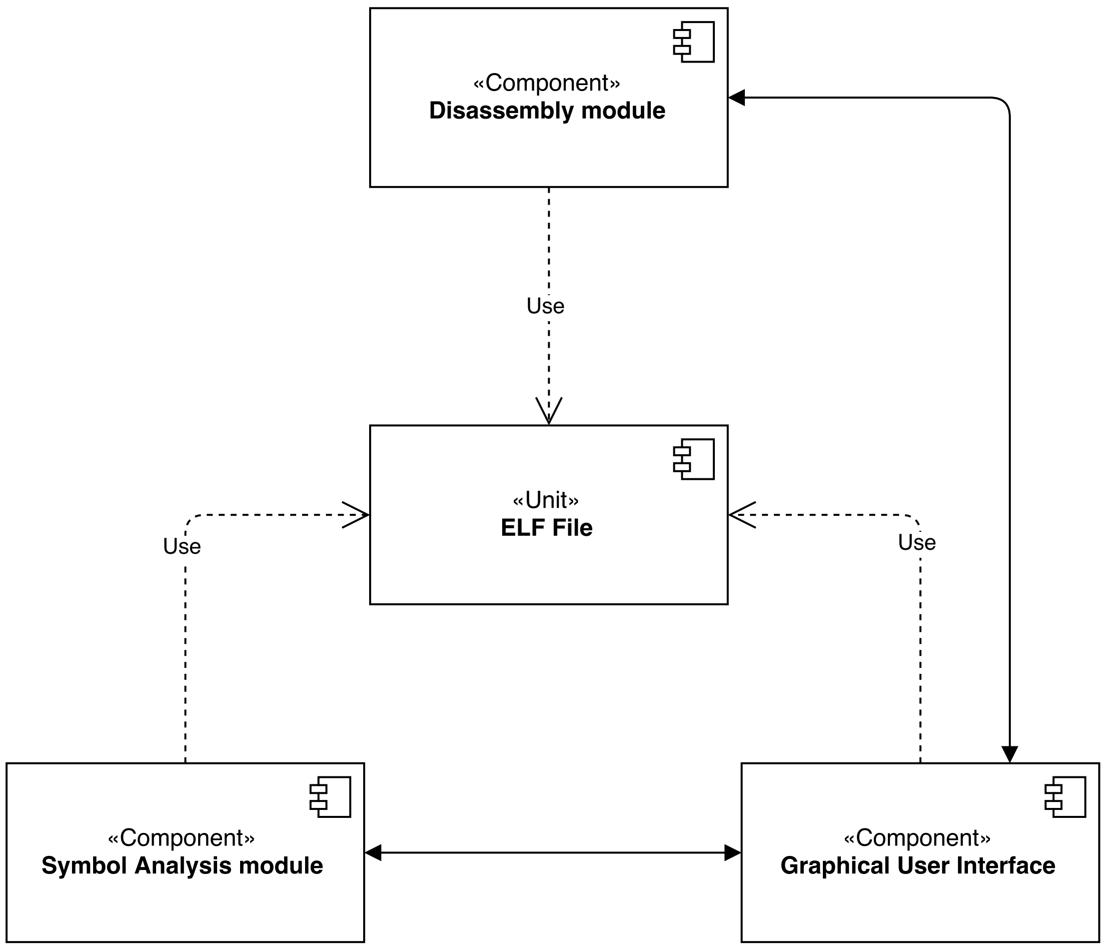


Figure 3.2: Architectural Diagram

BFD format so that any required data can be cached as it can be seen in Listing 3.1. As an example, the most used data is the symbol table, which is cached as an array of **asymbol*** (the BFD representation for symbols, it contains details about the symbol such as name, address, if it is known and a pointer to its section). 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.

The data structure representing the ELF files is gradually initialized. Each task that's run before showing any data will add more information to it so that anything can be easily found in a friendly format.

```
class ELFFile
{
private:
        bfd *abfd = nullptr;
        asymbol **syms = nullptr;
        asymbol **dynsyms = nullptr;
        asymbol *synthsyms = nullptr;

        long symcount;
        long dynsymcount;
        long synthcount;

        std::vector<Function *> functions;

        QWidget *view;
};
```

Listing 3.1: The data cached in the ELFFile class

3.2.2 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. A skeleton of the code can be seen in Listing 3.2, and while this is not complete due to the complexity of the actual code, it still represents well how functions form and get in the files.

```
Function *f = new Function();
f->setName(sym->name);

... // gathering all the necessary data for disassembly

while (addr_offset < stop_offset)
{
        crtLine = new CodeLine();
        ... // add values to the code line
        f->addCodeLine(crtLine);
}

crtFile->addFunction(f);
```

Listing 3.2: Disassembly skeleton

3.2.3 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 used 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. The way the Hash Map is formed can be seen in Listing 3.3.

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.

```
for (ELFFile *E : objfiles) {
  current = E->getSyms();

  for (int i = 0; i < E->getSymcount(); ++i) {
    Symbol entry;
    ... // check current symbol for validity
    if (bfd_is_und_section((*current)->section)) {
      ... // checks for symbol
      entry.undefined_in.push_back(E->getName());
    }
    else {
      entry.name = bfd_asymbol_name(*current);
      entry.defined = true;
      entry.defined_in = E->getName();
      entry.def_value = bfd_asymbol_value(*current);
      entry.defined_section = (*current)->section->name;
      entry.def_section_value = bfd_asymbol_base(*current);

      symbolTable[name] = entry
    }
  }
}
```

Listing 3.3: Symbol selection from object files

3.2.4 Graphical User Interface

The GUI is what makes this project unique and gives it a purpose. It is implemented with Qt[1] (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'.

Adding files, running or clearing the project are easy tasks as well due to buttons designed with suggestive icons, which have shortcuts as well for a more advanced user.

The design of the GUI was implemented with **XML** and **Qt Style Sheet**[2], or **QSS**, which is a powerful style language that allows the customization of Qt widgets. A sample of what it looks like is presented in Listing 3.4

```
QTreeView::branch:selected {
        background: #99ff99;
}
QTreeView::branch:selected:active {
        background: #99ff99;
}
QTreeView::branch:has-children {
        image: url(icons/icon-arrow-right.ico);
}
QTreeView::branch:has-children:!closed {
        image: url(icons/icon-arrow-down.ico);
}
QTreeView::item:selected,
QTreeView::item:selected:!active {
        background: #99ff99;
        color: #000;
}
```

Listing 3.4: Style of a QTreeView - the function container -

[1] https://www.qt.io/qt-framework/
[2] http://doc.qt.io/qt-4.8/stylesheet.html

The GUI was the most discussed topic during the testing phase since the project targets unexperienced users and for that it has to have a great user experience design. Most of the feedback we received from the testers can be seen in Section 5.5.

Chapter 4

Implementation Details

Now that we explained all the modules of the project, 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 reinventing the wheel is almost never a good idea, we decided to base our work on another open source project, **objdump**. Some of its code was used, but it was modified and added some more to make it more specific for this project, for 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.

4.1 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.

4.1.1 Binary File Descriptor library

The BFD package is part of the **GNU Project**[1] and represents a mechanism that offers binary file manipulation. It supports multiple file formats on many processor architecture[2]. 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)."[3].

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**[4], 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.

4.1.2 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.

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.

[1] https://www.gnu.org/gnu/thegnuproject.html
[2] As of 2013, it supports around 50 file formats for 25 processor architectures
[3] https://sourceware.org/binutils/docs/bfd/Overview.html
[4] https://gcc.gnu.org/onlinedocs/libiberty/Overview.html

4.1.3 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**[1] 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.

ELF Detective was designed using the Qt Creator IDE and the Qt 5.6 library.

[1] https://www.qt.io/ide/

Chapter 5

Testing

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. Some of their feedback will be presented in Section 5.2.

5.1 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[2] 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.

As it can be seen in the code snippet below, the script generates symbols for each file. It starts with the generation of unique symbol names, each one gets additional attributes such as a type, variable or function, and a keyword. The script is hardcoded to generate a bigger amount of extern variables since they are of interest. The functions bodies are generated while writing the C files since all symbols must be known in order to reference some from other files.

```
for i in range(files_no):
        symbols[i] = generate_symbols(i)
```

[2]https://www.python.org/

```

        for sym in symbols[i]:
                if sym['Name'] == "main":
                        continue

                if randint(0,1) == 0:
                        sym['Type'] = VARIABLE
                        sym['Keyword'] = external
                        external_symbols.append(sym)
                else:
                        sym['Type'] = randint(0,1)
                        sym['Keyword'] = keywords[randint(0,2)] if
                            sym['Type'] == 0 else ""

                if sym['Type'] == FUNCTION:
                        external_symbols.append(sym)

for i in range(files_no):
        generate_file(i)
```

Listing 5.1: Core code of the test generator

The resulting files are compiled and linked, creating the input for the application to inspect.

5.2 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.

The look and feel of the project was a matter of discussion during beta testing and some of the received feedback was used to increase the level of interactivity of the program. This feedback can be found in Section 5.5.

5.3 Correctness

The most important part of this project was to respect all the specifications. The main requirements were for the program to be interactive, this was discussed in Section 5.2, 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.

5.4 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.

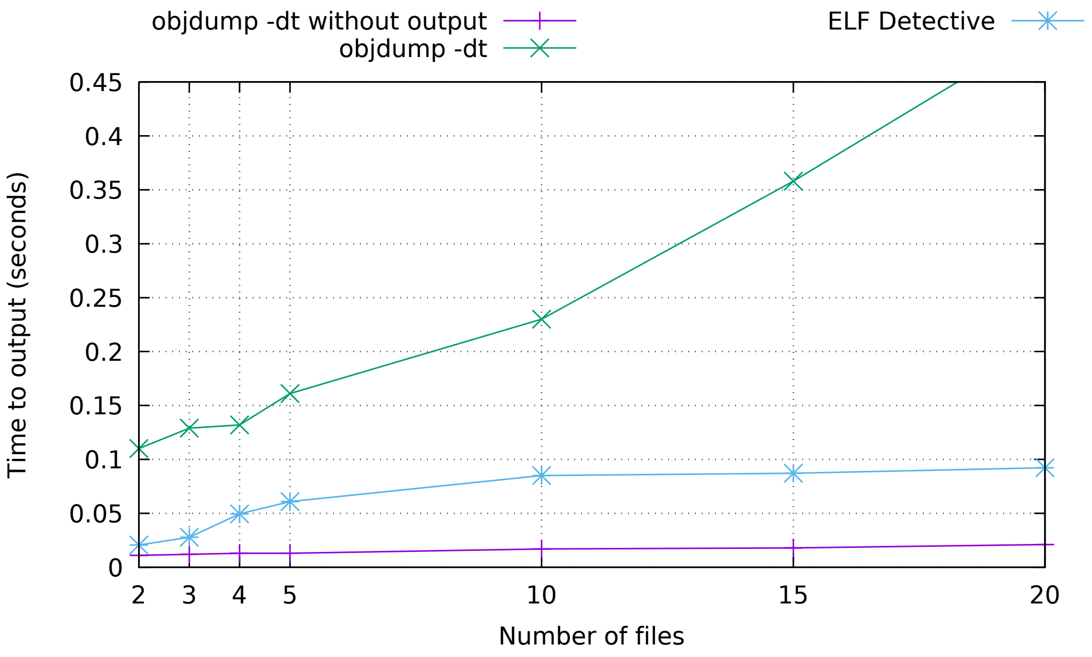


Figure 5.1: Comparision between ELF Detective and objdump -dt

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, and put the results on a chart, shown in Figure 5.1, along the data of ELF Detective. The response times of ELF Detective were obtained by using the C++ clock()[1] function which allows to compute the total

[1]http://www.cplusplus.com/reference/ctime/clock/

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**[1] 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.

As it shows on this chart, 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 20 object files and one executable file project, and it was run multiple times. The data in the pie chart from Figure 5.2 is the average of these runs. 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.

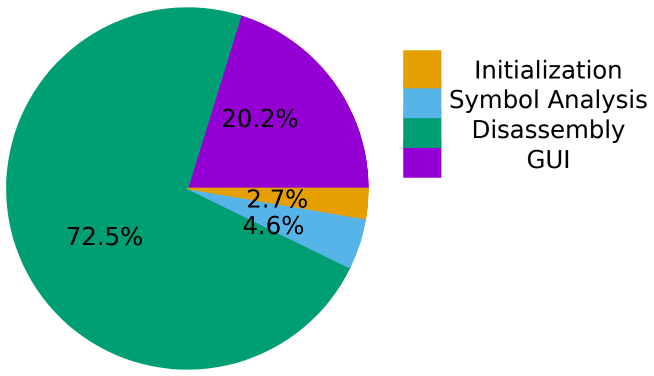


Figure 5.2: Time spent by each module

As it can be seen, 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

[1] http://linux.die.net/man/1/time

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. This can be found in Figure 5.3. 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.

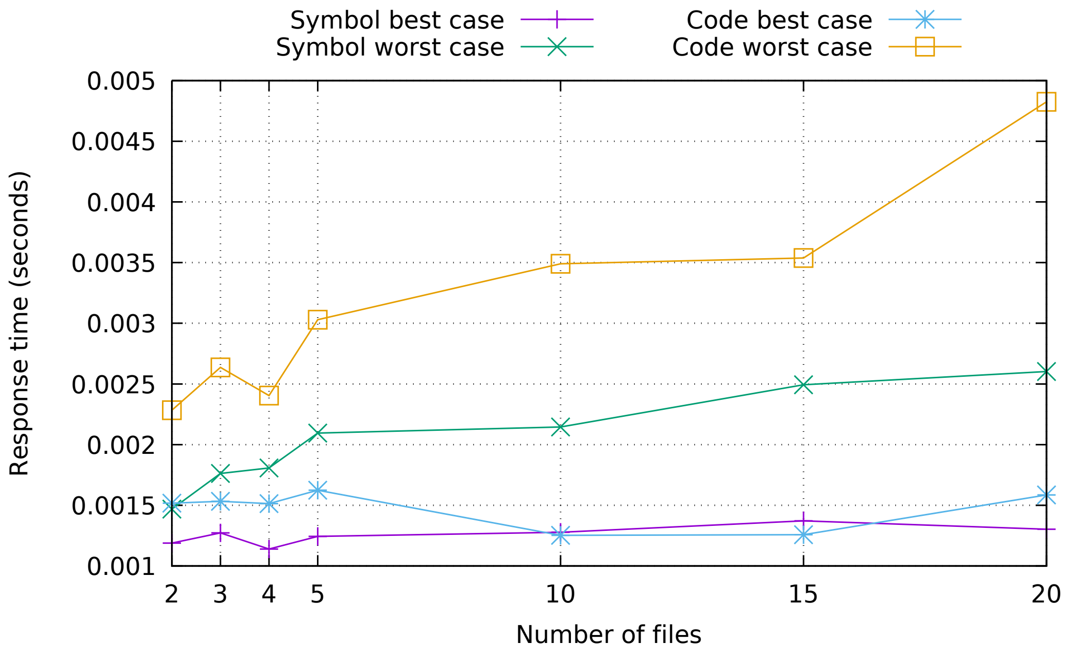


Figure 5.3: Best and worst cases while inspecting data

Each file on the graphic represent an additional amount of symbols computed by:

$$\mathbf{random}(\mathbf{5}, \mathbf{10}) + \sum_{\mathbf{0}}^{\mathbf{F}} \mathbf{random}(\mathbf{0}, \mathbf{2}), F = numberoffunctions$$

As it can be seen in the plot, 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. This of course can be improved, but that will be discussed in Section 6.2.

5.5 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.

Chapter 6

Current Status and Further Work

In this chapter we will present the current state of the project and they ways it can be improved in the near future. Even though the project respects all the required specifications, it can still be greatly improved code-wise, scalability-wise and design-wise. In Section 6.2 will be presented what these improvements are and how can they make the project gradually better.

6.1 Current Status

Seeing this project slowly develop from a basic text dump to an elaborate program with a complex graphical interface, enables us to say that the project in its current state is ready for release. Functionality and scalability wise, the project does well, as it was stated in Chapter 5. In the same section it was presented that the look and feel of it seem straight forward according to the received feedback.

Despite the fact that all the tests show great results, ELF Detective can get even better. In the next subsection we will present what should be improved and why.

6.2 Further Work

As stated earlier, the project is in a great state, but anything can be improved in time. These improvements are separated by category so they can be easier to isolate for the next person that will work on this project.

6.2.1 Coding

Code-wise there are two things that can improve how easy the program's code can be understood and it would bring another abstraction layer making it easier for anyone to continue the development of the project.

1. The core library of this project is BFD, which is an old C package that could work better with C++ than it does. It feels that by creating wrappers over the API of this library it would make the code much easier to understand because at its current state it relies on working with a lot of pointers, some of which are allocated, some are not. Replacing small things like **asymbol**** with **vector<asymbol>** would make a lot more sense and it would make the data much easier to access. These wrappers would indeed make the program run a bit slower, but this should be tested since most likely the difference should be inconsiderable.

2. Another thing that should be improved is the way the correspondences between the executable and object files work. At this time, these connect on a symbol interrogation level, meaning that each symbol is looked for in a hash map that provides the necessary information. It would be a lot better, and the GUI might work faster if these links are between GUI elements that hold the data, enabling instant access.

6.2.2 Functionality

3. As seen in Figure 5.2, the disassembly code is the bottleneck for this project. This is mostly due to working on strings with high-level functions that split and replace iteratively. There could be various ways to improve this, but they all need to be tested to find the best way to implement it. Some solutions might be:

- Using pointers instead of strings
- Using compiled regular expressions
- Finding a pattern that allows to change the way strings are currently parsed
- Keeping the strings as raw data and parsing them on request

4. There is a corner case when the same mov instruction looks differently in another file. Sometimes a simple mov from a register to another might be split in two different mov instructions that use the memory as a buffer. This should be find before displaying the code and it should be marked so it can't be highlighted.

6.2.3 Design and User Experience

The user interface currently works as it should and it has a nice look and feel, although there are some features that could provide a better user experience for a niche category of users.

5. Even though the purpose of this program is to provide an educational overview of the linking process, more advanced users might find it to be a good tool to easily inspect ELFs. This category of users are used to using the mouse as little as possible since it slows the work process. In its current state, ELF Detective cannot be used only with the keyboard. It has shortcuts for buttons, it allows arrow movement while inspecting symbols, but it has no support for opening the functions to see the code, to open a certain page or the use the check box. This would bring a better feel to the project and it would increase the amount of users it targets.

6. While testing the application it became clear that adding files repeatedly every time can be annoying. A solution for this would be to allow the user to export the opened files as a project in a file, holding any data needed to allow the reopening of the project. Implementing this would take little time and it would have a great effect in the case of reinspecting the same project often.

7. Another thing that's somewhat similar with the previous point is to add a "Refresh Project" button. This would be useful in case of inspecting a project that you currently work on because refreshing the project would parse all the opened files again, instantly accessing the latest version of the files.

8. Another thing that feels it could be useful is to add the functionality that allows for certain parts of the GUI to pop up in a new window. This would allow the users to isolate only the information they need and by adding the option to export this as raw data as well would let anyone interested to parse the data in any way they need.

Bibliography

[1] Yale FLINT Group. Executable and linkable format. http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf.

[2] John R. Levin. *Linkers and Loaders.* Morgan Kaufmann, October 1999.

[3] Milan Stevanovic. *C and C++ Compiling.* Apress, April 2014.