Detouring program execution in binary applications

MEng Final Report

Mike Kwan `mk08`
Department of Computing,
Imperial College London
`michael.kwan08@imperial.ac.uk`

Supervisor: Dr. Cristian Cadar
Co-supervisor: Petr Hosek
Second marker: Dr. Paolo Costa

June 19, 2012

Abstract

Extending an existing system and altering its behaviour are tasks conventionally associated with modification of source code. At this level, such changes are trivial but there are many cases where there is either no access to source code or it is not feasible to modify the system from that level.

We investigate the possibilities and challenges of providing a solution to this problem on the x86-32 and x86-64 Linux platforms. We present `libbf`, an elegant and lightweight solution using binary rewriting to provide a user with the ability to modify behaviour and weave new functionality into existing systems. We demonstrate its effectiveness on production systems by evaluating it on large-scale real-world software.

Acknowledgements

I would like to thank the following people:

- My supervisor **Dr. Cristian Cadar**, for his consistent advice, suggestions and guidance throughout.
- My co-supervisor **Petr Hosek**, for the many thorough discussions, invaluable report feedback and his constant enthusiasm.
- My second marker **Dr. Paolo Costa**, for spending time to give feedback on the various drafts of the report.

Contents

Chapter 1

Introduction

The ability to modify the behaviour of an executable is a powerful aspect of software development that is often not fully exploited. While editing the source code of executables is a routine practice in software development, there are many cases where there is either no access to source code or it is inconvenient to perform modifications at that level.

One such example is the situation where a bug is displayed only in the production environment. The classical approach is to recompile the code with integrated logging and to reproduce the bug. This allows a developer to isolate and gather information about the circumstances under which the bug occurs and to 'catch it in the act'. Unfortunately, this requires access to the source code and is often impractical if the system under test is already in production. Furthermore, when deployment is a heavyweight process, giving a user a logged or debug build can be cumbersome and disruptive.

An ideal solution would be to have some form of injectable logging which can be performed by an end-user in a manner that resembles the weaving of aspects in aspect-oriented programming [32]. That is, the cross-cutting behaviour of logging becomes modularised. In our exemplary scenario, the user would run some tool which patches logging functionality to the target system, either altering it or duplicating it and altering the copy. The following example illustrates what such a tool would allow us to achieve. Given the target system:

```
#include <stdlib.h>
#include <time.h>

int func1(int num)
{
  return 10 / num;
}

int main(void)
{
  srand(time(NULL));

  for(int i = 0; i < 5; i++) {
    func1(rand() % 5);
  }

  return EXIT_SUCCESS;
}
```

Listing 1.1: Target System

In this circumstance, `rand()% 5` will occasionally evaluate to 0 leading to `func1` performing a divide-by-zero causing the FPU to signal a hardware exception. The tool should allow a user to determine whether `func1` was invoked, and if so when, how many times and with what parameters/return values.

As a concrete real-world example, the issue tracker of *lighttpd* illustrates several cases with the exact scenario we are describing[1][2][3]. The tool would make it possible to provide users with a piece of code which would be injected into the binary. The tool would then patch the binary to make calls to the injected code which would collect the information necessary to hunt down the bug. Observation through tracing is only a single use of a tool capable of injecting arbitrary code into a binary. The ability to modify the flow of execution of systems is not a new concept and is known as execution detouring. Previous example usages include:

1. **Debugging** - *IF* [23] is a Windows tool instrumenting *Microsoft Detours* into a stealthy debugger. The tool provides the emulation of breakpoints through execution detouring rather than using standard implementations of software and hardware breakpoints. IF provides users with the ability to target an executable and mine information on the data/code flow, order of execution and parameters/return values of functions.

2. **Profiling** - Detouring has been used extensively for profiling and tracing. Notable implementations include *Valgrind*, *pixie* and *qpt* [20, 31, 33], which inject instrumentation to record execution frequencies and trace memory references.

3. **Interception** - *NOZZLE* is a tool which performs runtime heap-spraying detection [24] through the use of detouring. Once again, it is implemented on top of *Microsoft Detours*. The tool intercepts calls to the memory manager in the Mozilla Firefox browser.

We want to be able to perform execution detouring on Linux x86-32 and x86-64, where related work is relatively sparse. Essentially, Microsoft Detours exists as the de facto standard for execution detouring on the Windows platform, whereas there is no such tool for Linux. Some existing solutions on Linux appeared to be potentially promising but were later dismissed for reasons such as discontinued development or having unnecessarily complex interfaces which were difficult to use. The latter is certainly true of several solutions which provide limited support for detouring but are unpopular, mainly because they are large toolchains simply incorporating detouring as one of many provided features. For this reason, their usage is cumbersome and their API complicated (compared to Microsoft Detours).

1.1 Requirements

We intend to produce a lightweight and dedicated library for Linux x86-32 and x86-64 which provides execution detouring functionality for existing binaries where source code is unavailable. The functional requirements and aims for the library are:

1. `[bb_det]` (Basic block level detouring) - The library must allow users to override arbitrary functions with user-defined routines. As well as working at the procedural level, the library should expose an interface that allows detouring at the basic block granularity. That is, it should be possible to detour an arbitrary basic block to a user-defined detour function. The detour function will be invoked each time *instead of* the basic block.

[1] Bug #1278 (http://redmine.lighttpd.net/issues/1278)
[2] Bug #1116 (http://redmine.lighttpd.net/issues/1116)
[3] Bug #605 (http://redmine.lighttpd.net/issues/605)

2. **[bb_tra]** (Basic block level trampolining) - The library must provide the capability to extend the semantics of functions in the original code through trampolining (see Section 2.1.1). Similar to **[bb_det]**, it should be possible to trampoline both procedures and basic blocks. From a higher level, the implication of this is that a trampoline function can be inserted before an arbitrary basic block. The trampoline function will be invoked each time *before* the basic block is executed.

3. **[cfg_gen]** - (Control flow graph generation). In order to be convenient to a user, the library will perform analysis on the target binary to enumerate the routines and basic blocks contained within it. This analysis should take advantage of any available symbol information. To effectively represent the target, a control flow graph (CFG) should be generated in the form of a directed graph where nodes consist of basic blocks. The user should have the ability to traverse the generated CFG through the use of visitor patterns:

```
bf_enum_basic_blk(struct bin_file * bf, void (*handler)(struct bin_file *,
    struct bf_basic_blk *, void *), void * param);
```

Listing 1.2: API for traversal of the basic blocks in a binary file

Listing 1.2 illustrates an example of the proposed API for enumerating basic blocks. The `handler` passed into the function will be called as each basic block in the binary is traversed. As is common with the usage of C callbacks, the user is able to provide a parameter which will be passed to the handler function for each invocation. Multiple values can be passed by aggregating them into a **`struct`** and passing a pointer to it. The API will provide similar variants for handling the enumeration and traversal of procedures and instructions.

The library should expose an interface to directly access the generated CFG. This will allow people to build further on top of the library, such as creating a tool to analyse the differences between two binaries by examining and comparing their respective control flow graphs.

4. **[api_expr]** - (API expressibility). One important aim of the library is to provide a simple and clean API that abstracts from implementation details. Some existing solutions cater for many architectures and for this reason, require users to directly write small chunks of assembly which will be patched into the executable. While it is useful to expose such functionality, it is convenient to abstract away entirely from assembly language. There is a tradeoff between having a simple and expressive API and having a solution which is portable across many architectures. Since we are only targeting two architectures, there is no need to expose such general functionality. Instead we should aim to provide a rich API specifically for detouring and trampolining since this is the main focus.

   As another aspect of expressibility, a user should not be required to provide raw addresses to patch. If this was required, the usefulness of the library would be greatly diminished and it would not be feasible to use the library standalone. Instead, the user should be able to express detouring and trampolining by selecting source and destination basic blocks or functions from the CFG generated from **[cfg_gen]**.

5. **[standalone]** - One specific use of the library is within an existing project, which has its own runtime environment. This means an optimal solution would require no extra runtime requirements or separate environment to be distributed. Section 2.3 discusses how detouring solutions are either static or dynamic. As we will see, dynamic methods invariably require specific runtime environments so effectively, **[standalone]** limits our scope to static solutions.

1.2 Contributions

The nature of binary rewriting means there are two main challenges to this project. Firstly, the technical complexity of working with very low-level binary analysis techniques which are often deliberately abstracted from users. Secondly, a difficulty lies in providing an elegant and lightweight solution that abstracts from these exact same implementation details.

We design and implement a solution which uses executable editing to provide a user with the ability to weave new functionality and modify the behaviour of existing systems. To provide this in a standalone fashion we investigate various techniques to provide adequate supporting functionality. We solve this problem by designing and implementing static analysis engine and object file injector components.

We look at the limitations of current solutions and make improvements in terms of features, overhead and expressibility.

1.3 Report Structure

We start by taking a deeper look into some of the technical specificities of related concepts in Background (Chapter 2). This will help with understanding what considerations we must bear in mind. This will also enable us to examine related work more critically and ultimately justify planning and design decisions we make. After this, Related Work (Chapter 3) will discuss in further detail the existing work that has been done in this field. We will present our findings and discuss why we assessed existing solutions to be unsuitable. As well as this, we will look at what we can take away from these solutions.

The architecture and design of the solution is discussed and a high-level view of its constituent components presented in Design (Chapter 4). The actual implementation is discussed in Implementation (Chapter 5), where the details about how the library works from a technical aspect are revealed. This section will also cover the optimisations that were made to the library. Any major deviations between the original design and the actual implementation are highlighted here.

The final solution must work robustly and not just for toy examples. The library contains a test suite which is discussed in Testing (Chapter 6). The Evaluation (Chapter 7) benchmarks the success of the project. We also discuss the limitations of the library. Finally, the Conclusion (Chapter 8) concludes the project and discusses future work.

Chapter 2

Background

Execution detouring is widely used throughout the industry with established library implementations already existing. This section will cover common methods of detouring and discuss how they relate to the functional requirements defined in Chapter 1. Furthermore, this section will provide the intuition for many of the design decisions of the project by covering the relevant technical details of an executable's life cycle.

2.1 Detouring

From a simplified technical viewpoint, a detour is set by replacing the first few instructions of a target function with an unconditional jump to a user-provided detour function. In this way the code path can be redirected for all given calls to some function.

Address	Bytes	Instruction
`Func1():`		
`0x0`	`55`	`push %ebp`
`0x1`	`89 e5`	`mov %esp, %ebp`
`0x3`	`83 ec 10`	`sub $0x10, %esp`
`0x6`	`56`	`push %esi`
`0x7`	`53`	`push %ebx`
`...`	`...`	`...`
`0x5c`	`c9`	`leave`
`0x5d`	`c3`	`ret`
`Func2():`		
`0x5e`	`55`	`push %ebp`
`...`	`...`	`...`
`0x7f`	`c3`	`ret`

Address	Bytes	Instruction
`Func1():`		
`0x0`	`e9 59 00`	`jmp 5e <Func2>`
	`00 00`	
`0x5`	`90`	`nop`
`0x6`	`56`	`push %esi`
`0x7`	`53`	`push %ebx`
`...`	`...`	`...`
`0x5c`	`c9`	`leave`
`0x5d`	`c3`	`ret`
`Func2():`		
`0x5e`	`55`	`push %ebp`
`...`	`...`	`...`
`0x7f`	`c3`	`ret`

The disassembly listings above illustrate a detouring example in x86-32. The x86-64 implementation of detouring is conceptually identical but has minor technicalities that must be considered. In this example, `Func1` is modified to detour execution to `Func2`. The definition of `Func1` starts at `0x0` and the detouring process works by placing a `jmp` to `Func2` at the start of this definition. This means whenever `Func1` is invoked, instead of executing the body of `Func1`, the execution is redirected to `Func2`.

When a function is called, the address of the instruction after the `call` is pushed on the

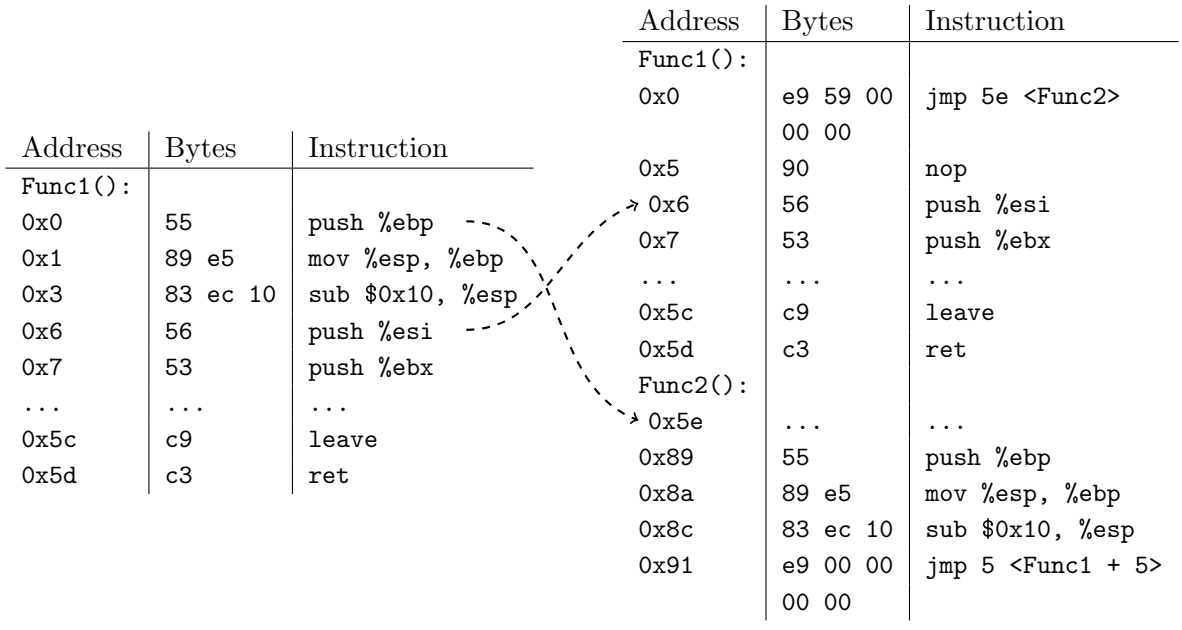
stack as the return address. When a `ret` instruction is executed, the top of the stack is popped and the value treated as the return address. Since the new code does not modify the stack, `Func2` will return using the original return address from the call to `Func1`. Effectively, a detour can be used to *replace* one function with another.

This example conveniently serves as a good demonstration of one of the challenges of patching code on the x86 architecture. The x86 architecture uses variable length instructions, which means instructions can not simply be swapped in one for one. In this example, the `jmp` instruction is five bytes long and patching it in overwrites two instructions completely (`push %ebp` and `mov %esp, %ebp`) and one instruction partially (`sub $0x10, %esp`). For clarity, we have padded the end of the partially overwritten instruction with `nop`. If the garbage byte (`0x10`) is left in, it would make the disassembly useless because the disassembler would attempt to disassemble an instruction starting from `0x5`. When modifying code, it is important to make sure execution is never detoured to the middle of an instruction because the instruction decoder will attempt to do the same thing.

Other methods of detouring exist but are less widely used. For example, instead of patching the target function, all calls to it can be replaced instead. However, to be reliable this technique requires substantial symbolic information which is not commonly available at the binary level.

2.1.1 Trampoline

Trampolining provides access to the original function by first executing the overwritten instructions and then unconditionally branching to the remainder of the target function. Hence, a detour either replaces the target function or extends its semantics by invoking the original function as a subroutine through the use of the trampoline.



In this case, the trampolining process works by first detouring `Func1` to `Func2`. In order to preserve the functionality of `Func1`, the overwritten instructions are then copied to the end of `Func2` to be executed before jumping back to the instruction directly after the first detour.

To provide trampolining functionality, some form of disassembling must occur. We must be able to calculate how many *whole* instructions have been replaced by the `jmp` instruction.

The reason for this is that it must be ensured that the destination of the trampoline's `jmp` at `0x91` is not to the middle of an instruction. This is achieved by `nop` padding the remainder of partially overwritten instructions. As well as providing us with the new instruction boundaries, disassembling is critical because it lets us know what instructions have been overwritten by the detour. Different instructions are relocated to the trampoline in a different way, such as relative jumps where it is not a simple case of copying bytes. Section 5.4 provides more details about this.

2.2 Executable Life Cycle

To approach the design of a suitable solution in a logical and methodical manner, it is essential to understand certain stages in the life cycle of an executable under Linux.

2.2.1 Compiling

Compilation marks the inception of a program, with the compiler providing the translation from the source file/s to an executable object file. The following is not intended as a comprehensive definition of compilation, but rather the conceptual fundamentals relevant to us.

2.2.2 Static Linking

When multiple `.c` source files are compiled, each one is first preprocessed (`cpp`) to generate intermediate `.i` files, then compiled by the C compiler (`cc1`) producing `.s` files, which are assembled (`as`) into `.o` relocatable object files. Since we have defined our scope outside of these stages, it is unnecessary to look at them in further detail. On the other hand, the linking stage is more interesting to us since some detouring methods emulate it to add code to the final executable. Static linking combines various relocatable object files to form the final executable object file. The process consists of two tasks [5]:

1. **Symbol resolution** - Object files typically contain three types of symbols:
   (a) **Defined symbols** - These allow other modules to call functions within the module defining the symbols.
   (b) **Undefined symbols** - These occur where the module calls other modules where the symbols are defined.
   (c) **Local symbols** - Used internally within the object file to allow functions to call each other and for the resulting code to be relocatable.

   The linker is responsible for collating all the symbols and ensuring a symbol reference is associated with exactly one definition. For linking to occur successfully, relocatable object files contain symbol tables which hold information about what symbols are defined and referenced by each module. It should be noted that the symbol table is often stripped fully or at least partially after the linking process.

2. **Relocation** - The compiler and assembler do not know where an object will reside in the final executable so generate code and data sections that start at address 0. The linker needs to *relocate* these sections, associating a memory location with each symbol definition. All symbol references then need to be updated to point to the assigned memory location. Relocation merges sections of the same type. For example the `.data` section of the final executable contains the aggregated `.data` sections from each of the input modules.

A common technique in executable editing is to introduce a new object file containing user defined routines to the final executable. This means that references from the original executable to symbols (usually functions) defined in the object file need to be resolved, as do references going the other way (trampolines). Naturally, a relocation stage must also occur but executable editing tools tend to avoid the complexity of further aggregation of sections by simply adding a new section.

2.2.3 Dynamic Linking

Dynamic linking postpones the resolving of undefined symbols until runtime. The executable contains undefined symbols, and a list of objects or libraries that will provide definitions to these. These objects are loaded at runtime and a final linking performed. Dynamic linking is a common technique for runtime detouring because of the ease of development. A shared object can be made with the same ease as a regular executable and the Linux loader will handle all the linking.

Position-Independent Code (PIC)

Shared objects need to be compiled such that they can be loaded and executed at any address. Such code is called position-independent code and can be generated with a simple *GCC* option (`-fPIC`). Although function addresses can be looked up at runtime by callers, the ELF system uses lazy binding which defers the binding of function addresses until the first time the function is called. The implication of this for modifying an existing binary to make calls to a new shared library is that the implementation would need to modify data structures within the executable (namely *Global Offset Table* and *Procedure Linkage Table*).

2.3 Approaches to Execution Detouring

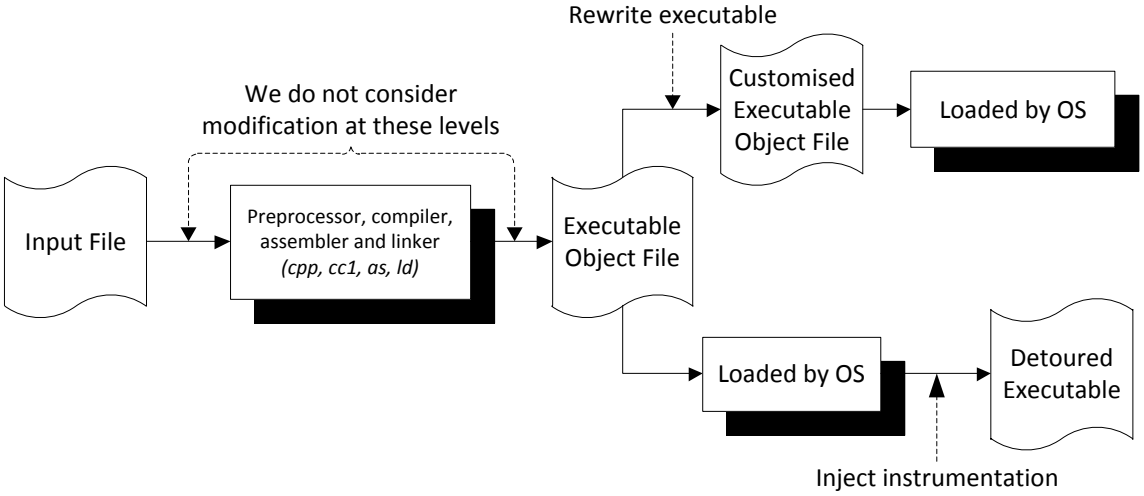


Figure 2.1: Our options for inserting instrumentation code. Since we only have access to the executable in its final compiled form, we focus solely on binary level options.

As shown in Figure 2.1, code modification can be performed at any stage of the compilation process: source, object or binary level. Source level modification allows instrumentation code to

be added directly into a program's source file before compilation occurs either by modifying the compiler or by using the preprocessor to insert special code [4]. Object level modification involves changing object files between the assembly and linking steps [10]. However, this requires a modified linker or relinking of the entire program and cannot be applied to programs where object files are unavailable. Since we are only considering executables after compilation, we can assume both source and object files are unavailable. This limits our scope to binary level execution detouring which falls under two categories:

1. **Executable Editing/Static Binary Rewriting** - This concerns the act of patching code into an executable before it is loaded by the operating system. The injected code is designed for the interception of functions and redirection of code execution.
2. **Runtime/Dynamic Detouring** - In this case, the interception code is applied dynamically at runtime, after the operating system has loaded the program into memory.

2.3.1 Executable Editing vs Runtime Detouring

Also known as static code annotation [15], executable editing is seen as the most convenient to the end user because a single command can be applied to one executable file generating a customised program which can be easily distributed. In comparison, with runtime detouring users are often required to set up a specific runtime environment which might be difficult to use and deploy. In the cases where the production system is on a cloud system such as EC2, this means reconfiguring many environments.

However, the attractiveness of executable editing is reduced when considering the complexity of implementation. Despite being conceptually trivial, executable editing is complex to implement in practice because of architectural and system-specific details. This increases the time and effort required to produce such a tool. Furthermore, at the binary level, symbol-table information is often incomplete and significant analysis must be performed to properly relocate code and data [16].

2.3.2 Executable Editing

Conceptually, binary rewriting generally entails a standardised procedure:

1. **Code discovery/analysis** - During this stage, a tool typically analyses the binary and provides an interface to enumerate routines and code blocks which can be split into their constituent instructions. The tool must be able to accurately distinguish between code and data in this case.
2. **Insertion of instrumentation code** - The insertion stage tends to happen separately and allows insertion/deletion of instructions from the structures discovered in the first stage. It is standard for a tool to allow both the modification of the original binary as well as to provide an option to create a new customised one.

Register Scavenging

When inserting code to an executable, it is critical not to break the original code by overwriting data or otherwise functionally changing the program in an unintended way. At the assembly language level, this means it is important not to overwrite registers that are in use. Register scavenging uses control flow analysis to provide information about unused registers at any point in a basic block [31]. If unused registers can not be found, a last resort is to `push` and `pop` registers to temporarily evict them so they are available for usage.

Binary Manipulation

When manipulating a binary, it is important to build on top of standard libraries where possible. Some existing solutions bundle their own implementations of libraries to deal with specific file formats. This presents issues because of the size of the codebase. If a bug is found and the internal library is no longer maintained, any code dependent on it breaks. This is exactly what happens with one of the existing tools we discuss in Chapter 3. Currently, there exist two libraries for dealing with binary access and manipulation.

ELF (Executable and Linkable Format) is the file format for executables, object code and shared libraries on Unix-based systems. *libelf* is a library used to directly read, modify and create ELF files in an architecture-independent way. It would be ideal to abstract away from the ELF file format since it locks down our solution. If our solution depends on `libelf`, it makes it more difficult to port to other architectures using different file formats in the future. Although currently we aim only to support x86-32 and x86-64 Linux, there is no reason to make a design decision which enforces this limitation for the long term.

BFD (Binary File Descriptor) is a package which allows applications to use the same routines to operate on object files whatever the object file format [7]. *libbfd* encapsulates many file formats over many different architectures and presents a common view of them and then provides a unified API to this view. As we will see, some existing Linux binary rewriters build on top of `libbfd` and others directly over the ELF file format, performing modification through `libelf`. While the abstractions BFD provide limit functionality in comparison to dealing with the specific file format directly, it makes it powerful for this precise reason, allowing any library built on top of it to effortlessly and automatically handle differences between and hence support the various formats.

2.3.3 Runtime Implementation Approaches

To perform detouring at runtime but achieve the same effect as if it was performed statically, the target must be forced to execute special code as part of its startup - before it starts execution. The purpose of this code is to ensure the instrumentation code is mapped into the target's address space and also to either perform the placement of the detours or implement some system which can perform the detouring on the fly. We now examine different implementation approaches employed by existing tools and research projects.

Shared Library and Position-Independent Code

Once we have the ability to execute arbitrary code in a target, the placing of the detours at runtime is a trivial matter. The issue we face is how to insert the instrumentation code in the first place and also how to override execution to make sure our special code is run before anything else. One method is to patch the entry point of the target to dynamically load a shared library containing the instrumentation code through the `dl` interface with functions such as `dlopen`. While this does require static insertion of the bootstrapping code, the amount of work required is minimal. One further advantage is that the shared library will already have been compiled to have position-independent code, which means that the Linux loader deals with all the linking of our extra library, something we are required to perform manually with binary rewriting.

```
void init(void) __attribute__ ((constructor));

void init(void)
{
   /*
    * This function is invoked as soon as the library
    * is loaded. In a dynamic implementation, we would
    * place the code to perform the detours here.
    */
}
```

Listing 2.1: This example would be compiled as a shared library

GCC conveniently provides a constructor attribute, `__attribute__((constructor))` which can be used to specify a function which is invoked upon loading of the shared library. With this approach, the constructor would set up the hooks before the main program starts executing as shown in Listing 2.1.

An alternative to patching the executable's entry point to load the bootstrapping library is to use the `LD_PRELOAD` environment variable which adjusts the runtime linking process by preloading a shared library into an arbitrary process. Unfortunately, `LD_PRELOAD` can be subverted by the application which means this option is unreliable.

Exception Handlers

Methods used for general debugging often require the breaking or redirection of code flow. One common form of this is breakpointing which comes in two flavours - hardware and software breakpoints. The breakpoint suspends execution before some given instruction is executed and the programmer inspects the program context at that point. As well as inspecting the context, it is possible to modify it, or more specifically the instruction pointer. In the context of a detouring implementation, software breakpoints are often used by replacing instructions with the `int 3` opcode which when executed triggers an interrupt handler. The interrupt handler can be overridden to modify the `eip`/`rip` register [4, 12, 14].

Similarly, the access protection of memory pages can be changed to become non-executable pages so each access invokes a global exception handler. As with breakpoint trapping, the ability to trap each instruction implies the capability to redirect code flow. Although this approach is interesting, the overhead is too large to be useful in practice.

Chapter 3

Related Work

Given the limited amount of related work in the Linux arena, it is well worth it to look at work that has been done for other operating systems as it may influence our final design. The related work will also act as case studies of current techniques used for detouring.

3.1 Microsoft Detours

Detours is a mature proprietary Microsoft library for the Windows operating system which provides functionality for the interception of arbitrary Win32 functions [11]. The library only allows detours to be inserted at execution time by modifying memory as with the dynamic approach described earlier. The rationale given for this is that it facilitates interception at a very fine granularity allowing detouring of one running instance of an application whilst another instance of the original application runs alongside. However, this is not an exclusive feature of dynamic detouring since executable editing tools often allow the creation of a new executable.

When Detours was initially released, it introduced trampolining, a feature not seen with other detouring packages released at the time. Other features supplied by Detours are Windows-specific such as the ability to edit import tables of binaries. Detours instruments code extremely efficiently with benchmarks placing interception overheads at less than 400ns on the 200MHz processors the library was initially tested on. Another advantage over binary rewriters is size, with Detours adding less than 18KB to an instrumentation package. However, this comes at the cost of an inability to insert code between instructions or basic blocks. Binary rewriters can typically insert instrumentation arbitrarily by performing sophisticated code analysis with techniques such as register scavenging. On the other hand, Detours relies on matching calling conventions to properly preserve register values at the function level. For example, `stdcall` designates `eax`, `ecx` and `edx` for use within a function and hence these registers are trashable by instrumentation code. This method would not be appropriate for our case since we require modification at the basic block level where looking solely at the call convention would not help determine free registers.

3.1.1 Advantages and Disadvantages

Detours is a lightweight library with a distinctly clean API which has no doubt contributed to its success on the Windows platform. The library is written such that a user can make good use of it with only a superficial understanding of Windows internals. Even though Detours is only available for Windows, we can likely borrow from or at least be influenced by its API if we implement dynamic detouring. From a technical level, Detours works in a similar way to the `LD_PRELOAD` method we described previously. It performs a very limited amount of static

manipulation on the target binary's import section to have it load the DLL (Windows' equivalent of a shared library). Since the details of editing the import section are Windows-specific, we shall not go into it further. However, later we will see the same technique employed on the Linux platform by LEEL.

3.2 EEL

EEL (Executable Editing Library) is a library for building tools to analyze and modify compiled executable programs on SPARC systems [17]. EEL works by removing existing instructions and adding foreign code that observes or modifies the original program's execution. Binary rewriting is non-trivial so it is worth it for us to discuss in more detail the intricacies of this technique. The tool provides five major conceptual abstractions in the form of C++ class hierarchies:

1. **Executable** - This is the top-level abstraction which represents any container of executable code, regardless of the specific format.

2. **Routine** - Executables contain many routines, which are discovered by the library through static code analysis.

3. **CFG (Control-Flow Graphs)** - A CFG is a directed graph with nodes as basic blocks and edges representing control flow between these blocks. The CFG is also generated through static control-flow and data-flow analysis.

4. **Instruction** - Basic blocks are composed of these machine-independent descriptions of opcodes. Each instruction object holds intimate information about what registers is reads/writes, which aids data-flow analysis.

5. **Snippet** - Snippets encapsulate the machine-specific foreign code that is to be added (instrumentation code). Snippets also take responsibility for register allocation through the use of register scavenging. Snippets often need to be written in assembly language although the authors argue this is not a drawback because the code is usually short and carefully written for efficiency. Useful work is done by separately inserting user-defined routines which the snippets call out to.

The abstractions EEL provide are not merely there to hide the complex and low-level implementation details but more so to provide machine-independence. All static analysis performed by the library is completely machine-independent and is supported by requiring every implementation of the EEL library to contain a separate backend-frontend mapping of architecture-specific instructions to EEL's own representation. Theoretically, the library could be ported to any architecture if the appropriate translations were made from EEL instructions to the machine-specific instructions. However, EEL was originally written for RISC instruction sets such as SPARC and MIPS. For this reason, CISC instructions are difficult to synthesize effectively in terms of EEL instructions. For example, string-manipulation instructions such as `movsbl`, `scas`, etc. do not interact with registers in a trivial way. It is not ideal to model the dynamic behaviour and internal control flow of x86 CISC instructions such as these against classic RISC instructions. One approach is to model a CISC instruction as multiple RISC instructions [30], but this is inconvenient, complex and exposes underlying implementation details thereby breaking abstractions.

Despite the focus on portability, EEL still has architecture-specific details even within its instruction class due to the non-generic nature of some of its target instruction sets. An example of this is representation for delayed branching which is inherent to SPARC but completely absent

in the x86 instruction sets. Details such as these unnecessarily add extra complexity to the final library if we were to port it to x86-32 and x86-64.

The portability in itself is a double-edged sword since EEL has catered for it to such an extent that the library and API has become clunky and complicated in comparison to libraries such as Detours. The last update to the source of EEL was well over a decade ago and it is likely the code is now too outdated to be of feasible use to us. Furthermore, the library is already far too bulky to consider extending and this is driven home by the unacceptable size markup when adding even basic instrumentation code. A simple program written in C with a size of 8KB is bloated to 350KB after insertion of a minimal amount of instrumentation code which simply instruments all routines to print their names [35]. The reason for this is that EEL inserts code from an object file containing the new routines. Since it is unknown whether these routines call sub-routines, the whole object file is inserted. One of the requirements for our tool is that it must be lightweight and have minimal impact to performance so we must avoid this problem in our final solution.

Advantages and Disadvantages

EEL presents a complete system for executable editing even though it is too vast to be of direct use to us. Despite the fact that the EEL source code is not available, it is interesting to consider the concept of the machine-independent representation which adds a layer of abstraction above machine-specific opcodes and allows the library to be ported to multiple architectures (previous versions also ran on MIPS, but the most recent version works only for SPARC). Some of the more novel design decisions such as the use of CFGs and the use of an abstraction layer are certainly tempting features to include in our implementation if we opt for executable editing. However, we are only considering Linux x86-32 and x86-64, so adding an extra layer of abstraction may not be necessary since the two instruction sets do not differ greatly. Instead, a more appropriate option as an abstraction layer might be to build on top of the BFD library.

3.3 LEEL

LEEL (Linux Executable Editing Library) is an executable editing library for Linux systems running on Intel x86-32 processors [35]. The project was motivated by EEL and the workflow of tools using both libraries are similar. LEEL identifies shortcomings of the original EEL library and addresses them by adapting a new design. The final product closely resembles EEL from a functional aspect so we will discuss mainly the additional features offered by LEEL[1]:

1. **Editing of multiple formats** - LEEL supports editing of executables, relocatable object files and shared libraries.

2. **Insertion of user-defined routines** - Insertion of user-defined routines differs significantly to what we have seen with EEL. It is useful to recall that EEL inserted user-defined routines by requiring them to be supplied in an object file which was then inserted into the target. Similarly, LEEL recommends users do most of their processing in an external routine instead of in the snippet itself. However, instead of defining the external routines in an object file, LEEL requires the routines to be contained in a shared library. As we will see later, this is a similar approach to that taken by Etch.

[1]It should be noted that the similarities and differences discussed are concerning LEEL acting upon executables. It is outside the scope of this project to consider editing relocatable object files and shared libraries.

3. **Insertion of snippets to CFG** - Snippets in EEL need to be written as small chunks of assembly, but LEEL expects most work to be done in external routines. With this expectation, it simply exposes an API which allow users to generate the assembly code for common tasks such as calling an external routine.

4. **Lightweight** - One of the problems with EEL was the huge file bloat after the injection of user-defined routines. Files generated by LEEL have a small overhead in comparison (a sample program with an original size of 11KB is increased to less than 17KB). The problem was solved by having users compile their routines into a shared library and calling out to these routines from instrumented code. A side effect of this is that the development of this shared library and the creation of the snippets can be performed independently as long as function names and signatures are agreed upon. Furthermore, after the new executable is generated, the shared library can be continually updated without requiring further executable editing as long as it continues to adhere to the mutually accepted function names and signatures.

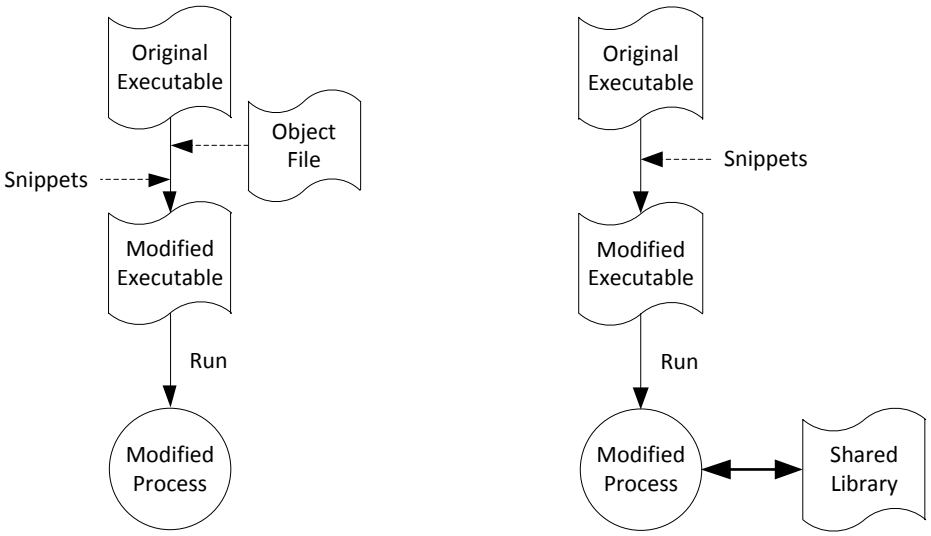


Figure 3.1: The left-hand side shows an executable being modified by EEL. An object file is inserted containing user-defined routines. Snippets allow the original code to be modified to make use of the new routines. The right-hand side shows the same process via LEEL. Snippets are inserted to make use of external routines in a shared library. The shared library is made available at runtime.

The only major stage we did not elaborate on is the static analysis stage of LEEL. As with EEL, the library must perform duties before modification of code occurs. Namely, routines need to be identified and a CFG generated - code flow analysis. However, it is uninteresting to delve deep into this since the process does not differ substantially from EEL. The main difference is in the data flow analysis, which LEEL does not perform at all. The justification for this is that with EEL, snippets are expected to be highly optimized and carefully written and this is why register scavenging is performed (which requires data flow analysis). With LEEL, the focus is less on the snippets and more on the external routines so the code is simply wrapped in `pushad/popad` to automatically save and restore the registers. Register scavenging is also less valuable on the Intel processor because of the relatively few number of registers in comparison

to SPARC and is not worth the extra complexity of analysis attributed to a more complex instruction set. One midway alternative that LEEL could have used is matching call conventions as part of the agreed external routine signatures, which would inherently allow the snippet to know which registers were used in the external routine. This is the technique used by Detours.

Advantages and Disadvantages

While LEEL offers many desirable features, it has a fundamental abstraction design which is unsuitable for us. The library achieves this by building on top of the ELF library. This was a deliberate design decision and reflects one of the key differences in the aims of the two libraries. Whereas one of EEL's main focuses was portability, LEEL intended only to achieve portability to Win32. Since there exists no backend port of BFD to Win32, this resulted in the decision to work directly with ELF instead. Porting LEEL to use BFD as an abstraction layer is infeasible. It would require almost entirely reprogramming the backend and it would take a lot of effort to avoid breaking the support for the specific cases of relocatable object files and shared libraries. In fact, even porting the library to support x86-64 would be very tough because of the tight coupling against the x86-32 instruction set. As a minor point, LEEL also does not support trampolining although this would not be overly difficult to implement. Despite these drawbacks, LEEL presents good solutions to the same drawbacks we identified with the EEL library.

Executable editing is a complicated process and both EEL and LEEL recognise this by demonstrating huge efforts to minimize the amount of code instrumented directly to the target (snippets) by allowing the compilation of the majority of the code (user-defined routines) to be performed separately by the compiler. This code is then integrated at two different stages, either by merging the object file with the executable statically (EEL) or merging the code in the form of a shared library at runtime (LEEL). Overall, LEEL can get away with performing less binary rewriting by shifting the focus to the external routines, but the price for this is that it needs to deal with the additional complexities of:

1. **Modifying the executable to use the shared library** - The target has to be modified to act as if it had originally been dynamically linked to the shared library. This process involves an intimate understanding of the file format.

2. **Calling of external routines from snippets** - The snippets must deal with the complexities of interacting with the position independent code of the shared library, where the absolute address of the external routines is not known statically.

3.4 Etch

Etch is a general-purpose tool for rewriting arbitrary Win32/x86 binaries without requiring source code [25]. Binary rewriting under the Win32 environment poses complications which are not present in UNIX-based environments due to system-specific features. It is outside the scope of this project to discuss such challenges, so we shall omit them from this report and focus on the higher level concepts of Etch.

Unlike EEL, Etch weaves the instrumentation phase tightly with the code discovery phase, but uses a separate analysis stage. Etch defines two phases: an instrumentation phase and an analysis phase. The instrumentation phase represents the insertion of instrumentation and modification of the program and the analysis stage represents the program at runtime. A tool created using Etch is similarly split into an instrumentation module and an analysis module with the analysis module being loaded with the executable at runtime. An approximation of the workflow is:

1. **Code Discovery** - Etch performs static code analysis to discover the components of the program. *Figure 3.2* illustrates how Etch views the program hierarchy.

2. **Callback** - The instrumentation module supplies a callback which Etch calls upon discovery of each component. Each invocation of the callback provides an opportunity to insert instrumentation or modify the executable. This modification can happen at the instruction, basic block or procedural level. Inserted instructions may include calls to procedures in the analysis module.

3. **Executable Generation** - The complete traversal of the executable concludes the instrumentation stage and a new executable is generated.

4. **Running the Executable** - The executable is run alongside the analysis module. Etch provides a hook providing notification to the analysis module of program completion. The module can then optionally run some analysis routines based on data collected during execution.

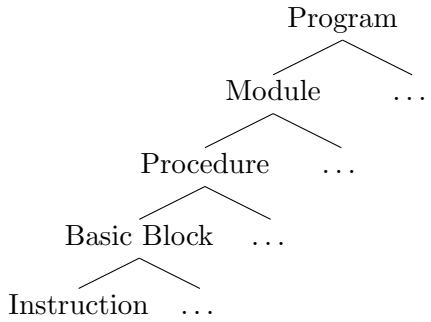


Figure 3.2: Etch views a program as a collection of modules. Each module contains procedures, which are composed of basic blocks which in turn are composed of instructions.

Etch provides additional features which are less related to executable editing. For example, it provides facilities to rewrite an executable in order to improve its performance by reordering instructions to optimize code layout to take advantage of spatial and temporal locality. This is a relatively complex example and application of data flow analysis alongside executable editing.

Advantages and Disadvantages

Etch presents a vastly different front-end interface compared to EEL, working less directly through the callback system established in the instrumentation module. Since Etch is not an open source project (nor even publicly available), to some extent, we can only speculate about the underlying implementation of the tool. However, it would appear that the separation of 'instrumentation' and 'analysis' simplifies the task of the editor slightly. By mapping the analysis module in at runtime, calls from the instrumentation can be dynamic which lightens the workload of the executable editing part of the tool by decreasing the amount of code that needs to be injected statically. Essentially, the task of code injection is partially delegated to the OS loader to be performed at runtime, which makes Etch more of a static/dynamic hybrid rather than a strict static binary rewriter.

Aside from the technical ramifications, the separation of the two duties reflects a different approach to the creation of a tool through Etch. It allows a user to treat the analysis module as a library and also decouples the instrumentation from the logic of analysing the data gleaned at

runtime. In a way, this is more natural than the approach taken with EEL which tends to focus on the injection of small and optimized snippets of code.

3.5 ELFsh

ELFsh is an interactive, modular, and scriptable ELF machine for static binary instrumentation of executable files, shared libraries and relocatable ELF objects [1]. ELFsh differs from most other implementations in that it does not come in the form of a library nor does it provide a programmatic interface other than in the form of a scriptable command-line. The main way to use ELFsh is interactively through the command prompt. The tool provides access to many low-level features such as reading and writing of ELF sections and symbol tables. However these features are not only file-format specific, but also do not directly provide useful functions to an end-user and should be abstracted in our tool. The reason these features are available in ELFsh from the user-level is mainly because of the implementation approach taken. From a high-level, ELFsh provides the following features:

1. **Injection of instrumentation** - The instrumentation must be in the form of an object file and is added into the target as if the binary has not been linked yet. Internally, this works by adding a library dependence to the main object [19].

2. **Function redirection** - By exploiting the way that dynamically linked function calls are resolved, existing procedures can be hijacked. When resolving symbols, the runtime linker iterates over a *link_map* list and resolves each symbol to an absolute runtime address where the function is mapped. By forcing some symbols to be resolved in priority, ELFsh is able to control the procedures to which symbols resolve to. ELFsh does not provide convenient access to the original function. Instead, the common practice to access it is to call `dlopen` and `dlsym` in the hook function. This allows a function address to be resolved at runtime based off a module and function name.

Advantages and Disadvantages

ELFsh demonstrates less conventional techniques for the binary rewriting process, which is due to its birth in the reverse engineering scene. For the most part, the tool is able to avoid dealing with concepts such as static code analysis by relying on pre-existing analysis from the user. For example, it provides no function to enumerate procedures so the parameters to `dlopen` and `dlsym` would have to be determined beforehand at the compile-time of the instrumentation object file. The tool is able to discover such information interactively but the lack of a proper programmatic interface makes the process much more manual. The tool is unsuitable for us for various other reasons:

1. **Lack of abstraction layer** - One of our primary irks with ELFsh is that it is built on top of an internal library called libelfsh which duplicates a lot of the functionality in libelf. This is exploited by using file-specific tricks to implement the features provided by the tool. The tool pays the price for this in portability, with ports requiring manual porting of the backend. The difficulty of this is evidenced in the fact that despite ports existing to other architectures, only the Intel version is fully featured.

2. **Lack of granularity** - ELFsh lacks the granularity and control given with other binary editing tools. It does not allow the insertion of instrumentation at the instruction level, nor even at the basic block level.

3. **Development state** - ELFsh is an old project that is no longer maintained or developed.

One concept we can take away from ELFsh is the scriptable command-line. Even though we want to create a library, it would be nice to be able to operate our tool through a command-line also.

3.6 Pin/DynamoRIO

Pin and *DynamoRIO* are two tools which perform run-time binary instrumentation of Linux and Windows applications [8, 18, 28]. The two tools are completely separate but due to the similarities of their implementation they have been grouped together here. We will talk about Pin, but the concepts are applicable to both tools unless explicitly stated.

Pin does not make any static modifications to the executable. Instead, it uses *dynamic recompilation* to run the target in a process-level virtual machine which intercepts execution at the entry point and injects a runtime agent which performs the insertion of the instrumentation.

Pin is used to create an architecture independent Pintool which can access architecture-specific details when necessary. The Pintools are written in C/C++ making them theoretically source compatible across different architectures. Pin is viewed as an efficient solution compared to other runtime detouring implementations because of its use of just-in-time (JIT) compiling to insert and optimize code.

Advantages and Disadvantages

The advanced process-level virtual machine Pin utilizes requires the distribution of a substantial environment so is not quite suitable for us. However, interception from this level allows for a high level of observability which is usually difficult to obtain. Whereas the tools we have looked at so far operate mostly at the instruction, basic block and procedural level, Pin allows higher level abstractions to be observed. For example, instrumentation can be notified upon events such as loading/unloading of shared libraries and creation/end of threads. It is possible to reproduce these effects with regular tools by detouring system library functions but this convenience and ease of use is one of Pin's selling points. If we have time, it would be good to mimic this concept of moving more responsibility from the user to the library.

Pin takes full advantage of the flexibility offered by dynamic detouring presenting features such as process attaching and detaching. Pin suffers from the overhead inferred from runtime attachment, but makes up for it with its powerful JIT compiler. Firstly, by performing code discovery at runtime, Pin is able to gather more comprehensive information about the program compared to using static control flow analysis. This extra information allows the tool to perform optimizations on instrumentation which previously had to be done by the user. Secondly, the JIT instruments code on the fly by taking the native executable as input and intercepting its execution, generating new code from it (which is cached) and transferring the target's execution to the generated sequence. Instrumentation can easily be inserted during the translation phase. This process essentially optimizes the code on the fly with the further advantage that the system can reflect the target's runtime environment accurately reducing any effects on the original program's behaviour.

Pin was designed with observation in mind, rather than modification. It can modify the behaviour of the executable by modifying registers and memory but its ability to do so is more limited than some of the other tools we have looked at. In a way, Pin has taken the typical inspections that users might make with regular detouring tools and made these accessible features as part of its API.

3.7 IDA Pro

IDA Pro is a popular commercial disassembler supporting a large variety of architectures and operating systems [2,9]. The focus with IDA Pro is not on the instrumentation phase, but rather on the automatic code analysis that it performs. The tool is able perform instrumentation, but since it does not present anything substantially different to what we have already seen, we shall draw our comparisons about its ability to perform static analysis.

Advantages and Disadvantages

As with all static binary rewriters, IDA Pro faces the problem of discovering functions that are only reachable through indirect control transfer. What is interesting in this case is not what IDA Pro provides, but what it lacks and how it contrasts with solutions provided by other tools.

IDA Pro and EEL both do not provide any support for discovery of functions that are reached via indirect calls. These functions become gaps in the code. LEEL approaches the problem by starting analysis from the program's entry point and recursively building a call graph (same so far). However, it provides two options to deal with the gaps [35]: either not to analyze them at all or assume that the first byte of a gap is the starting address of a function. It can be argued that this provides two extremes, providing either low code coverage or a high chance of incorrect analysis of non-code blocks as functions. Other tools such as RAD [22] take the more aggressive approach but use conservative heuristics to prevent false positives such as only analysing blocks if it starts with a known prologue. If we perform static analysis, we will need to take these methods into account and select one appropriately.

3.8 Others

There exist various other noteworthy libraries and tools which either perform detouring or make use of it including *Valgrind*, *IDtrace*, *radare2*, *qpt*, *ATOM* and *pixie* [3,20,21,29,31,33]. However these tools do not introduce significant concepts or techniques interesting to us that we have not already seen so we shall not be covering them in further depth.

Chapter 4

Design

This section discusses the design and architecture of `libbf` (**bi**nary **f**ile) and presents its constituent components. This section also provides the justification for the design decisions made. In order to make these decisions, we re-evaluate the functional requirements taking into account the techniques gleaned and the lessons learnt from Background and Related Work. The actual implementation will be covered separately so there may be minor inaccuracies and inconsistencies to keep things simple. Particularly, one key aspect of the library that should be noted is the support for both x86-32 and x86-64. While there is a high degree of overlap in code dealing with these architectures, the library delegates the work to architecture-specific sub-components for certain tasks such as instruction manipulation. Such differences will be addressed in Implementation (Chapter 5), but ignored for now.

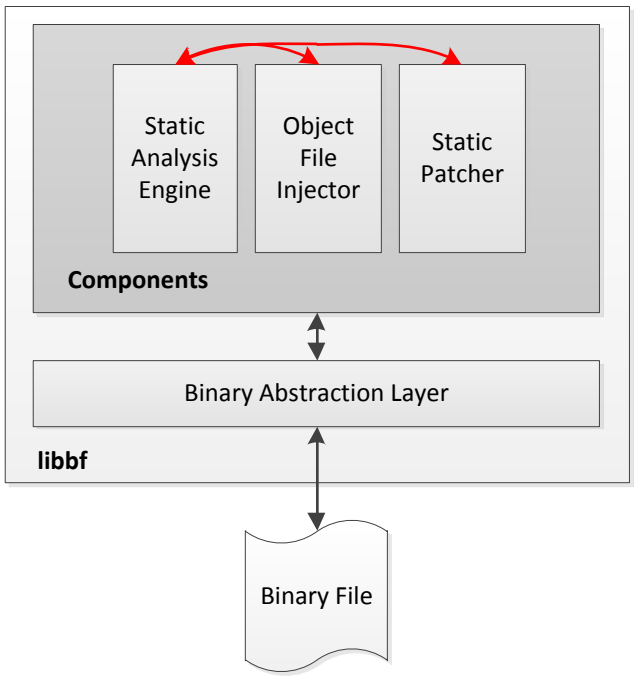


Figure 4.1: The high level architecture of `libbf`. The three components of `libbf` interact with a binary file through an internal binary abstraction layer. Within the library, the static analysis engine collaborates with the object file injector and static patcher.

`libbf` works through binary rewriting mainly because all the runtime methods we have looked at require some form of environment, whether this is through the distribution of a shared library, process-level virtual machine or otherwise. Since a dynamic solution would compromise [**standalone**], we narrow the scope of the solution to binary rewriting.

Figure 4.1 illustrates the three main components of `libbf`. The choice for the different components of `libbf` is mainly due to the natural and inherent separation in duty between the three aspects. The components interact through a well-defined interface which means the implementations are fully decoupled. This allows different implementations to be 'plugged in' increasing the portability value of the library. For example, it is trivial to write a static analysis engine plugin for a different architecture as long as the new implementation adheres to the original interface.

For now, we can treat the binary abstraction layer as `libbf`'s internal representation of a binary file. Most importantly, the binary abstraction layer encapsulates the internal representation of the code of a binary as a CFG as required by **`[cfg_gen]`** of the specification. As such, any modifications to the code of a binary are done through the addition, deletion or modification of nodes/edges in the CFG. Hence, a simplified and abstract way of considering each component of `libbf` is in terms of the operations it performs upon the CFG.

4.1 Static Analysis Engine

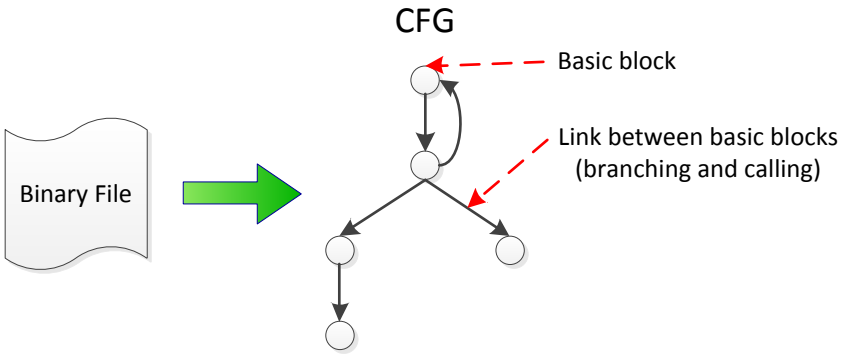


Figure 4.2: The primary role of the static analysis engine is to *generate a CFG*.

While the fundamental goal of `libbf` is to provide static detouring, the usefulness of such a feature is greatly diminished without the appropriate supporting functionality. The static analysis engine satisfies **`[api_expr]`** and essentially plays a supporting role which can be split into three parts:

Disassembler Engine This is required for the construction of the CFG. The sole purpose of the disassembler engine is to perform the heavy lifting in terms of mapping the raw bytes of a binary to useful semantic information. For example, given an arbitrary address, the disassembler engine can disassemble the instruction held at that address and determine whether that instruction is a branching one and if so, decode the operand and attempt to figure out the branch destination.

CFG Analysis The generation of a CFG starts by pointing the disassembler engine to a root of disassembly (such as the entry point). Using the information provided by the disassembler

engine, the analysis uses an algorithm to perform a depth-first discovery of the program structure.

Symbol Table Reader A symbol table is not guaranteed to exist in a binary because it can be completely stripped during or after compilation. The symbol table reader checks for the existence of a symbol table and if found stores the symbol information alongside the CFG.

To summarise, there are two phases to the static analysis performed by `libbf`. Firstly, a CFG is generated and then in the second pass, symbol information is added to the CFG. The CFG exposes an access interface through which a user is able to locate and express the source and destination for static patching. This is a concrete example of how the presence of a symbol table is beneficial. If we first consider a CFG without symbol information, the only way of identifying a basic block or function is to specify a heuristic in terms of instructions and iterate all basic blocks till a match is found. For example:

```
/*
 * Extracts the first instruction of each basic block and checks whether it is
 * "push %rbp". This has to be done by checking the mnemonic is push_insn
 * then checking the first operand is a register and that the register is %ebp.
 */
struct bf_basic_blk * bb;

bf_for_each_basic_blk(bb, bf) {
  struct bf_insn * first_insn = bb->insn_vec[0];

  if(first_insn->mnemonic == push_insn &&
      first_insn->operand1->tag == OP_REG &&
      first_insn->operand1->operand_info->reg == rbp_reg &&
      ...) {
    bf_trampoline_basic_blk(bf, bb, dest_bb);
    break;
  }
}
```

Listing 4.1: Identifying basic blocks via instruction heuristics

Basic block (and function) identification through instruction heuristics is verbose but necessary in order to be able to search at an instruction granularity. If available, a symbol table is a valuable resource that allows a far more convenient method of locating the same basic blocks/functions. Assuming the basic block we are searching for is actually the first basic block of a function named `func1`, the identification could be expressed alternatively:

```
/*
 * Locates a function by its name.
 */
struct bf_func * func = symbol_find(bf, "func1");
bf_trampoline_func(bf, func, dest_func);
```

Listing 4.2: Identifying basic blocks via symbols

The functionality provided by the static analysis engine is powerful and can be used in many other applications. For example, it is possible to compare the CFG of two binaries to generate statistics for the number of changed functions/basic blocks/instructions. This is not a feature that is provided directly by `libbf` because it aims to be a general purpose library. It is a specific application of the static analysis engine and we demonstrate this as part of our evaluation.

4.2 Object File Injector

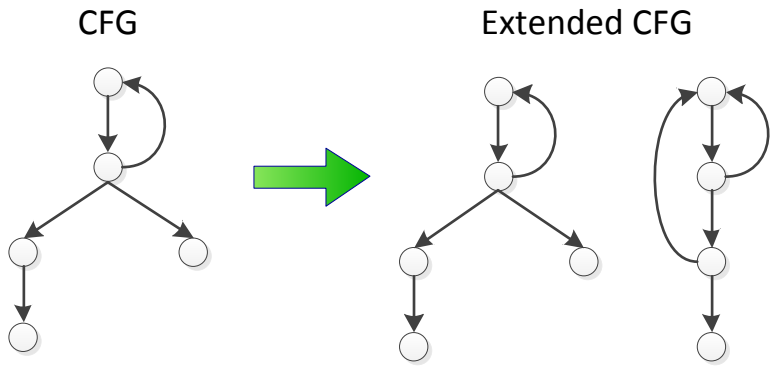


Figure 4.3: The role of the object file injector is to extend the CFG such that the CFG after injection is a superset of the CFG before injection. That is, a set of *nodes and edges are added* which are disjoint to the original CFG.

The function of the object file injector is to load extra code into the target binary. Immediately after injection, the new code is unreachable from the original code which explains the two disjoint parts of the resulting CFG. The implication is that the injected CFG can be hooked up with the original CFG via the static patcher. Trampolining from the original code to the new code and back allows the semantics of target to be extended while preserving existing functionality.

The ability to load external code into the target binary is necessary in order to instrument the code at all. However, object file injection is a highly non-trivial process as we will see in later sections.

4.3 Static Patcher

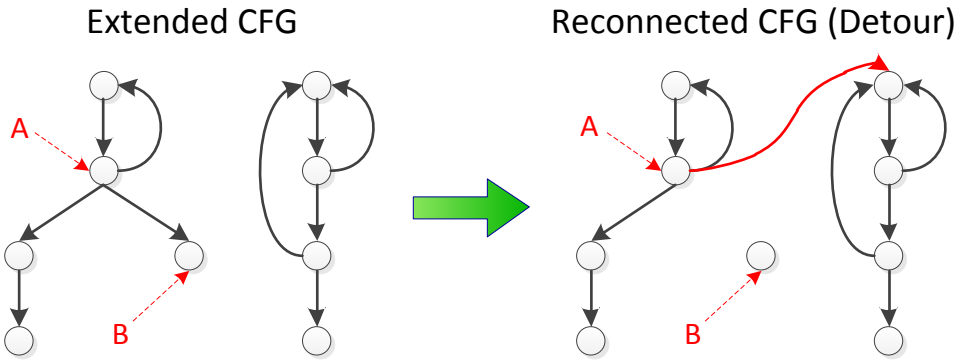


Figure 4.4: The role of the static patcher is to take a binary which is assumed to have all necessary code and manipulate its CFG to detour or trampoline execution. This is done by *adding edges* to a CFG. This is the most basic case in which a single edge has been added to detour the execution of one basic block (A) to another (B). This satisfies `[bb_det]`. Note that the basic block B is no longer reachable.

Similarly, the static patcher places a trampoline by adding two edges to a CFG - one to detour execution to the destination and another to trampoline back:

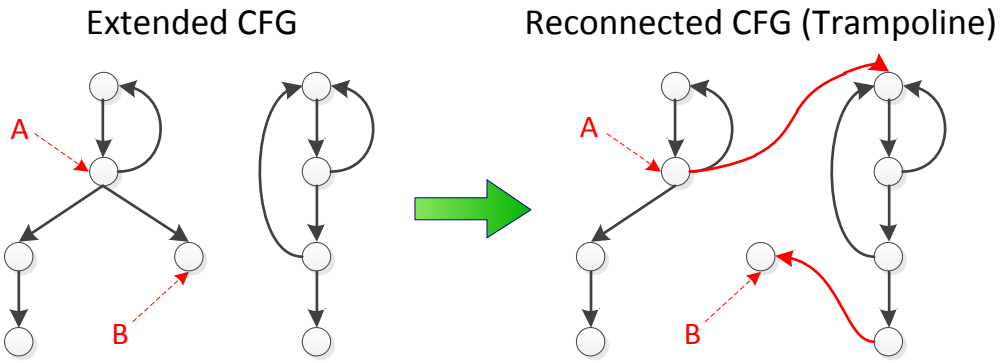


Figure 4.5: The static patcher adds two edges to the CFG to make a trampoline. This satisfies `[bb_tra]`.

Strictly speaking, in our implementation, adding an outgoing edge to a CFG requires modifying the node the edge is to be added to. The intuition for this is that an edge simply represents a branch in the code so adding an edge writes a branch instruction to the start of a basic block. This is more of an implementation detail and in fact, we have already seen less aggressive methods of detouring execution which do not modify code in Chapter 2 (e.g. exception handlers).

To clarify the terminology, the static patcher does not actually edit the binary. It requests the modifications from the binary abstraction layer which fulfils this task. In practice, the static patcher is responsible for deciding what bytes need to be changed in order to achieve a detour/trampoline.

4.4 Summary

In summary, we present an implementation of executable editing to satisfy `[standalone]`. The solution contains a static analysis engine and object file injector which work together to satisfy `[cfg_gen]` and `[api_expr]`. The final component of the library is a static patcher which satisfies `[bb_det]` and `[bb_tra]`.

Chapter 5

Implementation

In this section, we will discuss the implementation of each of the components of `libbf`. Any major deviations from what has been covered in Design will be highlighted.

5.1 Binary Abstraction Layer

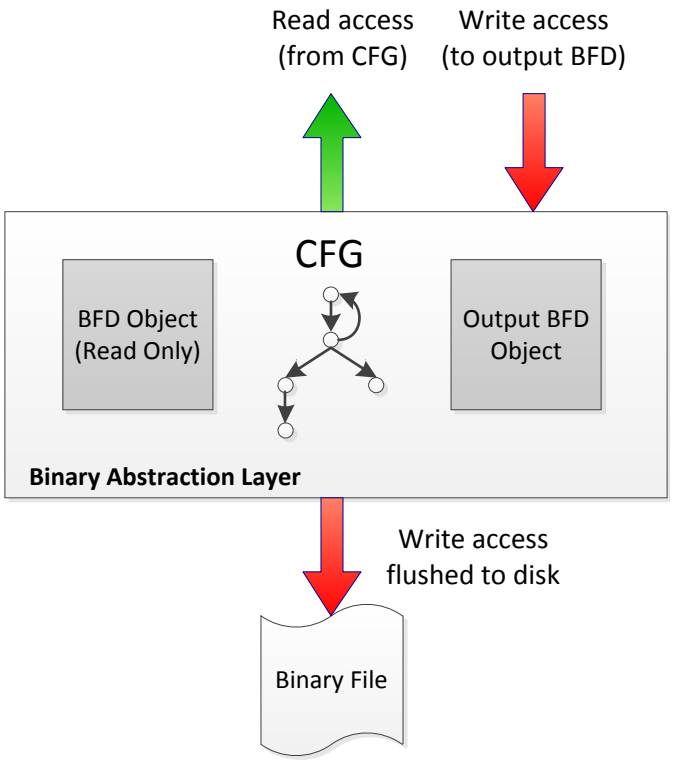


Figure 5.1: The original approach for the binary abstraction layer. The read only BFD object is the abstraction of the original binary file. The CFG is generated from this BFD. Any changes need to be outputted to the output BFD independently.

The binary abstraction layer was originally designed to act as a bridge between the components and the target file. Essentially, `libbf` encapsulates a binary file in an internal structure called a `bin_file`. A `bin_file` contains two important elements: the BFD abstraction of a binary and

its corresponding CFG (which is generated by the static analysis engine). In general, when a component needs to read information about the binary, it will read from the generated CFG. As we have seen, writing to the binary is equivalent to modifying the CFG. After a change to the CFG occurs, this change is flushed to the underlying binary through the BFD object. This was the original concept but it had to be modified to work in practice. Figure 5.1 illustrates the original approach to creating the binary abstraction layer.

5.1.1 Original Approach

The problem with the original design was that it only encapsulated a single BFD object. With `libbfd`, it is possible to open a file in two ways:

1. An existing file can be opened with read-only access.
2. A *new and empty* file can be opened with write access.

This restriction of `libbfd` is by design, but does not make it ideal for patching. Tools such as *ld* and *objcopy* work well above `libbfd` because the files they output are written in a single pass. Although BFD is unsuitable for our usage, it is possible (albeit very awkward) to patch a binary file. Figure 5.2 illustrates the steps required to patch a file through `libbfd`.

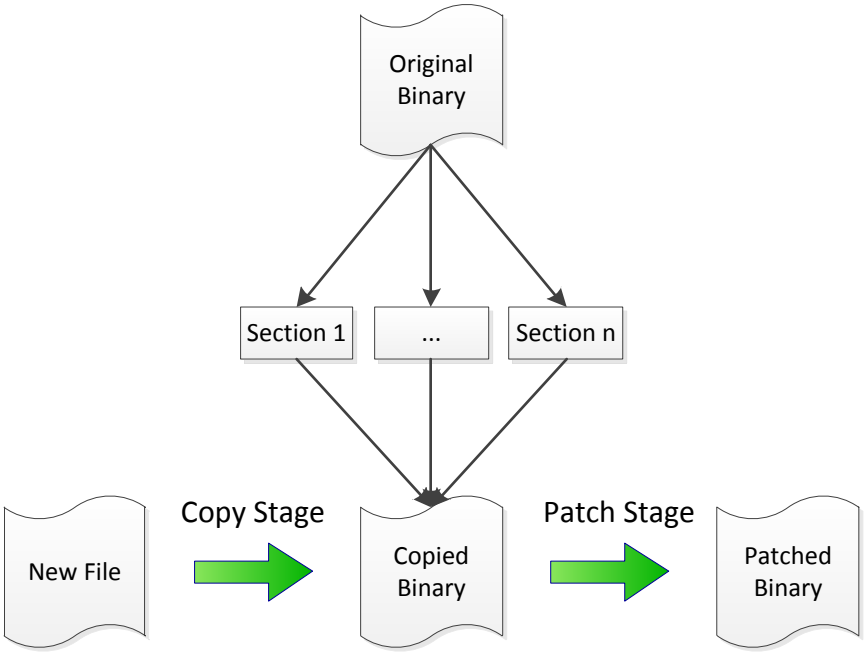


Figure 5.2: The process required to patch a file using `libbfd`.

The abstraction used within BFD is that an object file has:

- A header
- A number of sections containing raw data (this data could represent code)
- A set of relocations

- Symbol information

As illustrated, patching a binary file with `libbfd` works through a series of steps:

1. The original binary must first be opened with read access.
2. A new BFD (which starts off empty) is created. The original binary is then reconstructed in this BFD by extracting the four different types of content (as defined above) and copying them. This reconstruction all happens in memory.
3. Before the new BFD object is flushed to disk, the user is able to perform patches by editing the contents of sections.
4. After all patches are performed, the BFD object is flushed to disk. If more modifications are needed after this, the entire process has to be restarted.

In terms of the implementation of this algorithm, the first two stages are essentially what *objcopy* does. Our implementation of static patching re-uses the objcopy code for the decomposition and reconstruction stage. The rest of the algorithm is trivial to implement. However, there are two drawbacks with this method:

1. The part of the code re-used from objcopy is large, approximately 1.5 KLOC.
2. BFD is merely an abstraction over many file formats. This means that it represents a more generalised abstract format. Unfortunately, it does not encapsulate all the information required for the full reconstruction of a binary. In order to obtain this information, the code relies on a dependency with the file format and ends up calling functions in `libelf`.

5.1.2 Final Approach

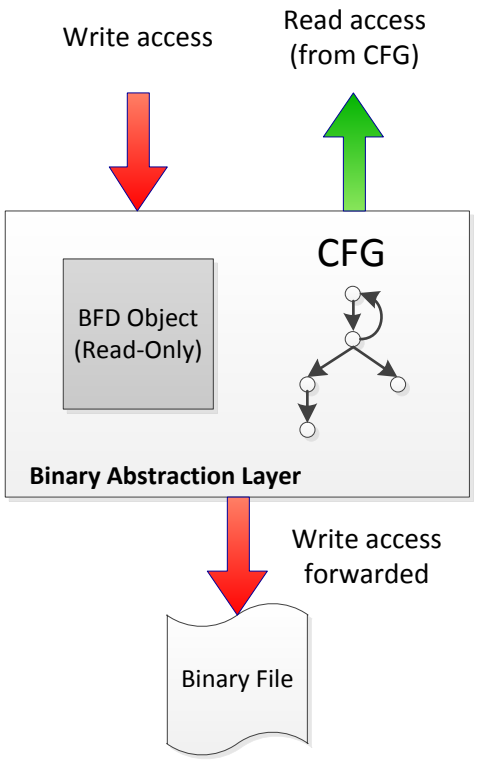


Figure 5.3: The final design for the binary abstraction layer. Read accesses are handled in the same way but in this design, we have removed the output BFD. Write accesses alter the target binary file directly.

Given the disadvantages of using the method from the original approach, there is no reason to use BFD to perform the patching. The method is overly complicated and more error-prone simply because of the amount of code that has to be written to perform a patch. Since a dependency on `libelf` is unavoidable, we can re-implement the solution by using `libelf` directly and save the hassle of forcing `libbfd` to do something it was not designed for. In terms of the binary abstraction layer, this means that we modify write accesses to directly modify the disk version instead of through the BFD object as seen in Figure 5.3.

In the final implementation of the binary abstraction layer where `libelf` is used, the same result as with the original approach can be achieved in under 100 LOC. The method is fairly intuitive and the concept is visualised in Figure 5.4:

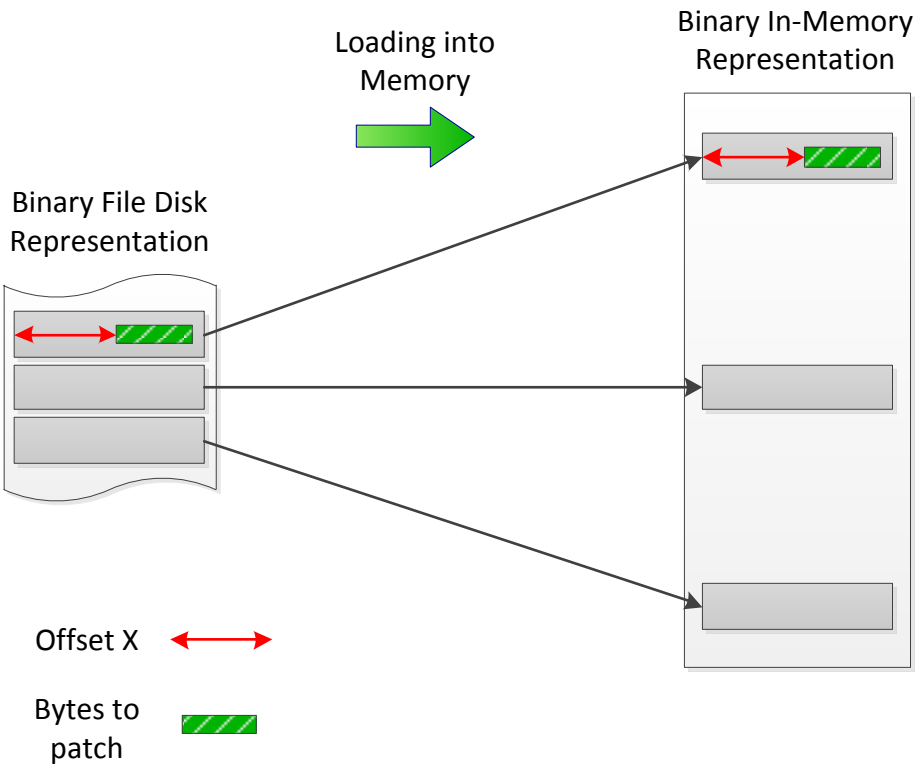


Figure 5.4: The concept of how patching works in the final implementation. The most important idea here is that a section in the disk representation is preserved (not split) when it is mapped into memory. This means data that is at a certain offset into the section in the disk representation will be offset by the same amount after loaded into memory.

In `libelf`, similarly to `libbfd`, an executable contains a set of sections in which both code and data can reside. When the operating system loads the executable into memory, each section is based at some base address. We can reverse this to map a virtual memory address to a physical file offset. If a physical file offset can be found, the bytes can be patched as a regular file.

1. First, the section header of the executable is parsed with `libelf`. This allows us to obtain a set of sections and most importantly, for each section `libelf` can tell us three things:

   Section size The size of the section.

   Section base address The address that the section will be based at when the executable on disk is loaded into memory.

   Section disk offset The disk offset of the section.

2. By iterating through the section information extracted in the previous step, it is possible to find out what section an arbitrary virtual memory address is contained in. During patching, the memory address we are interested in is the address of the code at which a detour is to be written. The offset of the virtual memory address into the section can be calculated by `(virtual_memory_address - section_base_address)`.

3. Therefore, the disk offset of the virtual memory address can be calculated by `(section_disk_offset + (virtual_memory_address - section_base_address))`.

This process allows an arbitrary virtual memory address to be mapped to its corresponding disk file offset. This offset can then be patched with regular C file I/O functions such as `fopen/fseek/fwrite/fclose`.

Fortunately the architecture of `libbf` was designed such that a change in the implementation of one component (even if that is the binary abstraction layer) would not impact the other components. The downside is that a lot of time was lost writing the BFD patch code, only to have it thrown out in favour of another method. However, this was unavoidable as the documentation did not make it clear how difficult the implementation of patching would be. The lack of clear documentation is a widely recognised drawback to `libbfd` and the required steps were only discovered by reading through the binutils library source.

5.2 Static Analysis Engine

As discussed in earlier sections, the static analysis engine allows `libbf` to discover the code of a binary. It is implemented in several layers and matches the design described in Design (Chapter 4). Figure 5.5 shows the internal implementation of the static analysis engine in more detail:

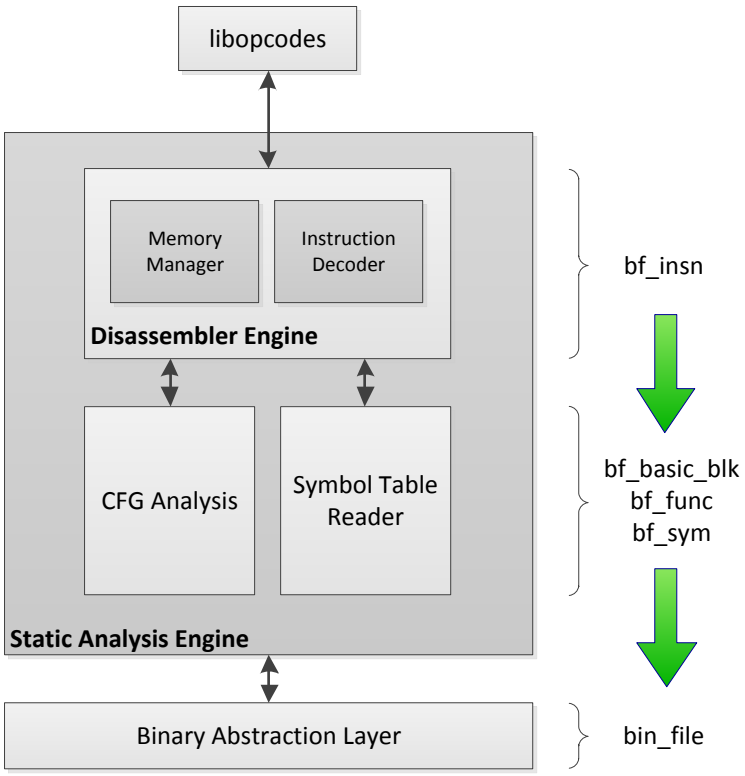


Figure 5.5: The internal implementation of the static analysis engine.

From a high level, the disassembler engine builds `bf_insn` objects to represent each disassembled instruction. The CFG analysis drives the disassembler engine to identify basic block boundaries and to generate the CFG. A CFG is simply a graph of `bf_basic_blk` objects. Certain `bf_basic_blk` objects correspond to the start of a function and can be additionally labelled as `bf_func` objects. The symbol table reader attaches symbol information to the generated CFG by associating `bf_sym` objects with the relevant `bf_basic_blk` objects.

In order to understand how the static analysis engine works as a whole, we will explain each component in further detail.

5.2.1 Disassembler Engine

In practice, the disassembler engine is responsible for parsing and translating the strings received from `libopcodes` and storing the information in its internal semantic representation. At the most basic level, a `bf_insn` corresponds to an assembly instruction. In order to use `libopcodes`, we first needed to solve the two problems outlined in Design:

Limited disassembly scope `libopcodes` can only disassemble a single address at a time which means the same applies for the disassembler engine. In order to make the disassembler engine useful, CFG analysis needs to be performed when an instruction is disassembled. The results of this analysis will then be used to drive the disassembler engine. For example, if the CFG analysis comes across a conditional branch instruction, it will need to instruct the disassembler engine to disassemble both the branch target and the next instruction. This analysis will be covered in detail later since it is not part of the disassembler engine.

Output redirection The default behaviour of `libopcodes` is to print to a stream using `fprintf`. It is possible to override this behaviour by instructing `libopcodes` to invoke a custom function with the same prototype as `fprintf`.

One quirk of `libopcodes` is that it invokes the `fprintf` function (or overridden substitute) for each of the following 'types': mnemonic, operand, separator and comment. As a concrete example, consider the following instruction:

```
ucomiss 0x9124(%rip), %xmm0 # 414d58
```

Listing 5.1: Example instruction disassembled by `libopcodes`.

The custom `fprintf` function is invoked 6 times with the following:

```
ucomiss (Mnemonic)
0x9124(%rip) (Operand)
, (Separator)
%xmm0 (Operand)
# (Separator)
414d58 (Comment)
```

Listing 5.2: Strings received by custom `fprintf`. Type information is included for the benefit of the reader but is not given by `libopcodes`.

The main goal of the disassembler engine is to take these inputs and produce something useful which can be used for CFG analysis. One option is to concatenate all these instruction parts to obtain a full instruction but we already decided in Design that we would not store instructions in string from. The main problem with storing strings is that semantic information needs to be extracted as part of the CFG analysis anyway. Furthermore, users of the library would need to reproduce the exact same extraction code. One advantage of storing strings is that it is convenient for printing instructions. This means an instruction decoder must be able to:

Map instruction parts to semantic information For example, instead of the string "$0x3ef", its parsed value (0x3ef) should be stored.

Map the stored semantic information back to its string equivalent For example, it should be possible for a mnemonic such as "mov" to be stored semantically (perhaps as an enum) as part of the first requirement. Mapping back to its string equivalent means mapping the enum value back to a string.

The implementation stores semantic information about each instruction by decoding each part as it is received from `libopcodes`. The decoded information is stored in a `bf_insn` object corresponding to the current address being disassembled. An issue with this approach is that `libbf` needs to know how to decode every type of string it receives. For example, an arbitrary string that is received might be a mnemonic, operand, separator or comment which all need to be decoded in different ways. It would be very inefficient to attempt to decode each string to determine what type it should be. `libbf` solves this issue by implementing a finite state machine (Figure 5.6) which allows it to know what type it is expecting to receive next.

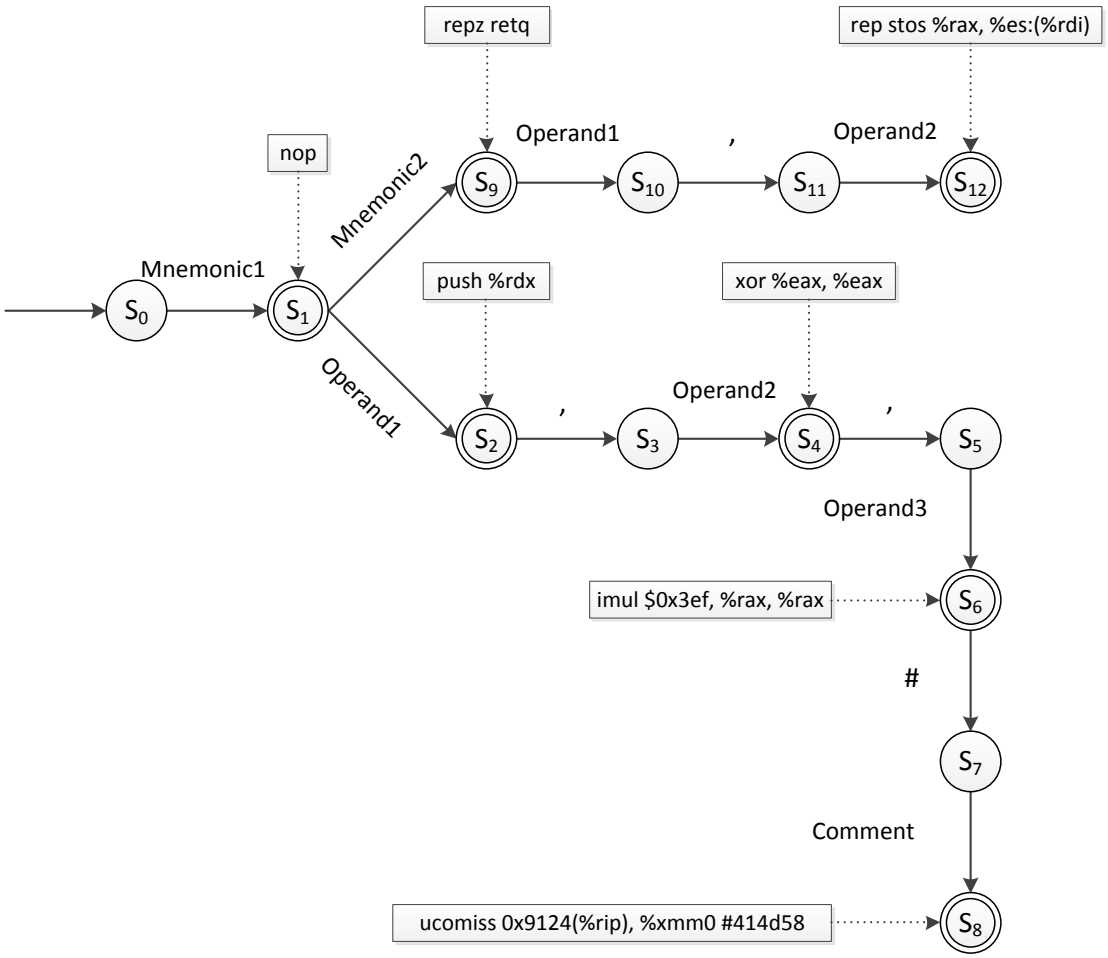


Figure 5.6: The instruction finite state machine used by the disassembler engine to determine how the received string should be decoded. A double circle denotes a possible terminal state. For each terminal state, an example instruction which would cause termination at that state is shown.

Each time a new instruction is disassembled, a new `bf_insn` object is constructed and the current state is restarted to the initial state of the instruction finite state machine. When `fprintf`

is called, the received string is decoded based on its expected type, the semantic information updated for the `bf_insn` and the current state of the state machine advanced. S_1 is a special case because two types can be received at this point: an operand or a mnemonic. In this case, we can only decode as one type and if that fails decode as the other type. Since an operand is more common, the disassembler engine attempts to decode as that first. The implementation of the instruction decoder is covered later.

Instruction Decoder

The two key aims of the instruction decoder have been discussed already, but to briefly recap, the instruction decoder is responsible for:

- Mapping strings representing instruction parts to a semantic representation.
- Mapping this semantic representation back to string form for printing.

We maintain that a semantic representation is a more natural way of working with assembly instructions as opposed to string comparisons. However, it is also understandable that people would want to use strings either out of preference and habit or in order to print onto the screen. This is why we require the mapping to go both ways. A further requirement is that any solution for the instruction decoder needs to be highly efficient and have minimal overhead, particularly to go from string to semantic representation since this is part of the disassembling process as seen above.

Every string received from `libopcodes` needs to be decoded which means there are four cases to handle:

Mnemonics "mov", "xor", "repz", ...

Operands "%rax", "%es:(%rdi)", "0x9124(%rip)", ...

Separators "," and "#"

Comments "414d58", "602030", ...

We will enumerate how each case is handled starting from the most basic case.

Separators A separator is defined either as "," or "#". This is the most simple case to handle. When the instruction decoder encounters a separator, it simply does nothing since this information does not need to be stored semantically. The disassembler will automatically advance the instruction finite state machine and the next part of the instruction will be decoded.

Comments Comments are extra information added by `libopcodes` that suggest the value of an operand. They always appear in the form of a hexadecimal value without the 0x prefix. These can be decoded without effort with `sscanf` with the `%lx` format specifier.

Operands Instruction operands are represented in `bf_insn` objects by a **`struct`** containing a **`union`** (Listing 5.3).

```
struct insn_operand {
  enum operand_type tag;
  union {
    bfd_vma  val;
    uint64_t imm;
    bfd_vma  addr_ptr;
    enum insn_reg reg;
    enum insn_reg reg_ptr;
    struct array_index arr_index;
    struct array_index arr_index_ptr;
    uint64_t index_into_fs;
    struct cs_index index_into_cs;
    enum insn_reg index_into_es;
    enum insn_reg index_into_ds;
    bfd_vma  index_into_gs;
  } operand_info;
};
```

Listing 5.3: The `insn_operand` structure which holds the semantic representation of operands within instructions.

All the different cases need to be enumerated when performing the decoding:

Values `0x9124`, `0x2348098`, ...

Immediates `$0x8f0a`, `$0x3ef`, ...

Address Pointers `*0x8f0a`, `*0x3ef`, ...

Array Indexes `0x200c2a(%rip)`, `-0x28(%rsp)`, `0x0(%rax,%rax,1)`, ...

Array Index Pointers `*0x200c2c(%rip)`, `*0x600e28(,%rax,8)`, `*(%r12,%rbx,8)`, ...

Index into CS/ES/DS/GS `%cs:0x0(%rax,%rax,1)`, `%es:(%edi)`, `%ds:(%rsi)`, ...

These operands can be decoded trivially by pattern matching with `sscanf`. When an operand consists of the conjunction of several types, the decoding is performed recursively. For example, decoding `0x0(%rax,%rax,1)` requires `0x0`, `%rax`, `%rax` and `1` to be decoded.

Registers `%rax`, `(%bl)`, `(%r14b)`, ...

Register Pointers `*%rax`, `*%rbx`, `*%rcx`, ...

Registers and register pointers are decoded in the same way as mnemonics. The reasoning behind this will become clear in the explanation of mnemonic decoding.

Mnemonics The decoding of mnemonics is the most tricky case. The most appropriate/intuitive C data type to be used for the representation of mnemonics is the enumerated type. Classically, a string can be mapped to an enumerated type and vice versa as shown by Listing 5.4. The forward case (string to enum) uses string comparisons and has a running time complexity of *O(n)*[1]. The backward case (enum to string) uses **`switch..case`** and has a running time complexity of *O(1)* in the best case[2]

[1]We ignore the complexity of the **`strcmp`** operation by assuming each **`strcmp`** takes the same amount of time. We are interested only in complexity with respect to the number of separate cases in the **`switch..case`** statement.

[2]Switch/case is usually optimised to O(1), but this is not defined by the C standard. In practice, GCC generally makes the O(1) optimisation based on a threshold of three or more case parts (excluding the default).

It is worth noting that efficiency for the forward case is more important than for the backward case. That is, users generally care more about the speed of disassembly than the time taken to print or dump instructions. While the classical implementation is trivial to implement, it should be possible to improve on the forward case complexity of *O(n)*.

```
/*
 * Running time complexity of O(n).
 */
enum insn_mnemonic str_to_mnemonic(char * str)
{
  enum insn_mnemonic = 0;

  if(strcmp(str, "aaa") == 0) {
    insn_mnemonic = aaa_insn;
  } else if(...) {
    insn_mnemonic = ...
  } else if(strcmp(str, "xorps") == 0) {
    insn_mnemonic = xorps_insn;
  }

  return insn_mnemonic;
}

/*
 * Running time complexity of O(1).
 */
char * mnemonic_to_str(enum insn_mnemonic mnemonic)
{
  char * str = NULL;

  switch(mnemonic) {
  case aaa_insn:
    str = "aaa";
    break;

  case ...

  case xorps_insn:
    str = "xorps";
    break;

  default:
  }

  return str;
}
```

Listing 5.4: Classical implementation for mapping strings to enumerated type and vice versa.

One way to reduce the forward case complexity is to create a perfect hash function, which maps all possible mnemonic strings to *unique* enum values. There exist tools such as GNU *gperf* [26] for generating perfect hash functions but this approach is brittle and rigid to change. The x86 instruction set has been extended many times throughout its history with many of these additions occurring with the release of a specific processor. Updating `libbf` under such scenarios would require regeneration of the perfect hash function.

`libbf` extends the concept of using a perfect hash function by using an approach which aims

to be more amenable to change and minimise the cost of the hash overhead. The decoding of mnemonics is done by generating an enumerated type where the values of members correspond to the integer representation of the mnemonic string representation as shown in Listing 5.5.

```
enum insn_mnemonic {
  mov_insn = (uint64_t)‘m’ | (uint64_t)‘o’ << 8 | (uint64_t)‘v’ << 16,
  ret_insn = (uint64_t)‘r’ | (uint64_t)‘e’ << 8 | (uint64_t)‘t’ << 16,
  sysenter_insn = (uint64_t)‘s’ | (uint64_t)‘y’ << 8 | (uint64_t)‘s’ << 16 |
    (uint64_t)‘e’ << 24 | (uint64_t)‘n’ << 32 | (uint64_t)‘t’ << 40 |
    (uint64_t)‘e’ << 48 | (uint64_t)‘r’ << 56
};
```

Listing 5.5: Example mnemonic string and integer representations.

The bit-shifting is abstracted through a macro which allows the same enumeration to be defined as follows:

```
enum insn_mnemonic {
  mov_insn = INSN_TO_ENUM(’m’,’o’,’v’),
  ret_insn = INSN_TO_ENUM(’r’,’e’,’t’),
  sysenter_insn = INSN_TO_ENUM(’s’,’y’,’s’,’e’,’n’,’t’,’e’,’r’)
};
```

Listing 5.6: Macro abstraction.

In `libbf`, the `insn_mnemonic` enumeration is actually generated by passing a list of mnemonics to a Java program which acts as a pre-preprocessor to generate the C enumeration. However, it is clear that adding a new instruction in this implementation is trivial. This approach allows the forward stage to be defined as so:

```
enum insn_mnemonic mnemonic = 0;
strncpy((char *)&mnemonic, str, sizeof(uint64_t));
```

Listing 5.7: Forward stage.

The backward stage is defined as so:

```
char str[sizeof(uint64_t) + 1] = {0};
memcpy(str, &mnemonic, sizeof(uint64_t));
```

Listing 5.8: Backward stage (`mnemonic` is of type `enum insn_mnemonic`). The extra byte in the `char` array is used as padding for the null terminator.

As a side-note, it may appear at a first glance that the backward stage can alternatively be defined equivalently as the following:

```
char str[sizeof(uint64_t) + 1] = {0};
*(uint64_t *)str = mnemonic;
```

Listing 5.9: mnemonic is of type `enum insn_mnemonic`. The array `str` is casted to a pointer to a `uint64_t` and assigned the value of `mnemonic`.

However this code is not standards complaint because of the type-punning. Such an implementation is illegal and violates strict aliasing rules. It would result in undefined behaviour because a pointer is dereferenced that aliases another of an *incompatible type*. The standards compliant way of performing the assignment is with `memcpy`. Empirically, GCC often compiles the two code snippets into identical assembly code by optimising the `memcpy`, but it is important that we do not rely on this behaviour since it is not defined by the standard.

The approach taken by `libbf` satisfies the following prerequisites:

Efficiency Both the forward and backward mapping are *O(1)*. On top of this, the implementa-

tion amounts to copying 8 bytes of memory with no extra calculations. It is unlikely any generated hash function would be able to beat this.

Uniqueness Every mnemonic in the x86 instruction set is uniquely identifiable from its first 8 characters. There are three exceptions where the mnemonic length is longer than 8 bytes (namely `cvttsd2si`, `cvttss2si` and `cvtsi2sdq`). These special cases are handled by manually truncating the `INSN_TO_ENUM` macro input. `cvttsd2si` and `cvttss2si` can not simply be truncated because the first 8 characters are shared so they are arbitrarily assigned different values. These special cases are checked in both the forward and backward cases to ensure that correct observable behaviour is preserved.

As mentioned during discussion of the operand decoder, registers are decoded in the same way as mnemonics, but stored in a separate enumeration (`enum insn_reg`). In order to analyse instructions easier, `libbf` exposes functions to get more information about enumeration values. For example, functions to check if a mnemonic represents an instruction which will break flow (an unconditional branch), branch flow (a conditional branch or `loop` mnemonic), call subroutines (`call`/`callq`) or end flow (`ret`, `sysret`, etc.).

The implementation of mnemonic decoding described above has portability implications which are discussed in Writing Standards Compliant Code (Section 5.6). This discusses how the approach relies on implementation-defined behaviour of GCC (which affects the portability of `libbf` across different compilers).

Memory Manager

From an architectural point of view, the memory manager does not play an important role. It is discussed here briefly for completeness. Under the BFD abstraction, all data (including code) resides in a section. The vast majority of interactions involving data, including disassembling with `libopcodes`, requires the section containing the data to be loaded. The `libopcodes`/`libbfd` interface then requires a pointer to the section object. In order to make the API of `libbf` as lightweight as possible, it was decided that the concept of a section should be abstracted because it should not be necessary for a user to know about them. This means the disassembler engine is invoked without section information. The memory manager intercepts all requests sent to the disassembler and examines the target address being disassembled. It then loads the appropriate section, adds this information and then forwards the request to `libopcodes`. For efficiency, the memory manager only unloads sections during the final cleanup when a `bin_file` is closed which means no section is ever loaded more than once.

5.2.2 Control Flow Graph Analysis

A control flow graph (CFG) represents a binary file using graph notation where nodes are basic blocks. There are many control flow analysis algorithms used to generate the CFG of a program. The vast majority of these build on top of the same algorithm to analyse the structure of a binary but differ in how hard they try to reach higher levels of coverage. In general, this corresponds to how much effort is put into performing analysis on indirect branching and indirect procedure invocation [34]. Static analysis is not the main goal of this project so we did not aim to maximise coverage. Instead, we establish a basic awareness of the general techniques available and ensure the CFG analysis code we write is amenable to extensions (such as support for indirect branch analysis). To demonstrate this, we implement a proof of concept example of one case of indirect branching. This shows that extending the CFG analysis of `libbf` to be more powerful in the future is feasible.

The aim of the CFG analysis in `libbf` is to generate a basic CFG, ignoring indirect branching. The purpose of the generated CFG is to help users locate and identify source and destination basic blocks for detours and trampolines. Instead of building our own CFG analysis, we could have used existing libraries but in the general case, they build above their own disassembler engine. We wanted to be able to use our own disassembler engine to have tighter control over the interface and the library architecture without having to wrap an existing library. A further advantage of using our own disassembler engine is that we can control the implementation. We take advantage of this by using our own instruction decoder to completely eliminate the use of strings.

In our CFG analysis, we consider only direct branching and direct calls. What this means is that indirect calls due to function pointers or jump tables can not be resolved. For this reason, analysing programs with characteristics such as packing or polymorphism will not produce useful results. As part of its API, `libbf` gives the user the ability to dump part of or the whole of a CFG. We will use these dumps to demonstrate how the analysis works.

CFG Generation Algorithm

The CFG analysis is a recursive process for detecting basic blocks and is defined as so:

1. The disassembly is started from a root. Initially, this is typically the entry point.
2. A new `bf_basic_blk` object is created and associated with the current address being disassembled. The CFG analysis then drives the disassembler engine in a linear fashion. This means if instruction `insn` is disassembled, the next instruction to be disassembled will be the one at `address_of(insn)+ size_of(insn)`. All disassembled instructions are added to the current `bf_basic_blk` object. The disassembly stops if any of the following special cases are encountered:

   Case 1: Flow ending (`ret`, `sysret`, ...). The end of the `bf_basic_blk` has been reached and disassembly terminates. This is the base-case.

   Case 2: Unconditional branching (`jmp`/`jmpq`). If resolvable, a new `bf_basic_blk` is generated from the branch target by going to **Step 1** with the branch target as the root. The current `bf_basic_blk` is linked to the returned one.

   Case 3: Conditional branching or procedure call If resolvable, two new `bf_basic_blk` objects are generated from the branch/call target and the next linear address by going to **Step 1** with the branch/call target and next linear address as the root. The current `bf_basic_blk` is linked to both new `bf_basic_blk` objects.

As new `bf_basic_blk` objects are created/discovered, they are added to a global hash table. If **Step 1** starts with a root that is already associated with a `bf_basic_blk`, the existing `bf_basic_blk` is returned (this is known as a back edge and is typical in loops). In order to understand each case better, we will visualise an example of each from a `libbf` CFG dump. Listing 5.10 is an example of **Case 1**:

```
int f1(void)
{
        int a = 1;
        int b = 2;

        a += b;
        return a;
}
```

Listing 5.10: The entire function is a single basic block.

The corresponding CFG is:

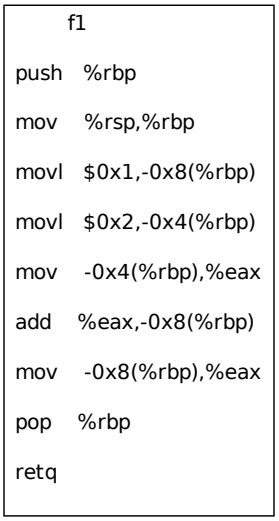


Figure 5.7: This CFG generation corresponds to the first case. A flow ending instruction (`retq`) was the first 'special' instruction to be reached. This terminates the disassembly of the current `bf_basic_blk`.

Listing 5.11 is an example of both **Case 2** and **Case 3**:

```
void f2(void)
{
        puts("f2");
}
```

Listing 5.11: This function results in a procedure call and subsequently an unconditional branch.

The corresponding CFG is:

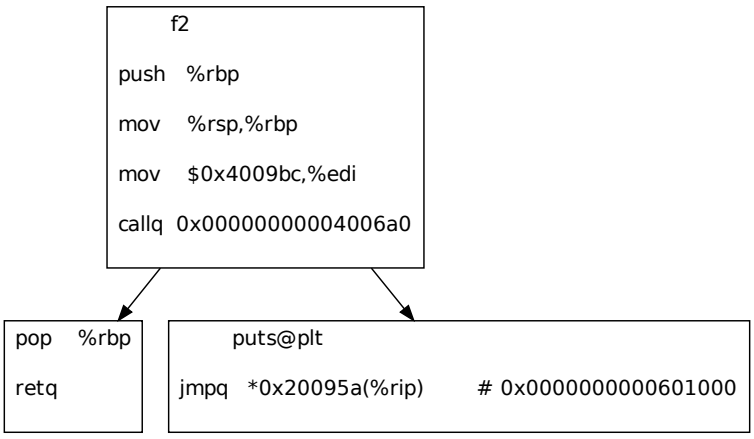


Figure 5.8: This CFG demonstrates the recursive nature of the generation process.

Figure 5.8 shows `f2` being disassembled. The first 'special' instruction reached is `callq`. This is an example of **Case 3** so the following rules are followed:

Next linear address The next linear address is disassembled as a new `bf_basic_blk` and linked to the current `bf_basic_blk` (`f2`). In this case, the next linear address holds `pop %rbp`. The new block is disassembled until the `retq` instruction, which falls into **Case 1** terminating the disassembly.

Call target The call target is `0x4006a0` which is an absolute value and therefore resolvable by the disassembler engine. This means `0x4006a0` is disassembled as a new `bf_basic_blk` and linked to the current `bf_basic_blk` (`f2`). The call target contains a single instruction, `jmpq` which falls into **Case 2**. However, the value of the operand (`*0x20095a(%rip)`) is unresolvable so the disassembly terminates.

The algorithm described above is slightly simplified. One special case not encapsulated are critical edges. A critical edge occurs when the middle of an existing basic block is disassembled as a recursive root. In these cases, the block must be split. The C code in Listing 5.12 will result in a critical edge.

```
void f3(void)
{
        int a = 1;
        int b = 2;

        do {
                a += b;
        } while(a < 10);
}
```

Listing 5.12: This function results in a critical edge during CFG generation.

The corresponding CFG is:

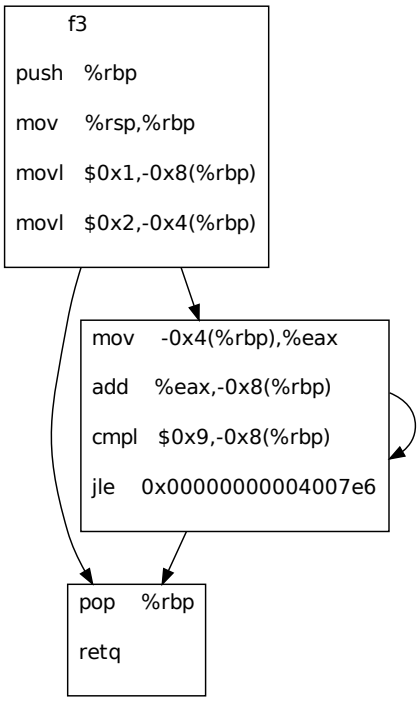


Figure 5.9: This CFG demonstrates block splitting.

Following the CFG generation algorithm, `f3` is disassembled to the first 'special' instruction, which in this case is the `jle 0x4007e6`. This is a conditional branch and falls under **Case 3**. By the rules, the following are disassembled:

Next linear address This case is uninteresting since we already saw a similar example earlier so we shall not cover it.

Branch target The branch target is to the `mov -0x4(%rbp), %eax` instruction. However, this is already part of an existing `bf_basic_blk` (the current one). This means there are should be two incoming links to a new `bf_basic_blk` starting at the address of the `mov -0x4(%rbp), %eax` instruction. The way this is achieved is by splitting the existing `bf_basic_blk`. After the split, two links are added: one from the first half of the split and one from the current `bf_basic_blk`.

In theory, a perfect CFG analysis would require only one root (the entry point) and would be able to discover all instructions in the binary file. This is because the entry point dominates all blocks. In practice, this does not happen for two reasons:

Dead code If code is unreachable by regular execution (such as a defined function which is never invoked), then it follows that the CFG analysis can not reach it.

Imprecise control flow analysis Static CFG analysis is limited in precision because indirect branches are difficult to resolve (as demonstrated by Figure 5.8).

As mentioned earlier, there are other CFG analysis techniques which will yield higher precision. Most notably, dynamic techniques can be used to gain precision. By running a binary under a monitor, indirect branches can be resolved by breaking code on the branch and checking memory and register values. The payoff is poor code coverage because often alternative code paths are not taken. It is outside the scope of this project to consider such dynamic techniques and other related methods such as symbolic and concolic execution [6, 27]. If such analysis were added to `libbf`, it could work independently to the static analysis and the extra CFG nodes found could be added through the current API.

One extra point to note is that so far we have only considered CFG generation from a single root. By starting recursive CFG generation from more roots, it is possible to discover more of the program and achieve higher code coverage. `libbf` allows the user to specify the roots of disassembly. For example, the user could choose to disassemble with every symbol as a root. Further details on the interface exposed by `libbf` are in Chapter 7.

Functions in the CFG

Generally, functions play no part in a CFG because it is tricky to identify a function from assembly code and it provides no extra syntactic information. For example, if the compiler optimises a function by inlining it, it will look no different to a regular basic block. Despite this fact, it is useful to have such semantic information available to the user if it can be obtained. There are three ways in which a `bf_basic_blk` can be identified as the start of a function:

1. If a `bf_basic_blk` is the target of a call site.
2. If a `bf_basic_blk` corresponds to an address identified as a `bf_sym` function.
3. If the user defined it as a root for CFG generation and explicitly stated it is a function.

The first two cases are intuitive. Regarding the second case, `bf_sym` objects will be covered in more depth in Section 5.2.3. The third case refers to the fact that when a user passes in a root for CFG generation, it is possible he knows the root is a function even though it can not be identified from the first two cases. In this case, the user can optionally pass a flag to notify `libbf` about this information.

When a `bf_basic_blk` is suspected to represent the start of a function by satisfying any of the three cases defined above, it is annotated by the library as a `bf_func` object. The main advantage of this is that the `libbf` has macros which allow the enumeration of `bf_func` objects separately to `bf_basic_blk` objects. As an example, this means the user can easily instrument every function instead of every basic block with the following iterator:

```
struct bf_func * func;

bf_for_each_func(func, bf) {
  bf_trampoline_func(bf, func, dest_func);
}
```

Listing 5.13: Iterating all `bf_func` objects.

Demonstration of Indirect Branching Analysis

So far, we have only brushed over the lack of support for indirect branching analysis in `libbf`. As discussed, advanced static analysis is outside the scope of this project. However, it is important that the library is implemented in a way that is amenable to extensions. To demonstrate this is

the case, we extend `libbf` to support one basic case of indirect branching. Consider the x86-64 architecture and the following sequence of assembly instructions:

```
pushq $0xabcdef12
movl $0x12345678,0x4(%rsp)
retq
```

Listing 5.14: Indirect branch.

In terms of changing the instruction pointer (`%rip`), the above sequence is equivalent to:

```
movabs $0x12345678abcdef12, %rax
jmpq *%rax
```

Listing 5.15: Changes the instruction pointer equivalently to Listing 5.14.

Functionally, the only difference is that Listing 5.15 clobbers `%rax`. Other than that, both sequences of instructions will branch execution to `0x1234567abcdef12`. If it is not clear why this is the case, this will be explained in Section 5.4. For now, it is not important exactly why such a mechanism results in a branch, just that it does. In both scenarios, the branch is indirect because the address of the next instruction is not specified as an operand. In the first case, the destination is specified on the stack and in the second case, in the `%rax` register.

Functionality was added to the CFG analysis of `libbf` to allow it to recognise sequences of instructions such as in Listing 5.14 and to treat it as an unconditional branch. For example, consider the following CFG:

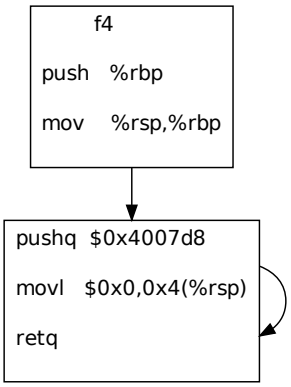


Figure 5.10: This CFG demonstrates the ability of `libbf` to resolve certain indirect branches.

The CFG shown in Figure 5.10 can be explained by following the CFG generation rules. Firstly, disassembly starts at `f4` where instructions are disassembled linearly till the first flow changing instruction. In this case, that instruction is `retq`. In the earlier description of the algorithm, the disassembly should terminate by **Case 1**. However, `libbf` automatically recognises the last three instructions as an indirect branch and resolves the branch location to `0x4007d8`.

In this example, `0x4007d8` is the address of the `pushq` instruction. As before, when a branch is detected, its target is disassembled and linked to the current basic block. The CFG analysis is able to recognise that disassembling `0x4007d8` in this case results in block splitting. This explains why there are two basic blocks in the above CFG with one block looping onto itself. From a higher level, `f4` creates a stack frame then infinitely loops in the second basic block.

This case of indirect branching is one of the more simplistic examples because it deals with a

very specific case of three consecutive instructions. The static resolution of indirect branching is more difficult to handle for general cases. For example, to resolve a register value, it might be necessary to analyse interactions with that register in predecessor basic blocks. If there are multiple predecessors, it might be impossible to statically resolve the value at all. Static analysis involving memory and multiple threads is even more difficult to analyse without the heavy use of heuristics.

The key concept demonstrated by this proof of concept is that the static analysis of instructions during CFG generation is not limited to a single instruction and can easily be extended. As an approximate measure of the complexity of this, the support for this case of indirect branching required around 40 lines of C code which was plugged directly into the CFG analysis.

5.2.3 Symbol Table Reader

The symbol table reader is responsible for providing an interface through which symbols (if any) contained in the binary file can be accessed. `libbfd` provides access to the symbol table as part of its API so the symbol table reader simply wraps this functionality. The main advantage of doing this is that the user does not need to learn the complicated API of `libbfd`.

When a binary file is loaded in `libbf`, the first step is usually to generate the CFG. After this, the library automatically augments the CFG by storing and associating additional symbol information with nodes (`bf_basic_blk` objects) in a similar way to how `bf_func` annotations work. For example, the following CFG has been augmented with symbol information:

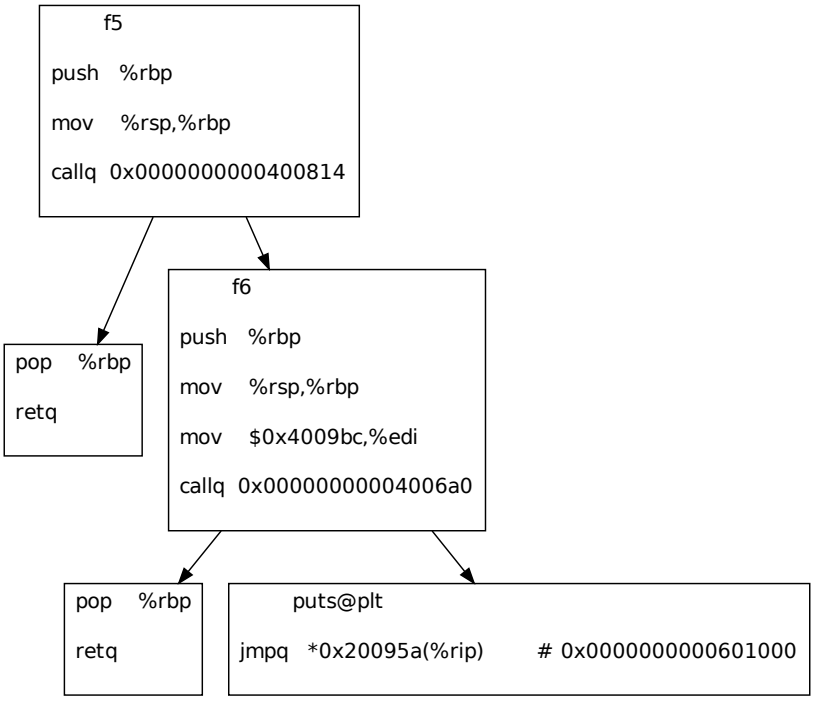


Figure 5.11: This CFG has been augmented with symbol information. `f5`, `f6` and `puts@plt` are all symbols extracted from the original binary. In this case, all three symbols denote a function so the three `bf_basic_blk` objects are additionally annotated as `bf_func` objects.

5.3 Object File Injector

The role of the object file injector is to inject an object file containing user-defined routines into the target binary file. The purpose of this is that it allows users to add extra functionality that is not existing in the binary. For example, if the user wanted to instrument a particular function to print a notification when it is invoked, he might trampoline the execution of that function to a routine defined like so:

```
void notify(void)
{
  puts("The instrumented function was invoked.");
}
```

Listing 5.16: User-defined routine for instrumenting notification.

Before we discuss the details of the approach we shall clarify one important concept. When we discuss object file injection, we consider a relocatable object file. Recall that in a relocatable object file, the code and data has relocation information which allows it to be linked at an arbitrary base. For this reason, when we speak of object file injection, we can equivalently consider the injection/merge of a fully compiled binary file since an object file can be transformed to one by fully linking it (for example against a dummy `main`). What this means is that if we can find a method to merge two binary files we will have solved the problem of object file injection. The reason it is useful to try to solve the binary merge problem rather than object file injection is that `libbfd` does not provide functionality which would make the injection of an object file convenient. That is, fully linking a relocatable object (with *ld*) and attempting a merge is easier than manually relocating with `libbfd` and then attempting the merge.

5.3.1 Original Approach

The original idea was to use `libbfd` to implement object file injection. The intuition behind this was the fact that EEL allows object file injection and works on top of `libbfd`. During the implementation stage, we realised that the BFD library does not offer any functionality directly for object file injection. In fact, as we saw in Section 5.1.1, `libbfd` is not designed for patching files at all. In our original approach, we tried to work around this limitation..

We consider two binaries containing code and data we want to merge together as shown in Figure 5.12. We take the following steps:

Create fresh binary We use the same trick as we originally used in the binary abstraction layer. That is, we create a fresh and empty binary and copy the sections of the original and routines binary.

Merge stage We deconstruct and extract the sections from each of the source binaries and create corresponding sections in the fresh binary. This is a simplified explanation of this step. In practice, conflicting section names need to be resolved (by renaming the section).

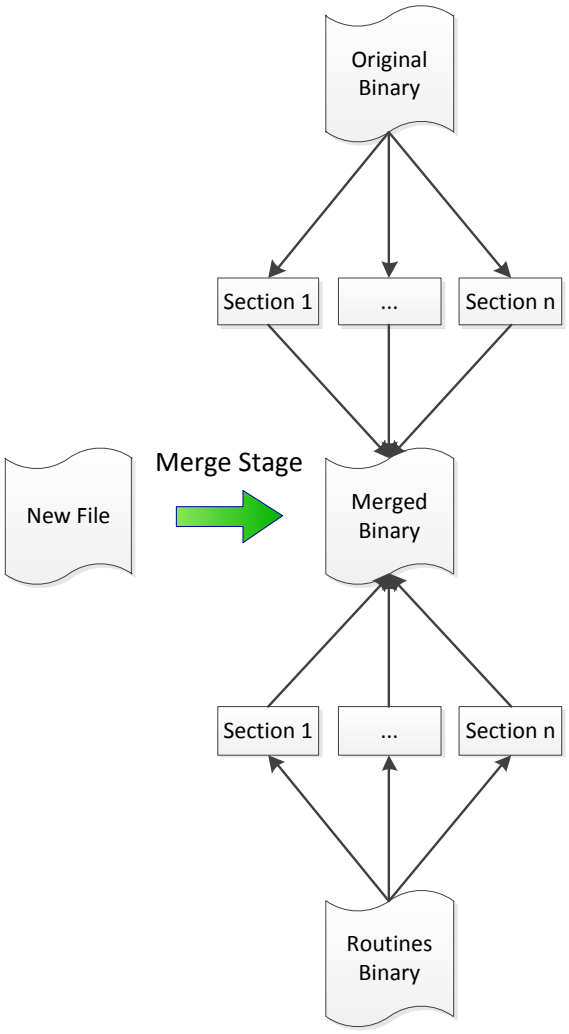


Figure 5.12: Our first approach to object file injection using `libbfd`. The original binary is the target binary and the routines binary is the fully linked version of the object file.

The problem with this approach is shown in Figure 5.13. The resulting merged binary can not have overlapped sections. To understand what an overlapped section is, consider that a section has a base address and a size. Hence, when a program is loaded into memory, each section represents a contiguous block of memory relocated to some base address. An overlapped section occurs when the base address of one section is defined in the middle of some other section. The method we used to resolve this issue is:

Compile and fully link routines binary This step is the same as before. The object file to be injected is fully linked such that all sections are relocated to some base. At this point, it is likely that there are section conflicts with the target binary.

Analyse binary section layouts `libbf` extracts the section information of both the target binary and the routines binary and stores this information.

Generate non-conflicting section layout In this step, we calculate non-conflicting base addresses for all the sections of the routines binary.

Relink the object file to use the new section layout The object file to be injected is relinked to use the new section layout generated from the previous step. This step can be achieved through the use of linker flags, specifying the section start address for each section (`--section-start=.secname=0xABCDEF, ...`). Effectively, this relocates all the sections in the routines binary to non-conflicting addresses.

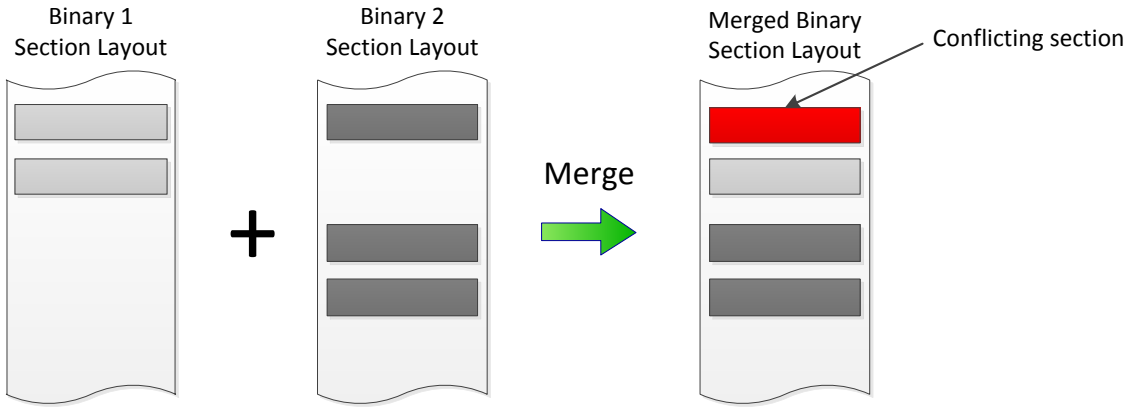


Figure 5.13: Section overlap is an issue which arises with the original approach.

Figure 5.14 and 5.15 illustrate the described method we used to solve the section overlap issue.

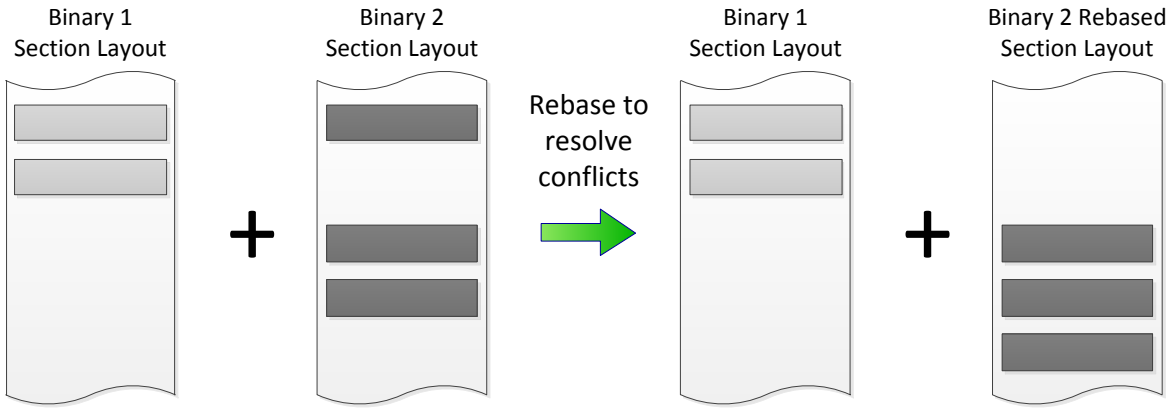


Figure 5.14: Method used to solve the section overlap issue shown in Figure 5.13. In this stage, the non-conflicting section layout is generated and the routines binary is relinked according to this.

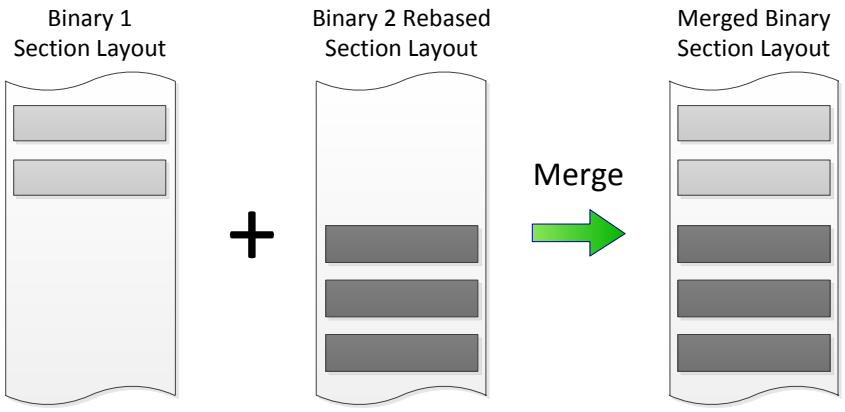


Figure 5.15: After relinking to relocate overlapping sections, the target binary and routines binary can be merged without conflict.

After solving the section overlap issue, we hit another problem which could not be resolved. It appears that `libbfd` is not designed for the merging process we used. The problem is that the symbol tables are not merged properly. Rather than merging the two symbol tables[3], `libbfd` simply creates two symbol tables in the merged binary. The code from the target binary uses one of these symbol tables to resolve its symbols and the code from the routines binary uses the other symbol table. However, the ELF file format mandates that a file can not contain multiple dynamic symbol tables. As we observed when writing the binary abstraction layer, sometimes the BFD abstraction abstracts too far and does not allow more specific manipulation at a finer granularity. We conclude that the problem of object file injection can not be fully solved with `libbfd`. Any solutions building on top of `libbfd` likely reach through to the underlying file format through file specific libraries such as `libelf`.

In hindsight, we can speculate that EEL was a mature project, and perhaps the reason they required the recursive object file injection method was because that is the best that is achievable through the BFD abstraction. In the general case, it is true that the higher the level of abstraction used, the easier the implementation will be. However, we have no restriction that forces us to use BFD and if it is the case that BFD abstracts too far to offer the functionality we need, there is no reason for us not to delve directly into the ELF layer. In fact, we did exactly this with the binary abstraction layer where we chose to use `libelf` instead of `libbfd` for the same reason. One option we could have taken is to pursue this approach further by using `libelf` to merge the symbol tables properly. However, the solution as it stands required over 3 KLOC of C code. We believe a cleaner and more elegant solution exists which leads us to the final approach.

5.3.2 Final Approach

After further investigation, we found that the problem of object file injection has already been solved. Of course, we know this problem has previously been solved because of its usage in tools like EEL, but the source code and method is unavailable in such cases. Instead, our final approach was discovered in one of the publications presented in an ezine written by Phrack [19]. The documented approach is the same one used by ELFsh.

The main algorithm of object file injection is as follows:

[3]Technically, there are 4 symbol tables in total because each binary contains a regular symbol table (`.symtab`) and a dynamic symbol table (`.dynsymtab`)

Insert .bss section The `.bss` section is used by many compilers and linkers to contain statically-allocated variables which are typically initialised to zero. The first step of the algorithm is to insert the .bss section from the object file into the target binary. Each static symbol from the .bss of the object file is inserted into the .bss of the target binary and the symbol table updated to reflect the additions.

Section injection The sections from the object file are loaded into the target binary. Different sections are injected in a different way but this is an implementation detail we will not cover. For now, the sections are injected without performing relocation.

Symbol table fusion This is the missing step from our original approach. Each entry in the symbol table of the object file is inserted into the symbol table of the target binary.

Relocation of injected sections The section injection earlier did not relocate the injected sections. This step locates the relocation information for each section and relocates the section. In the original approach, relocation required two passes (to avoid section overlap) whereas conflicts can be resolved in a single pass with this method.

This algorithm is implemented in ELFsh but we can not use it directly because the ELFsh project has not been updated for over three years and the project is unmaintained and no longer builds. The implementation builds on top of `libelfsh`, an internal library of the project which reproduces much of the functionality in `libelf`. This library is now out of date which means porting the algorithm to `libbf` would require either updating `libelfsh` or preferably, re-implementing the required functionality above `libelf`. Due to time constraints, the port of the algorithm from `libelfsh` to `libbf` was not fully completed. We do not regard this as a significant problem because the rest of library is usable as is demonstrated in Evaluation (Chapter 7). Instead of using an object injector, we statically link the code against the target by modifying the source code. Most importantly, we have found a working solution which can feasibly be implemented in `libbf`.

5.4 Static Patcher

The static patcher is perhaps inaptly named because the role it plays is not the actual binary rewriting, which is performed by the binary abstraction layer. Instead, the static patcher is responsible for deciding what bytes need to be changed in order to achieve a detour/trampoline and then requesting these modifications from the binary abstraction layer. In this section, we will cover how the following features are achieved in `libbf`:

1. Detouring/trampolining x86-32 code
2. Detouring/trampolining x86-64 code

`libbf` allows both `bf_basic_blk` and `bf_func` objects to be detoured. We already know `bf_func` objects are really just annotations on `bf_basic_blk` objects but the importance of this is at the code level and how much flexibility we have when trampolining. If we only considered trampolining `bf_func` objects, we would be allowed to clobber certain registers in our detour. Consider the *System V AMD64 ABI*, which is the calling convention used under Linux x86-64:

3.2.1 Registers and the Stack Frame

[...]

This subsection discusses usage of each register. Registers `%rbp`, `%rbx` and `%r12` through `%r15` "belong" to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function.[a] If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

[...]

3.2.3 Parameter Passing

[...]

Passing Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:

1. ...
2. If the class is INTEGER, the next available register of the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` is used[b].
3. ...

[a] Note that in contrast to the Intel386 ABI, $\%rdi$ and $\%rsi$ belong to the called function, not the caller.

[b] Note that $\%r11$ is neither required to be preserved, nor is it used to pass arguments. Making this register available as scratch register means that code in the PLT need nto spill any registers when computing the address to which control needs to be transferred. $\%rax$ is used to indicate the number of vector arguments passed to a function requiring a variable number of arguments. $\%r10$ is used for passing a function's static chain pointer.

If we were only trampolining functions, we would be allowed to clobber all registers which "belong" to the called function which are not being used for parameter passing. In fact, we can see there is always at least one general purpose register which we are allowed to use as a scratch register (`%r11`). On the other hand, when detouring a basic block, there is no way to determine which registers can be clobbered without performing register scavenging to find unused registers. At the basic block level, the worst case scenario is that there are no unused registers at all.

As we will see later, having a scratch register available when trampolining x86-64 is an advantage. However, we aim to additionally trampoline at the basic block level. The implication of this is that any trampolining solution we use must preserve all registers. The intuition behind this is that the context of a program is stored in its registers at any given point in time. As an example, at some given point in time a live register might be storing an intermediate value of a calculation. If our trampolining clobbers this value, the behaviour of the program will change unexpectedly in a way we do not want when we return execution to the trampolined basic block.

In summary, any instruction or sequence of instructions we use for implementing detouring must satisfy two conditions:

Arbitrary branch destination The detour must be able to branch to any destination in the virtual address space. This is an implicit requirement which we have not covered in depth but we want to allow the user to be able to detour *from* anywhere *to* anywhere.

Register preservation In order to ensure basic blocks are trampolined without corrupting the behaviour of the original program, all registers must be preserved across detours.

5.4.1 x86-32 Detouring/Trampolining

Under x86-32, every process has a virtual address space from `0x00000000-0xffffffff`. The relative branch instruction is capable of jumping to any destination in this address space. The relative branch is a 5 byte instruction encoded with the first byte as `0xe9` and the last 4 bytes as the delta between the address of the next instruction and the branch target. For example:

Address	Bytes	Instruction
`0x2904`	`e9 ba 04 08 83`	`jmp 83082dc3`

The address of the jump instruction is `0x2904` and the address of the next instruction is 5 bytes forward because the current instruction is 5 bytes (`0x2904 + 0x5 = 0x2909`). The branch target is `0x83082dc3` which means the delta is calculated by `0x83082dc3 - 0x2909 = 0x830804ba`. This encoding is exhibited in the above table. The reason the last 4 bytes are in reverse is because the x86 architecture uses the little-endian format.

x86-32 Detouring

The relative branch described above can be used for detours because it satisfies both our requirements (arbitrary branch destination and register preservation). In order to see in more detail what happens when a detour is set, we shall look at an example C program:

```
#include <stdlib.h>
#include <stdio.h>

/*
 * func1 is invoked by the regular execution of this program.
 */
void func1(void)
{
  puts("func1 was invoked");
}

/*
 * func2 is not invoked by the regular execution of this program. The aim is to
 * detour execution to this function.
 */
void func2(void)
{
  puts("func2 was invoked");
}

int main(void)
{
  func1();
  return EXIT_SUCCESS;
}
```

Listing 5.17: Example process to be detoured.

The corresponding CFG generated by `libbf` is:

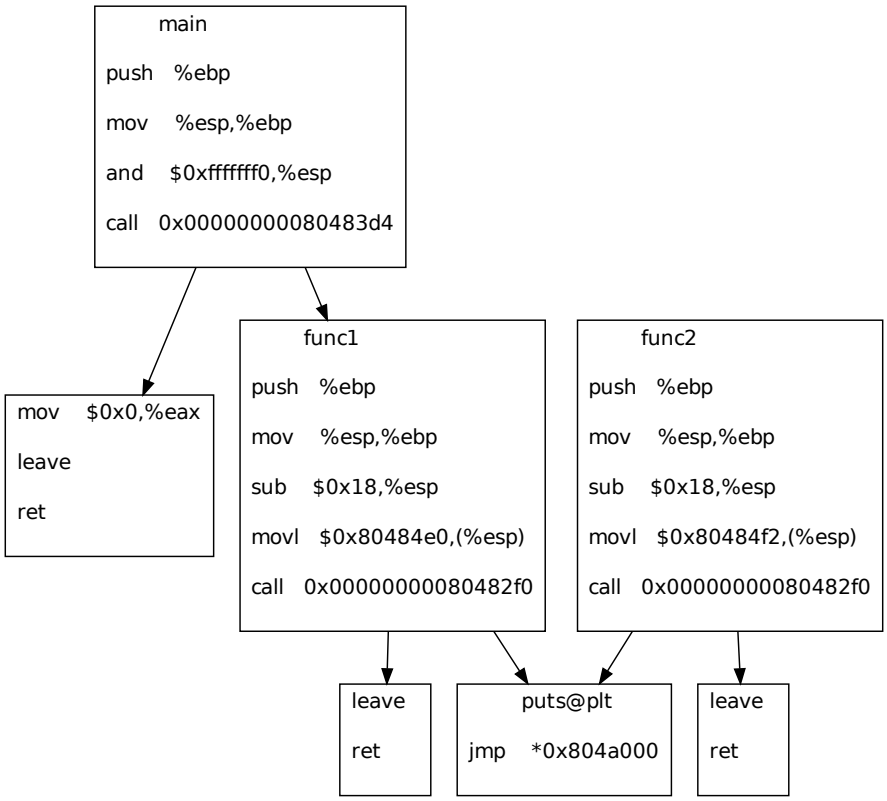


Figure 5.16: The corresponding CFG to Listing 5.17.

It is clear from both the code and the CFG that `func2` will not be invoked by regular execution because it is not reachable. During the regular execution of the code, we would expect "`func1 was invoked`" to be printed. The goal is to detour the execution at the start of `func1` to `func2`. If this is successful, the modified program should print "`func2 was invoked`" instead.

Detouring `func1` to `func2` means `libbf` needs to write 5 bytes at the address of `func1` to place a relative branch to `func2`. The CFG of the detoured program is shown in Figure 5.17.

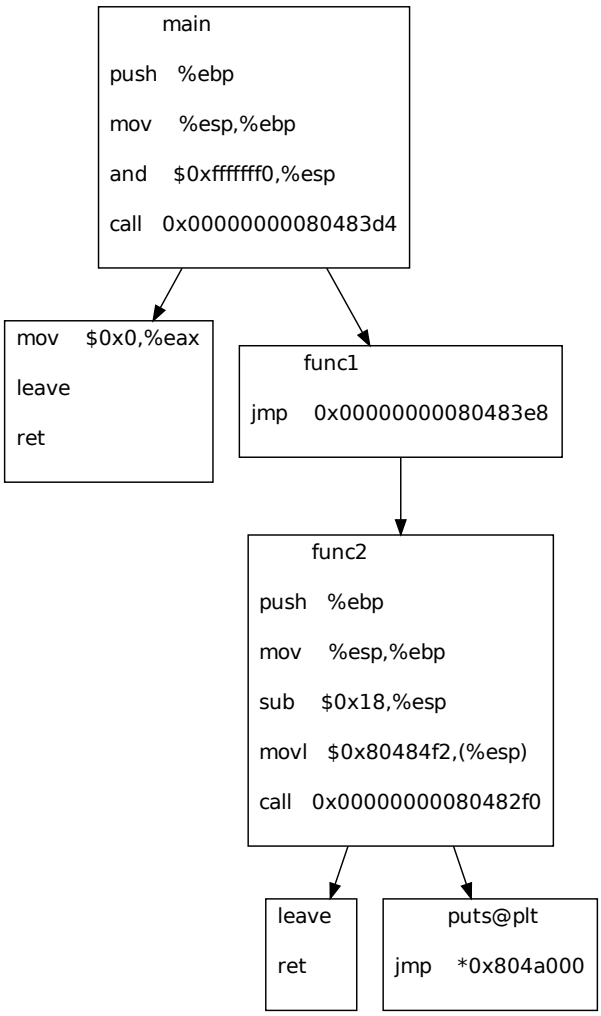


Figure 5.17: The CFG after `func1` is detoured to `func2`.

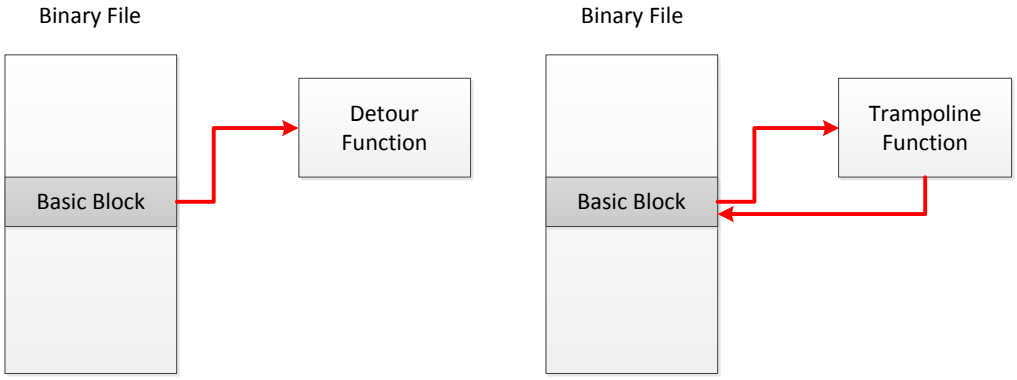


Figure 5.18: A comparison of detouring and trampolining. In detouring, the execution is redirected to a detour function. The difference with trampolining is that after the trampoline function is executed, the execution is returned to the basic block that was detoured.

x86-32 Trampolining

To briefly recap, trampolining refers to the ability to detour execution of a function or basic block to an arbitrary function, which we shall call the trampoline function. The difference between detouring and trampolining is illustrated in Figure 5.18. In order to achieve trampolining from a source basic block, the following steps need to be taken:

Place forward detour The forward detour is the detour from the source basic block to the trampoline function. As with regular detouring, a 5 byte relative branch is written at the start of the source block causing execution to jump to the trampoline function.

Execute trampoline function The contents of the trampoline function are executed. As an example, the trampoline function might print logging information.

Trampoline function epilogue relocation This stage will be covered later to simplify the explanation. For now, it is enough to know an extra step exists.

Overwritten instructions need to be executed The detour from the source block to the trampoline function overwrites the first 5 bytes of the source block for the relative branch. These instructions need to be copied into the trampoline function and executed.

Place backward detour The backward detour is the detour from the end of the trampoline block to the rest of the source block.

The key concept here is that when the forward detour is placed, it overwrites 5 bytes at the start of the source basic block. For example, this is the disassembly of `func1`:

Address	Bytes	Instruction
0x80483d4	55	push %ebp
0x80483d5	89 e5	mov %esp, %ebp
0x80483d7	83 ec 18	sub $0x18, %esp
0x80483da	c7 04 24 e0 84 04 08	movl $0x80484e0, (%esp)
0x80483e1	e8 0a ff ff ff	call 80482f0 <puts@plt>
0x80483e6	c9	leave
0x80483e7	c3	ret

After a detour, the disassembly of the same function is:

Address	Bytes	Instruction
0x80483d4	e9 0f 00 00 00	jmp 80483e8 <func2>
0x80483d9	90	nop
0x80483da	c7 04 24 e0 84 04 08	movl $0x80484e0, (%esp)
0x80483e1	e8 0a ff ff ff	call 80482f0 <puts@plt>
0x80483e6	c9	leave
0x80483e7	c3	ret

The first 5 bytes show the relative branch instruction that was written. The next instruction is now `0x90`. The reason for this is that when the relative branch instruction was written, the last instruction it overwrote was only partially overwritten. One last byte should remain which would be `0x18` from `sub $0x18, %esp`. When `libbf` detects that an instruction was overwritten partially, the remaining bytes are padded to `nop` instructions. This serves two purposes:

Aids disassembly If there is are random junk bytes existing, linear disassembly from most disassemblers will assume these bytes are the start of the next instruction. This generates junk instructions which are not useful.

Simplifies backward detour By replacing junk bytes with `nop`, the backward detour simply needs to branch back to `source_block + 5`.

This means when we copy the overwritten bytes, we may have to copy more than 5 bytes. If the relative branch partially overwrites an instruction, that whole instruction must be copied to the trampoline function. In this case, the following instructions all have to be copied to the trampoline block:

Address	Bytes	Instruction
`0x80483d4`	`55`	`push %ebp`
`0x80483d5`	`89 e5`	`mov %esp, %ebp`
`0x80483d7`	`83 ec 18`	`sub $0x18, %esp`

It should be noted that speaking of copying these instructions is a simplification. In practice, these instructions have to be individually *relocated* to the trampoline function. The reason for this is that some instructions (such as the relative branch) encode relative information as part of the instruction bytes. In order to preserve the meaning of these instructions when copying, this relative information has to be updated accordingly. `libbf` detects which instructions require special relocation and which instructions can be simply copied and deals with them separately.

When setting up trampolines, `libbf` requires that the trampoline function ends in a long block of `nop` instructions. The reason for this is that the number of bytes that require relocation/copying is unknown until the function to be trampolined is disassembled. `libbf` caters for the worst case scenario:

Address	Bytes	Instruction
`0x0`	`xx xx xx xx`	`Some 4 byte instruction`
`0x4`	`xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx`	`Some 15 byte instruction`

The worst case scenario is when a very long instruction starts on the 4th byte of the source basic block. The longest instruction on both x86-32 and x86-64 is 15 bytes long which means in the worst case 19 bytes need to be copied. For this reason, `libbf` requires all trampoline functions to be padded with 24 `nop` instructions. The first 19 are for copying overwritten instructions and the last 5 are for the backward detour. If the padding is not present, `libbf` will not attempt to perform the trampoline.

Listing 5.18 shows an example of a trampoline function. We demonstrate how trampolining works in `libbf` by working through this example. When the original executable is executed, it will output "`func1 was invoked`". The aim is to trampoline execution to `func2` which should cause "`func2 was invoked`" to be outputted followed by "`func1 was invoked`".

```
#include <stdlib.h>
#include <stdio.h>

#ifdef __i386__
  #define TRAMPOLINE_BLOCK \
    ({ \
        asm volatile ("nop\n");\
        ...
        asm volatile ("nop\n");\
    });
#endif
/*
 * func1 is invoked by the regular execution of this program.
 */
void func1(void)
{
  puts("func1 was invoked");
}

/*
 * func2 is not invoked by the regular execution of this program. The aim is to
 * trampoline execution to this function.
 */
void func2(void)
{
  puts("func2 was invoked");
  TRAMPOLINE_BLOCK
}

int main(void)
{
  func1();
  return EXIT_SUCCESS;
}
```

Listing 5.18: Example process to be trampolined. There should be 24 `nop` instructions but they have been removed for brevity.

The corresponding CFG for Listing 5.18 is shown in Figure 5.19. Before we look at the CFG after trampolining, we shall go back to the epilogue relocation stage we previously skipped over. In order to understand what the epilogue is and why it needs to be relocated, we shall briefly go over stack frames.

A stack frame is a common mechanism in assembly language that allows access to function parameters and automatic function variables. It is most easily demonstrated with a simple example. Consider the following C function:

```
void func(int a, int b)
{
       int x, y, z;

       x = a;
}
```

Listing 5.19: Example C function to illustrate stack frames.

The disassembly for the function shown in Listing 5.19 is:

Address	Bytes	Instruction	Comment
`0x80483b4`	`55`	`push %ebp`	Save old stack base pointer
`0x80483b5`	`89 e5`	`mov %esp, %ebp`	Set stack base pointer as top of stack
`0x80483b7`	`83 ec 10`	`sub $0x10, %esp`	Allocate space for local variables
`0x80483ba`	`8b 45 08`	`mov 0x8(%ebp), %eax`	Access argument through + offset
`0x80483bd`	`89 45 fc`	`mov %eax, -0x4(%ebp)`	Access automatic variable through - offset
`0x80483c0`	`c9`	`leave`	Epilogue
`0x80483c1`	`c3`	`ret`	Return

The concept behind stack frames is that each subroutine wants to act independently of its location on the stack. Ideally, each subroutine can act as if it is at the top of the stack. Linux x86-32 uses the *System V i386 ABI* which passes arguments through the stack. This means that if each subroutine assumes it is at the top of the stack then any arguments must be at known positive offsets. The reason for this is that the stack grows downwards. This is a more detailed explanation of the code:

Save old stack base pointer The stack frame is created by assigning the base pointer (held in `%ebp`) as the address of the top of the stack (held in `%esp`). Before assigning the new stack base pointer, the old one is saved on the stack so it can be restored at the end of the subroutine.

Set stack base pointer as top of stack This assignment changes `%esp` to point to the top of the stack.

Allocate space for local variables The stack grows downwards so subtracting the stack pointer extends the size of the stack. In this case, the stack is extended by `0x10` bytes. This is more than enough space for `3 * sizeof(int)` because in this case `sizeof(int)` evaluates to 4. The reason the stack is extended further than it needs to be does not concern us since it is simply an artifact of the code generation process. These first three instructions are collectively known as the *prologue*.

Access argument through + offset This line simply shows that arguments can be accessed through a positive offset from the base pointer because they appear further up the stack.

Access automatic variable through - offset Similarly, automatic variables can be accessed through a negative offset from the base pointer.

Epilogue This is the most important part of the code. The `leave` instruction in this case is equivalent to a `mov %ebp, %esp` followed by a `pop %ebp`. This restores the stack and base pointer to their values before the prologue.

Return Strictly speaking, the `ret` instruction is part of the epilogue but there is no need for us to relocate it. We will see why its presence is important.

While other compilers such as the Visual Studio compiler allow generation of naked functions (without prologue and epilogue) through attributes such as `__declspec(naked)`, GCC does not provide such functionality for the x86 architecture. This means all non-optimised functions will have a prologue and epilogue.

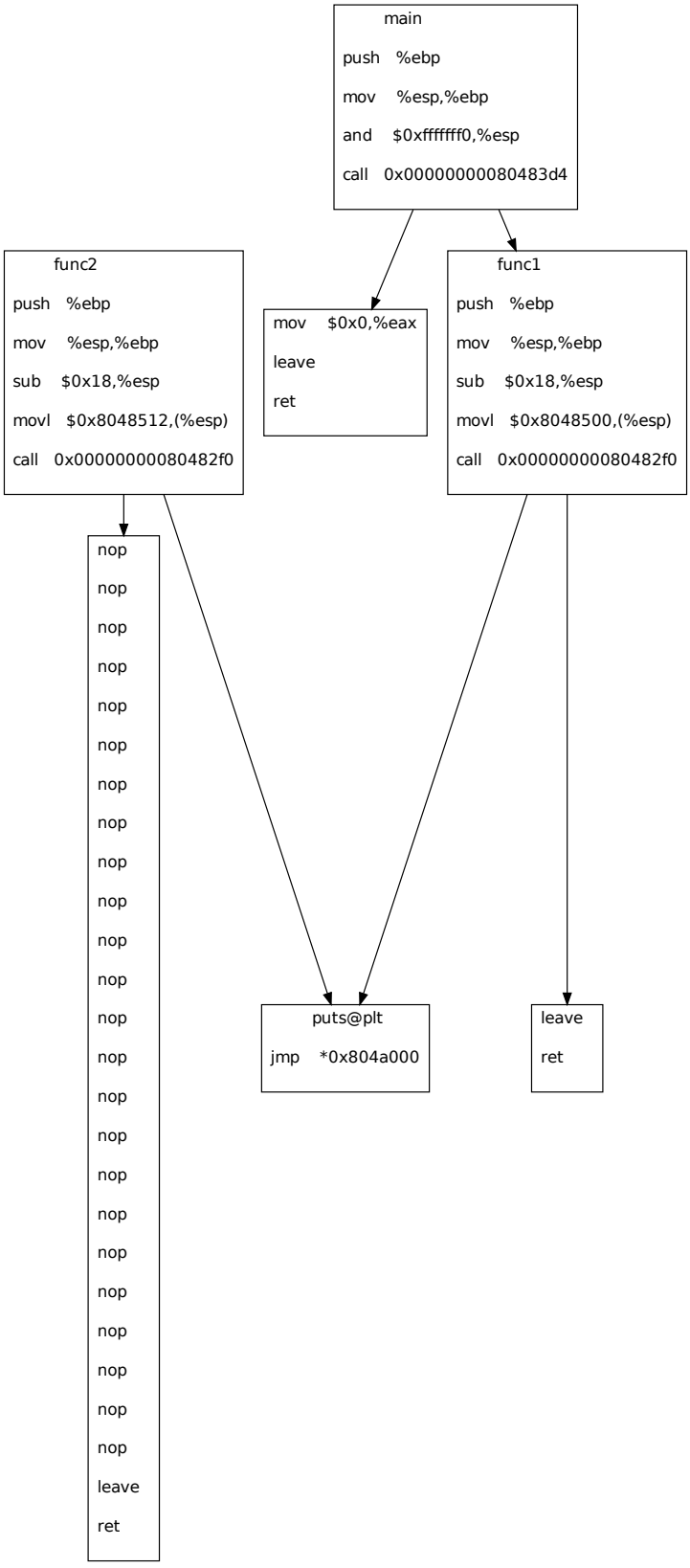


Figure 5.19: A trampoline function padded with 24 `nop` instructions.

If we relate the stack frame concepts to the code in 5.19, it is clear that the prologue is always invoked before the `nop` block and the epilogue is always at the end of the `nop` block. At least it should now be clear that the overwritten instructions from the forward detour can not simply be relocated to the start of the `nop` block. Doing so will cause them to use an incorrect stack frame. The solution to this is to:

Relocate the epilogue The epilogue (minus the `ret`) should be relocated to the start of the `nop` block. This restores the stack frame from before the prologue which is the one required by the overwritten instructions.

Relocate overwritten instructions The overwritten instructions should be relocated directly after the relocated epilogue. This allows them to use the stack frame they are expecting.

Backward detour The backward detour jumps execution back to `source_block + 5`.

The relocation of the epilogue happens dynamically by locating the `nop` block and `ret` instruction through disassembly and treating the instructions in between as the epilogue. The technique used for trampolining is verbose to explain but easier to understand by reading the code:

Address	Bytes	Instruction	Comment
func1():			
0x80483d4	55	push %ebp	*
0x80483d5	89 e5	mov %esp, %ebp	* Overwritten insns
0x80483d7	83 ec 18	sub $0x18, %esp	*
0x80483da	c7 04 24 00 85 04 08	movl $0x8048500, (%esp)	
0x80483e1	e8 0a ff ff ff	call 80482f0 <puts@plt>	
0x80483e6	c9	leave	
0x80483e7	c3	ret	
func2():			
0x80483e8	55	push %ebp	*
0x80483e9	89 e5	mov %esp, %ebp	* Prologue
0x80483eb	83 ec 18	sub $0x18, %esp	*
0x80483ee	c7 04 24 12 85 04 08	movl $0x8048512, (%esp)	
0x80483f5	e8 f6 fe ff ff	call 80482f0 <puts@plt>	
0x80483fa	90	nop	
...	...	...	
0x8048411	90	nop	
0x8048412	c9	leave	* Epilogue
0x8048413	c3	ret	

In the above case, the first relocation which must be performed is the epilogue to `0x80483fa`. The overwritten instructions should then be relocated directly after the relocated epilogue. The backward detour should be inserted directly after the relocated overwritten instructions. The prologue does not require relocation but is labelled for completeness. The following code is the result of `libbf` performing the trampoline:

Address	Bytes	Instruction	Comment
func1():			
0x80483d4	e9 0f 00 00 00	jmp 80483e8 <func2>	* Forward detour
0x80483d7	90	nop	*** nop padding
0x80483da	c7 04 24 00 85 04 08	movl $0x8048500, (%esp)	
0x80483e1	e8 0a ff ff ff	call 80482f0 <puts@plt>	
0x80483e6	c9	leave	
0x80483e7	c3	ret	
func2():			
0x80483e8	55	push %ebp	*
0x80483e9	89 e5	mov %esp, %ebp	* Prologue
0x80483eb	83 ec 18	sub $0x18, %esp	*
0x80483ee	c7 04 24 12 85 04 08	movl $0x8048512, (%esp)	
0x80483f5	e8 f6 fe ff ff	call 80482f0 <puts@plt>	
0x80483fa	c9	leave	* Relocated epilogue
0x80483fb	55	push %ebp	*** Relocated
0x80483fc	89 e5	mov %esp, %ebp	*** overwritten
0x80483fe	83 ec 18	sub $0x18, %esp	*** instructions
0x8048401	e9 d3 ff ff ff	jmp 80483d9 <func1+0x5>	***** Backward detour

The corresponding CFG generated by `libbf` is shown in Figure 5.20.

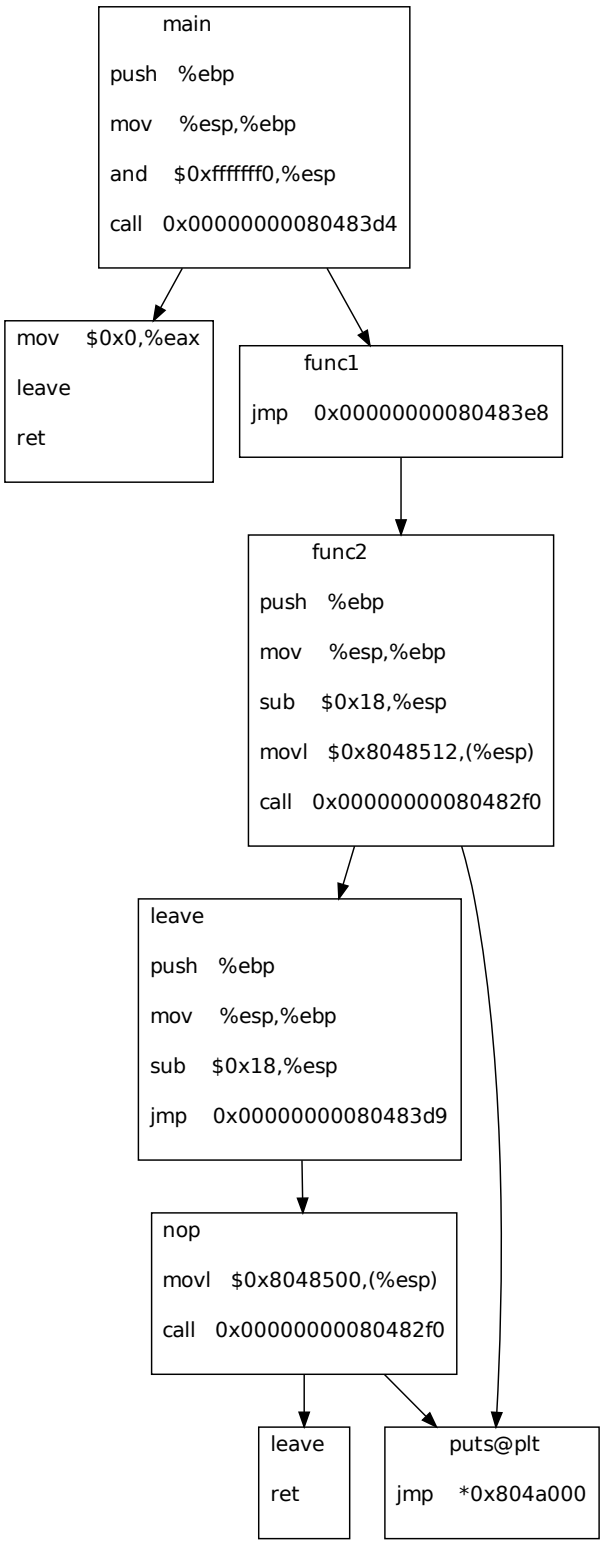


Figure 5.20: The CFG shown in Figure 5.19 after trampolining `func1` to `func2`.

5.4.2 x86-64 Detouring/Trampolining

Conceptually, detouring and trampolining basic blocks on x86-64 is the same as x86-32. The only difference is that there does not exist a single instruction which satisfies our detouring conditions (arbitrary branch destination and register preservation). A relative jump exists on x86-64 but it only allows branching within a 2GB range which means it fails the first condition. Some existing tools work around this by assuming all branch destinations will be in the 2GB range but we wish to solve this problem fully. In practice, the work-around is usually acceptable because of how linkers lay out code but we want to future-proof `libbf` and allow it to deal with larger binaries where sections may be arranged either more sparsely or across a greater range.

One common way to branch to any destination in the full address space of a x86-64 process is to load the address into a register and branch to that address:

```
movabs $0x12345678abcdef12, %rax
jmpq *%rax
```

Listing 5.20: Allows arbitrary branching but clobbers a register.

This is a convenient technique which we have briefly seen earlier. However, the reason this method is not acceptable for us is because it violates our second condition (register preservation). There are several alternatives involving memory accesses or saving/restoring of registers but we shall not cover these in further detail. Ultimately, the method we settled on was this sequence of instructions:

```
pushq $0xabcdef12
movl $0x12345678,0x4(%rsp)
retq
```

Listing 5.21: Allows arbitrary branching and preserves all registers.

The reason this technique was chosen is because it does not violate either of our conditions and also does not require modification of memory or dirty any registers. One particular advantage is that this method only requires 14 bytes. Requiring less bytes for a detour is important because basic blocks can not be detoured if they are less than the detour size. The reason for this is that if the detour overlaps to the next basic block, any incoming branches to the overlapped block will end up in the middle of a detour instruction or instruction sequence. Fixing up all incoming branches is a highly non-trivial process because it would entail finding *all* incoming branches including indirect ones.

Listing 5.21 works by pushing the absolute `qword`-sized address of the destination onto the top of the stack. The `ret` then treats this as a return address, popping it and branching to the destination. The `qword` is not pushed on all at once because the encoding of the instruction only allows a `dword`-sized immediate value to be pushed. The x86-64 stack is required to be 64-bit aligned which means the upper 32-bit half is present, but not populated. In other words, when the `pushq` executes, only a `dword` value is written, but the stack is extended by 64 bits. The `dword` value pushed corresponds to the lower half of the return address. Hence, the `movl` instruction is used to populate the upper half of the return address.

The fact that the detour is now 14 bytes means that the `nop` block must be longer to accommodate for the worst case scenario. The length of the required `nop` block can be calculated by summing the size of the forward detour (14), the size of the longest instruction (15) and the size of the backward detour (14). The required length is one less than this because the worst case scenario is if the longest instruction overlaps the first 14 bytes of the source basic block by 1 byte. Hence, the required length is `14 + 15 + 14 - 1 = 42`.

Other than these differences, the x86-64 static patcher works exactly the same as the x86-32 version. For this reason, it is uninteresting to go through the detouring/trampolining process in

as much detail as we did for x86-32. Instead, we will go through a single example of trampolining. As our example, we will use the 64-bit version of the example covered in Section 5.4.1. The disassembly of the original target binary is as follows:

Address	Bytes	Instruction	Comment
func1():			
0x4004f4	55	push %rbp	*
0x4004f5	48 89 e5	mov %rsp, %rbp	* Overwritten
0x4004f8	bf 3c 06 40 00	mov $0x40063c, %edi	* insns
0x4004fd	e8 ee fe ff ff	callq 4003f0 <puts@plt>	*
0x400502	5d	pop %rbp	
0x400503	c3	retq	
func2():			
0x400504	55	push %rbp	
0x400505	48 89 e5	mov %rsp, %rbp	
0x400508	bf 4e 06 40 00	mov $0x40064e, %edi	
0x40050d	e8 de fe ff ff	callq 4003f0 <puts@plt>	
0x400512	90	nop	
...	...	...	
0x40053c	90	nop	
0x40053d	5d	pop %rbp	* Epilogue
0x40053e	c3	retq	

After trampolining, the disassembly is as follows:

Address	Bytes	Instruction	Comment
func1():			
0x4004f4	68 04 05 40 00	pushq $0x400504	*
0x4004f9	c7 44 24 04 00 00 00 00	movl $0x0, 0x4(%rsp)	* Forward detour
0x400501	c3	retq	*
0x400502	5d	pop %rbp	
0x400503	c3	retq	
func2():			
0x400504	55	push %rbp	
0x400505	48 89 e5	mov %rsp, %rbp	
0x400508	bf 4e 06 40 00	mov $0x40064e, %edi	
0x40050d	e8 de fe ff ff	callq 4003f0 <puts@plt>	
0x400512	5d	pop %rbp	* Relocated epilogue
0x400513	55	push %rbp	***
0x400514	48 89 e5	mov %rsp, %rbp	*** Relocated
0x400517	bf 3c 06 40 00	mov $0x40063c, %edi	*** overwritten
0x40051c	e8 cf fe ff ff	callq 4003f0 <puts@plt>	*** instructions
0x400521	68 02 05 40 00	pushq $0x400502	*****
0x400526	c7 44 24 04 00 00 00 00	movl $0x0, 0x4(%rsp)	***** Backward detour
0x40052e	c3	retq	*****

The before and after CFG dumps are shown in Figure 5.21 and 5.22.

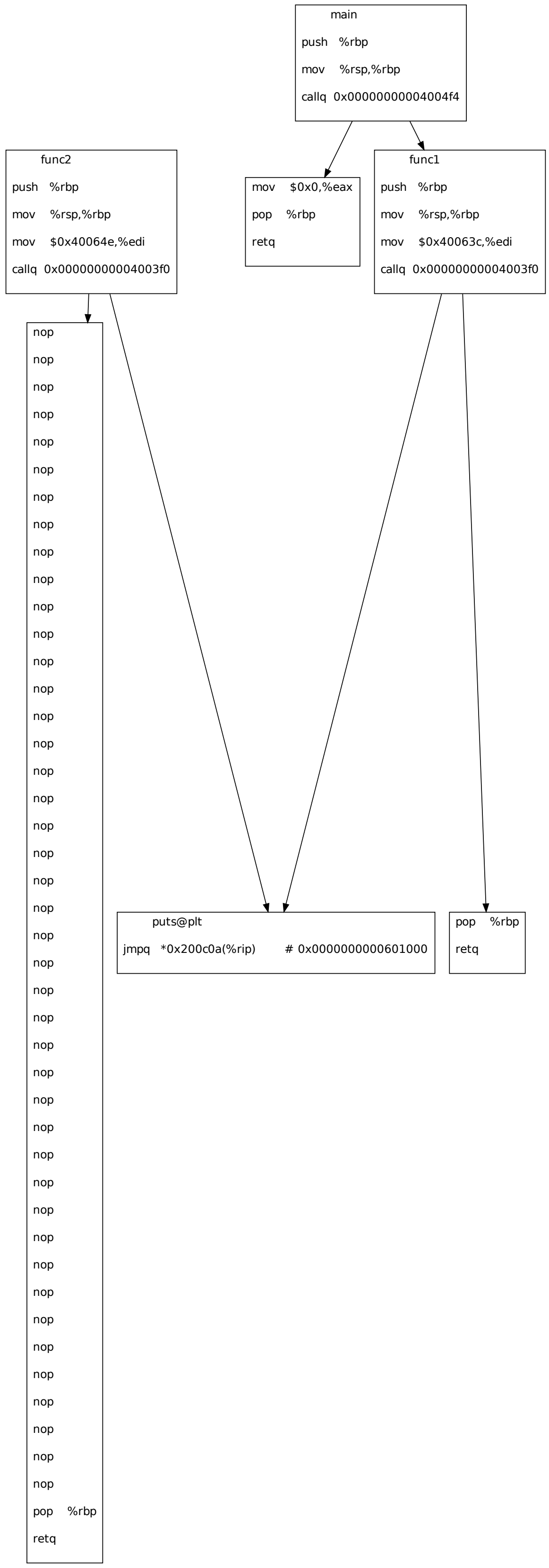


Figure 5.21: The 64-bit example CFG before trampolining.

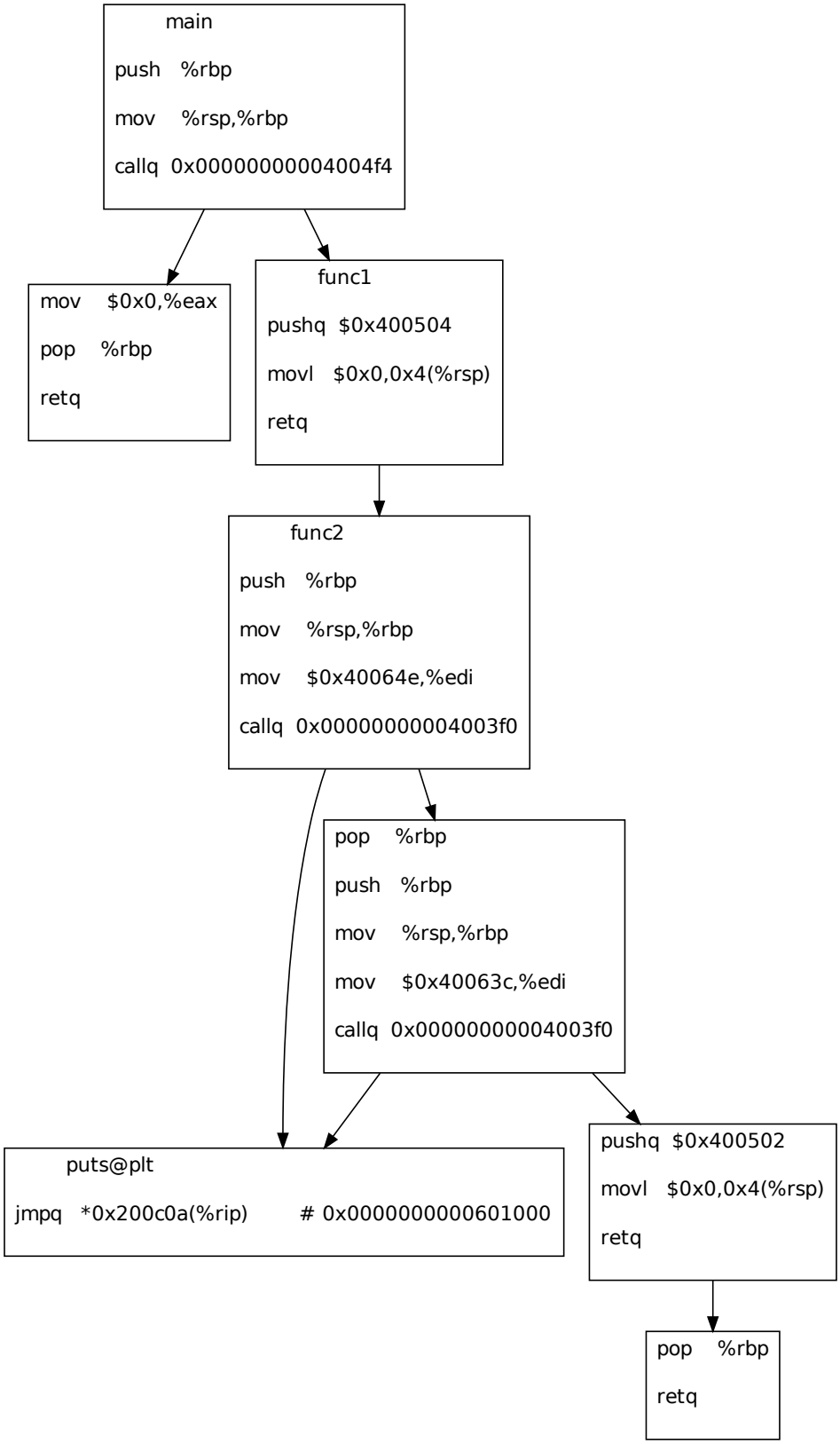


Figure 5.22: The 64-bit example CFG after trampolining.

5.5 Distribution and Documentation

Using *GNU Autotools* to create portable makefiles is so popular with C projects that it has essentially become the industry standard. Autotools takes care of automatically generating the dependency information for the project and ensures a compilation will only start if these dependencies exist. In `libbf`, the dependencies are `libelf`, `libbfd`, `libopcodes` and `libkern` (a generic collection library). Autotools allows for the automated generation of makefiles tailored to the software on the system on which `libbf` is to be compiled and installed. This means `libbf` can be downloaded and installed on any supported system as a library through the following:

```
./autogen.sh
./configure
make
make install
```

Listing 5.22: Generic shell commands to configure and install `libbf`.

Similarly, the test suite of `libbf` can be run with the shell command `make check`. `libbf` is extensively documented through *Doxygen* which is integrated to the make process. The full documentation can be generated by running `make doxygen-doc`. Two sets of documentation are provided, one for end users who are only interested in the external API and one for developers who additionally want to see documentation of all internal functions/structures. By default, internal documentation is not generated but this can be changed by simply toggling the `INTERNAL_DOCS` flag of the *Doxyfile*.

5.6 Writing Standards Compliant Code

Writing standards compliant code is important to ensure the library's functionality is robust and does not rely on undefined behaviour. `libbf` aims for compliance with C99 with GNU extensions (`-std=gnu99`). The GNU extensions are required because the library relies on several language features not found in ISO standard C such as statement expressions.

We recognise that GCC is not fully C99 compliant and in a few rare cases is not strictly conformant to the standard but in practice, `-pedantic` is an accepted way to check for standards compliance. We ensure compliance by compiling with the `-Wall -pedantic` flags and checking the warnings emitted only apply to the GNU C extensions used.

While we aim for standards compliance, there is one exception where we rely on implementation-defined behaviour. This is in our use of 64-bit wide enumeration values in the instruction decoder. In regards to the size of enumeration values, the C99 standard states:

> **6.7.2.2 Enumeration specifiers**
> [...]
> **Semantics**
> [...]
> Each enumerated type shall be compatible with **char**, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined,[a] but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until after the **}** that terminates the list of enumerator declarations.
>
> [a] An implementation may delay the choice of which integer type until all enumeration constants have been seen.

GCC allows a 64-bit enumeration type by specifying a 64-bit lvalue for one of the members

and then choosing `int` as the enumerated type. While legal, this behaviour is **implementation-defined** because the specification of C does not dictate the exact size of an integer type, only specifying the minimum size for integral types but making no guarantee on the maximum size. The implication of this is that if `libbf` is compiled on a compiler other than GCC, there are no guarantees that the instruction decoder will work as expected.

Chapter 6

Testing

As a library, it is important that `libbf` is guaranteed to be robust and reliable. In particular, we need to make sure that the library is scalable and does not fail on large scale systems. The testsuite included with `libbf` includes a set of functional tests where we try to test on real-world systems as opposed to toy examples where possible.

The `GNU Core Utilities` are the basic file, shell and text manipulation utilities of the GNU operating system. The latest version of `Coreutils` (8.17) includes 102 binaries including well-known utilities such as `ls`, `rm` and `cp`. The quantity and variety in size of the utilities make them an appropriate choice for our functional tests.

6.1 Static Analysis Engine Test

This test downloads and builds the latest version of `Coreutils`. The test then disassembles all the built binaries. The purpose of this test is to ensure that the disassembler engine has enumerated all instructions and that the CFG analysis is robust enough to properly handle large complex programs. The test is timed and allows us to guage whether any changes to the static analysis engine will cause a drastic effect on the disassembly time.

As an example of the approximate speed of the disassembler engine, when the disassembler engine was first built, it spent 7 seconds to disassemble all the `Coreutils` binaries. At this point, the instructions were all stored in string form. After the instruction decoder was added the time was measured again and the overhead was found to be only 0.5s.

6.2 Instruction Decoder Test

This test builds and disassembles `Coreutils` as with the static analysis engine test but has a few extra steps. Firstly, it keeps hold of all the instructions in their direct string form as received from `libopcodes` before they are decoded and dumps them in one file. The test then uses the instruction decoder to perform a backward mapping (semantic information to string form) of the instructions stored in the CFG and dumps these strings in another file. The two files are then differenced to check for discrepancies.

The purpose of this test is to ensure that the instruction decoder performs a *lossless* conversion for instructions which pass through its forward stage then back through its backward stage.

6.3 Static Patching (Detouring)

```
void func1(void)
{
  puts("func1 was invoked");
}

/*
 * func2 is not invoked by the regular execution of detour_test. The aim is to
 * detour execution to this function.
 */
void func2(void)
{
  puts("func2 was invoked");
  TRAMPOLINE_BLOCK
}

int main(void)
{
  func1();
  return EXIT_SUCCESS;
}
```

Listing 6.1: Detour/trampoline test target

This test asserts basic detouring works. The test first runs the original binary and asserts that "`func1 was invoked`" is printed. The test then attempts to detour the execution of `func1` to `func2`. Finally the modified binary is run and the test asserts that "`func2 was invoked`" is printed. For this test, `TRAMPOLINE_BLOCK` is redundant.

6.4 Static Patching (Trampolining)

This test asserts basic trampolining works. The test shares the same test target as the previous test. The test first asserts that "`func1 was invoked`" is printed. The test then attempts to instrument `func1` with `func2`. The test runs the modified binary and asserts the trampoline was inserted by checking that "`func2 was invoked`" is first printed, followed by "`func1 was invoked`".

6.5 Summary

In all tests, a set of x86-32 binaries and another set of x86-64 binaries are generated. Hence there are 8 functional tests in total. These tests also act as regression tests and code is only pushed to the master repository if it passes all tests. The tests essentially assert the various fundamental functionalities of the library.

Chapter 7

Evaluation

One of the easiest ways to evaluate the success of the project is to show that we achieved each of the functional requirements in turn. In order to do this, we shall first demonstrate the library on the example given in the Introduction before moving onto larger real-world targets.

7.1 Example 1: Target System from Introduction

Recall the target system shown in Introduction:

```
#include <stdlib.h>
#include <time.h>

int func1(int num)
{
  return 10 / num;
}

int main(void)
{
  srand(time(NULL));

  for(int i = 0; i < 5; i++) {
    func1(rand() % 5);
  }

  return EXIT_SUCCESS;
}
```

Listing 7.1: Target System

We aim to demonstrate `[bb_det]` and `[bb_tra]` by showing it is possible to detour and trampoline `func1`.

7.1.1 Showing `[bb_det]`

We use detouring to replace `func1` with a version which fixes the problem. The original problem is that if `func1` is invoked with `num` as 0, a divide-by-zero is performed leading to an exception. One solution is to check the input value and only perform the divide if the value is non-zero as shown in Listing 7.2. The code required to perform the detour is shown in Listing 7.3.

```
int new_func1(int num)
{
  return num == 0 ? num : 10 / num;
}
```

Listing 7.2: Detour function which replaces `func1`.

```
struct bin_file * bf = load_bin_file("rand", NULL);
struct bf_func * src_func, * dest_func;

disasm_all_func_sym(bf);
src_func = bf_get_func_from_name(bf, "func1");
dest_func = bf_get_func_from_name(bf, "new_func1");

bf_detour_func(bf, src_func, dest_func);
close_bin_file(bf);
```

Listing 7.3: The code required to use `libbind` to detour `func1` to `new_func1`.

In the patched system where the detour has been performed, a divide-by-zero exception never occurs and nothing is outputted. It is unhelpful that `new_func1` does not output any information but we will see how to solve this later.

7.1.2 Showing `[bb_tra]`

We use trampolining to extend the semantics of the target system in order to trace all calls to `func1` and to print out the argument passed in. The trampoline function is shown in Listing 7.4 and the code required to perform the trampoline is shown in Listing 7.5.

```
void log_func1(int num)
{
        printf("func1 was invoked with %d\n", num);
        TRAMPOLINE_BLOCK
}
```

Listing 7.4: Trampoline function which is invoked before `func1`.

```
struct bin_file * bf = load_bin_file("rand", NULL);
struct bf_func * src_func, * dest_func;

disasm_all_func_sym(bf);
src_func = bf_get_func_from_name(bf, "func1");
dest_func = bf_get_func_from_name(bf, "log_func1");

bf_trampoline_func(bf, src_func, dest_func);
close_bin_file(bf);
```

Listing 7.5: The code required to use `libbind` to trampoline `func1` to `log_func1`.

An example run of the instrumented binary is shown in Listing 7.6. The run shows

```
~$ ./rand
func1 was invoked with 1
func1 was invoked with 2
func1 was invoked with 0
Floating point exception (core dumped)
```

Listing 7.6: An example run after `func1` has been trampolining to `log_func1`.

7.1.3 Showing [bb_det] and [bb_tra]

As noted earlier, there is a problem with our detour because it seems to fix the bug but we are not sure because new_func1 does not generate any output. libbind is designed to solve exactly these types of problems. By combining both of the examples above, we can find out more about what happens. First, we perform the detour from func1 to new_func1 and then instrument new_func1 by trampolining log_func1. The code required for this is shown in Listing 7.7.

```
struct bin_file * bf = load_bin_file("rand", NULL);
struct bf_func * src_func1, * new_func1, * log_func1;

disasm_all_func_sym(bf);
src_func1 = bf_get_func_from_name(bf, "func1");
new_func1 = bf_get_func_from_name(bf, "new_func1");
log_func1 = bf_get_func_from_name(bf, "log_func");

bf_detour_func(bf, src_func1, new_func1);
bf_trampoline_func(bf, new_func1, log_func1);

close_bin_file(bf);
```

Listing 7.7: The code required to use libbind to fix func1 and to instrument logging on the replacement, new_func1.

7.2 Example 2: GNU Core Utilities

In Introduction, it was stated that a successful [cfg_gen] should expose an interface to directly access to generated CFG. Furthermore, such access should be powerful enough to allow people to build an analysis tool on top of the library which compares binaries by examining and comparing their respective CFGs.

7.2.1 Showing [cfg_gen]

The ability to perform static analysis to detect changes in two binaries is not part of the libbf itself nor should it be. Such a specific case of static analysis is not the main focus of the library and adding such functionality would detract from the aim for a lightweight and dedicated library. However, the library should provide an interface which allows this analysis to be performed and we demonstrate this by using the static analysis engine component to analyse and produce statistics of how Coreutils changed over time starting from version 8.1 up to 8.17. We created a project built on top of libbf which uses the following algorithm to generate the data to be analysed:

1. **Build Coreutils binaries** - A shell script fetches the sources for all versions from 8.1 to 8.17 and compiles a x86-32 and x86-64 builds for each version.[1]

2. **Compare CFGs for subsequent versions** - For each binary, CFG comparison is done with the corresponding binary in the subsequent version. For example, ls (v8.1) will be compared to ls (v8.2), which in turn will be compared to ls (v8.3)...

[1]Some versions of Coreutils have a broken build or perhaps one which requires fiddling to work. We ignore these cases and just allow the builds to fail. Out of the 17 versions, 13 versions build successfully and 4 versions fail.

This generates the cumulative changes in the CFGs through the history of the utility. CFG comparison then compares all function points which we expect to be the same. In the general case, we expect two functions to be equivalent if they have the same name in the symbol information. The CFG comparison algorithm is detailed below.

The CFG comparison algorithm is used to extract change information between two binaries. The aims are to find how many functions were added, modified or deleted. An added function is identified if symbol information appears for a function name which was not present in the previous version. Similarly, a function is identified as removed if the symbol information exists for a function name in one version and does not appear in the next. The algorithm for detecting modifications is fairly primitive but serves its purpose well as a proof of concept. The algorithm accepts two basic blocks as input and performs a depth-first recursively comparison of the CFGs rooted from these basic blocks and returns either `true` or `false` to indicate equality.

1. **Basecase 1** - If both basic blocks are `null`, return `true`. Assuming the initial input is not `null`, this case is reached only if the depth-first traversal of both trees reaches a terminating node at the same time (either the function returned or an indirect branch/call occurred and the CFG analysis was halted).

2. **Basecase 2** - If one basic block is `null`, but the other is not, return `false`. This occurs if one CFG has more links than the CFG it is being compared to. This might happen if one CFG has a unconditional branch and the other CFG has a conditional branch at the same place.

3. **Basecase 3** - If both basic blocks have already been compared against each other, return `true`. This prevents infinite comparisons when a CFG loops over itself. A `while` loop would translate to a looping structure in the CFG.

4. **Recursive case** - Compare each instruction in the two basic blocks. If there is a difference in the order or number of instructions, return `false`. If the basic block comparison evaluates to `true`, recurse down the CFG. The recursive step is to compare the left link (if any) of both CFGs and compare the right link (if any) of both CFGs.

As a partial demonstration of `[api_expr]`, we show the implementation of this algorithm in Listing 7.8.

It should be noted that the instruction comparison does not compare operands. An instruction comparison will evaluate to `true` if both instructions have the same mnemonic. This allows us to treat relocated code the same even though relative branch offsets might be different. It also means the same code which refers to relocated data is treated as equivalent.

Applying the comparison algorithm on the x86-32 and x86-64 builds generates the graphs shown in Figure 7.1 and 7.2 respectively.

```
/*
 * Recursive CFG comparison.
 */
extern bool bb_cmp(struct bb_cmp_info * info, struct bf_basic_blk * bb,
    struct bf_basic_blk * bb2)
{
  /*
   * Both NULL.
   */
  if(bb == NULL && bb2 == NULL) {
    return TRUE;
  /*
   * Branch in one but not the other.
   */
  } else if(bb == NULL || bb2 == NULL) {
    return FALSE;
  /*
   * Already visited.
   */
  } else if(has_visited_bb(info, bb, bb2)) {
    return TRUE;
  } else {
    unsigned int length = bf_get_bb_length(bb);

    /*
     * Different num instructions.
     */
    if(bf_get_bb_length(bb2) != length) {
      return FALSE;
    }

    /*
     * Check each instruction mnemonic.
     */
    for(int i = 0; i < length; i++) {
      if(bf_get_bb_insn(bb, i)->mnemonic !=
          bf_get_bb_insn(bb2, i)->mnemonic) {
        return FALSE;
      }
    }

    /*
     * Update visited bbs and compare the next bbs in the CFG.
     */
    add_visited_bb(info, bb, bb2);
    return bb_cmp(info, bb->target, bb2->target) &&
       bb_cmp(info, bb->target2, bb2->target2);
  }
}
```

Listing 7.8: The implementation of the recursive CFG comparison algorithm.

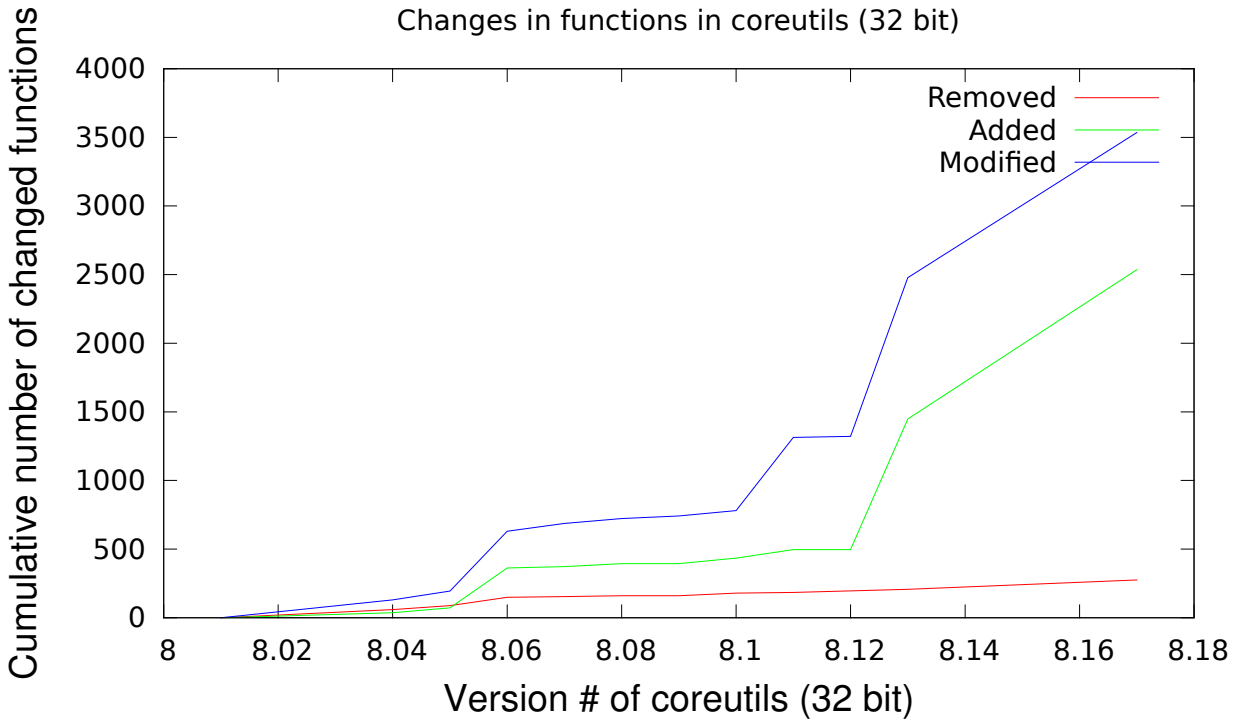


Figure 7.1: Graph of cumulative amounts of added, modified and removed functions across `Coreutils` version 8.x x86-32

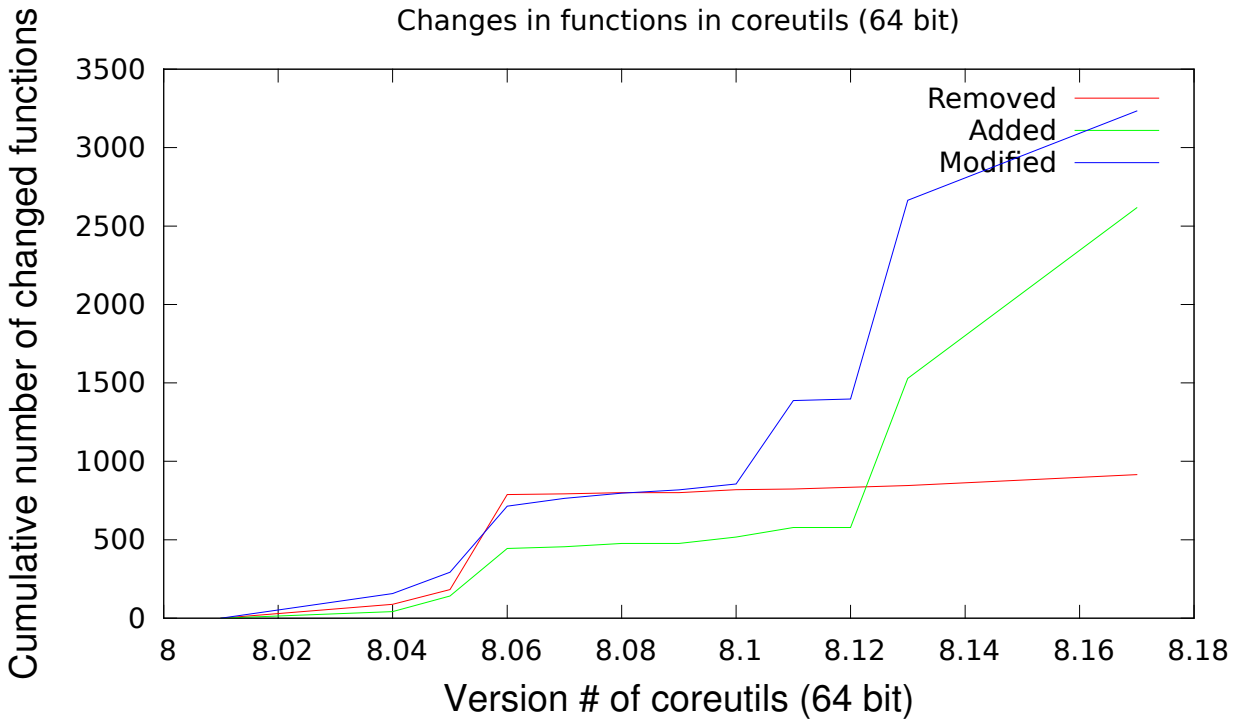


Figure 7.2: Graph of cumulative amounts of added, modified and removed functions across `Coreutils` version 8.x x86-64

An interesting observation is that the graphs are close to identical except for the fact that between v8.05 and 8.06, a lot of functions were removed from `Coreutils` x86-64, but not from x86-32. In order to make this observation meaningful, it is important to check whether one specific utility changed a lot and skewed the results. To do this, we generate a graph showing how much each individual utility contributed to the sharp increase between v8.05 and 8.06. Two such graphs are generated and shown in Figure 7.3 and 7.4 for x86-32 and x86-64 respectively. Similar graphs were also generated for modified and new functions but the results were uninteresting.

The generated graphs are difficult to read in detail because of the amount of content represented. However, there are two important observations to be made:

Scale The scale of changes is vastly different between the two graphs. The maximum amount of functions deleted under x86-32 between any two versions was less than 70. Individually, no single utility has more than 20 functions deleted in a single update. On the other hand, under x86-64, even when excluding the outlying statistic, there are three cases where over 70 functions were deleted between two versions. For some reason, the source code level deletion of functions is either more common in x86-64 or how the code generation works makes it appear so.

Anomalous results It can be seen very clearly that a large number of functions were deleted between v8.05 and 8.06 in `Coreutils` x86-64. In fact, it is not just one utility that skewed the result, but 5 different ones.

By applying a filter for more than 75 deletions, we found that the anomalous utilities were the following:

Utility Name	Number of deleted functions
`csplit`	110
`expr`	110
`nl`	110
`ptx`	106
`tax`	110

We could analyse the reason behind these anomalies if we wanted. For example, we could browse the commit history for the 5 anomalous utilities and find the reason there were so many function deletions at that point. However, this is serves no purpose to the evaluation of `libbf`. The key concept to take away from this is that the static analysis engine in `libbf` is able to robustly handle real-world examples of binaries of non-trivial sizes and complexity. It is able to produce useful analytics for such binaries even when using very basic heuristics. The static analysis engine is scalable and provides enough information to generate statistics to draw conclusions from. This concludes our evaluation of `[cfg_gen]`.

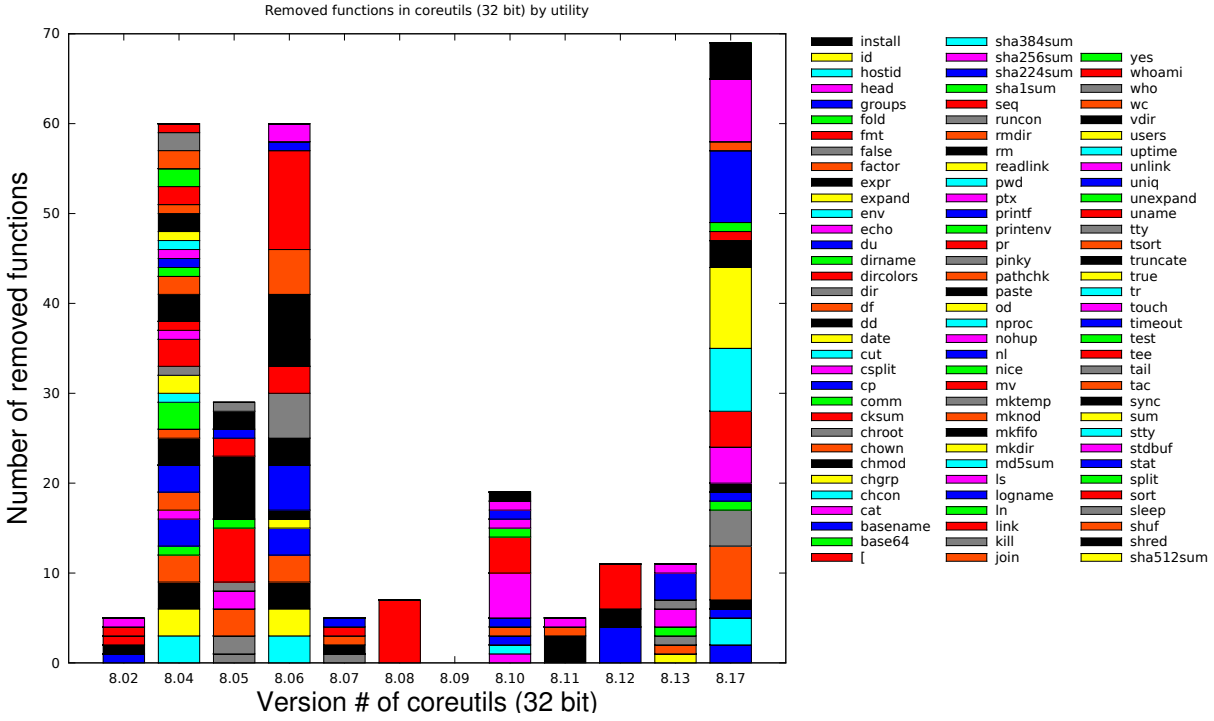


Figure 7.3: Graph of number of removed functions by utility across `Coreutils` version 8.x x86-32

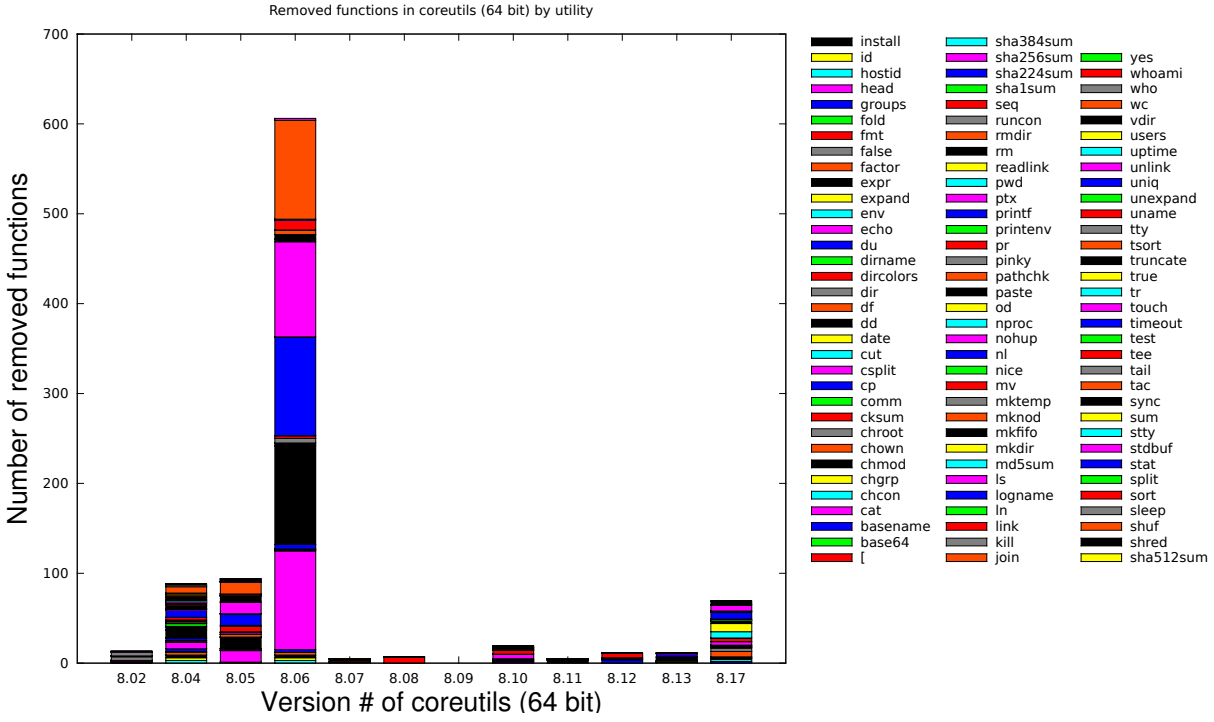


Figure 7.4: Graph of number of removed functions by utility across `Coreutils` version 8.x x86-64

7.3 Example 3: `du`

It has been re-iterated several times during this project that we wish to test on real-world examples as much as possible. Hence, we reproduce the tests we performed in *Example 1: Target System from Introduction* on a `Coreutils` utility called `du`. This utility is used to estimate file space usage and its simplest usage is `du FILE` where `FILE` can be a file or a directory, in which case the estimation is recursively applied.

7.3.1 Showing Overhead

When `du` is called, its default behaviour is to print a line for every file it checks the size of. For example, a typical output from `du` is shown in Listing 7.9.

```
$ du libexec
20 libexec/coreutils
24 libexec
```

Listing 7.9: A typical output produced from invoking `du`.

The example output shows `du` being invoked on a directory called `libexec`. Each time the utility recurses, it prints a new line. Internally, the printing mechanism is implemented in a function called `print_size` which is shown in Listing 7.10.

```
static void
print_size (const struct duinfo *pdui, const char *string)
{
  print_only_size (pdui->size);
  if (opt_time)
    {
      putchar ('\t');
      show_date (time_format, pdui->tmax);
    }
  printf ("\t%s%c", string, opt_nul_terminate_output ? '\0' : '\n');
  fflush (stdout);
}
```

Listing 7.10: The internal `print_size` function which is invoked for each line of output.

In this example, we instrument the `print_size` function. However, we have already seen the results of trampolining and detouring from earlier so it is redundant to go over this in further detail. Instead, we can use the trampoline to evaluate the overhead of instrumentation from `libbf`. In order to get more meaningful results, we should try to ensure the instrumented function is invoked as many times as possible. We do this by invoking `du` on the root directory `/` which recurses over the entire file system. The results of various stages of the instrumentation process are shown in the table below.

Stage	Time	Elapsed
`du` (Original)	Real	4.326s
	User	0.528s
	Sys	1.432s
	Total	**6.286s**
`du2` (After injection stage)	Real	4.353s
	User	0.464s
	Sys	1.500s
	Total	**6.317**
`du3` (After trampoline)	Real	4.351s
	User	0.514s
	Sys	1.462s
	Total	**6.327s**

We have only shown the results for the x86-64 version of `du` because the results from the x86-32 version follow the same pattern. In our tests, invoking `du` on the root directory causes `print_size` (and hence the trampoline) to be invoked over 40,000 times. Profiling or benchmarking any usermode binary has the problem that the current system load will affect the timings. We try to avoid taking a misrepresented timing by running each test 5 times and taking an average.

The behaviour we would intuitively expect is that the original executable is the fastest. After injection, the size of the executable is slightly larger which may affect the time it takes the system to load it into memory. The extra size also has implications on spatial locality and caching. For example, an extra cache cost would be incurred if the extra code is injected between two functions which constantly call each other which would normally be part of the same page. Finally, we would expect the binary to be the slowest after trampolining simply because more code needs to be executed.

The results obtained show a 0.4% overhead from `du` to `du2` and a 0.6% overhead from `du` to `du3`. Despite the fact that these timings have the same relationship in terms of timings as we expected, the percentage difference is so small that these results do not really illustrate a concrete pattern at all. The only thing that we can say is that there is not a drastic performance hit and even this is not certain.

Although this example does not tell us much about the overhead, it does demonstrate trampolining on a real-world application. Given more time, it would be ideal to find a better indication of the true overhead by testing on an application which invokes the trampoline more.

7.4 Example 4: Showcasing the Workflow `[api_expr]`

The expressibility of the API is a difficult requirement to evaluate quantitavely. The elegance and simplicity of an API is quite a subjective matter, but we can give examples of the workflow with the library. We have already seen several examples of the API but in this section we will try to formalise a typical workflow with `libbf`. The intention of this section is not to document the entire API, but to give a general idea about the level of abstraction provided by the library.

7.4.1 Opening a Binary File

The first step is always to show `libbf` the location of the target binary.

```
struct bin_file * bf = load_bin_file("input", "output");
```

Listing 7.11: Selecting a binary file.

The first parameter is the location of the target binary and the second parameter optionally specifies an output location. If an output location is specified, a copy of the target is created in the output location and all binary rewriting actions are redirected to this file. Otherwise, the input binary is edited directly.

Behind the scenes, this function also initialises various internal components of the library.

7.4.2 Generating the CFG

The next step is to generate the CFG. One option that was considered was to attempt to generate the entire CFG as greedily as possible as soon as the binary is loaded. The downside is that on very large binaries, this step could potentially take a while to complete. In these cases, the user would want to select his own roots for disassembly. The API tries to cater for both cases by providing a function to invoke the greedy disassembly in a single step.

```
void disasm_all_func_sym(struct bin_file * bf);
```

Listing 7.12: Greedy disassembly.

The `bin_file` backend keeps track of all previously analysed instructions which means there is no need to generate a CFG from the same root more than once. For manual CFG generation, the API allows the user to either disassemble from the file entry or from a symbol.

```
/*
 * Both functions return a bf_basic_blk object associated with the root of the
 * generate CFG. This is provided as a convenience. The user can choose to
 * ignore the return value completely and re-extract it later.
 */
struct bf_basic_blk * disasm_bin_file_entry(struct bin_file * bf);

struct bf_basic_blk * disasm_bin_file_sym(struct bin_file * bf,
    struct bf_sym * sym, bool is_func);
```

Listing 7.13: Selective/manual disassembly.

The simplest way to obtain a `bf_sym` object is simply by searching its name. The symbol table can be manually traversed with iterator macros:

```
for_each_symbol(struct bf_sym * sym, struct bin_file * bf) {
  if(strcmp(sym->name), "..." == 0) {
    ...
  }
}
```

Listing 7.14: Iterator macro to traverse the symbol table.

Alternatively, the library wraps such actions. This is a consistent property of the interface of `libbf`. There are usually several ways to achieve the same result. By writing many examples for functional tests and the evaluation examples, we were able to identify such common actions and provide wrappers for them.

```
struct bf_sym * sym = symbol_find(bf, "...");
```

Listing 7.15: Wrapper for fetching a symbol from the symbol table by its name.

7.4.3 Detouring and Trampolining

Since detouring and trampolining is the main focus of the library, we needed to encompass all possible capabilities of the library. However, we also wanted to provide an API which was narrow enough to allow a user to express these actions in a very simple way.

```
bool bf_detour_func(struct bin_file * bf, struct bf_func * src_func,
    struct bf_func * dest_func);
bool bf_detour_basic_blk(struct bin_file * bf, struct bf_basic_blk * src_bb,
    struct bf_basic_blk * dest_bb);
bool bf_trampoline_func(struct bin_file * bf,
    struct bf_func * src_func, struct bf_func * dest_func);
bool bf_trampoline_basic_blk(struct bin_file * bf,
    struct bf_basic_blk * src_bb, struct bf_basic_blk * dest_bb);
```

Listing 7.16: Detouring and trampolining functions.

We consider what has been covered to be the typical workflow with the library. The user is able to delve into lower levels (`bf_sym`) by accessing structure members of the high level abstraction objects (`bf_basic_blk`, `bf_func`).

The many examples shown demonstrate that `libbf` can be used standalone, both in terms of containing a complete and contained set of functionality and not having to depend on a particular external or runtime environment. As such, it adequately satisfies **`[standalone]`**.

7.5 Limitations

The original problem we set out to solve does not have a black-and-white solution and while we solved many of the problems we originally set out to tackle, there are also limitations to `libbf`.

7.5.1 Precision of CFG Analysis

`libbf` provides limited support for indirect branches and calls. Some of the more established tools discussed in Related Work put a heavy emphasis on indirect branch analysis. Indirect branch analysis is very important in order to achieve significant coverage in binaries with stripped symbol information but fuzziness of implementations means it would not have been a constructive use of time to focus too much effort in this arena. Instead, we recognise this drawback and focus on ensuring the static analysis engine is amenable to be extended to include more indirect branch analysis. We demonstrate this by adding a single example of indirect branch analysis to `libbf`.

7.5.2 Completeness

In some areas, `libbf` aims to provide functionality that is closer to proof of concept than a refined solution. For example, during epilogue relocation, individual instructions are relocated in different ways. Some instructions can simply be copied byte-for-byte but others have to be decoded and have relative offsets updated as part of the instruction encoding. `libbf` provides code to relocate a few of the exception cases but does not handle every case. Adding such support is technically trivial now that the framework is in place but involves a tedious process of identifying the instructions and then looking up documentation to understand their encoding.

7.5.3 Handling Packed Executables

Packers compress the executable file to allow developers to distribute smaller versions of binaries [13]. At the entry point, a small packer stub unpacks/decompresses the binary at runtime then bootstraps the execution of the decompressed code. Binary rewriters in general are not able to handle packed executables because any code representations that are statically generated (such as CFGs) will only refer to the packed version of the executable. This is a general drawback of executable editing as opposed to dynamic analysis and instrumentation. The best way to approach this limitation would be to add a dynamic component alongside the

static analysis engine of `libbf`. Using a hybrid solution would also allow for higher precision of CFG analysis.

7.5.4 Summary

In general, the design decisions of the project have helped provide solutions to the functional requirements. There are certain aspects, that had we known beforehand we would have designed differently. Most notably, it was previously unknown how cumbersome it would be to manipulate binary files with `libbfd`. However, this could not have been avoided and ultimately formed part of the development and learning process, which helped guide us to the final solution. Overall, we finish with a stable and highly maintainable codebase which can be extended in many ways.

Chapter 8

Conclusion

Writing `libbf` has provided the opportunity to develop a large and consistent software solution employing good engineering practices. When dealing with low level manipulation at the file or assembly language level, much of the challenge lies in fiddly implementation details. Particularly when it comes to the x86 architecture, there is certainly no shortage of exceptional corner cases. However, being able to abstract these concerns completely by providing a clean and simple API has proved to be very rewarding.

The challenge of binary rewriting seemed daunting at first but ultimately, it seems to have been an appropriate design choice for this project. `libbf` has proved itself on large, real-world systems and the payoff is the low overhead inherent to executable editing.

Our solution fills a gap that is cluttered with outdated and cumbersome products. This project has demonstrated that creating a lightweight and elegant binary rewriter is a feasible task and it has genuine benefits. `libbf` presents an elegant solution to a specific problem by borrowing the best aspects from various projects and combining them with our own novel ideas and investigations.

8.1 Future Work

Although we achieved the functional requirements defined at the start of this project, there are many ways in which `libbf` can be extended. In fact, the code was built with this in mind and this can clearly be seen from a software engineering perspective. Where possible, we always tried not to force ourselves into a position where we were locked down to any particular library, abstraction or even architecture. The plugin system has proved its value when the binary abstraction layer was re-implemented from scratch. `libbf`. Given more time, these are some of the extensions and additional work I would have liked to add.

Hybrid analysis techniques

Combining static techniques with dynamic techniques can create an extremely powerful solution as has been proved in the past. Not only would this allow more precise and broader CFG analysis but in terms of instrumentation, each technique has its own limits. Combining the two methods deals with the various trade-offs from both sides. Such an endeavour requires a very significant effort and time commitment to be achieved but would greatly increase the breadth of application of `libbf`.

Scriptable interface

It is often the case that low level manipulation and assembly language are viewed sceptically as unnecessarily dangerous tools. Providing a proper abstraction and interface could well lead to wider adoption of such techniques. One way might be to provide a scriptable interface to attract a wider audience. Being able to achieve a higher level of abstraction and simplicity without compromising functionality would be extremely valuable.

Bibliography

[1] The ELF shell. `http://www.eresi-project.org/wiki/TheELFsh`.

[2] IDA Pro. `http://www.hex-rays.com/products/ida/index.shtml`.

[3] radare2. `http://radare.org/y/?p=examples&f=itrace`.

[4] Matt Bishop. Profiling under unix by patching. *Softw. Pract. Exper.*, 17:729–739, October 1987.

[5] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[7] Steve Chamberlain. *libbfd: The Binary File Descriptor Library*. Cygnus Support, Free Software Foundation, Inc., 1st edition, April 1991.

[8] Timothy Garnett. Dynamic optimization of ia-32 applications under dynamorio.

[9] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33:63–68, December 2005.

[10] Reed Hastings and Bob Joyce. *Purify: Fast Detection of Memory Leaks and Access Errors*, pages 125–136. 1992.

[11] Galen Hunt and Doug Brubacher. Detours: binary interception of win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[12] Peter B. Kessler. Fast breakpoints: design and implementation. *SIGPLAN Not.*, 39:390–397, April 2004.

[13] Min-Jae Kim, Jin-Young Lee, Hye-Young Chang, SeongJe Cho, Yongsu Park, Minkyu Park, and Philip A. Wilsey. Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 80–86, Washington, DC, USA, 2010. IEEE Computer Society.

[14] R. Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux J.*, 2005:11–, May 2005.

[15] Thierry Lafage and André Seznec. Combining light static code annotation and instruction-set emulation for flexible and efficient on-the-fly simulation (research note). In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 178–182, London, UK, 2000. Springer-Verlag.

[16] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24:197–218, February 1994.

[17] James R. Larus and Eric Schnarr. Eel: machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 291–300, New York, NY, USA, 1995. ACM.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[19] mayhem. The Cerberus ELF Interface. *Phrack*, (61), August 2003. `http://www.phrack.org/issues.html?id=8&issue=61`.

[20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42:89–100, June 2007.

[21] Jim Pierce and Trevor N. Mudge. Idtrace - a tracing tool for i486 simulation. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS '94, pages 419–420, Washington, DC, USA, 1994. IEEE Computer Society.

[22] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[23] J. Raber and E. Laspe. Emulated breakpoint debugger and data mining using detours. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 271 –272, oct. 2007.

[24] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[25] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[26] Douglas C. Schmidt. More c++ gems. chapter GPERF: a perfect hash function generator, pages 461–491. Cambridge University Press, New York, NY, USA, 2000.

[27] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[28] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. Dynamic program analysis of microsoft windows applications. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 2 –12, march 2010.

[29] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39:528–539, April 2004.

[30] Amitabh Srivastava and David W Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, 1992.

[31] Computer Staff. Efficient program tracing. *Computer*, 26:52–61, May 1993.

[32] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 481–497, New York, NY, USA, 2006. ACM.

[33] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29:128–170, June 1997.

[34] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. 2010.

[35] Lu Xun. A linux executable editing library. Masters Thesis, 1999. available at http://reocities.com/SiliconValley/bay/3922/leel.tar.gz.