University POLITEHNICA of Bucharest

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

BACHELOR THESIS

Interactive Inspection of the Dynamic Linker and Loader

Scientific Adviser:
S.l.dr.ing. Răzvan Deaconescu

Author:
Lucia Mădălina Cojocaru

Bucharest, 2016

Abstract

Dynamic linking and loading are some of the most obscure processes which happen when an executable is run. However, tracing them by means of debuggers is a complex task which is not suitable for inexperienced users. In order to facilitate understanding of dynamic linking and loading, I created an interactive inspection tool with a user-friendly graphical user interface. The DynInspector tool offers support to view the relevant areas of the program memory by making use of the existing functionality of debuggers. It is an educational tool which helps Computer Science students and hobbyists gain a better understanding on the subject of dynamic linking and loading.

Contents

List of Figures

Chapter 1

Introduction

Linkers and loaders are an essential component of the software development process. While linking handles symbol resolution within the caller's object code for object files and shared libraries, loading consists of copying the executable into the main memory in order for it to be run. Clearly, a program could not be run on modern operating systems without the help of these tools. Nevertheless, linking and loading remain obscure to most users as their interaction with the program address space is transparent. The program runs, but we should ask ourselves how and why.

Take the example of an inexperienced user, a computer science freshman student. He is presented with the stages of compilation and he is shown how he can write and run ELF executables. He uses what he sees as basic functions, such as *printf*, which he never defined in his own source files. If pointed out, it might appear quite wondrous to the user how the implementation of *printf* is found, especially in the context of address space layout randomization. Moreover, since the process can only access its own address space, he might wonder how, when and where the library containing *printf*, in this case *libc*, is mapped.

Understanding how a program is run, when its symbols are linked and how the address space is modified is essential for programmers. It is a useful skill in the context of debugging, it can help improve code quality and can prove crucial in the context of security. However it is difficult to observe the effects of the linker and loader using command line debuggers, the default one provided on Linux systems being *gdb*[1]. Previous knowledge of their functionality is required, as well as understanding of what is to be debugged. Moreover, I found no tool that can highlight the job of linkers and loaders so as to fit an educational purpose.

1.1 Project Description

In order to facilitate understanding of linking and loading for inexperienced users, I created such an educational tool. It provides a user-friendly interface with dedicated functionality for

[1] https://www.gnu.org/software/gdb/

observing linking and loading exclusively.

The main objectives for this tool are to present in a detailed and accessible manner each step of dynamic linking and of dynamic loading. The information is presented as a tutorial where the user can follow a set of steps, read insightful information and observe the process hands-on. Ease of use is achieved by a minimal user interface with intuitive gadgets.

This DynInspector tool is thoroughly presented in this paper.

Chapter 2

Basic Concepts

In the previous chapter I presented a short introduction to this project and its main objectives. Before I present a more detailed description of the main goals, it is necessary to discuss some basic concepts on linkers and loaders. This knowledge correlates with the motivation behind the project.

2.1 Theoretical Background

To begin with, I will make a description of the theory behind linkers and loaders. This includes presenting the different types of linking and loading, the concept of shared libraries and the lazy procedure linkage mechanism.

2.1.1 Linkers and Loaders

As stated in the previous chapter, the purpose of the DynInspector tool is to highlight what linkers and loaders do. Basically, the linker handles symbol and address resolution. The loader copies the executable in the main memory for it to be run [2]. It also handles mapping shared libraries into the address space of the executable. Although they perform separate tasks, the functionality of linkers and loaders is intertwined.

Linking can take place at compile time, load time and runtime based on where the required symbols are located. Linking takes place at compile time when resolving symbols from object code libraries or static libraries. This is static linking. The issue of this approach is that large libraries, such as *libc*, would occupy a significant portion of a the address space of each executable.

For this reason, a widely used solution is that of shared libraries which are loaded into the memory only once and multiple executables can link to it simultaneously at load or runtime. The shared library is then mapped into the address space of each process that requires it.

Symbol and address resolution at load and runtime are tasks of the dynamic linker and are part of the dynamic linking process.

Loading is also static and dynamic. Static loading is responsible for loading the executable into the main memory for it to be run and mapping the required shared libraries into the address space of the program. It happens at load time, before the program execution is actually started. Dynamic loading takes place at runtime and is invoked by the user manually. It handles mapping shared libraries into the address space of the program.

Figure 2.1 highlights the differences between static and dynamic linking and loading. It also presents when the library is mapped into the memory (loading) and when the symbols and addresses are resolved (linking) for a call to *printf* and a call to a custom function *X* from a user defined dynamic shared library *Y*.

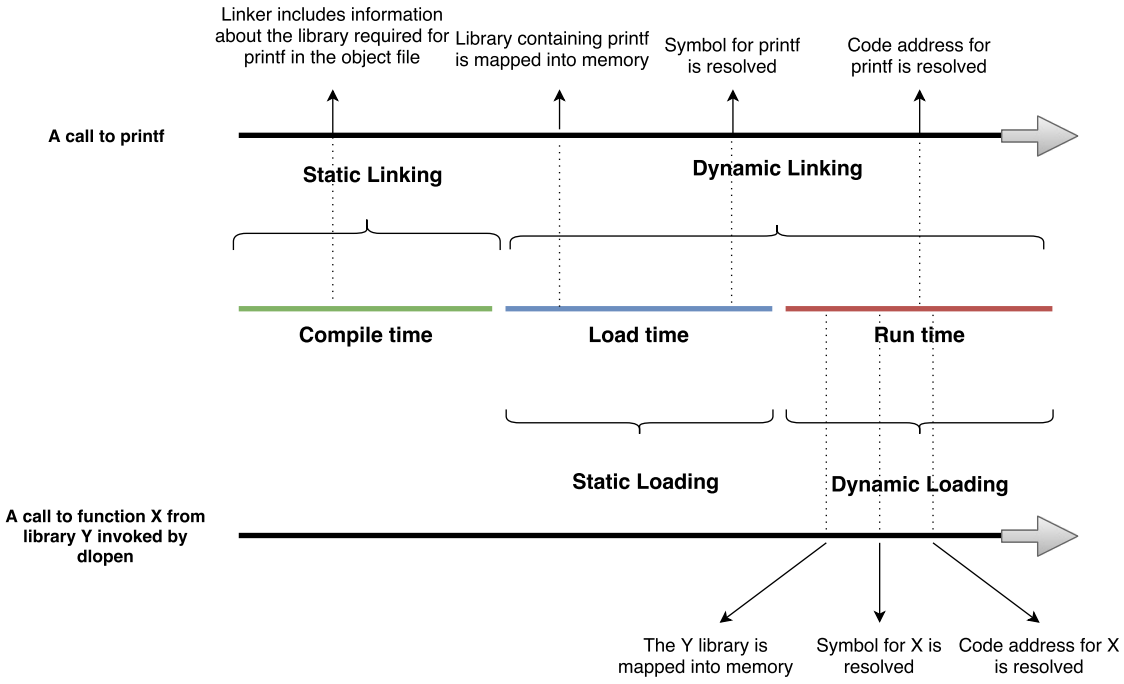

Figure 2.1: Static vs Dynamic Linking and Loading

2.1.2 Shared Libraries

Note that a distinction is made between static shared libraries and dynamic shared libraries [2]. Static shared libraries are mapped into the memory immediately before the executable is run. Their symbols are resolved immediately after the library is loaded. Address resolution for functions is postponed to the first call to the routine. The mechanism is called *lazy procedure linkage* or *lazy binding*.

On the other hand, dynamic shared libraries are an extremely powerful mechanism as they can be loaded and unloaded at runtime. After this, the dynamic linker resolves the necessary

symbols.

Therefore, whether a shared library is static or dynamic depends on how its contents is loaded in the program and how it is accessed.

2.1.3 Lazy Procedure Linkage

Dynamically linked programs use lazy procedure linkage with shared libraries. This means that binding to the actual address of the function happens on the first call to the function [2]. Before that point, the function address is unknown to the program.

For ELF files on Linux systems, the mechanism by which this is achieved uses the *.got* and *.plt* sections. The *.got*, particularly the *.got.plt* section, is an array of pointers to the functions located in the shared libraries. Initially, these pointers do not indicate the actual address of the functions, but they point to the *.plt* section. The *.plt* section contains entries for each of the functions found in *.got.plt*. Each entry is a stub code which has three roles:

- Jumps to the address indicated by the *.got.plt*. If the address is the actual function routine, then the execution continues as expected. If the address points back to the *.plt* stub, then the next two steps are executed and the dynamic linker is invoked.
- Stores the address of the *.got.plt* entry on the stack for the dynamic linker to access and update.
- Invokes the dynamic linker which finds the start address of the function code and updates the pointer value in the *.got.plt* section.

As can be observed, the *.plt* section acts as a trampoline for function calls. It introduces a level of indirection. Note that the last two steps are done only if the address is not valid, meaning only on the first call to the function.

The importance of shared libraries in modern day operating systems is based on the fact that they can be shared between multiple executables/processes ensuring an efficient utilisation of the main memory, as well as a common code base within the system. It is essential for developers to understand the mechanism in order to make use of its powerful features.

2.2 Tools of the Trade

The linker and the loader play an important role in creating a functional executable and moving it into the main memory to be run. Moreover, linking is essential in the case of shared libraries as it allows programs to address code which was not defined in their object files. There is a set on non-dedicated tools for inspecting dynamic linking and loading.

2.2.1 Basic Unix Tools

At present, in order to make a preliminary assessment of how the linker and loader work, there are tools which make it possible to examine the contents of an executable before it is run.

On Unix systems, basic command line tools such as *readelf*[1] and *objdump*[2] achieve the functionality described above for ELF executables. They can display the sections of a program with their corresponding offsets within the program address space. This can be useful as the loader can keep this arrangement, although this is not guaranteed.

However, they cannot offer detailed information, or none at all, about particular sections of interest in the dynamic linking and loading process, such as the *.plt* or *.got.plt* sections.

In order to properly understand how the linker and loader function and interact, tools which have the possibility to instrument an executable are necessary. At present, these tools are debuggers.

2.2.2 GDB

Unix platforms provide by default *gdb* which is a powerful debugging tool for ELF executables. *Gdb* includes standard commands for displaying the shared libraries used by the program, as well as the offsetted locations of the segments in the running ELF.

However, in order to observe the detailed process of dynamic linking and loading, the user must manually interact with *gdb*. For example, in the case of a function call to a shared library, the workflow involves setting breakpoints and obtaining the address of the *.plt* section and *.got.plt* entry for function.

This implies that knowledge of the process is a prerequisite in order to follow the subtle changes happening at the time of dynamic linking.

2.2.3 LLDB

Despite its powerful features, *gdb* lacks integration with other software and cannot be included as a module or library in custom programs. It is designed as a manually operated command line tool for the main purpose of debugging.

For this reason, the LLVM project[3] has started developing a high performance debugger which broadly covers the existing *gdb* functionality. *Lldb*[4] supports command-line debugging, but can also be included as a module in scripting languages such as Python[5].

[1] https://sourceware.org/binutils/docs/binutils/readelf.html
[2] https://sourceware.org/binutils/docs/binutils/objdump.html
[3] http://llvm.org/
[4] http://lldb.llvm.org/
[5] https://www.python.org/

Therefore, in order to observe the linking and loading process, *lldb* offers a similar set of steps as *gdb* through its command line feature. However, its main advantage is that its functionality can easily be extended to fit more specific needs.

Chapter 3

Motivation and Objectives

3.1 Motivation

The tools described in the previous section allow visualisation of the linking and loading processes as long as the user is aware of what needs to be followed. There are two main drawbacks of the usage of debuggers as a means to understand linking and loading. The first is that debuggers do not highlight any process in particular. They are tools with no purpose until the user gives them one. The second is that debuggers are very complex and offer a multitude of functions. This discourages inexperienced users from using them.

3.1.1 Tutorial Based Approach

Firstly, stepping in a debugger is not particularly useful for somebody who first encounters these concepts and needs a hand-on application to fully grasp what is happening at each stage.

Take for example the calls to the function *printf* which has the code located in the shared library *libc*:

- In order to set a breakpoint, the user must have knowledge that the dynamic linker steps in at the first call of this function.
- After this, the user must correctly identify in the assembly code the address for the *.plt* section entry of the function *printf*. This is where the code jumps next.
- At this point, knowledge of what the *.plt* code is doing is imperative: first there is a jump to a function pointer in the *.got.plt* section. On the first call, this point leads back to the *.plt* section and the dynamic linker is invoked. On the second call, the linker will have modified the pointer value to the correct address of the *printf* procedure in its corresponding shared library, which is *the Standard C Library*.

The steps described above are illustrated in Figure 3.1. They highlight how the loader and linker make use of lazy procedure linking which was described in Section 2.1.3.

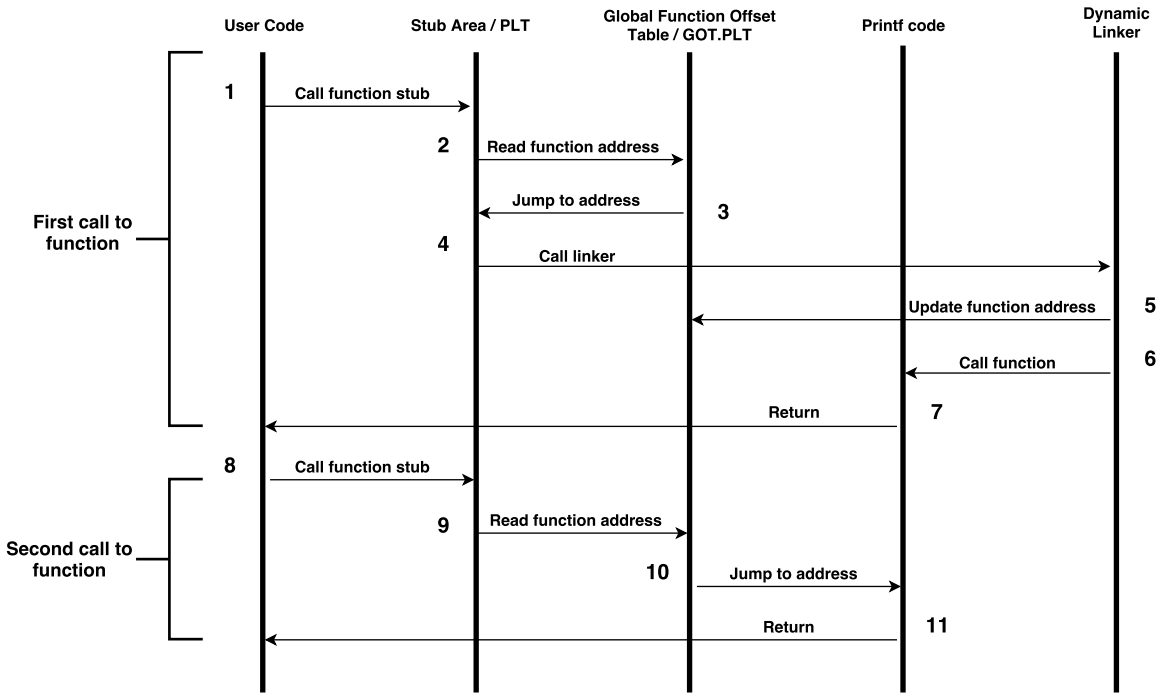

Figure 3.1: Function call and lazy procedure linking

Therefore, debuggers cannot be used as a learning tool properly, as they do not offer pointers as to what is happening in the linking and loading processes. Previous knowledge is required.

3.1.2 Complexity of Command Line Debuggers

Secondly, in order to use specific debuggers, the user must be accustomed to their functionality. Both *gdb* and *lldb* have a wide range of commands which make them very powerful tools, at the cost of user experience.

Prior to using them, one must learn what the proper commands are for what is intended, as well as what to understand from the output returned. For an inexperienced user whose goal is to understand the mechanics of dynamic linking and loading, this can be a time consuming and unnecessary process.

Moreover, using the debugger can prove cumbersome even for a user who is well accustomed to its functionality so long as the purpose is to simply observe the job of the linker and loader.

Debuggers are designed to be powerful and versatile tools which let the user decide what task they should be used for. They do not offer specific complex functionality, but rather provide a set of means to achieve that functionality.

3.2 Objectives

The preceding section underlines the issues with the existing approaches to understanding how the dynamic loader and linker work. Despite providing extensive functionality, debuggers are not a solution to the specific problem of inspecting the dynamic linker and loader, but rather a possible workaround.

This project aims to fill the need for a dedicated solution for analysing the dynamic linking and loading processes. DynInspector is a tool which provides full functionality for exploring dynamic linking and loading after an ELF executable is launched. It consists of two major parts:

- an extension to the *lldb* python module which implements dedicated methods for analysing the behaviour of the dynamic linker and loader;
- a user-friendly graphical interface which isolates the necessary functionality for interacting with the program during dynamic linking and loading;

3.2.1 Extend LLDB Functionality for Dynamic Linker and Loader Inspection

The *lldb* extension module is a wrapper over the python *lldb* module. It provides functionality for:

- Instrumenting a target ELF executable (eg. running, setting breakpoints)
- Retrieving and encapsulating data from the *.plt* and *.got.plt* sections.
- Offering information about the modules mapped in the program address space. (eg. shared libraries)
- Tracing dynamic loading calls. (eg. on Linux systems these are *dlopen, dlclose, dlsym*)
- Modifying custom memory locations.

The module is independent of the rest of the project and can be reused easily.

3.2.2 Create a Graphical User Interface

The user interface essentially provides the same information as the *lldb* or *gdb* debuggers. However, the data is retrieved through the *lldb* wrapper module and displayed on an easy to read interface in a descriptive format. This eliminates the need for direct interaction between the user and the command line debugger. Thus, the focus shifts to the content, rather than the form.

Moreover, the interface serves the user with a mechanism to control the execution flow of the target: adding breakpoints, continuing execution, restarting. However, the initial functionality of a debugger is substantially reduced and adapted to fit the specific needs of investigating

the dynamic linker and loader. For example, a user is not allowed to use the *step instruction* command in order to avoid following code irrelevant to the dynamic linking and loading process. Rather, the *step instruction* command is modified to continue to the next point of interest for analysing what the dynamic linker and loader do.

It is important that the distinction between dynamic linking and dynamic loading is clear to the user. In Section 2.1.1 I described how the two processes have intertwined tasks during the load and runtime stages of an executable. Presenting both modes simultaneously is confusing and does not help the user gain detailed understanding of each process individually. For this reason the graphical user interface offers two different inspection modes: one for understanding dynamic linking and the lazy binding mechanism and one for highlighting the key points of dynamic loading. The graphical user interface has a simple and similar aspect for both modes in order to maintain user-friendliness.

3.2.3 Objectives Summary

To sum up, the main objective of this thesis is to introduce an easy to use tool which benefits two category of users: those who aim to understand the functionality of a loader and those who want the functionality of a loader isolated. The tool is called DynInspector .

Chapter 4

Use Cases

The DynInspector tool is focused on providing an enhanced experience for observing the dynamic loading and linking processes. In order to clearly distinguish between dynamic loading and dynamic linking, it offers two different inspection modes: *Dynamic Linking / Lazy Binding Inspector* and *Dynamic Loading Inspector*.

4.1 Inspection of Dynamic Linking / Lazy Binding

The *Dynamic Linking / Lazy Binding Inspector* mode follows the steps the program takes in order to make a call to a function from a shared library. A complete use scenario of the mode is detailed below:

1. The user opens the tool and reads helpful information from the *Help Section*.
2. The user loads an executable of his choice. The tool makes sure the executable has the correct format and displays a proper notification if not.
3. The user starts the program. Execution is paused upon entering the main function.
4. The user chooses a function from a shared library to set a breakpoint on. A list of such functions from the given executable is presented to him. The program sets the breakpoint and displays a proper notification.
5. The user continues the program execution until the next breakpoint is reached. The breakpoint corresponds to a call to a function from a shared library. The user continues execution into the corresponding *.plt* stub. The user can view the contents of the *.got.plt* table. The inital function pointer indicates back to the next instruction in the *.plt*.
6. The user continues execution and the dynamic linker is invoked. The pointer from the *.got.plt* changes to the actual function address. After the function call, the user continues execution and returns to the caller context.

7. The user continues to the next call to the function. The program enter the *.plt* stub again and jumps to the address of the pointer from *.got.plt*. The actual function is called directly, without use of the dynamic linker. The user reads the descriptive messages from the console and observes how and when the *.got.plt* table content changes.
8. The user sees where the shared libraries are mapped into the memory of the program and can easily trace where the current instruction is. For example, while executing the code from the function *printf* the user sees the instruction pointer is in the code section of the *Standard C Library*.
9. The user can change the value of the *.got.plt* pointer for the function. This gives insight about how altering the address modifies the program control flow. On the next call, the code from the address introduced by the user is executed, rather than the actual routine.

The use case for this inspection mode comprises the following functionality:

1. Visualisation of the shared libraries required by target executable, including their offset within the address space.
2. Instrumentation of the dynamic loading and linking process which can be detailed into:
   (a) Setting breakpoints for a function which makes a call to a shared library.
   (b) Displaying the *.plt* stub at the point of each call to the function.
   (c) Displaying the *.got.plt* entry for the given function. It represents the pointer to the procedure in the shared library.
   (d) Basic stepping in the trampoline code until the linker (*ld.so*[1]) is called.
3. Displaying the detailed program memory map: shared libraries and executable sections.
4. *GOT Hijacking* simulation by modifying the *.got.plt* content.

4.2 Inspection of Dynamic Loading

The *Dynamic Loader Inspector* mode traces user calls to the dynamic loading mechanism. On Linux systems these calls are represented by the C functions provided by the *dlfcn* header[2].

A standard use scenario corresponds to the following steps:

1. The user opens the tool and reads helpful information from the *Help Section*.
2. The user loads an executable of his choice. The tool makes sure the executable has the correct format and displays a proper notification if not.
3. The user starts the program. Execution is paused upon entering the main function.
4. Breakpoints are automatically set on calls to *dlopen*, *dlsym* and *dlclose*.

[1] http://man7.org/linux/man-pages/man8/ld.so.8.html
[2] http://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html

5. On a *dlopen* breakpoint, the user is informed that a new shared library has been mapped in the address space. The user looks at the table of modules mapped in the program memory and recognizes the new dynamically loaded library. He also sees the first and last address where it has been mapped.

6. On a *dlsym* breakpoint, the user is informed of the symbol name and corresponding address returned by the *dlsym* function.

7. On a *dlclose* breakpoint, the user is informed that the shared library has been removed from the address space. The user looks at the table of modules mapped in the program memory and notices it has been removed.

The use case for this mode comprises the following functionality:

1. Tracing *dlopen*, *dlsym* and *dlclose* calls within the executable and displaying the information described in the previous points.

2. Displaying a simplified program memory map: shared libraries and executable, with the occupied address range and their size in bytes.

For each of the modes described above, the inspection tool provides the user with detailed information regarding all the steps of the linking and loading process. It offers guidelines as to what is being acted upon, why it is being done and when it is required.

Chapter 5

Architectural Overview

As detailed in the preceding chapters, the DynInspector tool focuses the capabilities of a debugger, in this case *lldb*, on the observation of and interaction with the dynamic linking and loading process. The focus is achieved through a set of design principles:

- **Create a user-friendly graphical interface.** The user should only interact with the tool through an intuitive and minimal graphical user interface. Unnecessary functionality should not be provided.
- **Ensure an interactive user experience.** The user should be able to control the target executable being inspected by means of the graphical user interface and nothing else. The GUI should provide the necessary functionality and be responsive at all times.
- **Extend the functionality of existing tools.** The functionality desired in order to observe the dynamic loader should be contained in a stand-alone module. The code should be reusable as a python module and should extend, not replicate, the existing *lldb* functionality.

The basic design of the DynInspector tool is illustrated in Figure 5.1. In the following sections I will present details about each component.

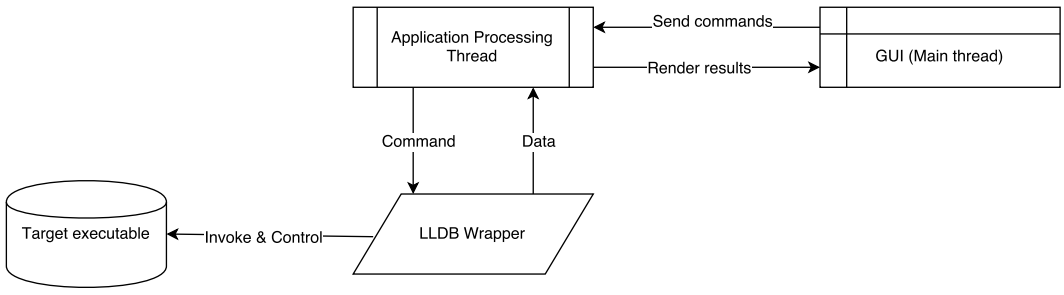

Figure 5.1: Architectural Overview

5.1 Graphical User Interface

The main purpose of the graphical user interface is to create a positive user experience. It achieves this by doing the following:

- **Displaying specific information** in its corresponding panel. For example, there are different panels for displaying the *.got.plt* table and the assembly code for the current frame.
- **Providing a 'Console Output Panel'** with detailed instructions and explanations. The user is aware at all times of what is happening in the code.
- **Interactive experience**. The user can restart the program, set breakpoints, continue the execution, alter the program memory to hijack the control flow.
- **Seamless interaction**. The GUI is responsive at all times as it runs on its dedicated thread. The thread it is run on is the main application thread.

5.2 Application Processing Thread

The application processing thread runs in the background of the user interface. It communicates with the GUI thread through signals and slots. The communication is bidirectional: the GUI sends the processing thread interpreted commands from the user and the processing thread provides the GUI with data to display.

Moreover, this thread is responsible for creating an instance of the *lldb* wrapper module. All the commands received from the GUI are executed by calling the appropriate methods from the wrapper module. The results from the wrapper are then interpreted and sent to the GUI thread to be displayed.

5.3 LLDB Wrapper Module

The *lldb* wrapper module is designed to offer stand-alone functionality for exploring dynamic linking and loading. Thus, it is a complementary feature to the existing *lldb* library.

Its functionality includes, but is not limited to:

- **Collecting and retrieving information related to linking and loading**, such as a detailed list of all the modules loaded in the executable, basic symbol information, the contents of the *.got.plt* section, a list of all the functions in the *.plt* section, as well as the stub of a specific *.plt* section entry or of the entire *.plt* section.
- **Control flow manipulation or instrumentation**. This refers to stepping at specific points of the program execution only relevant to the dynamic linking and loading process:
  - Starting / restarting the program

  - Continuing to the next breakpoint
  - Setting breakpoints in the *.plt* section
  - Setting breakpoints on *dlopen*, *dlsym*, *dlclose* calls

- **Printing** the assembly code of a frame or the C source code of a frame if compiled with debug symbols.
- **Altering the contents of a memory location from the program**.

The purpose of the module is to offer functions which directly provide specific information about the dynamic linking and loading process.

Chapter 6

Implementation

Previously I described the overall architecture of the DynInspector tool, presenting the main components and their roles. In the current chapter I offer technical details about the actual implementation of each part and their interaction.

6.1 Building Blocks

This section presents the hardware and software requirements of the project, as well as a detailed description of all modules used.

6.1.1 Hardware Requirements

The DynInspector tool aims to offer support for all 32 bit Linux platforms with the Intel x86 architecture. It is not compatible with 64 bit systems. Hardware dependencies are imposed by a tight coupling with the Instruction Set Architecture (*ISA*) of Intel x86 systems [1].

Firstly, the analysis is done based on the format and structure of the assembly code. Changing the architecture also changes the assembly instruction set to one the tool does not understand. Take for example a simple conditional jump, if two values are not equal, the x86 ISA defines a *jne* instruction, whereas the ARM ISA [3] offers the *bne* instruction. Searching for a *jne* instruction on an ARM architecture will result in an incorrect functioning of the tool.

Secondly, the tool requires direct access to registers which must be accessed by name. A simple use case is obtaining the return value of a function. On the x86 architecture, this is passed through the *eax* register, while ARM uses the *r0* register for this purpose. Since each architecture uses its distinct set of registers, they are not compatible.

Therefore hardware requirements are due to the low-level nature of the DynInspector project as it needs to systematically disassemble and analyse executables. In order to make the project compatible with other architectures, further development is required.

6.1.2 Software Requirements

The DynInspector tool is developed for 32 bit Linux systems. This choice is justified by the fact that Linux is an open-source community, as is the current project which makes a contribution in the form of a piece of software.

The project is strongly based on the existing *llvm* infrastructure and makes use of the *lldb* debugger to instrument the execution of a program and collect relevant data. Therefore the requirements for the *llvm* framework must also be satisfied in order to successfully use the DynInspector tool. The *lldb* version used in by the tool may render it incompatible with newer 32 bit Linux systems, such as Ubuntu 16.04, as feedback has revealed.

Another requirement is to have minimum Python version 2.7 on the system, as well as the Python Qt bindings module, PySide, with a minimum version of 1.2.1. Any system with newer versions of the mentioned software needs to make sure that they are backwards compatible with the ones presented here.

6.1.3 Development Environment

To achieve the desired functionality, the DynInspector tool is designed and built upon the following software support and hardware specifications:

- **OS platform**: Ubuntu 14.04 LTS
- **OS type**: 32 bit / i686
- **Target executable format**: ELF 32-bit LSB executable, Intel 80386
- **Programming language**: **python 2.7**
- **Graphical User Interface**: **PySide** python module **1.2.1**
  - Based on **QtCore 4.8.6**
- **lldb**: **lldb** python module **3.6**
- **Dependencies**: **llvm-3.6** and **clang 3.6**

As can be seen, the tool is based on the *lldb* debugger which was mentioned in Section 2.2.3. The purpose of DynInspector is to extend the *lldb* features to support specific interaction methods for studying the dynamic loader, while providing a user-friendly interface.

6.2 Implementation Details

The DynInspector tool is written in the Python scripting language. It is structured on two Python threads, one for the graphical user interface and one for background computations. The choice for using two threads is justified by the following two reasons:

- The graphical user interface is required to run on a dedicated thread in order to be responsive. The other computations require a background thread.
- There are no intensive background computations which makes one thread alone suitable for the task.

6.2.1 Graphical User Interface

As was previously described, the graphical user interface is the means by which the user interacts with the application. Its design has three main goals:

- **Display relevant information to the user**, such as:
  - The code currently begin executed
  - The information stored in the Global Offset Table (*got* on Unix) for each function
  - Explanatory messages about the linking and loading process
- **Offer an interactive interface to the user** by:
  - Allowing the user to step through the program
  - Allowing the user to restart the program
  - Allowing the user to set breakpoints for specific functions
  - Giving the user access to the standard input and output of the executable
  - Allowing the user to alter the memory content of the *.got.plt* section
- **Be responsive.** The graphical user interface communicates with the background seamlessly, without disturbing the user.

To achieve these goals, the graphical user interface is implemented using Python PySide bindings to Qt. PySide offers widgets which are classified within this application as either passive or active. Passive widgets are intended to display information to the user and can only receive commands from the background of the application. A text panel is an example of a passive widget. Active widgets can be interacted with by the user. They offer callback functions which are called on each interaction and which send commands to the background of the application. A button is an example of an active widget.

The graphical user interface is run on the application main thread and the background computations are run on a different thread. The graphical user interface registers and sends commands to the background thread. The thread runs the commands and notifies the GUI thread to update its information. This ensures responsiveness.

As stated in Chapter 3.2.2, it is important that the graphical user interface differentiates between dynamic linking, along with lazy binding, and dynamic loading. For this purpose there are two different application modes which I will detail below.

Dynamic Linking / Lazy Binding Mode. The Dynamic Linking / Lazy Binding mode highlights the dynamic linking process, more specifically address resolution with the lazy binding mechanism. Basically it shows what happens when a function from a dynamically shared library is called from a program.

Figure 6.1 shows the design of the graphical user interface for the Dynamic Linking / Lazy Binding mode. As can be seen, it contains a control and status bar, a window for displaying the code, a table with the *.got.plt* pointer values (or *intermediate stubs* table), a sections table for currently loaded modules and a console output window.

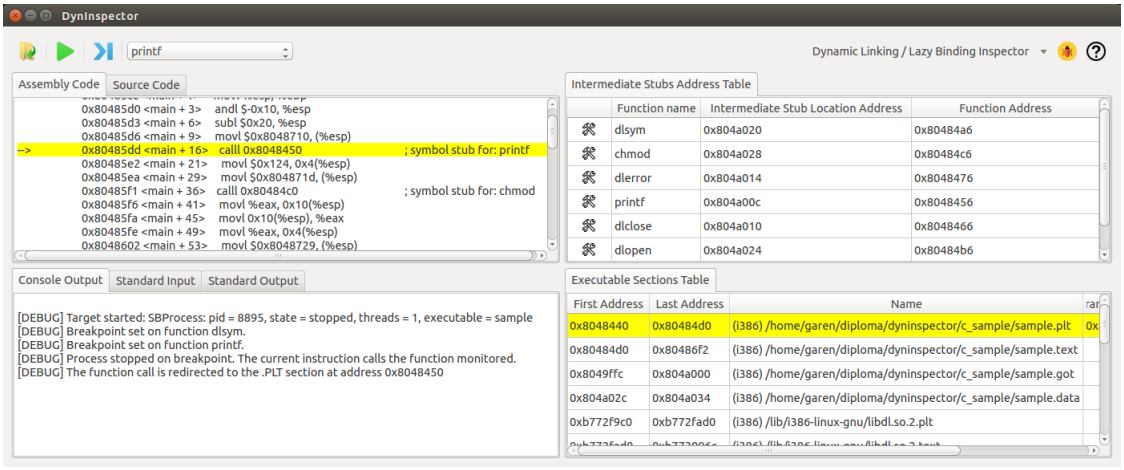

Figure 6.1: Graphical User Interface - Dynamic Linking / Lazy Binding Mode

The Status and Control bar offers a set of buttons for loading an executable, starting or continuing the inspection and a drop-down for selecting a breakpoint. The buttons have intuitive icons to describe their purpose, as well as text tooltips when the user hovers the mouse over them. On the right side of the bar, there is a button for switching the application mode. The current mode is indicated by the actual selection. Next to the right there is the *Debug Symbols Indicator*. This helps the user see whether the executable was compiled with debug symbols or not by hovering over the icon or by visual confirmation. The icon is either coloured or gray depending on the presence of debug symbols. To the far right, *the Status and Control bar* has a help window which opens a pop-up window with summarised information about the dynamic linking and dynamic loading processes structured on tabs.

The Assembly Code window displays the assembly code for the current frame of the program. The current instruction, as indicated by the instruction pointer, is highlighted with a different colour. Similarly, **the Source Code window** displays the original C code for the current frame. I considered this was useful for students who do not have experience with x86 assembly in order to follow the program execution more easily.

The Intermediate Stubs Address Table lists all entries from the *.plt* section of the executable. For each function, the location of the *.got.plt* entry is displayed in the *Intermediate Stubs Location Address* column and the routine pointer value found at that location can be seen in the *Function Address* column.

Executable Sections Table lists all the modules loaded in the current executable and displays the sections relevant to the dynamic linking process for each module. These are *.text*, where the code is, *.plt*, where the intermediate stubs are, *.got*, where the pointer values are located and *.data*. We can trace the program execution flow as it jumps through some of these sections by following the instruction pointer, which is displayed in the last column of the table. The row of the section where the current instruction is is highlighted in the table in order to make it more visible.

The Console Output window offers descriptive messages at each step. The messages are generated by the background thread based on the status of the debugging process.

The Standard Input window allows the user to introduce a keyboard input for the program. **The Standard Output window** shows the output of the target executable.

Dynamic Loading Mode. The Dynamic Loading mode highlights memory mappings for shared libraries and presents the user initiated function call mechanism for functions in shared libraries. Basically it traces *dlopen*, *dlsym* and *dlclose* calls.

Figure 6.2 shows the design of the graphical user interface for the Dynamic Loading mode. It is similar to the Dynamic Linking mode in order to help the user get accustomed to it.

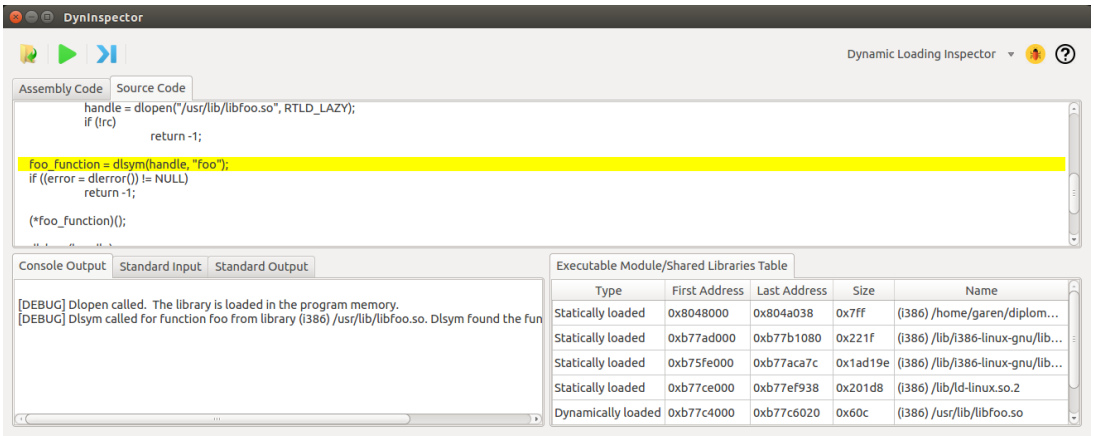

Figure 6.2: Graphical User Interface - Dynamic Loading Mode

The main differences are detailed below:

- **The function selector was removed from the status and control bar.** The nature of dynamic loading prevents us from knowing what symbols will be called dynamically until the call is actually made. For this reason, it is not possible to offer the user a list of functions to set breakpoints on. As the user cannot set breakpoints, the mode automatically stops on *dlopen*, *dlsym* and *dlclose* calls. Breakpoints to calls to functions from shared libraries are set automatically based on the return value of *dlsym* calls.
- **The Intermediate Stubs Address Table was removed.** This is only relevant for dynamic linking and lazy binding.

- **The Executable Sections Table was simplified to the Executable Modules / Shared Libraries Table**. Dynamic Loading is not concerned with particular sections of modules, only with their address mappings.

Overall, the graphical user interface has a simple and intuitive aspect which allows the user to use it without previous experience. Moreover, it offers all the functionality necessary for analysing the dynamic linking and dynamic loading processes with a minimum complexity by use of basic interactive widgets.

6.2.2 Background Work

As mentioned in the previous section, the application runs on two threads. One is responsible with managing the graphical user interface and the other with handling background computations.

In this particular case, background computations consist of:

- **Handling a debugger instance which runs an ELF executable.** This means sending commands to it and obtaining status information about the process being run. The debugger is an instance of the *lldb* Python module. It is wrapped in a stand-alone Python module which extends the basic functionality of *lldb* to include support for linking and loading analysis.
- **Processing the information from the debugger instance.** This step is necessary as the debugger instance offers the same amount of information as a command line debugger. However, the information is structured in appropriate data objects and is not a text string. The purpose of the current application is to provide selective information to the user. The parsed data is forwarded to the graphical user interface thread.

The background thread acts as an intermediate between the graphical user interface and *lldb* debugger. It keeps an internal state in order to correctly identify the state of the process currently run in the debugger. The state is changed based on the feedback from the debugger and the input from the graphical user interface (take for example a restart command from the user).

The background thread acts as a state-machine in order to follow the debugging process necessary for analysing dynamic linking and loading. The transition between the states is done when the user clicks the *Start* or *Continue* buttons from the GUI.

6.2.3 Communication and Synchronization

The applications threads communicate using Python *Signals and Slots* from *PySide*. It is a mechanism for user-defined callbacks. Each thread registers, or *connects*, a *slot* function to a *signal* object. The purpose of this function is to handle a certain event. For the signal, the event is *emitted* on demand.

Figure 6.3 illustrates the signals and slots mechanism.

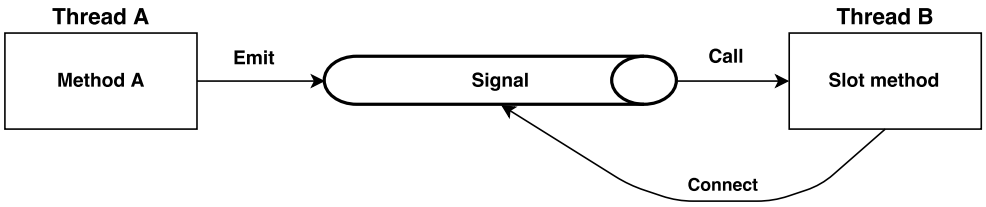

Figure 6.3: PySide Signals and Slots

The background thread defines all signals for communicating with the GUI thread. In this way, the functionality remains independent of the actual implementation of the GUI. The required signals represent an interface to the background of the application.

There are two kind of signals defined:

- **Backend to frontend signals.** These signals emit events from the background thread to the user interface thread. The events consist of updating the graphical user interface widgets with the given data.
- **Frontend to backend signals.** These signals emit events from the graphical user interface thread to the background thread. They represent commands from the user to the application.

The response of the background thread are instantaneous due to the performance of the *lldb* module and, thus, further synchronization is not necessary at this stage.

6.2.4 LLDB Wrapper Module

It was mentioned in Section 6.2.2 that the background thread creates and interacts with an instance of the *lldb* debugger. The instance is not used directly, but through a wrapper module which extends the functionality of the Python *lldb* module.

This module offers support for:

- **Printing the current frame as assembly code in a format which highlights what the linker and loader do**. For example, it displays the instruction addresses as <*function-name* + *offset*> and points out the calls to the *.plt* stubs for function with comments.
- **Printing the current frame as C source code**. This is useful for users who are not familiar with assembly code.
- **Returning the program counter from a frame.**
- **Obtaining symbol information.** For example, the name of the containing module.
- **Finding all the functions with stubs in the *.plt*.** These are the only ones relevant to the process of dynamic linking and loading.
- **Reading Global Offset Table entries.**

- **Providing information about shared libraries** mapped in the program memory (modules).
- **Creating breakpoints for the functions with stubs in the *.plt* exclusively**. These breakpoints are set on the first instruction in the function stub. The call to the function, identified by the previous instruction, can be viewed by printing the previous frame.
- **Creating breakpoints for *dlopen*, *dlsym*, *dlclose* calls**. Tracing these functions is useful for observing the dynamic loading process.
- **Execution handling**: run, continue, step. This limits the points where the program is stopped to events of interest for the linker and loader.
- **Printing of the shared libraries mapped in the program address space**.
- **Accessing the Standard Input and Standard Output of the target executable**.
- **Writing a data word to a given memory location within the program**. This is particularly useful for exemplifying a simple *.got hijack* where the *.got.plt* pointer is replaced with a custom address, usually to insert a hook for the function. The hook itself runs before actually calling the function.

The main advantage of isolating the above functionality in a stand-alone module is that it is independent of the current application and can be imported in other projects as well. The background thread aggregates the wrapper module and interacts with it by calling its methods.

6.3 Problems Encountered

Implementing the features described above was straight-forward in general. Using the Python language simplified the coding phase, whereas the existing functionality of *lldb* and PySide covered most of the needs of the application. Below I will provide details about the main issues encountered while working on this project.

6.3.1 Application Structure

The main issue is structuring the application in such a manner that it is properly designed and easy to understand. This is extremely important for sharing the code with other collaborators and further work on the project. The background thread needs to handle both interaction with the debugger instance and communication with the graphical user interface . For this reason its methods tend to perform several disjoint tasks which ultimately leads to a bad design.

This problem was solved by refactoring the code in several steps:

- Firstly, I created a function for each purpose.
- Then I removed unnecessary or duplicate code.

- Static data which needed to be shared across modules was made static within its corresponding class.
- Callback methods were set for events such as hitting breakpoints.

6.3.2 Setting Breakpoints

Another problem was setting breakpoints based on the function names in the *.plt* section. When setting a breakpoint on a function name, *lldb* sets the breakpoint on the symbol of the function. This means that several instructions must be executed before the actual code of the *.plt* stub is called which are irrelevant to the process of linking and loading and only confuses the user.

The solution was to set the breakpoints on the first instruction in the *.plt* stub for the specified function. However, this skips the call to the *.plt* stub, which might confuse the user. To make the process clearer, an intermediate step was introduced on breakpoint hit. It first shows the previous frame, where the call is made from, and only then moves on to the actual frame, which is the *.plt* stub one.

6.3.3 Control Flow

A constant issue was making the control flow of the program as clear as possible to the user. A setback in this sense was the fact that skipping the calls to the dynamic linker did not allow the program to return to the point where the *.plt* stub was invoked, but rather skip directly to the next breakpoint, if any. Based on the feedback received, this makes the process unclear for the user.

The problem was solved by setting additional breakpoints on the next instruction after each call to the *.plt* stubs. The address of this instruction is obtained by getting the instruction pointer from the previous frame immediately after entering the *.plt* stub.

Another control flow issue comes from blocking functions. The debugger instruments a target executable by skipping between breakpoints with a call to *continue*. The graphical user interface is updated only after the *continue* call returns. If a blocking call is intercepted between two breakpoints, the graphical user interface will not be updated and the user must first unblock the ELF executable before the DynInspector debugger resumes execution. This is unclear for the user. One possible solution is to completely remove the call to *continue* and gradually *step in* the program, instruction after instruction, and update the graphical user interface after each step. However this may introduce overhead in the application and consume system resources. This issue has not been addressed yet.

6.4 Lessons Learned

The problems described above emphasise the importance of a clean implementation which makes it easier to modify the code without breaking the existing functionality. Each module should be

written with the idea of 'Do one thing well!' in mind and the methods should respect a 'single purpose' principle, rather than aggregate as much functionality as possible.

The most important conclusion is that feedback on the application should be a top priority as only the user can give valuable insight on what is unclear or complicated. The DynInspector tool has an educational purpose and should be entirely developed based on user experience in order to effectively serve its purpose.

Chapter 7

Evaluation and Testing

Receiving constant feedback is extremely important in order to develop DynInspector as a useful educational tool. For this reason the main focus in the testing and evaluation stage was engaging the target users and soliciting feedback. Moreover, this phase was intertwined with the development process in order to comply to the user feedback and shape the final form of the application to meet the target users' needs.

7.1 Testing Plan

A testing plan for the DynInspector tool included deciding what is relevant for the feedback, who the target users are and how they should be approached.

7.1.1 Feedback Points

Relevant feedback points cover:

- **The graphical user interface design.** It should be intuitive and user-friendly.
- **Tool usage.** The user should be able to use the tool without additional support and understand what is happening.
- **Educational purpose.** After using the tool, the user should gain knowledge of what the dynamic linker and loader do.
- **Additional feedback.** This includes any particular user experience comments which can potentially enhance the functionality of the tool.

7.1.2 Target Users

The DynInspector tool is intended for both first and second year computer science students and hobbyists who are interested in understanding the functionality of dynamic linkers and loaders.

The purpose of the tool is primarily educational.

For this reason, the testing plan involved approaching the target users and allowing them to run and test the application. Further development was done based on the feedback received.

7.2 Feedback Received

Feedback was collected from a diverse set of users in order to form a complete image over the utility of the DynInspector application and how its objectives have been achieved. The majority of feedback was received from second and third year students which emphasised the actual utility of the tool to the target user. Other students and testers provided valuable insight on potential features and offered improvement suggestions for the existing software. Based on the feedback points mentioned in Section 7.1.1, I will present the most relevant observations received.

7.2.1 User Interface

In Section 3.2.2 I described the main objectives for the graphical user interface and in Section 6.2.1 the actual implementation was presented in detail. In order to asses if the interface satisfies the objectives, I evaluated the users' difficulty in using the interface and understanding what each component, or widget, does.

After the first version of DynInspector was tested, initial feedback remarked that the graphical user interface design was primitive and discouraged users from using the tool. This referred to the simplistic aspect of buttons which only had a text label and were placed in a vertical column to the upper-right side of the window. The vertical placement was not ideal as it wasted a lot of space and left little space for the *Intermediate Stubs Table*. I tackled this feedback by creating a dedicated *Status and Control Bar* with a similar design to that of IDEs such as *Eclipse*[1] in order to give the user a familiar sensation when using the tool. I changed the buttons to display icons, rather than text, with intuitive images which indicate their functionality. For example, the *Start* button has a green arrow icon with resemblance to the play button in media players. I also added tooltip messages for all the widgets in the bar to indicate what their functionality is. Not only did the tool become more visually appealing with these, but it also left more space for the other widgets to expand and display relevant content.

Another feedback point was that users were not aware that the application has two inspection modes. When asked, the two initial testers mentioned that neither did they know about a second mode, nor did they find the button for switching modes. The button was placed in the window bar in a similar fashion to program menus, such as *File->Save as...*. The first user who was running the tool on Ubuntu found the button when told, whereas the second user could not find it at all on Linux Mint. In order to make it simple and clear for all users, regardless of their operating system, I moved the button to the *Status and Control Bar*. I turned the label

[1] https://eclipse.org/

which displayed the current mode into a button which displayed a selection menu when clicked. The ten users who tested the application in its final form were able to switch the application mode without being told about the mode beforehand and without help in locating the widget.

Further feedback on the graphical user interface indicated that for most users it was not clear what information each window from the application, such as the Intermediate Stubs Table, was displaying because they had no title. Setting a tooltip message was not an appropriate solution in this case as the user expected to know what the windows were displaying at first glance, without having to move the mouse above the window widgets in turns. Therefore I set an appropriate title for each window, or panel, displayed on the interface and users informed that it was more helpful.

One of the more experienced users remarked that it is difficult for first year students to follow the program execution flow through the assembly code alone. This is because most of them take assembly courses in their second year. Moreover, both second and third year students displayed difficulty in understanding the mapping between the assembly code and the original source code. For this reason, I added an additional *Source Code* tab window to the *Assembly Code* tab window, functional only for executables compiled with debug symbols. It shows the content of the source file and also highlights the current instruction. The users can now look at both the assembly and source code which makes it easier to understand where the program is and what is happening.

Another feedback introduced an issue with programs which require console input. As the program is instrumented with the *lldb* python module, it does not have direct access to its own standard input and standard output. The user interface did not offer means to introduce console input to the program and therefore the program got stuck on the call. The issue was solved by adding a *Standard Input* tab window next to the *Console Output* tab window in the graphical user interface . Thus, when the user gets stuck on a blocking *getchar*[1] or *scanf*[2] call, he can introduce the values required from the *Standard Input* window and the program resumes execution. When the user introduces the input, the graphical user interface spawns an extra python thread. In this way, the graphical user interface thread is not affected. This new thread puts the user data to the standard input of the executable, which runs on the background thread, and allows it to exit the blocking call.

Further on, I added a *Standard Output* tab window to allow the user to follow the output of the executable, but also to distinguish it from the *Console Output* messages which have an informative purpose. The window is complementary to the *Standard Input* one.

The changes described above resulted in a positive final feedback. Users described the graphical user interface of DynInspector as being intuitive and straight-forward. This indicates that the objective of creating an easy-to-use tool was achieved.

[1] http://linux.die.net/man/3/getchar
[2] http://man7.org/linux/man-pages/man3/scanf.3.html

7.2.2 Tool Usage

In the previous chapters I argued that DynInspector was designed to be intuitive and self-explanatory and that an inexperienced user should be able to use it without previous knowledge and without assistance.

Overall feedback revealed that despite the straight-forward use of the interface, five out of twelve users were not certain what they were expected to do at first glance. They mentioned that only after a couple of consecutive runs did they understand what the tool was doing. To solve this problem, I created the *Help* button which offers a crash-course on the theory of dynamic linking and loading, as well as a short description of what each application mode does. As a result, users instinctively clicked *Help* and read the instructions when they first used the program. They figured out more easily what they were expected to do with the tool, where to look and how to interpret the data displayed.

One feedback mentioned that the tool had a confusing behaviour on calls to blocking functions. The program tested was the code for a C server with blocking functions such as *accept*[1] and *read*[2]. After further enquiries, I noticed that the behaviour was due to calls to blocking functions which led to the tool being stuck in the debugger *continue* call. As the user interface was updated only on return from *continue*, the user saw nothing happen on the interface and believed the program had crashed. However, in that particular case, it was an *accept* call from the code of server. Connecting with *telnet*[3] from a terminal window allowed the program to resume execution and display the proper information on the user interface.

The simple and intuitive design of the graphical user interface helped users figure out how to use the tool in the majority of situations. The feedback received on tool usage was tackled with a short tutorial on dynamic linking and loading and this proved to be an effective solution.

7.2.3 Educational Purpose

The most important objective of the project is to create a simple and straight-forward inspection tool for dynamic linking and loading which is adequate for inexperienced users interested in learning about these processes. In order to asses if DynInspector properly emphasises the essential aspects of dynamic linking and loading, I have analysed the feedback received from the users.

Initial feedback revealed that even though users could easily use the tool, they did not understand what dynamic linking and dynamic loading were. As their theoretical background was missing, they assumed they had to rely on external sources to find out more information, rather than use the DynInspector tool alone. For example, one of the first testers mentioned he had opened twenty browser tabs with information on dynamic linking and loading in order to fully understand what was happening. This problem was fixed when I created the *Help* section which offered a crash-course on dynamic linking, dynamic loading and lazy procedure linkage. This

[1] http://man7.org/linux/man-pages/man2/accept.2.html
[2] http://man7.org/linux/man-pages/man2/read.2.html
[3] http://linux.die.net/man/1/telnet

short introduction proved helpful as further feedback revealed the users had no need to seek information from other sources. By reading the *Help* manual, users gain the basic theoretical background necessary to use the tool and understand the concepts it presents.

The most recent feedback revealed that all the users who reviewed the DynInspector tool agreed that what is happening is simple and clear. They properly distinguished between dynamic linking and dynamic loading by using the two different inspection modes. The *Help* manual offers a basic theoretical understanding of the process, whereas the inspector itself helps deepen the knowledge with a hand-on tutorial on dynamic linking and dynamic loading.

Moreover, ten out of twelve users confirmed that they would not have been able step through the entire process by themselves, without using the tool and with no previous knowledge, as it was too complicated. There are two main reasons they presented. The first one is that they are not familiar enough with debugger tools such a *gdb* and, in most cases, the complexity of the command line debugger discouraged them from attempting to use it. It is important to notice that this feedback belongs not only to first year students, but also to third year ones. Secondly, they had no knowledge of the dynamic linking and dynamic loading processes and, as a result, they did not know what they should look at in the debugger output and what each step meant. This validates the motivation for creating this project and the final positive feedback supports that the main objectives have been achieved.

7.2.4 Additional Feedback

Aside from the main feedback points which validate the main purpose of the project, users have presented valuable feedback regarding the functionality of the project and compatibility with their systems.

Feedback revealed that seven out of twelve users had 64 bit systems which made the DynInspector tool unusable for them. In order to test the tool, they had to use a virtualization solution such a VmWare[1] and install a 32 bit machine. This process is cumbersome and might discourage users who do not have virtualization software already installed on their system from using the tool. For this reason, most users requested that further development includes support for 64 bit machines. This is discussed in Chapter 8.

Moreover feedback pointed out that there were issues with the *lldb python module* on more recent 32 bit systems such as *Ubuntu 16.04* [2]. On this particular system, lldb 3.6 does not recognize the symbols of the functions from the *.plt* section and therefore does not properly return them to the tool. Therefore the user cannot set any breakpoints in the *Dynamic Linking / Lazy Binding Inspector* mode. Additional testing proved that the *lldb-3.6* debugger has the same behaviour in the command line on *Ubuntu 16.04*. I verified that the same compiler version, *gcc 4.8.4*[3], was used to generate the executable and that the executable had the same format and the same assembly instructions as the one on the development platform, *Ubuntu 14.04*.

[1] http://www.vmware.com/
[2] http://releases.ubuntu.com/16.04/
[3] https://gcc.gnu.org/gcc-4.8/

Further enquires need to be made into this issue and a possible transition to a newer version of *lldb* which is compatible with all 32 bit systems should be considered.

Valuable feedback was also received with regard to the project repository on Github[1]. All users mentioned that the installation process was simple and clear by means of the *setup* script provided. They also appreciated the instructions on how to deal with possible issues during the installation process. Two users mentioned that the repository lacked a project description which made it unclear what the purpose of the DynInspector tool was. To solve this issue, I added a short description in the project *readme* file.

Overall, the feedback revealed that DynInspector has made an impact in the sense that its goal of presenting dynamic loading and dynamic linking in an accessible manner to inexperienced users has been completed.

[1] https://github.com/somzzz/dyninspector

Chapter 8

Conclusions and Further Work

In this thesis I presented DynInspector which is an educational tool for analysing the obscure process of dynamic linking and loading. DynInspector has a user-friendly user interface and allows an interactive inspection of ELF executables. The project has created a positive user experience and has achieved its purpose of transmitting relevant knowledge. The current implementation status has met all the objectives proposed in Chapter 3. However feedback revealed that there are further issues to be addressed in order to make the tool accessible for users on various other platforms.

In the following sections I will briefly present the current implementation status along with some of the most important conclusions drawn from working on the DynInspector project. I will also detail features that need attention, as well as compatibility aspects which need to be addressed in further work on the project.

8.1 Current Status

The tool can be run on Linux systems with Intel x86 architecture, for inspecting ELF executables, with the exception of Ubuntu 16.04 as feedback has revealed. Its dependencies include *python* with a minimum version of 2.7, the *llvm* infrastructure with *lldb* 3.6 and the *Pyside* python module for Qt bindings.

At present, the DynInspector tool offers support for inspecting the dynamic loading and dynamic linking process along with the lazy procedure linkage mechanism. This is done by using the two inspection modes of the tool.

8.1.1 Inspection of Dynamic Linking and Lazy Binding

This mode allows the user to step through the program between key points for dynamic linking. It also highlights the lazy binding mechanism. At present it comprises of the following features:

- Opening a target ELF executable
- Setting breakpoints on functions from shared libraries
- Visualisation of the Global Offset Table (*.got.plt*) and Trampoline Intermediate Stubs (*.plt*)
- Visualisation of the program sections
- Custom event-based stepping through the program

8.1.2 Inspection of Dynamic Loading

The Dynamic Loading Inspection mode presents the events on user calls to the dynamic loading procedure. There are *dlopen*, *dlsym* and *dlclose* for C code. The mode has the following functionality:

- Opening a target ELF executable
- Visualisation of the program modules (shared libraries) mapped in the program address space
- Custom event-based stepping through the program (on *dlopen*, *dlsym* and *dlclose* calls)

The above functionality can be achieved by using the fully functional graphical user interface which can be seen in Figure 6.1 and Figure 6.2.

8.2 Further Work

Further development of DynInspector should focus the main issues addressed by user feedback which are described in Chapter 7. After that, work on self-standing features should be considered. Below I will detail the main areas of future development.

8.2.1 Support for 64 bit Linux Systems

Systems nowadays comprise of 64 bit software with increased virtual memory address space to make use of the increased hardware capabilities. For example, these include 64 bit CPUs with 64 bit registers and buses and bigger RAM memories. Commercial computers have transitioned to 64 bit systems both hardware and software wise in recent years and as user feedback revealed, most users have 64 bit software.

DynInspector should offer support for 64 bit Linux systems in order to be used by a wider range of users. The current implementation is valid for 32 bit machines only. One of the main setbacks is that *lldb-3.6* does not properly extract the symbol stubs for the functions in the *.plt* section for 64 bit systems. An approach to solving this issue is switching to a more recent *lldb* version and making any necessary adaptations in the codebase. The *lldb* version chosen must be able to inspect 64 bit ELF executables on any 64 bit system.

Moreover, offering support for 64 bit systems also requires a thorough investigation of the executable format as generated by 64 bit compilers. It must be ensured that any difference in the ELF file is reflected by the manner in which the tool analyses it.

8.2.2 Migration to a Newer LLDB Version

Feedback highlighted some issues with the way *lldb-3.6* inspects 32 bit ELF executables on Ubuntu 16.04 specifically. Both command line debugger and python module do not find the symbol stubs for the functions in the *.plt* section. The symbols are clearly present as *gdb* can properly identify them and assembly comments inserted by the compiler can be visualised.

Similarly, one of the main issues of the current implementation with 64 bit systems is that *lldb-3.6* does not identity the symbol stubs in the *.plt* section. As was tested on 64 bit ELF executables, the DynInspector graphical user interface runs without errors and all widgets work as expected, displaying the correct information. The only setback is that the user cannot set any *.plt* breakpoints as no functions are identified in the *.plt*.

For the reasons stated above, further investigation is necessary into finding a *lldb* version which is compatible with all 32 bit systems, primarily, as well as all 64 bit systems. In case this is not possible, it should be considered that an adequate *lldb* version is installed depending on the user's system. Furthermore, compatibility issues between different *lldb* versions should be documented and resolved in the DynInspector project code.

8.2.3 Support for Blocking Calls

Feedback has pointed out that calls to blocking functions freeze the tool until the blocking call returns. Examples of such calls include server *accept* functions, socket *reads* and console input reads with functions such as *scanf*. This issue is thoroughly described in Section 6.3.3 and Section 7.2.2.

DynInspector should handle blocking calls in a manner which allows the user to understand what is happening and what is required to unblock the target executable.

8.2.4 Support for Parallel Execution

At present, DynInspector only supports inspection for single threaded executables. The tool extracts an instance of the first thread from the process of the executable. This thread is then used to obtain various information such as frame assembly code, program counter and so on.

Support for multi-threaded applications must tackle issues such as the correct management of all threads within the *lldb* wrapper module, as well as thread synchronization during the debugging session.

8.2.5 ARM Support

Currently DynInspector can analyse the dynamic linking and loading process for the x86 architecture. It would be useful to extend support to various other architectures such as ARM.

There are two main reasons why the current DynInspector implementation does not offer ARM support. Firstly, ARM defines its own ISA [3] which is fundamentally different from the x86 one. ARM is a *RISC* architecture, whereas x86 is basically *CISC*. Instructions differ from a conceptual point of view and not just by names. Secondly, ARM compilers generate a different executable format which cannot be analysed in the same way as its x86 counterpart. Being a low-level inspection tool, DynInspector relies on a very strict instruction order for the assembly code. Any minor difference may result in unexpected behaviour.

Therefore, offering ARM support must be done in two stages. Firstly ARM executables must be analysed and differences from the x86 code should be noted. Secondly, the current implementation should be adapted to reflect the differences between the two architectures. A more general approach is preferred to simply duplicating code.

Moreover, as there are more ARM ISA versions ranging from *ARMv1* to *ARMv8*, further documentation is necessary on their differences. A decision should be made as to whether it is possible to analyse executables in a similar manner for all ARM versions or not. If it is not the case, one option is to gradually offer support for all, or some, individually.

8.3 Conclusions

DynInspector is a dedicated educational tool for inspecting dynamic linking and dynamic loading. It is a light debugger with an user-friendly interface aimed at helping computer science students gain knowledge on the rather obscure dynamics of linking and loading.

The tool was evaluated by second and third year students primarily and the feedback received helped assess the quality of the software and improve user experience.

Firstly, the tool repository on Github was easily accessible and provided easy and clear installation instructions. The process was neither long, nor difficult and all the users who tested the application were able to get the tool with ease. To make the purpose of the tool clear at first glance, a short project description was also provided on the Github wiki.

Furthermore, the users were able to use the graphical user interface without additional help. The intuitive design and placement of the widgets produced an effortless interaction between the user and the application. Moreover, any difficulty in understanding the purpose of the tool was eliminated by means of the *Help* informative window and the console messages. Users who had no background on dynamic linking and loading found it most difficult to understand what the interface was showing them. The lack of theoretical knowledge was filled by the brief introduction to the dynamic linker and loader from the *Help* window. Afterwards, users were able to properly follow and understand the process by means of the widgets on the graphical user interface . Therefore, while the console messages provided descriptive information about

each step, the various tables, code view windows and buttons gave the user a hands-on learning experience.

Most importantly, users assessed the tool positively regarding its educational purpose. They understood what dynamic linking and what dynamic loading were hands-on and gained detailed knowledge, rather than theoretical concepts alone. They stated that the mechanisms were straightforward and the process overall was easy to understand using DynInspector . This underlines the fact that the key purpose of an educational tool was achieved and that a rather obscure mechanism was presented in a clear and easy manner.

Bibliography

[1] Joseph Cavanagh. *X86 Assembly Language and C Fundamentals.* CRC Press, January 2013.

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

[3] David Seal. *ARM Architecture Reference Manual 2nd Edition.* Addison-Wesley, 2001.