Compositional Decompilation using LLVM IR

Robin Eklind

2015-04-21

Abstract

Decompilation or reverse compilation is the process of translating low-level machine-readable code into high-level human-readable code. The problem is non-trivial due to the amount of information lost during compilation, but it can be divided into several smaller problems which may be solved independently. This report explores the feasibility of composing a decompilation pipeline from independent components, and the potential of exposing those components to the end-user. The components of the decompilation pipeline are conceptually grouped into three modules. Firstly, the front-end translates a source language (e.g. x86 assembly) into LLVM IR; a platform-independent low-level intermediate representation. Secondly, the middle-end structures the LLVM IR by identifying high-level control flow primitives (e.g. pre-test loops, 2-way conditionals). Lastly, the back-end translates the structured LLVM IR into a high-level target programming language (e.g. Go). The control flow analysis stage of the middle-end uses subgraph isomorphism search algorithms to locate control flow primitives in CFGs, both of which are described using Graphviz DOT files.

The decompilation pipeline has been proven capable of recovering nested pre-test and post-test loops (e.g. `while`, `do-while`), and 1-way and 2-way conditionals (e.g. `if`, `if-else`) from LLVM IR. Furthermore, the data-driven design of the control flow analysis stage facilitates extensions to identify new control flow primitives. There is huge potential for future development. The Go output could be made more idiomatic by extending the post-processing stage, using components such as Grind by Russ Cox which moves variable declarations closer to their usage. The language-agnostic aspects of the design will be validated by implementing components in other languages; e.g. data flow analysis in Haskell. Additional back-ends (e.g. Python output) will be implemented to verify that the general decompilation tasks (e.g. control flow analysis, data flow analysis) are handled by the middle-end.

Acknowledgements

My heartfelt gratitude goes to Janka Chlebíková for supervising this project and showing me the beauty of Theoretical Computer Science. Your joyful enthusiasm is inspiring!

I would like to dedicate this work to my grandfather Morgan Dominius, who taught me that *anything worth doing, is worth doing with care.*

Contents

This page is unintentionally left blank.

"What we call chaos is just patterns we haven't recognized. What we call random is just patterns we can't decipher."
— Chuck Palahniuk [1]

1 Introduction

A compiler is a piece of software which translates human readable high-level programming languages (e.g. C) to machine readable low-level languages (e.g. Assembly). In the usual flow of compilation, code is lowered through a set of transformations from a high-level to a low-level representation. The decompilation process (also referred to as reverse compilation [2]) moves in the opposite direction by lifting code from a low-level to a high-level representation.

Decompilation enables source code reconstruction of binary applications and libraries. Both security researchers and software engineers may benefit from decompilation as it facilitates analysis, modification and reconstruction of object code. The applications of decompilation are versatile, and may include one of the following uses:

- Analyse malware
- Recover source code
- Migrate software from legacy platforms or programming languages
- Optimise existing binary applications
- Discover and mitigate bugs and security vulnerabilities
- Verify compiler output with regards to correctness
- Analyse proprietary algorithms
- Improve interoperability with other software
- Add new features to existing software

As recognised by Edsger W. Dijkstra in his 1972 ACM Turing Lecture (an extract of which is presented in figure 1), one of the most powerful tools for solving complex problems in Computer Science is the use of abstractions and separation of concerns. This paper explores a compositional approach to decompilation which facilitates abstractions to create a decompilation pipeline of self-contained components. Since each component interacts through language-agnostic interfaces (well-defined input and output) they may be written in a variety of programming languages. Furthermore, for each component of the decompilation pipeline there may exist multiple implementations with their respective advantages and limitations. The end user (e.g. malware analyst, security researcher or reverse engineer) may select the components which solves their task most efficiently.

1.1 Project Aim and Objectives

The aim of this project is to facilitate decompilation workflows using composition of language-agnostic decompilation passes; specifically the reconstruction of high-level con-

"We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of their powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worthwhile to point out that the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."

Figure 1: An extract from the ACM Turing Lecture given by Edsger W. Dijkstra in 1972 [3].

trol structures and, as a future ambition, expressions.

To achieve this aim, the following objectives have been identified:

1. Review traditional decompilation techniques, including control flow analysis and data flow analysis.
2. Critically evaluate a set of Intermediate Representations (IRs), which describes low-, medium- and high-level language semantics, to identify one or more suitable for the decompilation pipeline.
3. Analyse the formal grammar (language specification) of the IR to verify that it is unambiguous. If the grammar is ambiguous or if no formal grammar exists, produce a formal grammar. This objective is critical for language-independence, as the IR works as a bridge between different programming languages.
4. Determine if any existing library for the IR satisfies the project requirements; and if not develop one. These requirements would include a suitable in-memory representation, and support for on-disk file storage and arbitrary manipulations (e.g. inject, delete) of the IR.
5. Design and develop components which identify the control flow patterns of high-level control structures using control flow analysis of the IR.
6. Develop tools which perform one or more decompilation passes on a given IR. The tools will be reusable by other programming language environments as their input and output is specified by a formally defined IR.
7. As a future ambition, design and develop components which perform expression propagation using data flow analysis of the IR.

1.2 Deliverables

The source code and the report of this project have been released into the public domain[1] and are made available on GitHub at `https://github.com/decomp/decomp` and `https://github.com/decomp/doc`.

The following document has been produced:

- Project report; see objective 1 and 2

[1]CC0 1.0 Universal: `https://creativecommons.org/publicdomain/zero/1.0/`

And the following system artefacts have been developed:

- Library for interacting with LLVM IR (*work in progress*); see objective 4 `https://github.com/llir/llvm`
- Control flow graph generation tool; see objective 5
- Subgraph isomorphism search algorithms and related tools; see objective 5
- Control flow recovery tool; see objective 6
- Go code generation tool (*proof of concept*); see objective 6
- Go post-processing tool; see objective 6

1.3 Disposition

This report details every stage of the project from conceptualisation to successful completion. It follows a logical structure and outlines the major stages in chronological order. A brief summary of each section is presented in the list below.

- Section 1 - **Introduction**
  Introduces the concept of decompilation and its applications, outlines the project aim and objectives, and summarises its deliverables.
- Section 2 - **Literature Review**
  Details the problem domain, reviews traditional decompilation techniques, and evaluates potential intermediate representations for the decompilation pipeline of the project.
- Section 3 - **Related Work**
  Evaluates projects for translating native code to LLVM IR, and reviews the design of modern decompilers.
- Section 4 - **Methodology**
  Surveys methodologies and best practices for software construction, and relates them to the specific problem domain.
- Section 5 - **Requirements**
  Specifies and prioritises the requirements of the project artefacts.
- Section 6 - **Design**
  Discusses the system architecture and the design of each component, motivates the choice of core algorithms and data structures, and highlights strengths and limitations of the design.
- Section 7 - **Implementation**
  Discusses language considerations, describes the implementation process, and showcases how set-backs were dealt with.
- Section 8 - **Verification**
  Describes the approaches taken to validate the correctness, performance and security of the artefacts.

- Section 9 - **Evaluation**
  Assesses the outcome of the project and evaluates the artefacts against the requirements.
- Section 10 - **Conclusion**
  Summarises the project outcomes, presents ideas for future work, reflects on personal development, and concludes with an attribution to the key idea of this project.

2 Literature Review

This section details the problem domain associated with decompilation, reviews traditional decompilation techniques, and evaluates a set of intermediate representations with regards to their suitability for decompilation purposes. To set the stage for binary analysis, a *"hello world"* executable is dissected in section 2.1.

2.1 The Anatomy of an Executable

The representation of executables, shared libraries and relocatable object code is standardised by a variety of file formats which provides encapsulation of assembly instructions and data. Two such formats are the Portable Executable (PE) file format and the Executable and Linkable Format (ELF), which are used by Windows and Linux respectively. Both of these formats partition executable code and data into sections and assign appropriate access permissions to each section, as summarised by table 1. In general, no single section has both write and execute permissions as this could compromise the security of the system.

Section name	Usage description	Access permissions
`.text`	Assembly instructions	`r-x`
`.rodata`	Read-only data	`r--`
`.data`	Data	`rw-`
`.bss`	Uninitialised data	`rw-`

Table 1: A summary of the most commonly used sections in ELF files. The `.text` section contains executable code while the `.rodata`, `.data` and `.bss` sections contains data in various forms.

To gain a better understanding of the anatomy of executables, the remainder of this section describes the structure of ELF files and presents the dissection of a simple *"hello world"* ELF executable, largely inspired by Eric Youngdale's article on *The ELF Object File Format by Dissection* [4]. Although the ELF and PE file formats differ with regards to specific details, the general principles are applicable to both formats.

In general, ELF files consist of a file header, zero or more program headers, zero or more section headers and data referred to by the program or section headers, as depicted in figure 2.

All ELF files starts with the four byte identifier `0x7F`, `'E'`, `'L'`, `'F'` which marks the beginning of the ELF file header. The ELF file header contains general information about a binary, such as its object file type (executable, relocatable or shared object), its assembly architecture (x86-64, ARM, ...), the virtual address of its entry point which indicates the starting point of program execution, and the file offsets to the program and section headers.

Each program and section header describes a continuous segment or section of memory respectively. In general, segments are used by the linker to load executables into memory

[2]Original image (CC BY-SA): `https://en.wikipedia.org/wiki/File:Elf-layout--en.svg`

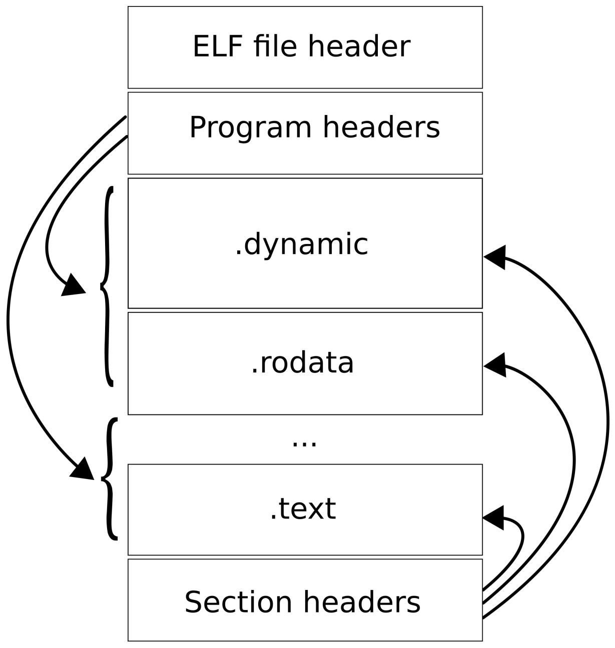


Figure 2: The basic structure of an ELF file.[2]

with correct access permissions, while sections are used by the compiler to categorise data and instructions. Therefore, the program headers are optional for relocatable and shared objects, while the section headers are optional for executables.

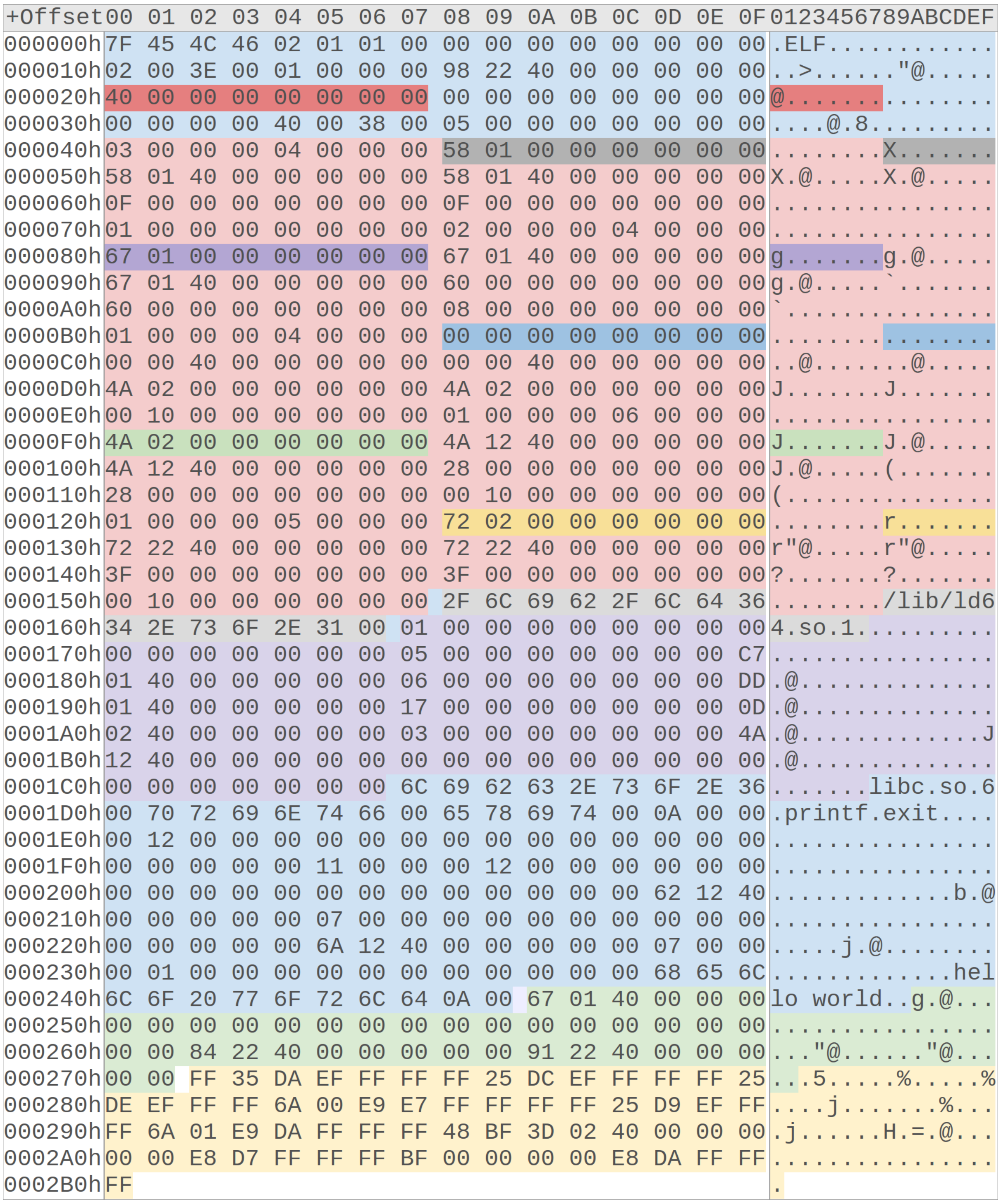
To further investigate the structure of ELF files a simple 64-bit *"hello world"* executable has been dissected and its content colour-coded. Each file offset of the executable consists of 8 bytes and is denoted in figure 3 with a darker shade of the colour used by its corresponding target segment, section or program header. Starting at the middle of the ELF file header, at offset `0x20`, is the file offset (red) to the program table (bright red). The program table contains five program headers which specify the size and file offsets of two sections and three segments, namely the `.interp` (grey) and the `.dynamic` (purple) sections, and a *read-only* (blue), a *read-write* (green) and a *read-execute* (yellow) segment.

Several sections are contained within the three segments. The *read-only* segment contains the following sections:

- `.interp`: the interpreter, i.e. the linker
- `.dynamic`: array of dynamic entities
- `.dynstr`: dynamic string table
- `.dynsym`: dynamic symbol table
- `.rela.plt`: relocation entities of the PLT
- `.rodata`: read-only data section

The *read-write* segment contains the following section:

- `.got.plt`: Global Offset Table (GOT) of the PLT (henceforth referred to as the GOT, as this executable only contains one such table)



```
+Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
000010h 02 00 3E 00 01 00 00 00 98 22 40 00 00 00 00 00 ..>......"@.....
000020h 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @...............
000030h 00 00 00 00 40 00 38 00 05 00 00 00 00 00 00 00 ....@.8.........
000040h 03 00 00 00 04 00 00 00 58 01 00 00 00 00 00 00 ........X.......
000050h 58 01 40 00 00 00 00 00 58 01 40 00 00 00 00 00 X.@.....X.@.....
000060h 0F 00 00 00 00 00 00 00 0F 00 00 00 00 00 00 00 ................
000070h 01 00 00 00 00 00 00 00 02 00 00 00 04 00 00 00 ................
000080h 67 01 00 00 00 00 00 00 67 01 40 00 00 00 00 00 g.......g.@.....
000090h 67 01 40 00 00 00 00 00 60 00 00 00 00 00 00 00 g.@.....`.......
0000A0h 60 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 `...............
0000B0h 01 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 ................
0000C0h 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 ..@.......@.....
0000D0h 4A 02 00 00 00 00 00 00 4A 02 00 00 00 00 00 00 J.......J.......
0000E0h 00 10 00 00 00 00 00 00 01 00 00 00 06 00 00 00 ................
0000F0h 4A 02 00 00 00 00 00 00 4A 12 40 00 00 00 00 00 J.......J.@.....
000100h 4A 12 40 00 00 00 00 00 28 00 00 00 00 00 00 00 J.@.....(.......
000110h 28 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 (...............
000120h 01 00 00 00 05 00 00 00 72 02 00 00 00 00 00 00 ........r.......
000130h 72 22 40 00 00 00 00 00 72 22 40 00 00 00 00 00 r"@.....r"@.....
000140h 3F 00 00 00 00 00 00 00 3F 00 00 00 00 00 00 00 ?.......?.......
000150h 00 10 00 00 00 00 00 00 2F 6C 69 62 2F 6C 64 36 ......../lib/ld6
000160h 34 2E 73 6F 2E 31 00 01 00 00 00 00 00 00 00 00 4.so.1..........
000170h 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00 C7 ................
000180h 01 40 00 00 00 00 00 06 00 00 00 00 00 00 00 DD .@..............
000190h 01 40 00 00 00 00 00 17 00 00 00 00 00 00 00 0D .@..............
0001A0h 02 40 00 00 00 00 00 03 00 00 00 00 00 00 00 4A .@.............J
0001B0h 12 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .@..............
0001C0h 00 00 00 00 00 00 00 6C 69 62 63 2E 73 6F 2E 36 .......libc.so.6
0001D0h 00 70 72 69 6E 74 66 00 65 78 69 74 00 0A 00 00 .printf.exit....
0001E0h 00 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0001F0h 00 00 00 00 00 11 00 00 00 12 00 00 00 00 00 00 ................
000200h 00 00 00 00 00 00 00 00 00 00 00 00 00 62 12 40 .............b.@
000210h 00 00 00 00 00 07 00 00 00 00 00 00 00 00 00 00 ................
000220h 00 00 00 00 00 6A 12 40 00 00 00 00 00 07 00 00 .....j.@........
000230h 00 01 00 00 00 00 00 00 00 00 00 00 00 68 65 6C .............hel
000240h 6C 6F 20 77 6F 72 6C 64 0A 00 67 01 40 00 00 00 lo world..g.@...
000250h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000260h 00 00 84 22 40 00 00 00 00 00 91 22 40 00 00 00 ..."@......"@...
000270h 00 00 FF 35 DA EF FF FF FF 25 DC EF FF FF FF 25 ...5.....%.....%
000280h DE EF FF FF 6A 00 E9 E7 FF FF FF FF 25 D9 EF FF ....j.......%...
000290h FF 6A 01 E9 DA FF FF FF 48 BF 3D 02 40 00 00 00 .j......H.=.@...
0002A0h 00 00 E8 D7 FF FF FF BF 00 00 00 00 E8 DA FF FF ................
0002B0h FF                                              .
```

Figure 3: The entire contents of a simple *"hello world"* ELF executable with colour-coded file offsets, sections, segments and program headers. Each file offset is 8 bytes in width and coloured using a darker shade of its corresponding segment, section or program header.

And the *read-execute* segment contains the following sections:

- `.plt`: Procedure Linkage Table (PLT)
- `.text`: executable code section

Seven of the nine sections contained within the executable are directly related to dynamic linking. The `.interp` section specifies the linker (in this case *"/lib/ld64.so.1"*) and the `.dynamic` section, an array of dynamic entities containing offsets and virtual addresses to relevant dynamic linking information. In this case the dynamic array specifies that *"libc.so.6"* is a required library, and contains the virtual addresses to the `.dynstr`, `.dynsym`, `.rela.plt` and `.got.plt` sections. As noted, even a simple *"hello world"* executable requires a large number of sections related to dynamic linking. Further analysis will reveal their relation to each other and describe their usage.

The dynamic string table contains the names of libraries (e.g. *"libc.so.6"*) and identifiers (e.g. *"printf"*) which are required for dynamic linking. Other sections refer to these strings using offsets into `.dynstr`. The dynamic symbol table declares an array of dynamic symbol entities, each specifying the name (e.g. offset to *"printf"* in `.dynstr`) and binding information (local or global) of a dynamic symbol. Both the `.plt` and the `.rela.plt` sections refers to these dynamic symbols using array indices. The `.rela.plt` section specifies the relocation entities of the PLT; more specifically this section informs the linker of the virtual address to the `.printf` and `.exit` entities in the GOT.

To reflect on how dynamic linking is accomplished on a Linux system, lets review the assembly instructions of the executable `.text` and `.plt` sections, as outlined in listing 1 and 2 respectively.

Listing 1: The assembly instructions of the `.text` section.

```
text:
  .start:
        mov     rdi, rodata.hello
        call    plt.printf
        mov     rdi, 0
        call    plt.exit
```

Listing 2: The assembly instructions of the `.plt` section.

```
plt:
  .resolve:
        push    [got_plt.link_map]
        jmp     [got_plt.dl_runtime_resolve]
  .printf:
        jmp     [got_plt.printf]
  .resolve_printf:
        push    dynsym.printf_idx
        jmp     .resolve
  .exit:
        jmp     [got_plt.exit]
  .resolve_exit:
        push    dynsym.exit_idx
        jmp     .resolve
```

As visualised in listing 1, the first call instruction of the `.text` section targets the `.printf` label of the `.plt` section instead of the actual address of the *printf* function in the *libc* library. The Procedure Linkage Table (PLT) provides a level of indirection between call instructions and actual function (procedure) addresses, and contains one entity per external function, as outlined in listing 2. The `.printf` entity of the PLT contains a jump instruction which targets the address stored in the `.printf` entity of the GOT. Initially this address points to the next instruction, i.e. the instruction denoted by the `.resolve_printf` label in the PLT. Upon the first invocation of *printf*, the linker replaces this address with the actual address of the *printf* function in the *libc* library. Any subsequent invocation of *printf* will target the resolved function address directly.

This method of external function resolution is called lazy dynamic linking as it postpones the work and only resolves a function once it is actually invoked at runtime. The lazy approach to dynamic linking may improve performance by limiting the number of symbols that require resolution. At the same time the eager approach may benefit latency sensitive applications which cannot afford the cost of dynamic linking at runtime.

A closer look at the instructions denoted by the `.resolve_printf` label in listing 2 reveals how the linker knows which function to resolve. Essentially the *dl_runtime_resolve* function is invoked with two arguments, namely the dynamic symbol index of the *printf* function and a pointer to a linked list of nodes, each referring to the `.dynamic` section of a shared object. Upon termination the linked list of the *"hello world"* process contains a total of four nodes, one for the executable itself and three for its dynamically loaded libraries, namely *linux-vdso.so.1*, *libc.so.6* and *ld64.so.1*.

To summarise, the execution of a dynamically linked executable can roughly be described as follows. Upon execution the kernel parses the program headers of the ELF file, maps each segment to one or more pages in memory with appropriate access permissions, and transfers the control of execution to the linker (*"/lib/ld64.so.1"*), which was loaded in a similar fashion. The linker is responsible for instantiating the addresses of the *dl_runtime_resolve* function and the aforementioned linked list, both of which are stored in the GOT of the executable. After this setup is complete the linker transfers control to the entry point of the executable, as specified by the ELF file header (in this case the `.start` label of the `.text` section). At this point the assembly instructions of the application are executed until termination and external functions are lazily resolved at runtime by the linker through invocations to the *dl_runtime_resolve* function.

2.2 Decompilation Phases

A core principle utilised in decompilers is the separation of concern through the use of abstractions, and extensive work involves translating into and breaking out of various abstraction layers. In general, a decompiler is composed of distinct phases which parse, analyse or transform the input. These phases are conceptually grouped into three modules to separate concerns regarding source machine language and target programming language. Firstly, the front-end module parses executable files and translates their platform-dependent assembly into a platform-independent intermediate representation (IR). Secondly, the middle-end module performs a set of decompilation passes to lift the IR, from a low-level to a high-level representation, by reconstructing high-level control structures and expressions. Lastly, the back-end module translates the high-level IR to a specific target programming language [2]. Figure 4 gives an overview of the decompilation modules and visualises their relationship.

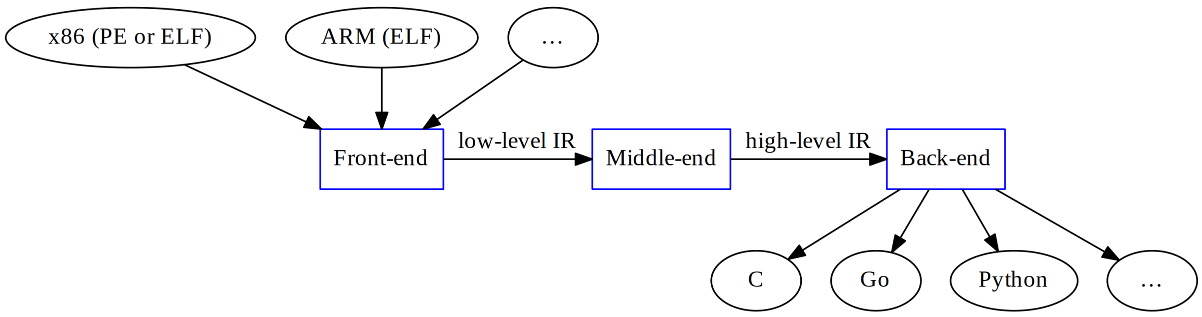


Figure 4: Firstly, the front-end module accepts several executable file formats (PE, ELF, ...) as input and translates their platform-dependent assembly (x86, ARM, ...) to a low-level IR. Secondly, the middle-end module lifts the low-level IR to a high-level IR through a set of decompilation passes. Lastly, the back-end module translates the high-level IR into one of several target programming languages (C, Go, Python, ...).

The remainder of this section describes the distinct decompilation phases, most of which have been thoroughly described by Cristina Cifuentes in her influential paper *"Reverse Compilation Techniques"* [2].

2.2.1 Binary Analysis

As demonstrated in section 2.1, parsing even a simple *"hello world"* executable requires extensive knowledge of its binary file format (in this case ELF). The binary analysis phase is responsible for parsing input files of various binary file formats, such as PE and ELF, and present their content in a uniform manner which preserves the relations between file contents, virtual addresses and access permissions. Later stages of the decompilation pipeline builds upon this abstraction to access the file contents of each segment or section without worrying about the details of the underlying file format. Information about external symbols, metadata and the computer architecture of the assembly may also be provided by this abstraction.

2.2.2 Disassembly

The disassembly phase (referred to as the *syntactic analysis phase* by C. Cifuentes) is responsible for decoding the raw machine instructions of the executable segments into assembly. The computer architecture dictates how the assembly instructions and their associated operands are encoded. Generally CISC architectures (e.g. x86) use variable length instruction encoding (e.g. instructions occupy between 1 and 15 bytes in x86) and allow memory addressing modes for most instructions (e.g. arithmetic instructions may refer to memory locations in x86) [5]. In contract, RISC architectures (e.g. ARM) generally use fixed-length instruction encoding (e.g. instructions always occupy 4 bytes in AArch64) and only allow memory access through load-store instructions (e.g. arithmetic instructions may only refer to registers or immediate values in ARM) [6].

One of the main problems of the disassembly phase is how to separate code from data. In the Von Neumann architecture the same memory unit may contain both code and data. Furthermore, the data stored in a given memory location may be interpreted as code by one part of the program, and as data by another part. In contrast, the Harvard architecture uses separate memory units for code and data [7]. Since the use of the Von Neumann architecture is wide spread, solving this problem is fundamental for successful disassemblers.

The most basic disassemblers (e.g. `objdump` and `ndisasm`) use linear descent when decoding instructions. Linear descent disassemblers decode instructions consecutively from a given entry point, and contain no logic for tracking the flow of execution. This approach may produce incorrect disassembly when code and data are intermixed (e.g. switch tables stored in executable segments) [2]; as illustrated in figure 5. More advanced disassemblers (e.g. IDA) often use recursive descent when decoding instructions, to mitigate this issue.

Recursive descent disassemblers track the flow of execution and decode instructions from a set of locations known to be reachable from a given entry point. The set of reachable locations is initially populated with the entry points of the binary (e.g. the `start` or `main` function of executables and the `DllMain` function of shared libraries). To disassemble programs, the recursive descent algorithm will recursively pop a location from the reachable set, decode its corresponding instruction, and add new reachable locations from the decoded instruction to the reachable set, until the reachable set is empty. When decoding non-branching instructions (e.g. `add`, `xor`), the immediately succeeding instruction is known to be reachable (as it will be executed after the non-branching instruction) and its location is therefore added to the reachable set. Similarly, when decoding branching instructions (e.g. `br`, `ret`), each target branch (e.g. the conditional branch and the default branch of conditional branch instructions) is known to be reachable and therefore added to the reachable set; unless the instruction has no target branches, as is the case with return instructions. This approach is applied recursively until all paths have reached an end-point, such as a return instruction, and the reachable set is empty. To prevent cycles, the reachable locations are tracked and added only once to the reachable set.

```
_start:
  mov   rdi, hello
  call  printf
  mov   rdi, 0
  call  exit
  ret
hello:
  push  qword 0x6F6C6C65  ; "hello"
  and   [rdi+0x6F], dh    ; " wo"
  jc    short 0x6D        ; "rl"
  or    al, [fs:rax]      ; "d\n\0"
```

(a) Disassembly from `objdump` and `ndisasm`[3].

```
_start:
  mov   rdi, hello
  call  printf
  mov   rdi, 0
  call  exit
  ret
hello:
  db "hello world",10,0
```

(b) Disassembly from IDA.

Figure 5: The disassembly produced by a linear descent parser (left) and a recursive descent parser (right) when analysing a simple *"hello world"* program that stores the `hello` string in the executable segment.

A limitation with recursive descent disassemblers is that they cannot track indirect branches (e.g. branch to the address stored in a register) without additional informa-

[3]The Netwide Disassembler: `http://www.nasm.us/doc/nasmdoca.html`

tion, as it is impossible to know the branch target of indirect branch instructions only by inspecting individual instructions (e.g. `jmp eax` gives no information about the value of `eax`). One solution to this problem is to utilise symbolic execution engines, which emulate the CPU and execute the instructions along each path to give information about the values stored in registers and memory locations. Using this approach, the target of indirect branch instructions may be derived from the symbolic execution engine by inspecting the values of registers and memory locations at the invocation site [8]. Symbolic execution engines are no silver bullets, and introduce a new range of problems; such as cycle accurate modelling of the CPU, idiosyncrasies related to memory caches and instruction pipelining, and potentially performance and security issues.

Malicious software often utilise anti-disassembly techniques to obstruct malware analysis. One such technique exploits the fact that recursive descent parsers follow both the conditional and the default branch of conditional branch instructions, as demonstrated in figure 6. The recursive descent parser cannot decode the target instructions of both the conditional branch (i.e. `fake+1`) and the default branch (i.e. `fake`) of the conditional branch instruction at line 3, because the conditional branch targets the middle of a `jmp` instruction which would be decoded if traversing the default branch. As both branches cannot be decoded, the recursive descent parser is forced to choose one of them; and in this case the `fake` branch was disassembled, thus disguising the potentially malicious code of the conditional branch [9].

```
 1 _start:
 2   xor  al, al
 3   jz   fake+1 ; true-branch always taken
 4 fake:
 5   db   0xE9   ; jmp instruction opcode
 6   mov  rdi, hello
 7   call printf
 8   mov  rdi, 0
 9   call exit
10   ret
11 hello:
12   db "hello world",10,0
```

(a) Original assembly.

```
 1 _start:
 2   xor  al, al
 3   jz   fake+1
 4 fake:
 5   jmp  0x029FBF4C
 6 db 0x40,0x00,0x00,0x00
 7 db 0x00,0x00,0xE8,0xCC
 8 db 0xFF,0xFF,0xFF,0xBF
 9 db 0x00,0x00,0x00,0x00
10 db 0xE8,0xD2,0xFF,0xFF
11 db 0xFF,0xC3,0x68,0x65
12 db 0x6C,0x6C,0x6F,0x20
13 db 0x77,0x6F,0x72,0x6C
14 db 0x64,0x0A,0x00
```

(b) Disassembly from IDA.

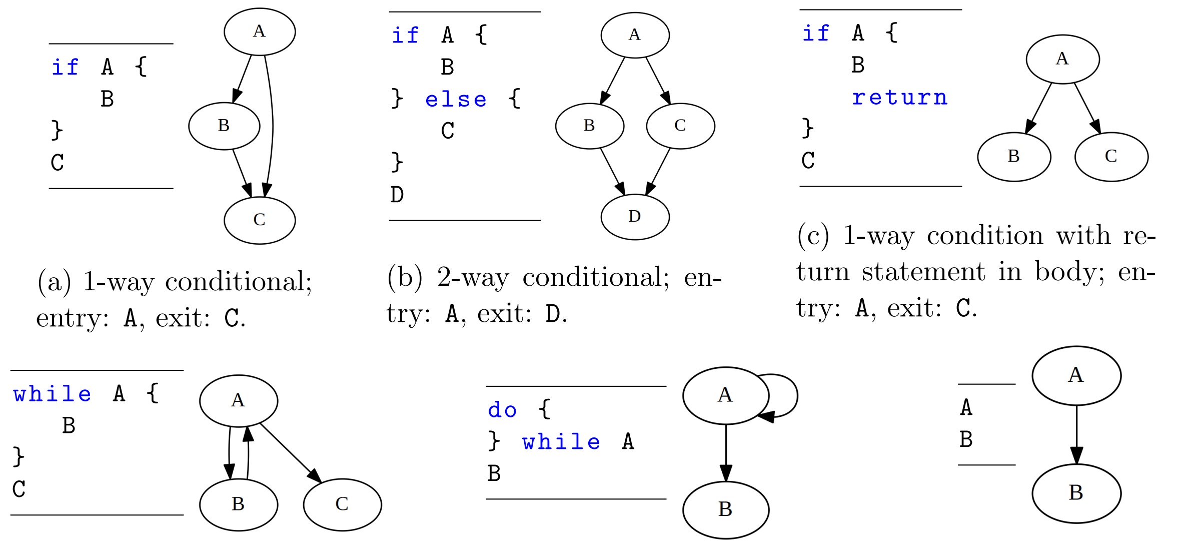
Figure 6: The original assembly (left) contains an anti-disassembly trick which causes the recursive descent parser to fail (right).

The anti-disassembly technique presented in figure 6 may be mitigated using symbolic execution. The symbolic execution engine could verify that the conditional branch instruction at line 3 always branches to the conditional branch (i.e. `fake+1`) and never to the default branch (i.e. `fake`). The conditional branch instruction may therefore be replaced with an unconditional branch instruction to `fake+1`, the target of which corresponds to the `mov` instruction at line 6. Please note that this is inherently a game of cat-and-mouse, as the anti-disassembly techniques could be extended to rely on network activity, file contents, or other external sources which would require the symbolic execution environment to be extended to handle such cases.

To conclude, the disassembly phase deals with non-trivial problems, some of which are very difficult to automate. Interactive disassemblers (such as IDA) automate what may reasonably be automated, and rely on human intuition and problem solving skills to resolve any ambiguities and instruct the disassembler on how to deal with corner cases; as further described in section 3.2.

2.2.3 Control Flow Analysis

The control flow analysis stage is responsible for analysing the control flow (i.e. flow of execution) of source programs to recover their high-level control flow structures. The control flow of a given function is determined by its branching instructions and may be expressed as a control flow graph (CFG), which is a connected graph with a single entry node (the function entry point) and zero or more exit nodes (the function return statements). A key insight provided by C. Cifuentes and S. Moll is that high-level control flow primitives (such as 1-way conditionals and pre-test loops) may be expressed using graph representations [2, 10], as illustrated in figure 7. The problem of recovering high-level control flow primitives from CFGs may therefore be reformulated as the problem of identifying subgraphs (i.e. the graph representation of a high-level control flow primitive) in graphs (i.e. the CFG of a function) without considering node names. This problem is commonly referred to as *subgraph isomorphism search*, the general problem of which is NP-hard [11]. However, the problem which is required to be solved by the control flow analysis stage may be simplified by exploiting known properties of CFGs (e.g. connected graph with a single entry node).



```
if A {
    B
}
C
```

(a) 1-way conditional; entry: A, exit: C.

```
if A {
    B
} else {
    C
}
D
```

(b) 2-way conditional; entry: A, exit: D.

```
if A {
    B
    return
}
C
```

(c) 1-way condition with return statement in body; entry: A, exit: C.

```
while A {
    B
}
C
```

(d) pre-test loop; entry: A, exit: C.

```
do {
} while A
B
```

(e) post-test loop; entry: A, exit: B.

```
A
B
```

(f) consecutive statements; entry: A, exit: B.

Figure 7: The pseudo-code and graph representation of various high-level control flow primitives with denoted entry and exit nodes.

When the subgraph isomorphism of a high-level control flow primitive has been identified in the CFG of a function, it may be replaced by a single node that inherits the predecessors of the subgraph entry node and the successors of the subgraph exit node; as illustrated in figure 8. By recording the node names of the identified subgraphs and the name of their corresponding high-level control flow primitives, the high-level control flow structure of a CFG may be recovered by successively identifying subgraph isomorphisms and replacing them with single nodes until the entire CFG has been reduced into a single node; as demonstrated by the step-by-step simplification of a CFG in appendix G. Should the control flow analysis fail to reduce a CFG into a single node, the CFG is considered irreducible with regards to the supported high-level control flow primitives (see figure 7). To structure arbitrary irreducible graphs, S. Moll applied node splitting (which translates irreducible graphs into reducible graphs by duplicating nodes) to produce functionally equivalent target programs [10]. In contrast, C. Cifuentes focused on preserving the structural semantics of the source program (which may be required in forensics investigations), and therefore used `goto`-statements in these cases to produce unstructured target programs.

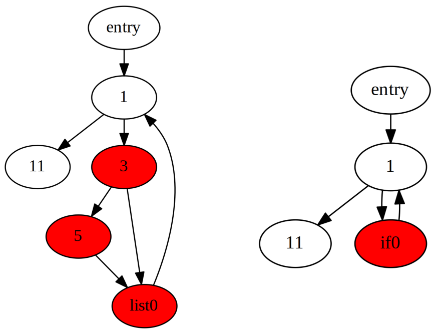


Figure 8: The left side illustrates the CFG of a function in which the graph representation of a 1-way conditional (see figure 7a) has been identified, and the right side illustrates the same CFG after the subgraph has been replaced with a single node (i.e. `if0`) that inherits the predecessors of the subgraph entry node (i.e. `3`) and the successors of the subgraph exit node (i.e. `list0`).

2.3 Evaluation of Intermediate Representations

Decompilers face similar problems as both binary analysis tools and compilers. Therefore, it seems reasonable that the intermediate representations (IRs) used in these domains may be well suited for decompilation purposes. This section evaluates one IR from each domain with regards to their suitability for recovering high-level control flow primitives (objective 5) and expressions (objective 7).

2.3.1 REIL

The Reverse Engineering Intermediate Language (REIL) is a very simple and platform-independent assembly language. The REIL instruction set contains only 17 different instructions, each with exactly three (possibly empty) operands. The first two operands are always used for input and the third for output (except for the conditional jump instruction which uses the third operand as the jump target). Furthermore, each instruction has at most one effect on the global state and never any side-effects (such as setting flags) [12, 13]. Thanks to the simplicity of REIL, a full definition of its instruction set has been provided in appendix A, which includes examples of each instruction and defines their syntax and semantics (in pseudo C-code).

When translating native assembly (e.g. x86) into REIL, the original addresses of each instruction is left shifted by 8 bits to allow 256 REIL instructions per address. Each native instruction may therefore be translated into one or more REIL instructions (at most 256), which is required to correctly map the semantics of complex instructions with side-effects. This systematic approach of deriving instruction addresses has a fundamental implication, REIL supports indirect branches (e.g. `call rax`) by design.

The language was originally designed to assist static code analysis and translators from native assembly (x86, PowerPC-32 and ARM-32) to REIL are commercially available. However, the project home page has not been updated since Google acquired zynamics in 2011. Since then approximately 10 papers have been published which references REIL and the adaptation of the language within the open source community seems limited. As of the 4th of January 2015, only three implementations existed on GitHub (two in Python[4][5] and one in C[6]), and the most popular had less than 25 watchers, 80 stars and 15 forks.

A fourth implementation was released at the 15th of March 2015 however, and in less than two weeks OpenREIL had become the most popular REIL implementation on GitHub. The OpenREIL project extends the original REIL instruction set with signed versions of the multiplication, division and modulo instructions, and includes convenience instructions for common comparison and binary operations. OpenREIL is currently capable of translating x86 executables to REIL, and aims to include support for ARM and x86-64 in the future. Furthermore, the OpenREIL project intends to implement support for translating REIL to LLVM IR, thus bridging the two intermediate representations [14].

2.3.2 LLVM IR

The LLVM compiler framework defines an intermediate representation called LLVM IR, which works as a language-agnostic and platform-independent bridge between high-level programming languages and low-level machine architectures. The majority of the optimisations of the LLVM compiler framework target LLVM IR, thus separating concerns related to the source language and target architecture [15].

There exist three isomorphic forms of LLVM IR; a human-readable assembly representation, an in-memory data structure, and an efficient binary bitcode file format. Several tools are provided by the LLVM compiler framework to convert LLVM IR between the various representations. The LLVM IR instruction set is comparable in size to the MIPS instruction set, and both use a load/store architecture [16, 17].

Function definitions in LLVM IR consist of a set of basic blocks. A basic block is a sequence of zero or more non-branching instructions (e.g. `add`), followed by a terminating instruction (i.e. a branching instruction; e.g. `br`, `ret`). The key idea behind a basic block is that if one instruction of the basic block is executed, then all instructions are

[4]Binary Analysis and RE Framework: `https://github.com/programa-stic/barf-project`

[5]REIL translation library: `https://github.com/c01db33f/pyreil`

[6]Binary introspection toolkit: `https://github.com/aoikonomopoulos/bit`

executed. This concept vastly simplifies control flow analysis as multiple instructions may be regarded as a single unit [10].

LLVM IR is represented in Static Single Assignment (SSA) form, which guarantees that every variable is assigned exactly once, and that every variable is defined before being used. These properties simplifies a range of optimisations (e.g. constant propagation, dead code elimination). For the same reasons, the Boomerang decompiler uses an IR in SSA form to simplify expression propagation [18].

In recent years other research groups have started developing decompilers [10, 19] and reverse engineering components [8] which rely on LLVM IR. There may exist an IR which is more suitable in theory, but in practice the collaboration and reuse of others' efforts made possible by the vibrant LLVM community is a strong merit in and of itself.

To conclude the evaluation, LLVM IR has been deemed suitable for the decompilation pipeline. The middle-end of the decompilation pipeline requires an IR which provides a clear separation between low-level machine architectures and high-level programming languages, and LLVM IR was designed with the same requirements in mind. Furthermore, the wide range of tools and optimisations provided by the LLVM compiler framework may facilitate decompilation workflows. The control flow analysis (see section 2.2.3) of the decompilation pipeline will benefit from the notion of basic blocks in LLVM IR. Similarly, the data flow analysis will benefit from the SSA form of LLVM IR.

3 Related Work

This section evaluates a set of open source projects which may be utilised by the front-end of the decompilation pipeline, to translate native code into LLVM IR (see section 3.1). Section 3.2 reviews the design of the de facto decompiler used in industry, to gain a better understanding of how it solves the non-trivial problems of decompilation (e.g. how to separate code from data).

3.1 Native Code to LLVM IR

There exist several open source projects for translating native code (e.g. x86, ARM) into LLVM IR. This section presents three such projects; Dagger, Fracture and MC-Semantics. The Fracture project is still in early development (e.g. recursive descent disassembler is on the roadmap), but shows a lot of promise and is currently capable of translating ARM binaries into LLVM IR [20]. The Dagger and MC-Semantics projects are reviewed in section 3.1.1 and 3.1.2, respectively.

3.1.1 Dagger

The Dagger project is a fork of the LLVM compiler framework, which extends its capabilities by implementing a set of tools and libraries for translating native code into LLVM IR. To facilitate the analysis of native code, the disassembly library of LLVM was extended to include support for recursive descent parsing (see section 2.2.2). Some of these changes have already been submitted upstream and merged back into the LLVM project. Once mature, the Dagger project aims to become a full part of the LLVM project.

The LLVM compiler framework defines a platform-independent representation of low-level machine instructions called MC-instructions (or `MCInst`), which may be used to describe the semantics of native instructions. For each supported architecture (e.g. x86-64) there exists a table (in the TableGen format) which maps the semantics of native machine instructions to MC-instructions. Similar to other project (e.g. Fracture and MC-Semantics), the Dagger project uses these tables to disassemble native code into MC-instructions as part of the decompilation process. The MC-instructions are then lazily (i.e. without optimisation) translated into LLVM IR instructions [21]. Appendix C demonstrates the decompilation of a simple Mach-o execute to LLVM IR, using using the Dagger project.

3.1.2 MC-Semantics

The MC-Semantics project may be used to decompile native code into LLVM IR. MC-Semantic conceptually consists of two components which separate concerns related to the disassembly stage (see section 2.2.2) from those of the intermediate code generation stage. Firstly, the control flow recovery component analyses binary files (e.g. ELF, PE files) and disassembles their machine instructions (e.g. x86 assembly) to produce a serialized CFG (in the Google Protocol Buffer format), which stores the basic blocks of each function and the native instructions contained within. Secondly, the instruction translation component converts the native instructions of the serialized CFG into semantically equivalent LLVM IR.

The clear separation between the two decompilation stages in MC-Semantics has enabled two independent implementations of the control flow recovery component in two different programming languages (i.e. C++ and Python), thus validating the language-agnostic aspects of its design. The C++ component is called `bin_descend` and it implements a recursive descent disassembler which translates the native code into serialized CFGs. As described in section 2.2.2, implementing a disassembler which correctly separates code from data is made difficult by a range of problems; e.g. indirect branches, intermixed code and data in executable segments, and callback functions. Interactive disassemblers (such as IDA) solve these issues by relying on human problem solving skills to resolve ambiguities and inform the disassembler. The second implementation of the control flow recovery component is an IDAPython script which produces serialized CFGs from IDA Pro [8]. The interaction between the components of the MC-Semantics project is illustrated in figure 9, and further demonstrated in appendix D.

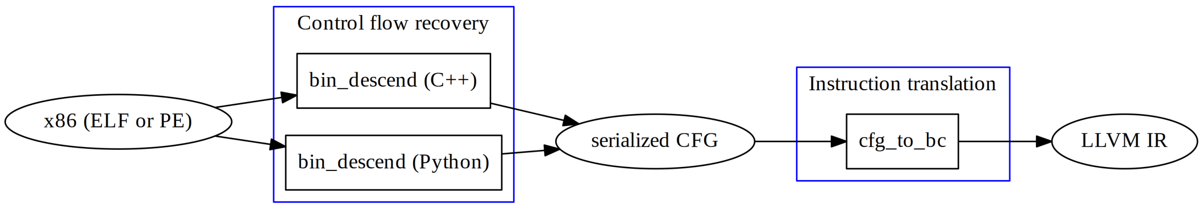


Figure 9: The MC-Semantics project is conceptually divided into two independent components. Firstly, the control flow recovery component disassembles binary files (e.g. executables and shared libraries) and stores their native instructions in serialized CFGs (in Google Protocol Buffer format). Secondly, the instruction translation component translates the native instructions of the serialized CFG into semantically equivalent LLVM IR.

3.2 Hex-Rays Decompiler

The Interactive Disassembler (IDA) and the Hex-Rays decompiler are the de facto tools used in industry for binary analysis, malware forensics and reverse engineering [22]. The interactive capabilities of IDA enables users to guide the disassembler through non-trivial problems (e.g. anti-disassembly techniques used by malware) related to the disassembly phase, some of which have been outlined in section 2.2.2. This approach turns out to be very powerful, as it is facilitated by human ingenuity and problem solving skills.

The Hex-Rays decompiler is implemented on top of IDA as a plugin, which separates concerns related to the disassembly phase from the later decompilation stages. The decompilation process of the Hex-Rays decompiler is divided into several distinct stages. Firstly, the microcode generation stage translates machine instructions into Hex-Rays Microcode, which is a RISC-like IR that is similar to REIL (see section 2.3.1). Secondly, the optimisation stage removes dead code (e.g. unused conditional flag accesses) from the unoptimised IR. Thirdly, the data flow analysis tracks the input and output registers of functions, to determine their calling conventions. Fourthly, the structural analysis stage analyses the CFGs of functions to produce a control tree containing the recovered high-level control flow primitives. The control flow recovery algorithm of the Hex-Rays decompiler handles irreducible graphs by generating `goto`-statements, which is similar to the approach taken by C. Cifuentes (see section 2.2.3). Fifthly, the pseudocode generation stage translates the IR into unpolished pseudocode (in C syntax). Sixthly, the pseudocode transformation stage improves the quality of the unpolished pseudocode by applying source code transformations; e.g. translate `while`-loops into `for`-loops by locating the initialisation and post-statements of the loop header. Lastly, the type analysis stage analyses the generated pseudocode to determine and propagate variable types, by building and solving type equations [23].

Unlike other decompilers, the type analysis stage is the last stage of the Hex-Rays decompiler. According to the lead developer of Hex-Rays, one benefit with postponing the type analysis stage (which is normally conducted in the middle-end rather than the back-end), is that more information is available to guide the type recovery and enforce rigid constraints on the type equations. A major drawback with this approach is that the type analysis has to be reimplemented for every back-end.

4 Methodology

No single methodology was used for this project, but rather a combination of software development techniques (such as test-driven development and continuous integration) which have been shown to work well in practice for other open source projects. This project has been developed in the open from day one, using public source code repositories and issue trackers. To encourage open source adaptation, the software artefacts and the project report have been released into the public domain, and are made available on GitHub; as further described in section 1.2. Throughout the course of the project a public discussion has been held with other members of the open source community to clarify the requirements and validate the design of the LLVM IR library, and to investigate inconsistent behaviours in the LLVM reference implementation; as described in section 7.2.

4.1 Operational Prototyping

The software artefacts were implemented using two distinct stages. The aim of the first stage was to get a better understanding of the problem domain, to identify suitable data structures, and to arrive at a solid approach for solving the problem. To achieve these objectives, a set of throwaway prototypes (see section 4.1.1) were iteratively implemented, discarded and redesigned until the requirements of the artefact were well understood and a mature design had emerged. The aim of the second stage was to develop a production quality software artefact based on the insights gained from the first stage. To achieve this objective, evolutionary prototyping (see section 4.1.2) was used to develop a solid foundation for the software artefact and incrementally extend its capabilities by implementing one feature at the time, starting with the features that were best understood.

This approach is very similar to the operational prototyping methodology, which was proposed by A. Davis in 1992. One important concept in operational prototyping is the notion of a quality baseline, which is implemented using evolutionary prototyping and represents a solid foundation for the software artefact. Throwaway prototypes are implemented on top of the quality baseline for poorly understood parts of the system, to gain further insight into their requirements. The throwaway prototypes are discarded once their part of the system is well-understood, at which point the well-understood parts are carefully reimplemented and incorporated into the evolutionary prototype to establish a new quality baseline [24]. In summary, throwaway prototyping is used to *identify* good solutions to problems, while evolutionary prototyping is used to *implement* identified solutions.

A major benefit with this approach is that it makes it easy to track the evolution of the design, by referring back to the throwaway prototypes which gave new insight into the problem domain; as demonstrated when tracking the evolution of the subgraph isomorphism search algorithm in section 7.4. A concrete risk with operational prototyping is that throwaway prototypes may end up in production systems, if not discarded as intended. As mentioned in section 4.1.1, the throwaway prototypes enable rapid iteration cycles by ignoring several areas of quality software (e.g. maintainability, efficiency and usability) and should therefore never end up in production systems. The use of revision control systems could help mitigate this risk, as they tracks old versions of the source code which may lower the psychological threshold for removing code (e.g. the code is not permanently removed, and may later be recovered if needed).

4.1.1 Throwaway Prototyping

Throwaway prototyping may be used in the early stages of development to gain insight into a problem domain, by rapidly implementing prototypes which will be discarded upon completion. These prototypes aim to challenge design decisions, stress test implementation strategies, identify further research requirements, and provide a better understanding and intuition for the problem domain and potential solutions. Throwaway prototypes are developed in an informal manner and are not intended to become part of the final artefact. This allows rapid iterations, as several areas of quality software (e.g. maintainability, efficiency and usability) may be ignored. When utilised appropriately, throwaway prototyping makes the development very time effective as costly changes are applied early on [24].

4.1.2 Evolutionary Prototyping

Evolutionary prototyping focuses on implementing the parts of the system which are well understood, as acknowledged by the quote from A. Davis presented in figure 10. This is in direct contrast to throwaway prototyping (see section 4.1.1), which aims to provide insight into the requirements of the poorly understood parts of the system. From the initial implementation, evolutionary prototypes are built as robust systems which evolve over time. The evolutionary prototypes may lack functionality, but the functionality they implement is generally of high enough quality to be used in production systems [24].

> *"... evolutionary prototyping acknowledges that we do not understand all the requirements and builds only those that are well understood."*

Figure 10: An extract from *Operational prototyping: A new development approach* by A. Davis in 1992 [24].

4.2 Continuous Integration

The Continuous Integration (CI) practice originated from the Extreme Programming methodology [25] but has reached a much broader audience in recent years. Today most large scale software projects rely on CI server farms to continuously compile and test new versions of the source code. This project makes heavy use of CI to monitor the build status, test cases and code coverage of each software artefact, as further described in section 8.4.

5 Requirements

The requirements of each deliverable have been outlined in the succeeding subsections, and are categorised using MoSCoW prioritization [26]; a definition of which is presented in table 2. Each requirement is directly related to an objective as indicated by the requirements tables of the deliverables. The only objectives not covered by these requirements are objective 1, 2 and 3 which relate to the literature review, and objective 7 which was intentionally left as a future ambition.

Priority	Description
MUST	An essential requirement that *must* be satisfied
SHOULD	An important requirement that *should* be satisfied if possible
COULD	A desirable requirement that *could* be satisfied but it is not necessary
WON'T	A future requirement that *will not* be satisfied in this release

Table 2: A summary of the MoSCoW (MUST, SHOULD, COULD, WON'T) priorities.

5.1 LLVM IR Library

The LLVM IR language defines several primitives directly related to code optimisation and linking, neither of which convey any useful information for the decompilation pipeline. It is therefore sufficient for this project to support a subset of the LLVM IR language and the relevant requirements should be interpreted as referring to a subset of the language.

The control flow recovery tool interacts with other components using LLVM IR. It is therefore required to support reading from and writing to at least one of the representations of LLVM IR. The representations of LLVM IR are isomorphic and the standard `llvm-as` and `llvm-dis` tools from the LLVM distribution may be used to convert between the assembly language and bitcode representation of LLVM IR. Access to the bitcode representation (**R6** and **R7**) has therefore been deferred in favour of the assembly language representation (**R1** and **R2**) which has the benefit of being human-readable.

The control flow analysis library will inspect and manipulate an in-memory representation of LLVM IR (**R3**) to locate high-level control flow patterns and store these findings respectively. Rather than working with sequential lists, the control flow analysis algorithms will operate on CFGs of basic blocks (**R4**). To facilitate the implementation and debugging of these algorithms, a visual representation of the CFGs would be beneficial (**R5**).

To guarantee the language-agnostic interaction between components, objective 3 stated that a formal grammar for the LLVM IR had to be located or produced (**R8**). Previous efforts have only managed to produce formal grammars for subsets of the LLVM IR language [27, 28] and no such grammar has been officially endorsed. The difficult nature of producing a formal grammar only became apparent after discussions with the project supervisor. With this in mind, objective 3 has been re-evaluated as a future ambition.

Obj.	Req.	Priority	Description
4	**R1**	MUST	Read the assembly language representation of LLVM IR
4	**R2**	MUST	Write the assembly language representation of LLVM IR
4	**R3**	MUST	Interact with an in-memory representation of LLVM IR
4	**R4**	MUST	Generate CFGs from LLVM IR basic blocks
4	**R5**	COULD	Visualise CFGs using the `DOT` graph description language
4	**R6**	WON'T	Read the bitcode representation of LLVM IR
4	**R7**	WON'T	Write the bitcode representation of LLVM IR
3	**R8**	WON'T	Provide a formal grammar of LLVM IR

Table 3: Requirements of the LLVM IR library.

5.2 Control Flow Analysis Library

A decision was made early on to only support decompilation of compiler generated code from structured high-level languages (**R9**). Support for arbitrary, unstructured and obfuscated code has been intentionally omitted (**R18**) to avoid a myriad of special cases.

The control flow analysis library must recover the high-level control flow primitives of pre-test loops (**R10**), infinite loops (**R11**), 1-way conditionals (**R12**) and 2-way conditionals (**R13**), as these are found in virtually every high-level language today [2]. Post-test loops (**R14**) and n-way conditionals (**R15**) are also common - but not found in every language (e.g. Go has no `do-while` loops and Python has no `switch` statements) - and should therefore be recovered. Support for multi-exit loops (**R16**) and nested loops (**R17**) could be included if time permits. The recovery of compound boolean expressions is intentionally deferred (**R19**) as it would require analysis of instructions within basic blocks in addition to the CFG analysis.

Obj.	Req.	Priority	Description
5	**R9**	MUST	Support analysis of reducible graphs
5	**R10**	MUST	Recover pre-test loops (e.g. `while`)
5	**R11**	MUST	Recover infinite loops (e.g. `while(TRUE)`)
5	**R12**	MUST	Recover 1-way conditionals (e.g. `if`)
5	**R13**	MUST	Recover 2-way conditionals (e.g. `if-else`)
5	**R14**	SHOULD	Recover post-test loops (e.g. `do-while`)
5	**R15**	SHOULD	Recover n-way conditionals (e.g. `switch`)
5	**R16**	COULD	Recover multi-exit loops
5	**R17**	COULD	Recover nested loops
5	**R18**	WON'T	Support analysis of irreducible graphs
5	**R19**	WON'T	Recover compound boolean expressions

Table 4: Requirements of the control flow analysis library.

5.3 Control Flow Recovery Tool

The primary intention of this project is to create self-contained components which may be used in the decompilation pipelines of other projects. It is therefore of vital importance that the components are able to interact with tools written in other programming languages (**R21**). The control flow recovery tool is one such component which aims to recover a set of high-level control flow primitives from LLVM IR (**R20**).

Obj.	Req.	Priority	Description
6	**R20**	MUST	Identify high-level control flow primitives in LLVM IR
6	**R21**	MUST	Support language-agnostic interaction with other components

Table 5: Requirements of the control flow recovery tool.

“The whole is more than the sum of its parts.” — Anonymous

6 Design

The principle of separation of concern has had a core influence on the design of the decompilation system. It has motivated a system architecture based on the composition of independent and self-contained components. End-users may either use the individual component in separation, or combine a set of components into a custom decompilation pipeline.

Several smaller components may conceptually be arranged in a pipeline of stages which transform, massage or interpret the input in a certain way to solve larger tasks. A well composed pipeline is capable of solving more complex problems than each of its components, problems which may not even have been envisioned by the original component authors [29]. This idea is embodied in the Unix philosophy and it has influenced software construction profoundly [30]. Furthermore, systems which expose their individual components to end-users facilitate dynamic workflows, as they enable users to adapt and extend each part of the system by adding, removing, replacing or refining components in one or more stages of the pipeline.

To enforce a strict separation of concerns, each component is given access to the least amount of information required to successfully accomplish its task (e.g. the control flow analysis stage operates on CFGs and is unaware of the underlying code).

The design of the decompilation system must allow language-agnostic interaction between components written in different programming languages (refer to the aim of the project in section 1.1). This requirement has been satisfied by communicating through well-defined input and output (e.g. JSON, DOT, LLVM IR). A more detailed view of the system architecture is presented in section 6.1.

6.1 System Architecture

The decompilation pipeline conceptually consists of three modules which separate the general decompilation tasks (e.g. control flow analysis) from concerns related to the source language and the target language. Firstly, the front-end translates a variety of source languages (e.g. x86 or ARM assembly, C or Haskell source code, . . .) to LLVM IR by utilizing several independent open source projects. Secondly, the middle-end structures the LLVM IR by identifying high-level control flow primitives in the CFGs generated from the intermediate representation. Lastly, the back-end translates the structured LLVM IR into a high-level target programming language (e.g. Go). The interaction between these modules is visualised in figure 11, and the individual components of the front-end, middle-end and back-end modules are further described in section 6.2, 6.3 and 6.4 respectively.

The main benefit with this decompiler architecture is that it scales well when implementing support for additional source languages (e.g. MIPS or PowerPC assembly) and target languages (e.g. Python), as the general decompilation tasks only have to be implemented once. The decompiler architecture is an adaptation of the one presented by

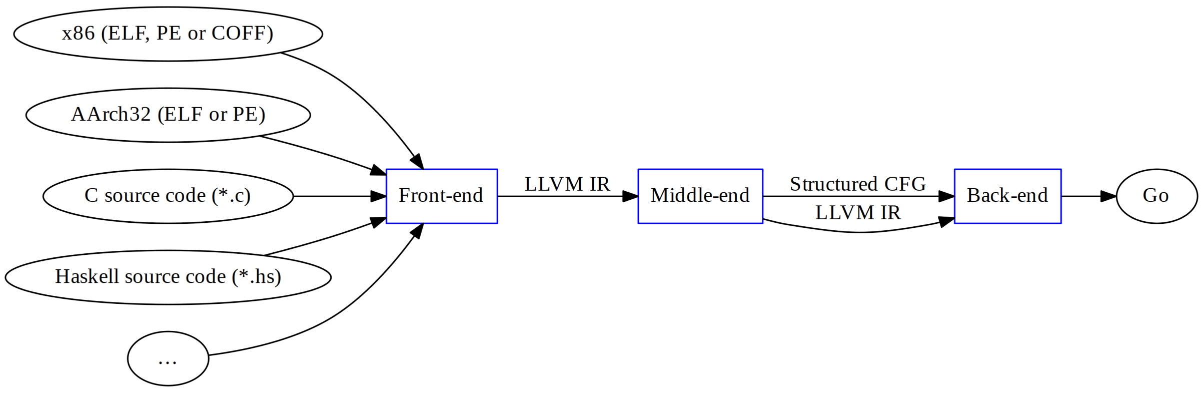


Figure 11: The front-end of the decompilation pipeline translates a variety of inputs (e.g. native code or source code) to LLVM IR; the middle-end structures the LLVM IR through control flow analysis; and the back-end translates the structured LLVM IR to a high-level programming language (e.g. Go).

C. Cifuentes back in 1994 (as described in section 2.2), which was heavily inspired by the architecture of compilers that separated general optimisation tasks (e.g. constant propagation) from concerns related to the source programming language (e.g. C) and the target computer architecture (e.g. x86). The compiler architecture has been proven so effective at separating concerns that it remains in use today by several production-quality compilers [15, 31].

6.2 Front-end Components

The front-end module is responsible for converting a variety of inputs into LLVM IR. Two common scenarios involve converting binary files (e.g. executables, shared libraries and relocatable object code) and converting source code (e.g. C, Haskell, Rust, . . .) into LLVM IR. The first scenario is presented in section 6.2.1 and the second in section 6.2.2.

6.2.1 Native Code to LLVM IR

There exist several open source projects which translate native code (e.g. x86 assembly of shared libraries in the PE file format) into LLVM IR. Three such projects have been reviewed in section 3.1, which support different input file formats and machine architectures. These projects may be used as-is by the front-end module to translate low-level source languages into LLVM IR, as illustrated in figure 12.

6.2.2 Compilers

One important aspect of utilizing the IR of a compiler framework, is that the decompilation pipeline automatically gains support for transpilation (i.e. translating one programming language into another) in addition to reverse compilation. An increasing number of open source compilers (e.g. Clang, GHC, `rustc`) are capable of translating a range of source languages (e.g. C, Haskell, Rust) into LLVM IR. These compilers may be used

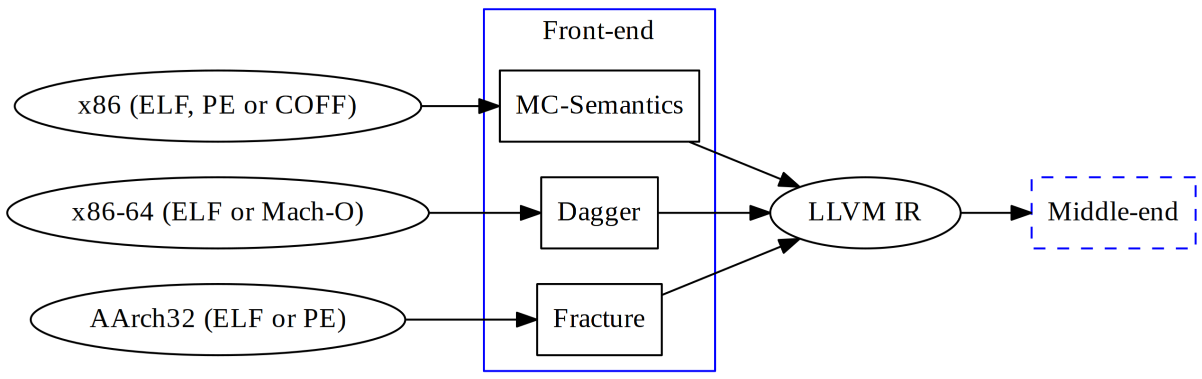


Figure 12: The three open source projects MC-Semantics, Dagger and Fracture translate native code of various architectures (e.g. x86, x86-64 and ARM) and file formats (e.g. ELF, PE, COFF and Mach-o) to LLVM IR.

as-is by the front-end module (see figure 13), thereby extending the supported source languages of the decompilation pipeline. Using this approach, the decompilation pipeline may translate n source languages into m target languages by implementing $n + m$ front-end and back-end modules, instead of $n \cdot m$ transpilers.

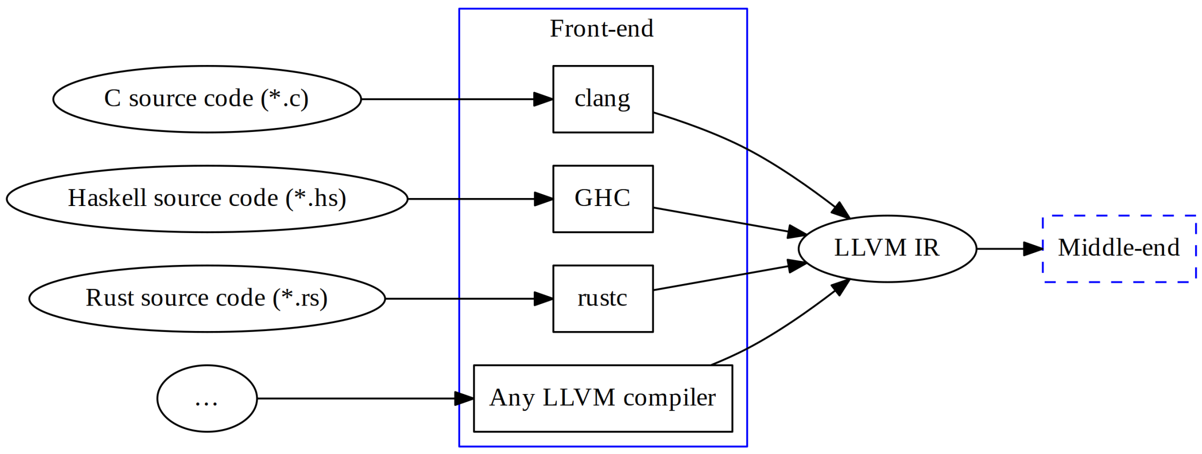


Figure 13: Several open source compilers translate high-level programming languages into LLVM IR. Three such compilers are Clang, the Glasgow Haskell Compiler and the Rust compiler which translate C, Haskell and Rust respectively into LLVM IR.

Another important aspect of utilizing LLVM IR, is that a wide range of optimisations have been implemented already by the LLVM compiler framework. This allows the front-end components to focus on translating the source languages into LLVM IR, without having to worry about producing highly optimised output. The LLVM IR may later be optimised by invoking the `opt` tool of LLVM to remove dead code, propagate constants, and promote memory accesses to registers, for instance.

6.3 Middle-end Components

The middle-end module is responsible for lifting the low-level IR generated by the front-end to a higher level. This is achieved through a set of decompilation stages, which identify high-level control flow primitives and, as a future ambition, propagate expressions. The former decompilation stage consists of two self-contained components which separate concerns related to the control flow analysis from the underlying details of LLVM IR. The first component generates unstructured CFGs from LLVM IR, as further described in section 6.3.1. And the second component structures the generated CFGs by identifying high-level control flow primitives, as further described in section 6.3.2. The interaction between the front-end, the `ll2dot` and `restructure` tools of the middle-end and the back-end is illustrated in figure 14.

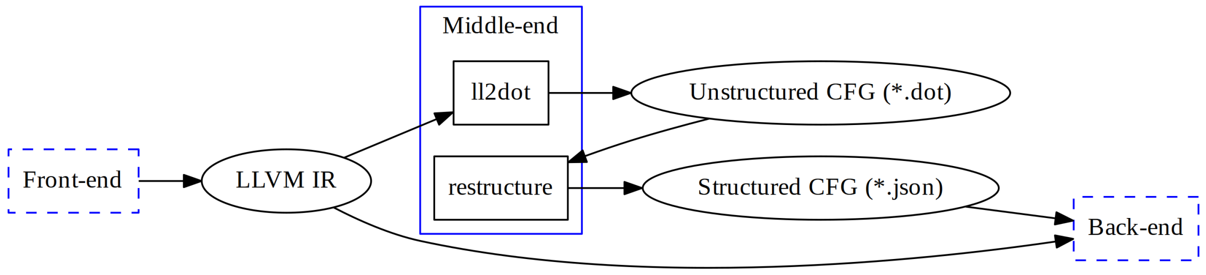


Figure 14: The middle-end module performs a control flow analysis on the LLVM IR in two steps. Firstly, the `ll2dot` tool generates unstructured CFGs (in the DOT file format) from LLVM IR. Secondly, the `restructure` tool produces a structured CFG (in JSON format) by identifying high-level control flow primitives in the unstructured CFG.

6.3.1 Control Flow Graph Generation

The control flow graph generation component generates a CFG for each function of a given LLVM IR assembly file. As described in section 2.3.2, a function definition in LLVM IR consists of a set of basic blocks; and a basic block consists of zero or more non-branching instructions followed by a terminating instruction (such as `br` or `ret`) which changes the control flow. Therefore, the control flow graph generation component may focus on analysing the last instruction of each basic block, as they will determine the control flow.

To generate the CFG of a given function, a directed graph is created and populated with one node per basic block, and with zero or more directed edges between the nodes of the graph. The node names are determined by the basic block labels, and the directed edges are determined by the terminating instructions, as illustrated in figure 15.

The `ll2dot` tool generates CFGs from LLVM IR in the DOT file format, which is a well-defined textual representation of graphs used by the Graphviz project. One benefit of expressing CFGs in this format, is that the existing Graphviz tools may be facilitated to produce image representations of the CFGs; as demonstrated in appendix F.

```
define i32 @f(i1 %cond) {
foo:
  br i1 %cond, label %bar, label %baz
bar:
  ret i32 42
baz:
  ret i32 37
}
```

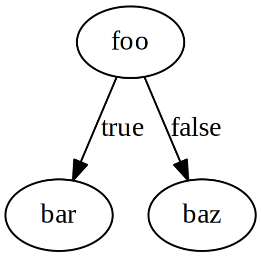
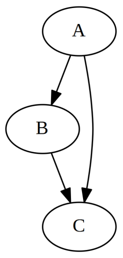


Figure 15: The return instructions of basic block `bar` and `baz` produces no directed edges, while the conditional branch instruction of basic block `foo` produces two directed edges in the CFG, one for each target branch (i.e. `bar` and `baz`).

6.3.2 Control Flow Analysis

The key idea behind the control flow analysis (see section 2.2.3), is that high-level control flow primitives may be represented using directed graphs. The problem of structuring low-level code may therefore be rephrased as the problem of identifying subgraphs (e.g. the graph representation of high-level control flow primitives) in graphs (e.g. the CFGs of low-level code) without considering node names, as illustrated in figure 16. This problem is generally referred to as *subgraph isomorphism search*, which has been well studied [11]. Rephrasing the problem in this manner aligns with the design principle of giving each component access to the least amount of information required to successfully accomplish its task. The control flow analysis component is only given access to control flow information (e.g. CFGs), and is oblivious of the underlying LLVM IR. This enables the component to be reused as-is when analysing the control flow of other languages, such as REIL.

```
if A {
    B
}
C
```



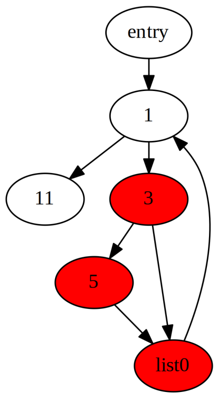


Figure 16: The left side contains the pseudo-code (top left) and graph representation (bottom left) of an if-statement; if `A` is true then do `B` followed by `C`, otherwise do `C`. The right side highlights (in red) an identified isomorphism of the if-statement's graph representation, in the CFG of the `main` function presented in appendix E.

The `restructure` tool uses subgraph isomorphism search algorithms to locate isomorphisms of the graph representations of high-level control flow primitives in the CFG of a given function. The CFG is simplified by recursively replacing the identified subgraphs with single nodes until the entire CFG has been reduced into a single node; a step-by-step demonstration of which is presented in appendix G. By recoding the node names of the identified subgraph isomorphisms and the name of their corresponding high-level control flow primitives, a structured CFG may be produced in which all nodes are known to belong to a high-level control flow primitive; as demonstrated in appendix H.

The pseudo-code and graph representations of the supported high-level control flow primitives are presented in figure 7 of section 2.2.3. Should the control flow analysis fail to reduce a CFG into a single node, the CFG is considered irreducible with regards to the supported high-level control flow primitives, in which case a structured CFG cannot be produced.

The `restructure` tool relies entirely on subgraph isomorphism search to produce structured CFGs (in JSON format) from unstructured CFGs (in the DOT file format). The supported high-level control flow primitives are defined using DOT files, thus promoting a data-driven design which separates data regarding the primitives from the implementation of the `restructure` tool. A major benefit with this approach is that the `restructure` tool may search for any high-level control flow primitive that can be expressed in the DOT file format, without any modification to the source code of `restructure`.

One limitation with this approach is that it does not support graph representations of high-level control flow primitives with a variable number of nodes, as they cannot be described in the DOT file format. For this reason, the `restructure` tool does not support the recovery of n-way conditionals (e.g. `switch`-statements). Furthermore, the current design enforces a single-entry/single-exit invariant on the graph representation of high-level control flow primitives. This prevents the recovery of infinite loops, as their graph representation has no exit node. A discussion of how these issues may be mitigated in the future is provided in section 10.2.1.

6.4 Back-end Components

The back-end module translates structured LLVM IR into a target high-level programming language, using two distinct stages. Firstly, the code generation stage translates LLVM IR into unpolished Go code by converting the individual instructions into equivalent Go statements and creating high-level control flow primitives for the various basic blocks, using the information of the structured CFGs (see section 6.3.2). Secondly, the post-processing stage improves the quality of the unpolished Go code, through a series of source code transformations. The interaction between the middle-end, and the `ll2go` and `go-post` tools of the back-end is illustrated in figure 17.

The clear distinction between the two back-end stages aligns with the design principle of separation of concern. The code generation stage may focus on converting LLVM IR into equivalent Go code, without having to worry about the quality of the produced code. Similarly, the post-processing stage may focus on simplifying the Go code and make it more idiomatic, without any knowledge of the underlying LLVM IR. This enables the post-processing component to be reused as-is by other projects to improve the quality of Go code.

A tighter integration between the two stages could potentially produce a higher quality output, but there are no known issues preventing the decoupled stages from producing output of equivalent quality.

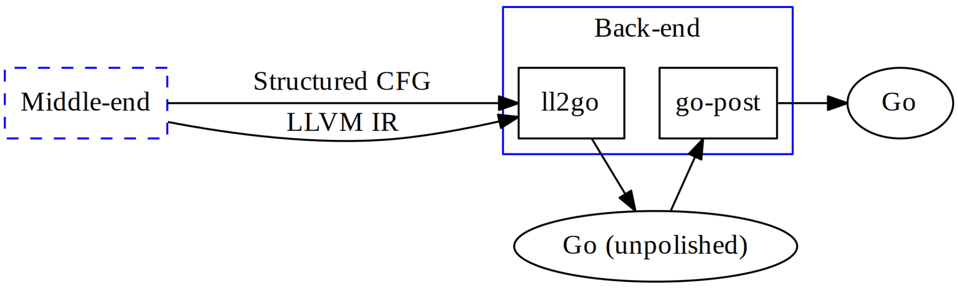


Figure 17: The back-end module decompiles structured LLVM IR into Go source code, using two components. The `ll2go` tool translates structured LLVM IR assembly into unpolished Go code, which is post-processed by the `go-post` tool to improve the quality of the output.

The decompilation pipeline aims to keep the back-end module as simple as possible, by delegating general decompilation tasks (e.g. control flow analysis, data flow analysis) to the middle-end module. This reduces the efforts required to implement additional back-ends, which add support for new target programming languages (e.g. Python).

6.4.1 Post-processing

The post-processing stage post-processes the unpolished Go source code from the earlier stages of the decompilation pipeline, by applying a set of source code transformations. The `go-post` tool improves the quality of Go source code by declaring unresolved identifiers, applying Go conventions for exit status codes, propagating temporary variables into expressions, simplifying binary operations, removing dead assignment statements, and promoting the initialisation statement and post-statement of for-loops to the loop header; as demonstrated by the step-by-step refinement of the unpolished source code in presented appendix J.

7 Implementation

This section motivates the language choice of the decompilation pipeline, describes the implementation process, and provides insight into how the software artefacts evolved from the challenges that were encountered.

7.1 Language Considerations

As stated by H. Mayer in 1989, *"No programming language is perfect. There is not even a single best language; there are only languages well suited or perhaps poorly suited for particular purposes. Understanding the problem and associated programming requirements is necessary for choosing the language best suited for the solution."* [32] This project seeks to explore the potential of a compositional approach to decompilation, and the components of the decompilation pipeline will require support for analysing and manipulating source code, and interacting with LLVM IR. The Go programming language emphasises composition at its core and provides extensive support for source code analysis, as indicated by the vast number of tools developed for analysing and manipulating Go source code (examples of such tools are given in section 8.4). As the LLVM compiler framework is written in C++, several projects (e.g. Dagger, Fracture, MC-Semantics) have chosen this language for interacting with LLVM IR. Meanwhile, a mature LLVM IR library is yet to be written for Go.

In 2012 Rob Pike (one of the Go language inventors) gave a talk titled *"Less is exponentially more"* which included a personal description of the historic events leading up to the inception of Go. The starting point of the language was C, not C++, which Go aimed to simplify further by removing cruft. This is in direct contrast to the direction of C++ which gains more features with each passing release. The *less is more* mindset is deeply rooted in the mentality of Go developers, and there is a strong emphasis on the use of composition to solve problems, as indicated by the following extract from Rob Pike's talk.

> *"If C++ and Java are about type hierarchies and the taxonomy of types, Go is about composition.*
>
> *Doug McIlroy, the eventual inventor of Unix pipes, wrote in 1964 (!):*
>
> > *"We should have some ways of coupling programs like garden hose–screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also."*
>
> *That is the way of Go also. Go takes that idea and pushes it very far. It is a language of composition and coupling."*
>
> — Rob Pike, 2012 [33]

Every aspect of Go development embodies the Unix philosophy (see figure 18), which is no surprise as Ken Thompson (one of the original inventors of Unix) is part of the core Go team.

"Write programs that do one thing and do it well. Write programs to work together."

Figure 18: The Unix philosophy [30].

To conclude the language considerations, Go has been chosen as the primary language for the decompilation pipeline based on its simplicity and emphasis on composition. Furthermore, the Go standard library includes production quality support for lexing and parsing of Go source code. The language may therefore be a good candidate for developing lexers and parsers for LLVM IR, as will be further discussed in section 7.2.

7.2 LLVM IR Library

Early on in the project it was believed that the control flow analysis stage would operate on CFGs that were tightly coupled with an in-memory representation of LLVM IR. This motivated the search for a LLVM IR library with a carefully considered API and set of data structures (objective 4). While there existed a library which provides Go bindings for LLVM, the API of this library felt awkward to use and was too heavily influenced by the underlying C and C++ libraries; as further described in section 7.3. The interaction with the LLVM IR library would critically influence the design and implementation of the decompilation components. For this reason it was decided that a set of pure Go libraries would be implemented for interacting with LLVM IR, even if it would require a considerable amount of work.

The LLVM IR libraries were intentially developed as reusable components for compilers, decompilers and other semantic analysis tools. To assess the requirements of LLVM-based compilers, a public discussion was held with the developers of the open source `llgo` compiler, who clarified its specific requirements[7]. Fredrik Ehnbom, who is one of the `llgo` developers, has remained involved with the development of the LLVM IR libraries, by participating in API discussions, conducting code reviews, and submitting patches for performance improvements[8] (these code changes have not yet been merged, as the project artefacts are required to be developed independently).

The first component to be implemented was the LLVM IR lexer, which tokenizes LLVM IR assembly. In addition to the LLVM IR language specification, the implementation of the reference lexer in LLVM was reviewed to establish a full listing of the valid tokens in LLVM IR. This review uncovered two tokens which were defined but never used in the code base of LLVM. In collaboration with members of the LLVM community, a patch was commited to the LLVM code base which removed these tokens[9].

The design of the LLVM IR lexer has been hugely influenced by a talk given by Rob Pike in 2011, titled *"Lexical Scanning in Go"* [34]. In this talk, Pike introduces the notion of a state function which is a function that returns a state function. The state of the lexer is represented by the active state function, which may transition into another stage by returning the corresponding state function. For instance, when `lexLineComment` is active, the context of the lexer is known to be a line comment. Any character is valid within

[7]Requirements: `https://github.com/llir/llvm/issues/3`
[8]Use binary search for keyword lexing: `https://github.com/llir/llvm/pull/11`
[9]Remove unused tokens from AsmParser: `http://reviews.llvm.org/D7248`

line comments, except new lines which terminate the token; at which point an unknown token on the succeeding line is lexed by returning the `lexToken` state function.

The LLVM IR assembly language requires no separators (e.g. whitespace characters, semicolons) between tokens. This made it very difficult to determine where one token ends and another starts, as further indicated by inconsistent behaviour for separating tokens in the reference implementation of the LLVM lexer. The solution to this issue was inspired by the Go language specification[10] which states that *"the next token is the longest sequence of characters that form a valid token"*, thus defining a consistent behaviour.

A formal grammar of the LLVM IR language would have facilitate the implementation of the LLVM IR libraries. As no such grammar had been officially endorsed, other sources were cross-referenced to learn about the low-level details of the language, such as its token set and details regarding the type system. This work uncovered a number of potential inconsistencies between the language reference, the implementation, and official blog posts. After discussing these issues with more experienced LLVM developers on the LLVM-dev mailing list, it could be concluded that some issues highlighted inconsistent behaviours while others were working as intended.

The cheer size of LLVM IR was at times discouraging and the project time constrains forced the implementation of subsets within every aspect of the language. LLVM IR may have started out as a simple, minimalistic and platform-independent low-level IR, but this is no longer the case. As time went by and as the project rose in popularity, more and more developers joined the project. In 2014 more than five hundred commits were submitted to the LLVM code base each month. Keeping these changes consistent and the overall system simple is a massive challenge.

It was at times very tempting to switch to REIL instead of LLVM IR, as REIL is a minimal, consistent and clean language. The adaptation of REIL in the open source community was however limited, and there had been no public news announcements since Google acquired the company back in 2011. Furthermore, the REIL language lacks the notion of basic blocks which would complicate the control flow analysis.

After months of development it had become clear that the task of implementing libraries for LLVM IR was way more time consuming than initially anticipated. The project time constrains forced the re-evaluation of using the Go bindings for LLVM, which gave rise to a seemingly small but hugely influential idea. The control flow analysis stage should operate entirely on graph data structures, thus making it unaware of LLVM IR. This idea effectively mitigated the risk of being influenced by the API of the Go bindings for LLVM, and gave rise to the data-driven design of the control flow analysis component, as further described in section 6.3.2. From this point on, the focus shifted to implementing working artefacts which utilised the Go bindings for LLVM IR.

7.3 Go Bindings for LLVM

In the beginning of February 2015 a decision was made to defer the development of the LLVM IR library (see section 7.2) and make use of the Go bindings for LLVM instead. This decision was motivated by time constraints and has had both positive and negative

[10]The Go Programming Language Specification: `https://golang.org/ref/spec`

implications for the project. Most importantly, it became possible to rapidly implement working prototypes of the artefacts once the focus shifted from utility library development to core component development. Furthermore, when LLVM 3.6 was released in the end of February 2015, the components automatically gained support for the new LLVM IR metadata syntax which was introduced in this release.

Several components (e.g. `ll2dot`, `ll2go`) use the Go bindings for LLVM[11] to interact with LLVM IR assembly. The API of the Go bindings mimics that of the C bindings, which feels awkward as it bypasses the safety of the type system. A fundamental concept of the API is the notion of a *value*, which represents a computed value that may be used as an operand of other values. This recursive description captures the semantics of instructions and constants. The API provides a unified `Value` type, which defines 145 methods for interacting with the underlying value types. It is the callers responsibility to only invoke the subset of methods actually implemented by the underlying type. In practice this approach results in fragile applications which may crash during runtime; with assertions such as *"cast<Ty>() argument of incompatible type!"* when invoking the `Opcode` method instead of the `InstructionOpcode` method on an instruction value. The former method may only be invoked on constants, which uses a subset of the instruction opcodes for constant expressions. A more sound approach would solve these issues by refining the `Value` interface be the *intersect* rather than the *union* of all methods implemented on the underlying value types.

The control flow analysis stage of the decompilation pipeline requires access to the names of unnamed basic blocks, but these names are not accessible from the API of the Go bindings for LLVM as they are generated on the fly by the assembly printer. To work around this issue, the assembly printer of LLVM 3.6 was patched to always print the generated names of unnamed basic blocks (see appendix B). Once patched, the debug facilities of LLVM could be utilised to print the assembly to temporary files, which may be parsed to gain access to the names of unnamed basic blocks. This solution works, but it is considered highly temporary and may cause security implications (as further described in section 8.3). When the pure Go LLVM IR library (see section 7.2) reaches maturity it will replace the use of the Go bindings for LLVM, thus mitigating the aforementioned issues.

7.4 Subgraph Isomorphism Search Library

Implementing the subgraph isomorphism search algorithm was without doubt the most difficult endeavour of the entire project. It took roughly five iterations of implementing, evaluating and rethinking the algorithm to find an approach which felt right and another two iterations to develop a working implementation which passed all the test cases.

As mentioned in section 8.2, an early throwaway prototype provided a partial implementation of the subgraph isomorphism algorithm proposed by Ullman. The prototype was intended to provide insight into the subgraph isomorphism problem domain, and was eventually discarded.

The second throwaway prototype was specifically designed to exploit known properties of CFG (e.g. connected graphs with a single entry node) to limit the search space.

[11] Go bindings for LLVM: `https://godoc.org/llvm.org/llvm/bindings/go/llvm`

Focusing on connected graphs drastically simplified the general problem of subgraph isomorphism search, and enabled algorithms which traverse the graph from a given start node to identify subgraph isomorphisms. The second prototype had many issues (e.g. non-deterministic, unable to handle graph cycles), but provided valuable insight into how a subgraph isomorphism search algorithm may be designed when focusing on connected graphs.

In contrast to its predecessor, the third prototype separated subgraph isomorphism candidate discovery from candidate validation logic. A subgraph isomorphism candidate is a potential isomorphism of a subgraph in a graph, which provides a mapping from subgraph node names to graph node names. Should the source and the destination nodes of each directed edge in the subgraph translate through the candidate node mapping to corresponding nodes (with a directed edge from the source to the destination node) in the graph, and should furthermore each node in the subgraph have the same number of directed edges as the nodes of the candidate (with a few caveats regarding entry and exit nodes), then the candidate is considered a valid isomorphism of the subgraph in the graph. The third prototype was still incomplete (mainly with regards to candidate discovery) when discarded, but the separation of candidate discovery and candidate validation logic has had a large influence on its succeeding prototypes.

As described in section 6.3.2 and further evaluated in section 9.2.1, the current implementation of the subgraph isomorphism search algorithm enforces a single-entry/single-exit invariant on the subgraphs to simplify control flow analysis. This allows identified subgraphs to be replaced with single nodes, which inherit the predecessors of the subgraph entry node and successors of the subgraph exit node. For this reason, the candidate validation logic disregards the directed edges from predecessors of subgraph entry nodes and the directed edges to successors of subgraph exit nodes, when validating subgraph isomorphism candidates; which should clarify the aforementioned caveats of the preceding paragraph.

Similar to the third prototype, the fourth throwaway prototype separated candidate discovery from candidate validation. In addition, it introduced the concept of treating candidate node mappings as equations which may be solved, or at least partially solved. The candidate node mappings were extended from one-to-one node mappings (one subgraph node name maps to exactly one graph node name) to node pair mappings, which represent one-to-many node mappings (one subgraph node name maps to zero or more graph node names). The candidate discovery logic was extended to record all potential candidate nodes for a given subgraph node, when traversing the graph in search of candidates. A simple equation solver was implemented which was capable of identifying unique node pair mappings and propagate this information to successively simplify equations until they are either solved or require other methods for solving; an example of which is presented in figure 19. The equation solver would however fail to find a solution if two node pair mappings had the same candidate nodes, as illustrated in figure 20

The fifth throwaway prototype extended the capabilities of the simple equation solver by trying different candidate node mappings recursively until a valid subgraph isomorphism was found or known not to exist. These equations were solved concurrently using Go routines (independently executing functions, which are multiplexed onto system threads) which relayed the answers back using channels (typed and synchronised communication channels).

```
"A": ["X", "Y", "Z"]
"B": ["Y", "Z"]
"C": ["Z"]
```

(a) Step 1.

```
"A": ["X", "Y"]
"B": ["Y"]
"C": ["Z"]
```

(b) Step 2.

```
"A": ["X"]
"B": ["Y"]
"C": ["Z"]
```

(c) Step 3.

Figure 19: In step 1, the unique node pair mapping between the subgraph node name C and the graph node name Z is identified, and the remaining node pair mappings are simplified by removing Z from their candidate nodes. Similarly, in step 2, the unique node pair mapping between B and Y is identified; thus simplifying the equation further. Lastly, in step 3, the unique node pair mapping between A and X is identified, and the equation is thereby solved.

```
"A": ["X", "Y"]
"B": ["X", "Y"]
```

Figure 20: An equation which the simple equation solver of the forth prototype would fail to solve, as it cannot be simplified by identifying unique node pair mappings.

At this stage, the algorithm design had started to feel mature and the focus shifted from implementing throwaway prototypes to building a solid foundation. Starting with the parts of the system that were best understood, one part or concept at the time were removed from the throwaway prototype and carefully reimplemented in a new library through a series of steps. Firstly, the API of the new library was taken into careful consideration, and a set of stub functions and core data structures were added and thoroughly documented. Secondly, test cases were written for each stub function of the library. Lastly, the stub functions were implemented one at the time and verified against the test cases.

While the new implementation passed most test cases, there were a few corner cases for which the library produced incorrect results. The concurrent nature of the library made it difficult to debug, and a decision was made to reimplement the equation solver without concurrency. This resulted in a cleaner implementation which was easy to debug and successfully passed all test cases.

The final implementation of the subgraph isomorphism search algorithm is a cleaned up and thoroughly tested version of the non-concurrent library, which has a 94.8% code coverage; as further described in section 8.1.1. The final implementation was developed in the *"isobug"* branch on GitHub, and merged[12] once stable into the *"master"* branch.

7.5 Documentation

A set of source code analysis tools are used to automate the generation and presentation of documentation. The main benefit of this approach is that only one version of the documentation has to be maintained and it is kept within the source code, thus preventing it from falling out of sync with the implementation. Unix manual pages are generated

[12]Fix subgraph isomorphism search: `https://github.com/decomp/decomp/issues/183`

for command line tools using `mango`[13], which locates the relevant comments and command line flag definitions in the source code. Library documentation is presented using `godoc`[14] (a tool similar to `doxygen`), and may be accessed through a web or command line interface.

The GoDoc.org server hosts an instance of `godoc` which presents the documentation of publicly available source code repositories. An online version of the documentation has been made available for each artefact using this service.

- Library for interacting with LLVM IR (*work in progress*)
  `https://godoc.org/github.com/llir/llvm`
- Control flow graph generation tool
  `https://godoc.org/decomp.org/decomp/cmd/ll2dot`
- Subgraph isomorphism search algorithms and related tools
  `https://godoc.org/decomp.org/decomp/graphs`
- Control flow recovery tool
  `https://godoc.org/decomp.org/decomp/cmd/restructure`
- Go code generation tool (*proof of concept*)
  `https://godoc.org/decomp.org/decomp/cmd/ll2go`
- Go post-processing tool
  `https://godoc.org/decomp.org/decomp/cmd/go-post`

[13]Generate Man pages from Go source: `https://github.com/slyrz/mango`

[14]Godoc extracts and generates documentation for Go programs: `https://golang.org/cmd/godoc`

8 Verification

This section describes the methods used to verify the correctness, measure the performance and assess security of the software artefacts. It concludes with a discussion on how CI is utilised to automatically and continuously verify these aspects.

8.1 Test Cases

As stated by Edsger W. Dijkstra in 1969, *"testing shows the presence, not the absence of bugs."* [35] For this reason, several independent methods were utilised to verify the correctness of the decompilation components and their utility libraries, including the automatic generation of C programs (with a large number of nested `if`-statements) which were used to stress test each component of the decompilation pipeline; as further described in section 8.2.

A lot of thought went into designing test cases which attempt to break the code, exploit assumptions, and exercise tricky corner cases (e.g. no whitespace characters between tokens in LLVM IR). These tests were often written prior to the implementation of the software artefacts, to reduce the risk of testing what was built rather than what was intended to be built (as specified by the requirements). The test cases have successfully identified a large number of bugs in the software artefacts, and even uncovered inconsistent behaviour in the reference implementation of the LLVM IR lexer; as further described in section 7.2. To facilitate extensibility, the test cases were often implemented using a table driven design which separate the test case data from the test case implementation.

An extract of the test cases used to verify the candidate discovery logic, the equation solver and the candidate validation logic of the subgraph isomorphism search library is presented in figure 21. These test cases are automatically executed by the CI service any time a new change is committed to the source code repository, as further described in section 8.4.

```
$ go test -v decomp.org/decomp/graphs/iso
=== RUN TestCandidates
--- PASS: TestCandidates (0.02s)
=== RUN TestEquationSolveUnique
--- PASS: TestEquationSolveUnique (0.00s)
=== RUN TestEquationIsValid
--- PASS: TestEquationIsValid (0.22s)
=== RUN TestIsomorphism
--- PASS: TestIsomorphism (0.18s)
=== RUN TestSearch
--- PASS: TestSearch (0.20s)
PASS
ok      decomp.org/decomp/graphs/iso   0.62s
```

Figure 21: An extract of the test cases used to verify the subgraph isomorphism search library, as visualised by `go test`.

8.1.1 Code Coverage

Code coverage is a measurement for tracking what parts of the code that gets executed when running test cases. The Go project takes an interesting approach to tracking code coverage, which utilises the production quality source code analysis and transformation libraries available in Go. Instead of generating platform-dependent assembly which tracks the execution of various branches, Go inject unique tracking statements on each line of the original source code. This approach is platform-independent by design, and may easily be extended to support heat maps which track how often each line gets executed [36].

As the lexer of any language is a fundamental building block on which other libraries and components depend, the test cases of the LLVM IR lexer aimed for a 100% code coverage of any code not related to input/output errors (e.g. "file not found"); as illustrated in figure 22. This rigorous testing uncovered several faulty assumptions in the lexing logic, which were later corrected.

```
$ go test -coverprofile=lexer.out github.com/llir/llvm/asm/lexer
$ go tool cover -func=lexer.out
llvm/asm/lexer/lexer.go:36:    ParseFile      80.0%
llvm/asm/lexer/lexer.go:118:   emit           100.0%
llvm/asm/lexer/lexer.go:139:   next           100.0%
llvm/asm/lexer/lexer.go:158:   accept         100.0%
llvm/asm/lexer/state.go:38:    lexToken       100.0%
llvm/asm/lexer/state.go:131:   lexComment     100.0%
...
llvm/asm/lexer/state.go:364:   lexQuote       100.0%
llvm/asm/lexer/state.go:585:   unescape       100.0%
total:                         (statements)   97.6%
```

Figure 22: A summary of the code coverage for a selection of the LLVM IR lexer functions, as visualised by the `cover` tool.

A brief summary of the code coverage for the various software artefacts is presented in figure 6.

Code coverage	Component
97.6%	LLVM IR lexer[15]
0.0%	Control flow generation tool
94.8%	Subgraph isomorphism search library[16]
40.0%	Control flow recovery tool[17]
0.0%	Code generation tool
38.0%	Post-processing tool[18]

Table 6: A summary of the code coverage of each component, presented roughly in the same order as they appear in the decompilation pipeline.

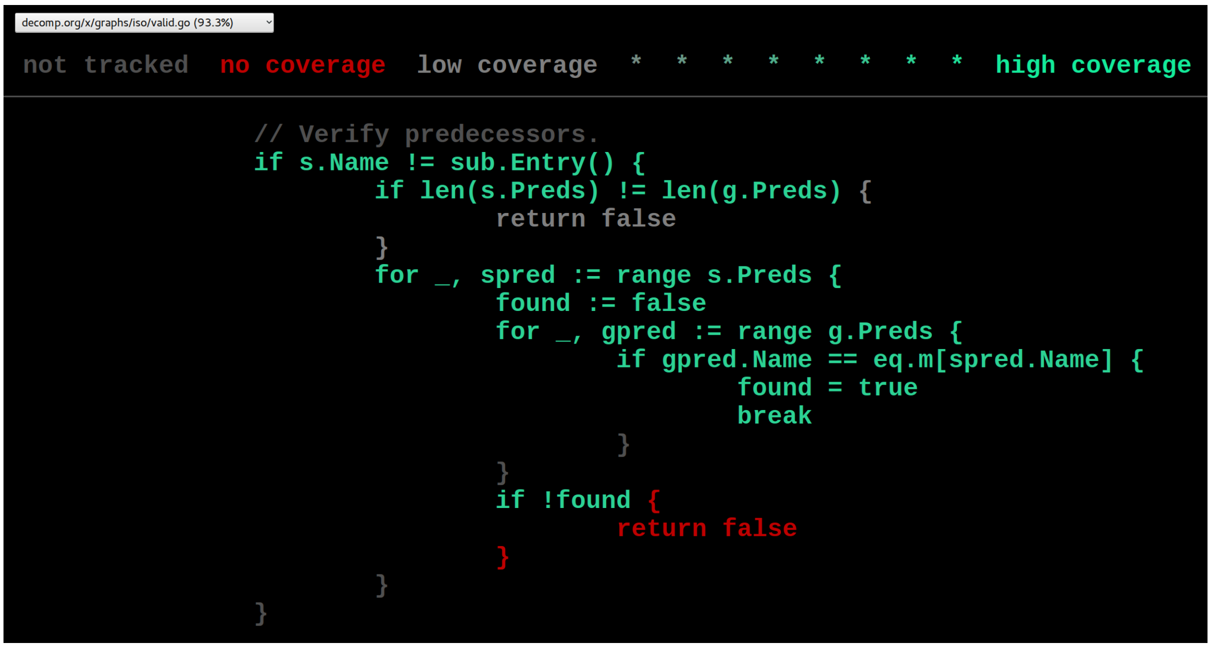
[15]As of Git revision `bbba2831ad079074516041907d16c347e388a310`

[16]As of Git revision `70967487ea73284c68a89e3fc566bc706603b6f7`

[17]As of Git revision `019e846bc7058f8fb9f7517568505c96eeed97bf`

[18]As of Git revision `02d38d1fbbf1c05bb11de89360a7cd8c38329a14`

Code coverage heat maps may be used to gain clarity in how often a line gets executed by the test cases. The use of heat maps were invaluable when stress testing the various prototypes of the subgraph isomorphism search algorithm, as they were able to identify several tricky corner cases; such as the ones illustrated in figure 23


decomp.org/x/graphs/iso/valid.go (93.3%)
not tracked no coverage low coverage * * * * * * * * high coverage

```
    // Verify predecessors.
    if s.Name != sub.Entry() {
            if len(s.Preds) != len(g.Preds) {
                    return false
            }
            for _, spred := range s.Preds {
                    found := false
                    for _, gpred := range g.Preds {
                            if gpred.Name == eq.m[spred.Name] {
                                    found = true
                                    break
                            }
                    }
                    if !found {
                            return false
                    }
            }
    }
```

Figure 23: An extract from the code coverage heat maps of the subgraph isomorphism search algorithm, which has identified two corner cases that may require further validation. The first return statement is seldomly executed (as indicated by the *grey* colour), and the second return statement is never executed.

8.2 Performance

The performance characteristics of the various components have been considered during every stage of the development process, but the initial prototypes have prioritised correctness and simplicity over performance. These prototypes have aimed at identifying suitable data structures and algorithms for the problems, through iterative redesigns and reimplementations. Once the major design decisions stabilised, production quality prototypes were being developed and thoroughly tested. To limit the risk of premature optimisations, micro-level performance work was intentionally postponed to the later stages of development.

Components with straight forward implementations (e.g. the LLVM IR library) have been profiled to identify performance bottle necks, as further described in section 8.2.1. When estimating the time complexity of various subgraph isomorphism search algorithms however, algorithm research and the use of intuition proved far more valuable. One of the first throw-away prototypes provided a partial implementation of the subgraph isomorphism algorithm proposed by Ullman. After further research the prototype was eventually discarded as the Ullman algorithm had been proven to scale poorly for randomly connected graphs with more than 700 nodes [37]. To put this into perspective, the `main` function of the c4[19] compiler consists of 248 basic blocks; in other words, the CFG of the `main` function is a connected graph (every node is reachable from the entry node) with 248 nodes. This leaves a margin (for the number of basic blocks in functions) of less than an order of magnitude before the Ullman algorithm starts to perform poorly.

There exist several subgraph isomorphism algorithms which scale better than the Ullman algorithm for graphs with a large number of nodes; such as the VF2 algorithm for dense graphs and McKay's nauty algorithm for sparse graphs [37, 11]. In the case of the c4 compiler, the CFGs are sparse with 1.35 edges per node in average, which would favour the nauty algorithm. The specific properties of CFGs (e.g. sparse connected graphs) guided the design of the subgraph isomorphism search algorithm, as further described in section 7.4.

To stress test the implementation of the decompilation components and to get an estimate of their time complexities, a set of C programs were automatically generated[20] with 2^x nested `if`-statements (where $7 \leq x \leq 12$). These C programs were converted to LLVM IR and decompiled into Go using the same steps as described in appendix E, F, H, I and J. The time complexity of each step may be estimated by monitoring how the execution time changes with regards to n, where n represents the number of nodes in the CFG of the generated programs; as summarised in table 7. The CFG of each generated program contains exactly twice as many nodes as nested `if`-statements; i.e. $n = 2^{x+1}$.

Component	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$
`ll2dot`	1.14s	4.01s	15.27s	1m 0s	3m 56s	15m 44s
`restructure`	0.97s	4.37s	21.37s	2m 6s	11m 1s	85m 58s
`ll2go`	2.85s	10.59s	40.95s	2m 41s	10m 47s	45m 13s
`go-post`						
unresolved	0.09s	0.29s	1.01s	3.70s	13.87s	53.35s
mainret	0.09s	0.28s	1.00s	3.70s	13.91s	52.91s
localid	3m 46s	57m 35s	-	-	-	-
assignbinop	0.02s	0.03s	0.07s	0.21s	0.72s	2.63s
deadassign	0.08s	0.26s	0.95s	3.40s	13.00s	49.96s
forloop	0.01s	0.03s	0.06s	0.15s	0.45s	1.63s

Table 7: The first column specifies the component being tested, and each consecutive column presents the execution time of the component (based on the average of three consecutive runs) in relation to n, which represents the number of nodes in the CFG. Each row below the `go-post` component represents a specific post-processing rewrite rule (e.g. *mainret*), as further described in appendix J. The steps which execute in polynomial time with regards to n have been highlighted in green, while the steps which execute in exponential time with regards to n have been highlighted in red.

Each step of the decompilation pipeline completed in reasonable time (i.e. polynomial time with regards to n) except for one, namely the *"localid"* rewrite rule (see figure 36 of appendix J) of the post-processing stage. As discussed in section 9, most of the post-processing rewrite rules are considered experimental and the *"localid"* rewrite rule suffers from both inaccuracy and performance issues. The *"localid"* rewrite rule optionally

[19]C in four functions: `https://github.com/rswier/c4`

[20]Generate nested C programs: `https://gist.github.com/mewmew/677994ee8da60bee1de9`

provides rudimentary expression propagation support, and will be removed when proper data flow analysis has been implemented by the middle-end (see section 10.2.1).

As the size of n doubles in table 7, the execution time of `ll2dot` roughly quadruples. The time complexity of `ll2dot` is therefore estimated to be $\Omega(n^2)$. Please note that this may not hold true for larger values of n, as a formal time complexity analysis of the algorithm is yet to be conducted. Similarly, as the size of n double, the execution time of `restructure` roughly octuples. The time complexity of `restructure` is therefore estimated to be $\Omega(n^3)$. The same logic and caveats may be applied to the other components to estimate their time complexities; a summary of which is presented below.

- `ll2dot`: $\Omega(n^2)$
- `restructure`: $\Omega(n^3)$
- `ll2go`: $\Omega(n^2)$
- `go-post`
  - *unresolved*: $\Omega(n^2)$
  - *mainret*: $\Omega(n^2)$
  - *localid*: exponential time complexity
  - *assignbinop*: $\Omega(n^2)$
  - *deadassign*: $\Omega(n^2)$
  - *forloop*: $\Omega(n^2)$

In summary, profiling is great for optimising the implementations of simple problems. Algorithm research, time complexity theory and intuition is essential for implementing performant solutions to complex problems. Furthermore, knowledge about specific properties of the problem may be exploited to design performant algorithms.

8.2.1 Profiling

The initial implementation of the LLVM IR lexer (see section 7.2) focused on correctness, and strived to be as simple and straight forward as possible. Once feature complete and thoroughly tested, the lexer was profiled for the first time and a major performance bottleneck was identified; as illustrated in figure 24. When scanning letters, the `lexLetter` function used a hash map to check if the scanned letters were part of a keyword. As letters make up the majority of the characters in LLVM IR source files, this caused an extensive number of hash map iterations which accounted for roughly 70% of the total execution time. To fix this issue, a benchmark test was implemented to measure the performance changes between the original and the updated version; as further described in section 8.2.2. At this stage, only CPU profiling has been utilised to identify performance bottlenecks. Future work may leverage memory profiling to further improve the performance of the decompilation components.

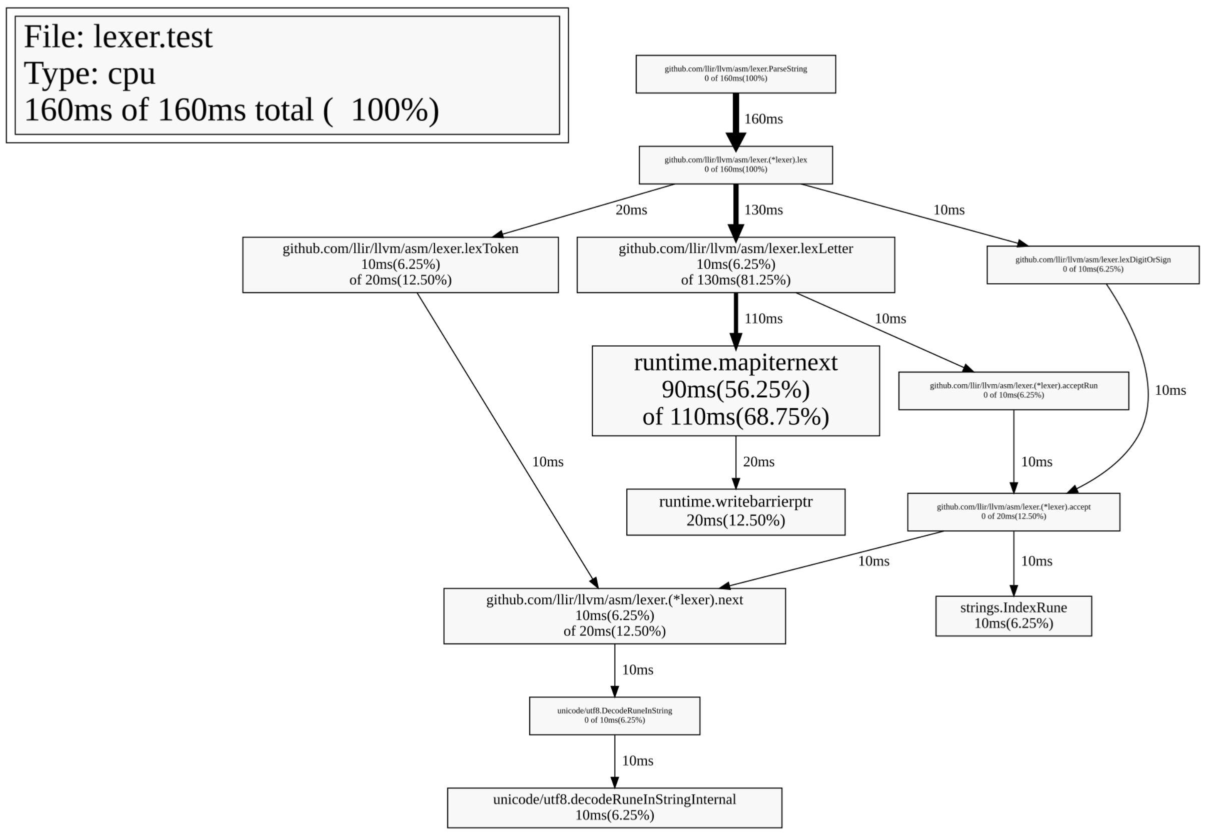

File: lexer.test
Type: cpu
160ms of 160ms total (100%)
github.com/llir/llvm/asm/lexer.ParseString 0 of 160ms(100%)
160ms
github.com/llir/llvm/asm/lexer.(*lexer).lex 0 of 160ms(100%)
20ms
130ms
10ms
github.com/llir/llvm/asm/lexer.lexToken 10ms(6.25%) of 20ms(12.50%)
github.com/llir/llvm/asm/lexer.lexLetter 10ms(6.25%) of 130ms(81.25%)
github.com/llir/llvm/asm/lexer.lexDigitOrSign 0 of 10ms(6.25%)
110ms
10ms
runtime.mapiternext 90ms(56.25%) of 110ms(68.75%)
github.com/llir/llvm/asm/lexer.(*lexer).acceptRun 0 of 10ms(6.25%)
10ms
10ms
20ms
10ms
runtime.writebarrierptr 20ms(12.50%)
github.com/llir/llvm/asm/lexer.(*lexer).accept 0 of 20ms(12.50%)
10ms
10ms
github.com/llir/llvm/asm/lexer.(*lexer).next 10ms(6.25%) of 20ms(12.50%)
strings.IndexRune 10ms(6.25%)
10ms
unicode/utf8.DecodeRuneInString 0 of 10ms(6.25%)
10ms
unicode/utf8.decodeRuneInStringInternal 10ms(6.25%)

Figure 24: A major performance bottleneck was located when profiling the LLVM IR lexer for the first time. Roughly 70% of the total execution time was spent doing hash map iterations (i.e. `runtime.mapiternext`).

8.2.2 Benchmarks

Benchmark tests were implemented to reliably measure any performance changes before trying to resolve performance issues. An updated version of the LLVM IR library used arrays instead of hash maps to identify keywords when scanning letters, which resolved the performance issue identified in section 8.2.1. The updated version of the LLVM IR lexer is roughly 3.6 times faster than the original version, as illustrated in figure 25.

```
$ git checkout old; go test -bench=ParseString > old.txt
$ git checkout new; go test -bench=ParseString > new.txt
$ benchcmp old.txt new.txt
benchmark                old ns/op     new ns/op     delta
BenchmarkParseString     737625        204010        -72.34%
```

Figure 25: Benchmark run time delta between the original and the optimised version of the LLVM IR lexer, as visualised by `benchcmp`[21]. The optimised version is roughly 3.6 times faster than the original version of the lexer.

8.3 Security Assessment

To assess the security of the decompiler pipeline, lets imagine a scenario in which users are given access to the implementation details and source code of the entire system and may provide arbitrary input to any of its components. A potential scenario could involve a web site which provides decompilation as a service and allows its users to interact with the various stages of the decompiler pipeline. The Retargetable Decompiler[22] provides such a service, except it only allows users to interact with the binary analysis stage of the pipeline (see section 2.2.1) and its source code is proprietary. The scope of this security assessment will be limited to the various components of the decompiler pipeline and their interaction. In particular security issues related to the operating system, network stack, web server and web site (e.g. SQL-injection and XSS vulnerabilities) of the decompilation service are intentionally excluded from the scope of the security assessment.

The objective of an attacker may be to escalate their privileges in a system by exploiting it to execute actions not intended by design. Since the security of any system is only as strong as its weakest link, it is critical to identify and isolate likely targets for attacks. Projects which consist of or depend on large C or C++ code bases may exhibit memory safety issues, such as buffer overflows or use-after-free vulnerabilities. These issues are considered low-hanging fruit for attackers and have a long history of successful exploitation [38]. Several modern programming languages (including Go) provide memory safety guarantees and may solve these issues by inserting bounds-checking for array accesses and using garbage collection for memory management. Code written in memory safe languages may still contain other security vulnerabilities caused by logic errors or insufficient validation, sanitation and parametrization of input (e.g. command injection and directory traversal vulnerabilities).

[21]Display performance changes between benchmarks: `https://golang.org/x/tools/cmd/benchcmp`
[22]Retargetable Decompiler: `https://retdec.com/`

The number of lines of code in a project may give an indication to the project's complexity and to some extent its potential attack surface. As summarised in table 8 every component, except `restructure`, of the decompiler pipeline depends on LLVM; the code base of which contains approximately 800 000 lines of C++ source code. Even if only a portion of the code will be linked into the executables it is an interesting target for attacks. One thing to keep in mind is that there are several high-end users of the LLVM project (such as Apple, Intel, NVIDIA and Sony [39]) and it has a well established code reviewing process. Some of the LLVM developers are also very well educated in common security vulnerabilities and have developed the Clang Static Analyzer, which is a static source code analysis tool that locates bugs (such as buffer overflows and use-after-free issues) in C and C++ programs [40]. The LLVM project may contain several security issues due to its size, but they are most likely difficult to discover since the low-hanging fruit have been caught already by the Clang Static Analyzer. Similarly, Google Protocol Buffers are used extensively by several companies and organisations and the likelihood of discovering a simple security vulnerability in its code base is low.

Project	**Language**	**Lines**	**Dependencies**
Front-end			
Dagger	C++	2 000	LLVM
Fracture	C++	20 000	LLVM
MC-Semantics	C++	25 000	LLVM and Google Protocol Buffers
Middle-end			
ll2dot	Go	500	LLVM and dot
restructure	Go	300	graphs and dot
Back-end			
ll2go	Go	1 500	LLVM and llvm (Go)
go-post	Go	3 000	-
Dependencies			
LLVM	C++	800 000	-
Google Protocol Buffers	C++	125 000	-
dot	Go	7 000	-
llvm (Go)	Go	5 000	-
graphs	Go	2 000	-

Table 8: A rough summary of each project specifying their programming language, total number of lines of code and dependencies.

There are still three potential targets which may contain memory related vulnerabilities, namely the front-end projects which translate binary executables, object code and shared libraries to LLVM IR. Insufficient validation during the parsing of headers (e.g. trusting the values of section header fields in ELF files) may lead to security vulnerabilities. The front-end projects rely extensively on parsing logic for the binary analysis stage (see section 2.2.1), and are therefore susceptible to security vulnerabilities.

The security implications of various design decisions have been taken into consideration during the development process of the Go components. The Go runtime guarantees memory safety by inserting bounds-checking for array accesses and using garbage collection for memory management. Furthermore, the Go project focuses on addressing security issues at the language-level rather than relying on security through obscurity (e.g. address space layout randomization) to mitigate these issues at the OS-level, as further explained and justified by the quote of Russ Cox (who works on the Go compiler and runtime) presented in figure 26.

> *"Address space randomization is an OS-level workaround for a language-level problem, namely that simple C programs tend to be full of exploitable buffer overflows. Go fixes this at the language level, with bounds-checked arrays and slices and no dangling pointers, which makes the OS-level workaround much less important. In return, we receive the incredible debuggability of deterministic address space layout. I would not give that up lightly."*

Figure 26: Reply by Russ Cox to a discussion regarding ASLR, on the Go mailing list[23].

There is one know issue with the Go bindings for LLVM IR which may compromise the integrity of the output from the decompilation pipeline. The `ll2dot` and `ll2go` components require access to the names of unnamed basic blocks, but these names are not accessible from the API of the LLVM IR library as they are generated on the fly by the assembly printer. As a work around, the debug facilities of the LLVM IR library have been utilised to print the assembly to temporary files, which are parsed to gain access to the names of unnamed basic blocks. These temporary files may be tampered with, if not sufficiently protected by access permissions, which may compromise the integrity of the control flow analysis stage. A pure Go library for interacting with LLVM IR is being developed (see section 7.2) which will include native support for calculating the names of unnamed basic blocks, thus mitigating this security issue.

To conclude, the security assessment was conducted to identify potential security issues and provide an intuition for the general security of the decompilation system. In scenarios such as the one described above (i.e. users may provide arbitrary input), it is advisable to further harden the decompilation system by utilizing defence in depth (i.e. several independent layers of security) and the principle of least privilege (e.g. use jails in FreeBSD and LXC in Linux).

8.4 Continuous Integration

This project makes use of Travis CI, which is tightly integrated with GitHub, to run a series of automated tests for each commit to the source code repository. The tests are varied and range from identifying source code formatting and coding style issues to monitoring the build status and test coverage. A future ambition is to run benchmarks for each commit to quickly identify performance regressions.

8.4.1 Source Code Formatting

Instead of relying on a formatting style guide the Go project enforces a single formatting style using `gofmt` (a tool similar to `indent`) which automatically formats Go source code. The adoption of this tool is widespread within the Go community as indicated by

[23]Secure Go binaries: `https://groups.google.com/d/msg/golang-nuts/Jd9tlNc6jUE/6dLasv0s4nIJ`

a survey conducted back in 2013. The survey found that 70% of the publicly available Go packages were formatted according to the rules of `gofmt` [41], a figure which is likely to have increased since then.

Using a single formatting style for all Go source code may at first seem like a small deal, but the advantages are vast. It becomes easier to write code as one may focus on the problem at hand rather than minor formatting issues. Similarly it becomes easier to read code when it is formatted in a familiar and uniform manner. Developers may focus their entire attention at understanding the semantics of the code, without being distracted by inconsistent or unfamiliar formatting. And perhaps most importantly, it prevents useless discussions about which formatting style is the right one.

Several text editors supports adding pre-save hooks which executes a command to pre-process the text before saving it. This mechanism may be used with the `gofmt` tool to automatically enforce its formatting style each time a source file is saved. One of the CI tests catches and reports incorrectly formatted source code, should a programmer forget to install such a hook.

8.4.2 Coding Style

Best practices for writing effective Go code have been outlined in the *Effective Go*[24] and *Go Code Review Comments*[25] documents. They supplement the Go language specification and describe a set of idioms and conventions used in idiomatic Go code. Great care has been taken to follow these principles when developing, and a CI test has been integrated which utilises the `golint`[26] tool to automatically detect and report issues related to coding style (e.g. naming, documentation and error message conventions).

8.4.3 Code Correctness

Source code analysis tools may be used to identify common coding mistakes, such as `printf` calls with conflicting arguments and format specifiers. A CI test has been integrated which utilises `go vet`[27] to examine the source code and report suspicious constructs (e.g. unreachable code, invalid `printf` calls).

8.4.4 Build Status

The Go build system relies on a few well-established conventions to remain configuration-free (e.g. no `Makefiles`, or `configure` scripts). These conventions specify how to locate the source code of a package based on its import path (e.g. `github.com/user/repo/pkg`). In combination with explicit import declarations in the source code, these conventions enable Go packages and tools to be built using only the information present in the source files [42].

[24]Effective Go: `https://golang.org/doc/effective_go.html`
[25]Go Code Review Comments: `http://golang.org/wiki/CodeReviewComments`
[26]Golint is a linter for Go source code: `https://github.com/golang/lint`
[27]Vet reports suspicious constructs in Go source code: `https://golang.org/x/tools/cmd/vet`

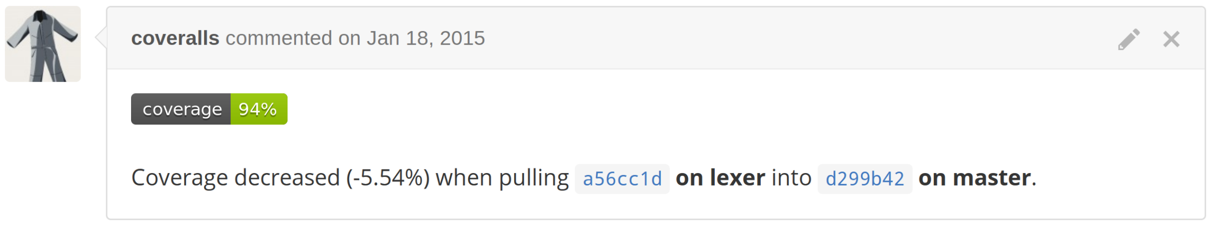
The CI service, which monitors the build status of each component and their dependencies, may leverage the configuration-free build system of Go to simplify its dependency management. Travis CI has been configured to invoke `go get`[28], which recursively downloads and installs the dependencies of each component; the location of which is derived from the source code.

8.4.5 Test Cases

Travis CI has been configured to run the test cases for each component with race detection[29] enabled. In addition to monitoring the status of test cases, this may help identify data race conditions which occur when concurrent code read from and write to the same memory locations.

8.4.6 Code Coverage

Tracking of changes to code coverage has been tightly integrated with the CI. Travis CI has been configured to send code coverage information to the Coveralls[30] service, after each successful build. The Coveralls service tracks changes in code coverage between commits, and reports these changes when merging branches on GitHub; as illustrated in figure 27.


coveralls commented on Jan 18, 2015

coverage 94%

Coverage decreased (-5.54%) when pulling a56cc1d **on lexer** into d299b42 **on master**.

Figure 27: The Coveralls service automatically reports code coverage changes when merging branches on GitHub. In this case the code coverage decreased from 100% to 94% when merging the LLVM IR lexer functionality.

[28]Download and install packages and dependencies: `https://golang.org/cmd/go/`
[29]Introducing the Go Race Detector: `https://blog.golang.org/race-detector`
[30]Coveralls - Test Coverage History & Statistics: `https://coveralls.io/`

9 Evaluation

This section evaluates the artefacts of the decompilation system against the requirements outlined in section 5. To assess the capabilities of the individual components, relevant decompilation scenarios have been considered. The current state of each component is summarised in the succeeding paragraphs, and future work to validate the design, improve the reliability, and extend the capabilities of the decompilation pipeline is presented in section 10.2.

The `ll2dot` component (see section 6.3.1) is considered stable, but there are known issues which may affect the reliability and the integrity of the produced CFGs; as further described in section 8.3. Future work which seeks to address these issues is presented in section 10.2.2.

The subgraph isomorphism search library (see section 7.4) is considered production quality, and the test cases of the `iso`[31] package have a code coverage of 94.8%; as outlined in section 8.1.1. The restrictions imposed by this library on the subgraph (e.g. single-entry/single-exit invariant and fixed number of nodes) limits infinite loops and n-way conditionals from being modelled, as further discussed in section 6.3.2. Section 10.2.1 presents a discussion of potential approaches which may relax these restrictions in the future.

The `restructure` component (see section 6.3.2) is considered production quality, and the test cases of the `restructure` command have a code coverage of 40.0%; as outlined in section 8.1.1. The `restructure` command is believed to be capable of structuring the CFG of any source program which may be constructed from the set of supported high-level control flow primitives (which are described in figure 7 of section 2.2.3), including source programs with arbitrarily nested primitives. Any future work which improves the reliability and the capabilities of the subgraph isomorphism search library will directly impact the `restructure` tool, as it relies entirely on subgraph isomorphism search to recover high-level control flow primitives.

The `ll2go` component (see section 6.4) is considered a *proof of concept* implementation. It was implemented primarily to stress test the design of the decompilation pipeline, and only supports a small subset (e.g. all arithmetic instructions and some terminator instructions) of the LLVM IR language. The `ll2go` tool is affected by the same reliability issues as the `ll2dot` command, which are caused by the Go bindings for LLVM; as further described in section 7.3. To address these issues a pure Go library is being developed for interacting with LLVM IR, as further described in section 10.2.2. A future version of the `ll2go` tool would discard the current implementation and start fresh, learning from the mistakes and the building upon the insights.

Lastly, the `go-post` component (see section 6.4.1) is considered alpha quality, and the test cases of the `go-post` command have a code coverage of 38.0%; as outlined in section 8.1.1. The `go-post` tool was primarily implemented to evaluate the feasibility of applying source code transformations to make the decompiled Go code more idiomatic. Implementing these post-processing rules were surprisingly easy, and it was often possible to go from the conceptual idea of a rewrite rule to a working implementation in a matter of hours. While some rewrite rules are reliable (e.g. the *"mainret"* rewrite rule, which is presented

[31]Subgraph isomorphism search library: `https://decomp.org/decomp/graphs/iso`

in 35 of appendix J), most are considered experimental. For instance, the *"localid"* rewrite rule (see figure 36 of appendix J) is known to produce incorrect rewrites when applied to complex programs, but it works for simple programs and provides rudimentary support for expression propagation. A proper implementation of expression propagation would rely on the future implementation of the data flow analysis component, which is mentioned in 10.2.1.

9.1 LLVM IR Library

In total four essential (**R1**, **R2**, **R3** and **R4**), one desirable (**R5**), and three future (**R6**, **R7** and **R8**) requirements were identified for the LLVM IR library (see section 5.1). The modified Go bindings for LLVM (see section 7.3), the control flow graph generation component (see section 6.3.1) and the `dot`[32] tool of the Graphviz project, collectively satisfies all five requirements (not counting future requirements); as summarised in table 9. Section 9.1.1 and 9.1.2 provides a detailed evaluation of the essential and the desirable requirements, respectively.

Sat.	Req.	Priority	Description
Yes	**R1**	MUST	Read the assembly language representation of LLVM IR
Yes	**R2**	MUST	Write the assembly language representation of LLVM IR
Yes	**R3**	MUST	Interact with an in-memory representation of LLVM IR
Yes	**R4**	MUST	Generate CFGs from LLVM IR basic blocks
Yes	**R5**	COULD	Visualise CFGs using the `DOT` graph description language
N/A	**R6**	WON'T	Read the bitcode representation of LLVM IR
N/A	**R7**	WON'T	Write the bitcode representation of LLVM IR
N/A	**R8**	WON'T	Provide a formal grammar of LLVM IR

Table 9: A summary of the evaluation against requirements of the LLVM IR library, which specifies what requirements (abbreviated as "Req.") that have been satisfied (abbreviated as "Sat.").

9.1.1 Essential Requirements

The modified Go bindings for LLVM (see section 7.3) includes read (**R1**) and write (**R2**) support for the assembly language representation of LLVM IR, and enables interaction with an in-memory representation of LLVM IR (**R3**). The `ll2dot` tool depends on **R1** and **R3** for parsing LLVM IR assembly files and inspecting their in-memory representation, which is required to gain access to information about the basic blocks of each function and their terminating instructions. This information determines the node names and the directed edges, when generating CFGs from LLVM IR; as further described in section 6.3.1. Appendix F demonstrates that the `ll2dot` tool is capable of generating CFGs from LLVM IR (**R4**), thus proving that **R1**, **R3** and **R4** have been satisfied.

To support generating CFGs for LLVM IR assembly which contains unnamed basic blocks, the `ll2dot` tool requires access to the names of unnamed basic blocks. These names are

[32]Drawing Graphs with dot: `http://www.graphviz.org/pdf/dotguide.pdf`

not available from the API of the original Go bindings for LLVM however, as they are generated on the fly by the assembly printer. To work around this issue, the assembly printer of LLVM 3.6 was patched to always print the generated names of unnamed basic blocks (see appendix B). With this patch in place, the debug facilities of the modified Go bindings for LLVM could be utilised to write (**R2**) the assembly to temporary files, which were parsed to gain access to the names of unnamed basic blocks; as further described in section 7.3. The generated CFG presented in appendix F contains the names of unnamed basic blocks (e.g. basic blocks with numeric names), thus proving that **R2** has been satisfied.

9.1.2 Desirable Requirements

The CFGs generated by the `ll2dot` tool (see section 6.3.1) are described in the DOT graph description language. One benefit of expressing CFGs in this format, is that it enables the reuse of existing software to visualise the CFGs (**R5**). Appendix F demonstrates how the `dot` tool of the Graphviz project may be utilised to produce an image representation of CFGs, which are expressed in the DOT file format.

9.2 Control Flow Analysis Library

In total five essential (**R9**, **R10**, **R11**, **R12** and **R13**), two important (**R14** and **R15**), two desirable (**R16** and **R17**), and two future (**R18** and **R19**) requirements were identified for the control flow analysis library (see section 5.2). The current implementation of the subgraph isomorphism search library satisfies six out of nine requirements (not counting future requirements), and fails to satisfy one essential, one important, and one desirable requirement; as summarised in table 10. Section 9.2.1, 9.2.2 and 9.2.3 provides a detailed evaluation of the essential, the important and the desirable requirements, respectively.

Sat.	Req.	Priority	Description
Yes	**R9**	MUST	Support analysis of reducible graphs
Yes	**R10**	MUST	Recover pre-test loops (e.g. `while`)
No	**R11**	MUST	Recover infinite loops (e.g. `while(TRUE)`)
Yes	**R12**	MUST	Recover 1-way conditionals (e.g. `if`)
Yes	**R13**	MUST	Recover 2-way conditionals (e.g. `if-else`)
Yes	**R14**	SHOULD	Recover post-test loops (e.g. `do-while`)
No	**R15**	SHOULD	Recover n-way conditionals (e.g. `switch`)
No	**R16**	COULD	Recover multi-exit loops
Yes	**R17**	COULD	Recover nested loops
N/A	**R18**	WON'T	Support analysis of irreducible graphs
N/A	**R19**	WON'T	Recover compound boolean expressions

Table 10: A summary of the evaluation against requirements of the control flow analysis library, which specifies what requirements (abbreviated as "Req.") that have been satisfied (abbreviated as "Sat.").

9.2.1 Essential Requirements

The current implementation of the subgraph isomorphism search library supports analysis of reducible graphs (**R9**), as demonstrated by the step-by-step analysis of a reducible CFG in appendix G.

The successful recovery of pre-test loops (**R10**) and 1-way conditionals (**R12**) is demonstrated in four steps, through the use of components which depend on the subgraph isomorphism search library. Firstly, the `ll2dot` tool (see section 6.3.1) is used to generate an unstructured CFG for each function of an LLVM IR assembly file; as demonstrated in appendix F. Secondly, the `restructure` tool (see section 6.3.2) analyses the unstructured CFG of an LLVM IR assembly function to produce a structured CFG; as demonstrated in appendix H. Thirdly, the `ll2go` tool (see section 6.4) uses the high-level control flow information of the structured CFG to decompile the LLVM IR function into unpolished Go code; as demonstrated in appendix I. Lastly, the `go-post` tool improves the quality of the unpolished Go code, by applying a set of source code transformations; as demonstrated in appendix J. The final Go output, which is presented on the right side of figure 40 in appendix J, contains both a `for`-loop and an `if`-statement, thus proving that pre-test loops and 1-way conditionals may be recovered.

The successful decompilation of 2-way conditionals (**R13**) is demonstrated in appendix K, which provides a contrived example that implicitly uses the same decompilation steps as described above. The final Go output, which is presented on the right side of figure 41 in appendix K, contains an `if-else` statement, thus proving that 2-way conditionals may be recovered.

The current design of the control flow analysis stage enforces a single-entry/single-exit invariant on the graph representation of high-level control flow primitives. In other words, high-level control flow primitives must be modelled as directed graphs with a single entry and a single exit node. This invariant simplifies the control flow analysis, as it allows identified subgraphs to be merged into single nodes, which inherit the predecessors of the entry node and the successors of the exit node; as demonstrated by the step-by-step simplification of the CFG in appendix G. This restriction prevents infinite loops (**R11**) from being modelled however, as they have no exit node. Future work will try to determine if this invariant may be relaxed to include single-entry/no-exit graphs, as further described in section 10.2.1.

9.2.2 Important Requirements

The successful decompilation of post-test loops (**R14**) is demonstrated in appendix L, which provides a contrived example that implicitly uses the same decompilation steps as described above. The final Go output, which is presented on the right side of figure 42 in appendix L, contains an infinite `for`-loop with a conditional break statement as the last statement of the loop body (which is semantically equivalent to a post-test loop), thus proving that post-test loops may be recovered. Even though Go does not provide native support for post-test loops, the back-end was capable of translating the source program into semantically equivalent Go code, by combining a set of primitives available in Go. The same approach may be used to support missing primitives for other target programming languages (e.g. `switch`-statements in Python).

A data-driven design separates the implementation of the control flow analysis component from the definition of supported high-level control flow primitives, which are expressed in the DOT file format. The design is motivated by the principle of separation of concern (e.g. the control flow analysis may be reused to analyse the control flow of REIL) and extensibility (e.g. support for new high-level control flow primitives may be added without changing the source code), as further described in section 6.3.2. One limitation with this design however, is that it does not support n-way conditionals (**R15**) or any other high-level control flow primitives with a variable number of nodes in their graph representations, as these cannot be expressed in the DOT file format. A discussion on how to mitigate this issue in the future is provided in section 10.2.1.

9.2.3 Desirable Requirements

Implementation strategies for desirable requirements were only considered as time permitted. The support for multi-exit loops (**R16**) was intentionally omitted from this release, to allocate time for the essential requirements. More research is required to determine how the current design of the control flow analysis stage may be refined to support the recovery of multi-exit loops.

The successful decompilation of nested loops (**R17**) is demonstrated in appendix K, which provides a contrived example that implicitly uses the same decompilation steps as described above. The final Go output, which is presented on the right side of figure 41 in appendix K, contains nested `for`-loops (one inner loop and one outer loop), thus proving that nested loops may be recovered.

9.3 Control Flow Recovery Tool

In total two essential (**R20** and **R21**) requirements were identified for the control flow recovery tool (see section 5.3).

The control flow analysis component (see section 6.3.2) satisfies both requirements (not counting future requirements); as summarised in table 11. Section 9.3.1 provides a detailed evaluation of the essential requirements.

Sat.	Req.	Priority	Description
Yes	**R20**	MUST	Identify high-level control flow primitives in LLVM IR
Yes	**R21**	MUST	Support language-agnostic interaction with other components

Table 11: A summary of the evaluation against requirements of the control flow recovery tool, which specifies what requirements (abbreviated as “Req.”) that have been satisfied (abbreviated as “Sat.”).

9.3.1 Essential Requirements

In collaboration, the `ll2dot` and `restructure` tools are capable of identifying high-level control flow primitives in LLVM IR (**R20**). Firstly, the `ll2dot` tool generates CFGs (in the DOT file format) for each function of the LLVM IR, as described in section 6.3.1

and demonstrated in appendix F. Secondly, the `restructure` tool structures the CFGs to recover high-level control-flow primitives from the underlying LLVM IR, as described in section 6.3.2 and demonstrated in appendix H.

The components of the decompilation pipeline support language-agnostic interaction with other components (**R21**), as they only communicate using well-defined input and output; specifically LLVM IR[33], DOT[34], JSON[35] and Go[36] input and output. The interaction between the components of the decompilation pipeline is demonstrated in four steps, when decompilation LLVM IR to Go. Firstly, the control flow graph generation component (see section 6.3.1) parses LLVM IR assembly to produce an unstructured CFG (in the DOT file format); as demonstrated in appendix F. Secondly, the control flow analysis component (see section 6.3.2) analyses the unstructured CFG (in the DOT file format) to produce a structured CFG (in JSON format); as demonstrated in appendix H. Thirdly, the code generation component (see section 6.4) decompiles the structured LLVM IR assembly into unpolished Go code; as demonstrated in appendix I. Lastly, the post-processing tool (see section 6.4.1) improves the quality of the unpolished Go code, by applying a set of source code transformations; as demonstrated in appendix J.

[33]LLVM Language Reference Manual: `http://llvm.org/docs/LangRef.html`
[34]The DOT Language: `http://www.graphviz.org/doc/info/lang.html`
[35]The JSON Data Interchange Format: `https://tools.ietf.org/html/rfc7159`
[36]The Go Programming Language Specification: `https://golang.org/ref/spec`

10 Conclusion

This section concludes the project report with subjective reflections from the author. For the remainder of this section I will switch to a first person narrative.

10.1 Project Summary

Reverse engineering has fascinated me for quite some time and I have experimented with a variety of tools, ranging from disassemblers and decompilers to debuggers and tracers. Many tools have been outstanding on an individual level and some have featured powerful extensibility through plugin and scripting support. I have never been bothered by the capabilities of these tools, but rather by the workflow they enforce.

The de facto tools for binary analysis are monolithic in nature with regards to the end-user, as they do not expose their individual components. Imagine trying to reuse the control flow analysis of IDA and the Hex-Rays Decompiler to implement control flow aware folding support in an IDE for x86 assembly development (e.g. group and toggle assembly code segments based on their corresponding high-level control flow structures). This idea is not limited by any technical issues; IDA Pro and the Hex-Rays Decompiler have support for recovering high-level control flow primitives from x86 assembly. If the IDE was given access to the control flow analysis component it would be trivial to implement sophisticated folding support for x86 assembly.

Having worked extensively within a Unix environment, I have grown accustomed to a workflow that allows you to *combine* individual tools in amazing ways; pipe the input from one tool into another which transforms, massages or interprets the data in a specific way to solve a given task. This background has instilled me with a belief that the decompilation workflow could be facilitated by implementing a decompilation pipeline composed of independent and reusable components. Throughout the course of this project several independent components have been implemented, including a control flow graph generation tool which generates CFGs (in the DOT file format) from LLVM IR assembly, a control flow recovery tool which identifies high-level control flow primitives in CFGs, a code generation tool which translates structured LLVM IR assembly into unpolished Go source code, and a post-processing tool which polishes Go source code to make it more idiomatic.

These components have been combined with open source tools from other projects to form the foundation of a decompilation pipeline, which may translate a variety of source languages into the Go programming language. The decompilation pipeline has been proven capable of recovering nested pre-test and post-test loops (e.g. `for` and `do-while` loops), and 1-way and 2-way conditionals (e.g. `if` and `if-else` statements) from LLVM IR assembly. While the project has succeeded at implementing a *proof of concept* decompilation pipeline, creating the One True decompilation pipeline has been considered a non-goal [43]. The aim of the project has always been to explore the feasibility and potential of a decompilation pipeline composed of independent and reusable components which interact through well-defined input and output.

To conclude, the project has demonstrated the feasibility of compositional decompilation through the implementation of a *proof of concept* decompilation pipeline which exposes its components to the end-user. The true potential of this approach is still being evaluated. It largely depends on the innovativeness of the end-users, as the design encourages end-users to replace, refine and interact with each stage of the decompilation pipeline, and to use the self-contained components for other purposes not yet envisioned by the component authors. Future plans for stress testing the design are detailed in section 10.2.

10.2 Future Work

The primary focus for planned future work is to stress test the design of the decompilation pipeline and its individual components. A secondary focus is to improve the quality and the reliability of the components. A tertiary focus is to extend the capabilities of the decompilation pipeline. This prioritization strives to validate the core of the system before extending it.

10.2.1 Design Validation

The principle of separation of concern has influenced every aspect of the design of the decompilation pipeline and its individual components. Conceptually, the components of the decompilation pipeline are grouped into three modules which separate concerns regarding the source language (front-end module), the general decompilation tasks (middle-end module), and the target language (back-end module). This conceptual separation is a vital aspect of the decompilation pipeline design, and it will therefore be thoroughly examined. Should a component violate the principle of separation of concern, either in isolation or within the system as a whole, it must be redesigned or reimplemented. To identify such issues, key areas of the decompilation pipeline will be extended to put pressure on the design.

Firstly, an additional back-end (e.g. support for Python as a target language) will be implemented to put pressure on the design of the middle-end module. The second back-end would only be able to leverage the target-independent information of the general decompilation tasks (e.g. control flow analysis) if the middle-end module was implemented correctly.

Secondly, a key component (e.g. data flow analysis) will be implemented in a separate programming language (e.g. Haskell, Rust, Prolog, . . .) to validate the language-agnostic aspects of the design. This component would only be able to interact with the rest of the decompilation pipeline, through well-defined input and output (e.g. LLVM IR, JSON, DOT, . . .), if the other components were implemented correctly.

The separation of the front-end and the middle-end has already been validated. These modules are only interacting through an intermediate representation (i.e. LLVM IR), and a variety of source languages are already supported by the front-end module which consists of components from several independent open source project (e.g. Dagger, Fracture, MC-Semantic, Clang, . . .).

The design of the control flow analysis component has both advantages and limitations, as discussed in section 6.3.2. The most significant limitation is the lack of support for control flow primitives with a variable number of nodes in their graph representations (e.g. n-way conditionals). To gain a better understanding of this issue, an analysis of control flow primitives from different high-level languages will be conducted. Should the n-node control flow primitives prove to be rare, hard-coded support for n-way conditionals (e.g. `switch`-statements) and similar control flow primitives would suffice. Otherwise, a general solution to the problem will be required (such as the introduction of a domain specific language which describes dynamic properties of the nodes and edges in DOT files). The OpenCL decompiler presented by S. Moll solved this problem by converting n-way conditionals into sets of 2-way conditionals [10].

As described in section 9.2, the current implementation of the control flow analysis component enforces a single-entry/single-exit invariant on the graph representations of high-level control flow primitives. This invariant prevents the recovery of infinite loops, as their graph representation has no exit node. At this stage it is unclear whether a refined implementation may relax the invariant to support single-entry/no-exit graphs, or if the limitation is inherent to the design. This issue requires further investigation before a potential solution may be proposed.

10.2.2 Reliability Improvements

As described in section 7.3, there are many reliability issues caused by the Go bindings for LLVM. To mitigate these issues a pure Go library is being developed for interacting with LLVM IR (see section 7.2). This library will be reusable by other projects, and the requirements of the third-party Go compiler `llgo`[37] are actively being tracked[38].

To ensure reliable interoperability between components written in different programming languages, the intermediate representation (i.e. LLVM IR) of the decompilation pipeline must be well-defined. Previous efforts to produce a formal grammar for LLVM IR have only focused on subsets of the language (as mentioned in section 5.1), and no such grammar has been officially endorsed. Producing an official formal specification for LLVM IR would require huge efforts, but it would enable interesting opportunities. For instance, with a formal grammar it would be possible to create a tool which automatically generates grammatically correct LLVM IR assembly which may be used to verify the various implementation of LLVM IR. This approach has been used by the GoSmith tool to generate random, but legal, Go programs which have uncovered 31 bugs in the official Go compiler, 18 bugs in the Gccgo compiler, 5 bugs in the `llgo` compiler, and 3 bugs in the Go language specification [44].

10.2.3 Extended Capabilities

The official Go compiler was automatically translated from C to Go in preparation for the 1.5 release (to be released in August 2015) [45]. To make the C-style Go source code more idiomatic, Russ Cox wrote a tool called `grind`[39] which moves variable declarations closer to their usage. This tool is a perfect fit for the decompilation pipeline, and may be used as-is to extend the post-processing stage of the Go back-end.

[37]LLVM-based compiler for Go: `https://llvm.org/svn/llvm-project/llgo/trunk/README.TXT`

[38]Requirements · Issue #3: `https://github.com/llir/llvm/issues/3`

[39]Grind polishes Go programs: `https://github.com/rsc/grind`

In the far future, a type analysis component will be implement to support type recovery during decompilation. As type analysis requires type constraints equations to be solved, the component will be implemented in a language with good support support for constraints programming (e.g. Prolog). At this stage, more research is required to determine how generic type inference algorithms (e.g. Algorithm W [46]) may influence the design.

10.3 Personal Development

This is the largest project I have undertaken in my life and I feel satisfied with the outcome and proud of what I have been able to accomplished. It has re-enforced my belief that any problem is solvable when broken into smaller subproblems and instilled me with a feeling that anything is possible. The project has allowed me to mature as a software engineer and I now feel more confident in utilizing best practices such as test-driven development and CI. I have also matured as a developer and gained experience with implementing a semi-large project and structuring it into several smaller self-contained projects.

If I were to redo the project today, I wish someone would have told me about the critical importance of effective time management, and then told me again to let it sink in. In the beginning of the project I used coarse time management and assigned bi-weekly deadlines to larger tasks. While some tasks were completed ahead of schedule, many were postponed. Later on it became apparent that accurate time estimates could only be given when larger tasks were broken down into sufficiently small sub-tasks. Towards the end of the project I used more granular time management, prioritised tasks, and split larger tasks into sub-tasks that could be completed in a couple of days. For software development tasks the former technique (coarse time management) worked well and for report writing tasks the latter technique (granular time management) was most effective.

10.4 Final Thoughts

My happiest moment during the project was when the larger components started working and could be connected to form a complete system. It feels great having started out with a vague idea of how the decompiler could work, gradually gaining new insights and refining its design after researching and building on the knowledge of others, developing and iteratively reimplementing the various components until they felt just right, finally arriving at a working prototype and seeing the full system in action! If there is one key idea I want to leave you with it is that the composition of independent components, each with a single purpose and well-defined input and output, is a powerful concept for solving complex problems.

References

[1] C. Palahniuk, *Survivor.* W. W. Norton, 1999.

[2] C. Cifuentes, *Reverse Compilation Techniques.* PhD thesis, Queensland University of Technology, 1994. [Online] Available: `http://www.ci.tuwien.ac.at/~grill/decompilation_thesis.pdf`. [Accessed: 22 Feb 2016].

[3] E. W. Dijkstra, "The humble programmer." ACM Turing Lecture 1972, 1972. [Online] Available: `http://dl.acm.org/ft_gateway.cfm?id=1283927&type=pdf`. [Accessed: 22 Feb 2016].

[4] E. Youngdale, "The ELF object file format by dissection," *Linux Journal*, vol. 13, 1995. [Online] Available: `http://www.linuxjournal.com/article/1060`. [Accessed: 22 Feb 2016].

[5] I. Intel, "Intel 64 and IA-32 architectures software developer's manual," 2015. [Online] Available: `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`. [Accessed: 22 Feb 2016].

[6] I. ARM, "ARMv8-A architecture reference manual," 2015. [Online] Available: `https://silver.arm.com/download/download.tm?pv=2113558`. [Accessed: 22 Feb 2016].

[7] I. ARM, "Harvard vs. Von Neumann." ARM home page, 2009. [Online] Available: `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka11516.html`. [Accessed: 22 Feb 2016].

[8] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instruction semantics to LLVM." Talk given at ReCON 2014, 2014. [Online] Available: `http://www.trailofbits.com/resources/McSema.pdf`. [Accessed: 22 Feb 2016].

[9] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[10] S. Moll, "Decompilation of LLVM IR," BSc thesis, Saarland University, 2011. [Online] Available: `http://www.cdl.uni-saarland.de/publications/theses/moll_bsc.pdf`. [Accessed: 22 Feb 2016].

[11] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," in *Proceedings of the VLDB Endowment*, vol. 6, pp. 133–144, VLDB Endowment, 2012. [Online] Available: `http://db.disi.unitn.eu/pages/VLDBProgram/pdf/research/p185-han.pdf`. [Accessed: 22 Feb 2016].

[12] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," *Proceeding of CanSecWest*, 2009. [Online] Available: `http://www.zynamics.com/downloads/csw09.pdf`. [Accessed: 22 Feb 2016].

[13] "REIL - the reverse engineering intermediate language." [Online] Available: `http://www.zynamics.com/binnavi/manual/html/reil_language.htm`. [Accessed: 22 Feb 2016].

[14] "Open source library that implements translator and tools for REIL (reverse engineering intermediate language)." GitHub repository. [Online] Available: `https://github.com/Cr4sh/openreil`. [Accessed: 22 Feb 2016].

[15] C. Lattner, "LLVM," *The Architecture of Open Source Applications*, 2011. [Online] Available: `http://www.aosabook.org/en/llvm.html`. [Accessed: 22 Feb 2016].

[16] "MIPS instruction reference," 1998. [Online] Available: `http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html`. [Accessed: 22 Feb 2016].

[17] "LLVM language reference manual." [Online] Available: `http://llvm.org/docs/LangRef.html`. [Accessed: 22 Feb 2016].

[18] M. J. Van Emmerik, *Static Single Assignment for Decompilation.* PhD thesis, The University of Queensland, 2007. [Online] Available: `http://www.backerstreet.com/decompiler/vanEmmerik_ssa.pdf`. [Accessed: 22 Feb 2016].

[19] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna, "Design of a retargetable decompiler for a static platform-independent malware analysis," in *Information Security and Assurance*, pp. 72–86, Springer, 2011. [Online] Available: `http://www.sersc.org/journals/IJSIA/vol5_no4_2011/8.pdf`. [Accessed: 22 Feb 2016].

[20] R. T. C. III, "Fracture: Inverting llvm's target independent code generator." LLVM Developer's Conference 2013, 2013. [Online] Available: `http://llvm.org/devmtg/2013-11/slides/Carback-Poster.pdf`. [Accessed: 22 Feb 2016].

[21] "Dagger - the DC layer." The Dagger home page, 2014. [Online] Available: `http://dagger.repzret.org/overview-dc/`. [Accessed: 22 Feb 2016].

[22] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, p. 16, 2013. [Online] Available: `http://users.ece.cmu.edu/~ejschwar/papers/usenix13.pdf`. [Accessed: 22 Feb 2016].

[23] I. Guilfanov, "Decompilers and beyond," *Black Hat USA*, 2008. [Online] Available: `https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf`. [Accessed: 22 Feb 2016].

[24] A. M. Davis, "Operational prototyping: A new development approach," *Software, IEEE*, vol. 9, no. 5, pp. 70–78, 1992. [Online] Available: `http://www-di.inf.puc-rio.br/~karin/pos/davis.pdf`. [Accessed: 22 Feb 2016].

[25] K. Beck, *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[26] K. Brennan *et al.*, *A Guide to the Business Analysis Body of Knowledger.* Iiba, 2009.

[27] R. Kotler, “A formal specification for LLVM assembly language.” Google Code repository, 2011. [Online] Available: `https://github.com/decomp-mirror/llvm-assembly-language-formal-specification`. [Accessed: 22 Feb 2016].

[28] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Formalizing the LLVM intermediate representation for verified program transformations,” *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 427–440, 2012. [Online] Available: `http://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf`. [Accessed: 22 Feb 2016].

[29] D. Cheney, “Simplicity and collaboration,” in *Proceedings of Gophercon India*, 2015. [Online] Available: `http://dave.cheney.net/2015/03/08/simplicity-and-collaboration`. [Accessed: 22 Feb 2016].

[30] E. S. Raymond, *The art of Unix programming.* Addison-Wesley Professional, 2003. [Online] Available: `http://www.faqs.org/docs/artu/ch01s06.html`. [Accessed: 22 Feb 2016].

[31] D. Novillo, “GCC internals,” in *International Symposium on Code Generation and Optimization (CGO), San Jose, California*, 2007. [Online] Available: `http://www.airs.com/dnovillo/200711-GCC-Internals/200711-GCC-Internals-1-condensed.pdf`. [Accessed: 22 Feb 2016].

[32] H. G. Mayer, *Advanced C programming on the IBM PC.* Windcrest, 1989.

[33] R. Pike, “Less is exponentially more.” Blog post, 2012. [Online] Available: `http://commandcenter.blogspot.co.uk/2012/06/less-is-exponentially-more.html`. [Accessed: 22 Feb 2016].

[34] R. Pike, “Lexical scanning in go.” Talk given at Google Technology User Group in 2011. [Online] Available: `https://www.youtube.com/watch?v=HxaD_trXwRE`. [Accessed: 22 Feb 2016].

[35] J. N. Buxton and B. Randell, *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee; available from Scientific Affairs Division, NATO, 1970. [Online] Available: `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`. [Accessed: 22 Feb 2016].

[36] R. Pike, “The cover story.” The Go Blog. [Online] Available: `https://blog.golang.org/cover`. [Accessed: 22 Feb 2016].

[37] P. Foggia, C. Sansone, and M. Vento, “A performance comparison of five algorithms for graph isomorphism,” in *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188–199, 2001. [Online] Available: `http://www.researchgate.net/profile/Maximo_Vento/publication/228854771_A_performance_comparison_of_five_algorithms_for_graph_isomorphism/links/0fcfd50acf6509462c000000.pdf`. [Accessed: 22 Feb 2016].

[38] A. One, “Smashing the stack for fun and profit,” *Phrack Magazine*, vol. 49, 1996. [Online] Available: `http://phrack.org/issues/49/14.html`. [Accessed: 22 Feb 2016].

[39] “LLVM users.” LLVM home page. [Online] Available: `http://llvm.org/Users.html`. [Accessed: 22 Feb 2016].

[40] "Clang static analyzer." LLVM home page. [Online] Available: `http://clang-analyzer.llvm.org/`. [Accessed: 22 Feb 2016].

[41] A. Gerrand, "go fmt your code." The Go Blog. [Online] Available: `https://blog.golang.org/go-fmt-your-code`. [Accessed: 22 Feb 2016].

[42] "About the go command." The Go home page. [Online] Available: `https://golang.org/doc/articles/go_command.html`. [Accessed: 22 Feb 2016].

[43] T. Gray, "Goals, non-goals, and anti-goals," 2013. [Online] Available: `https://docs.google.com/document/d/1Y3Q1ySsHxNOk_WaPcQhROYXwJQjZ4wPip_w69VB9Eb8/edit`. [Accessed: 22 Feb 2016].

[44] D. Vyukov, "GoSmith - random go program generator." Google Code repository. [Online] Available: `https://github.com/dvyukov/gosmith`. [Accessed: 22 Feb 2016].

[45] R. Cox, "Go 1.3+ compiler overhaul." Google Docs. [Online] Available: `https://golang.org/s/go13compiler`. [Accessed: 22 Feb 2016].

[46] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, ACM, 1982.

Appendices

A The REIL Instruction Set

Listing 3: A full definition of the REIL instruction set.

```
; === [ Arithmetic instructions ] ===

;     ADD (Addition)
;        Syntax: add op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 + op2
         add      (t1, b4), (t2, b4), (t3, b8) ; t3 = t1 + t2

;     SUB (Subtraction)
;        Syntax: sub op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 - op2
         sub      (t1, b4), (42, b4), (t2, b8) ; t2 = t1 - 42

;     MUL (Unsigned multiplication)
;        Syntax: mul op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 * op2
         mul      (37, b4), (t1, b4), (t2, b8) ; t2 = 37 * t1

;     DIV (Unsigned division)
;        Syntax: div op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 / op2
         div      (4000, b4), (20, b4), (t1, b4) ; t1 = 4000 / 20

;     MOD (Unsigned modulo)
;        Syntax: mod op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 % op2
         mod      (t1, b4), (3, b4), (eax, b4) ; eax = t1 % 3

;     BSH (Logical shift)
;        Syntax: bsh op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 << op2   (right shifts if op2 < 0)
         bsh      (ebx, b4), (-4, b4), (t1, b8) ; t1 = ebx >> 4

; === [ Bitwise instructions ] ===

;     AND (Bitwise AND)
;        Syntax: and op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 & op2
         and     (t1, b4), (t2, b4), (t3, b4) ; t3 = t1 & t2

;     OR (Bitwise OR)
;        Syntax: or op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 | op2
         or      (t1, b4), (4, b4), (t2, b4) ; t2 = t1 | 4

;     XOR (Bitwise XOR)
;        Syntax: xor op1, op2, dst
;           op1: int or reg
;           op2: int or reg
;           dst: reg
;        Semantics: dst = op1 ^ op2
         xor     (0, b4), (eax, b4), (t1, b4) ; t1 = 0 ^ eax

; === [ Data transfer instructions ] ===

;     LDM (Load from memory)
;        Syntax: ldm op1, , dst
;           op1: int or reg
;           op2: empty
;           dst: reg
;        Semantics: dst = mem[op1]   (load a value the size of dst from↩
    memory)
         ldm     (16392, b4), , (t1, b2) ; t1 = *(uint16 *)mem[0x4008]

;     STM (Store to memory)
;        Syntax: stm op1, , dst
;           op1: int or reg
;           op2: empty
;           dst: int or reg
;        Semantics: mem[dst] = op1   (store a value the size of op1 to ↩
   memory)
         stm     (t1, b8), , (16392, b4) ; *(uint64 *)mem[0x4008] = t1

;     STR (Store to register)
;        Syntax: str op1, , dst
;           op1: int or reg
;           op2: empty
;           dst: reg
;        Semantics: dst = op1
         str     (12, b4), , (t1, b4) ; t1 = 12

; === [ Conditional instructions ] ===

;     BISZ (Boolean is zero)
;        Syntax: bisz op1, , dst
;           op1: int or reg
;          op2: empty
;          dst: reg
;       Semantics: dst = (op1 == 0)
        bisz    (t1, b4), , (t2, b1) ; t2 = (t1 == 0)

;    JCC (Conditional jump)
;       Syntax: jcc op1, , dst
;          op1: int or reg
;          op2: empty
;          dst: subaddr
;       Semantics: if (op1 != 0) goto dst
        jcc     (t1, b4), , (11008, b1) ; if (t1 != 0) goto 0x2B00

; === [ Other instructions ] ===

;    UNDEF (Undefine a register)
;       Syntax: undef , , dst
;          op1: empty
;          op2: empty
;          dst: reg
;       Semantics: // Undefine dst, its value is now unknown
        undef   , , (t1, b4) ; Undefine t1, its value is now unknown

;    UNKN (Unknown instruction)
;       Syntax: unkn , ,
;          op1: empty
;          op2: empty
;          dst: empty
;       Semantics: // Unknown instruction from source architecture
        unkn    , , ; Unknown instruction from source architecture

;    NOP (No operation)
;       Syntax: nop , ,
;          op1: empty
;          op2: empty
;          dst: empty
;       Semantics: // No operation
        nop     , , ; No operation
```

B Patch for Unnamed Basic Blocks of LLVM

The following patch ensures that the assembly printer of LLVM 3.6.0 always prints the generated names of unnamed basic blocks.

Listing 4: Always print the generated names of unnamed basic blocks.

```
diff --git a/lib/IR/AsmWriter.cpp b/lib/IR/AsmWriter.cpp
index c494d6c..1a96956 100644
--- a/lib/IR/AsmWriter.cpp
+++ b/lib/IR/AsmWriter.cpp
@@ -2025,13 +2025,14 @@ void AssemblyWriter::printBasicBlock(const ↩
   BasicBlock *BB) {
     Out << "\n";
     PrintLLVMName(Out, BB->getName(), LabelPrefix);
     Out << ':';
-  } else if (!BB->use_empty()) {      // Don't print block # of no ↩
   uses...
-    Out << "\n; <label>:";
+  } else {      // Print block # for unnamed basic blocks.
+    Out << "\n";
     int Slot = Machine.getLocalSlot(BB);
     if (Slot != -1)
       Out << Slot;
     else
       Out << "<badref>";
+    Out << ':';
   }

   if (!BB->getParent()) {
```

C Dagger Example

To demonstrate the capabilities of Dagger, the relocatable object code of a simple *"hello world"* C program was analysed. For reference, the disassembly of the relocatable object code is presented in listing 5. Firstly, the `llvm-dec` tool parsed the relocatable object code (i.e. Mach-o file), disassembled its machine instructions (i.e. x86-64 assembly) and converted each native instruction to semantically equivalent, unoptimised LLVM IR; the listing of which was omitted for the sake of brevity. Lastly, the optimiser of the LLVM compiler framework analysed the unoptimised LLVM IR to produce an optimised version, which is presented in listing 6.

Listing 5: Disassembly of the relocatable object code which was produced from a simple *"hello world"* C program.

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        lea     rax, hello
        mov     [rbp-4], 0
        mov     [rbp-8], edi
        mov     [rbp-16], rsi
        mov     rdi, rax
        mov     al, 0
        call    printf
        mov     ecx, 0
        mov     [rbp-20], eax
        mov     eax, ecx
        add     rsp, 32
        pop     rbp
        ret

hello:
        db 'hello world',10,0
```

Listing 6: Decompiled LLVM IR, which was produced by Dagger when analysing the relocatable object code which contained the disassembly presented in listing 5.

```
; ModuleID = 'hello.ll'

%regset = type { i16, i16, i32, i16, i16, i16, i16, i64, i64, i64, i64, i64, i64, i64, i64, i64, i16, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i32, i32, i32, i32, i32, i32, i32, i32, i80, i80, i80, i80, i80, i80, i80, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i80, i80, i80, i80, i80, i80, i80, i80, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512 }

define void @fn_0(%regset* noalias nocapture) {
entry_fn_0:
  %RIP_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 13
  %RBP_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 8
  %RBP_init = load i64, i64* %RBP_ptr, align 4
```

```
%RSP_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 ↩
    15
%RSP_init = load i64, i64* %RSP_ptr, align 4
%EFLAGS_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, ↩
    i32 2
%EFLAGS_init = load i32, i32* %EFLAGS_ptr, align 4
%RAX_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 7
%RDI_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 ↩
    11
%RDI_init = load i64, i64* %RDI_ptr, align 4
%RSI_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 ↩
    14
%RSI_init = load i64, i64* %RSI_ptr, align 4
%RCX_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 ↩
    10
%RCX_init = load i64, i64* %RCX_ptr, align 4
%1 = add i64 %RSP_init, -8
%2 = inttoptr i64 %1 to i64*
store i64 %RBP_init, i64* %2, align 4
%RSP_2 = add i64 %RSP_init, -40
%3 = add i64 %RSP_init, -12
%4 = inttoptr i64 %3 to i32*
store i32 0, i32* %4, align 4
%EDI_0 = trunc i64 %RDI_init to i32
%5 = add i64 %RSP_init, -16
%6 = inttoptr i64 %5 to i32*
store i32 %EDI_0, i32* %6, align 4
%7 = add i64 %RSP_init, -24
%8 = inttoptr i64 %7 to i64*
store i64 %RSI_init, i64* %8, align 4
%RSP_3 = add i64 %RSP_init, -48
%9 = inttoptr i64 %RSP_3 to i64*
store i64 39, i64* %9, align 4
%ZF_0 = icmp eq i64 %RSP_2, 0
%10 = tail call { i64, i1 } @llvm.ssub.with.overflow.i64(i64 %1, i64↩
    32)
%OF_0 = extractvalue { i64, i1 } %10, 1
%11 = tail call { i64, i1 } @llvm.usub.with.overflow.i64(i64 %1, i64↩
    32)
%CF_0 = extractvalue { i64, i1 } %11, 1
%12 = trunc i64 %RSP_2 to i8
%13 = tail call i8 @llvm.ctpop.i8(i8 %12)
%14 = and i8 %13, 1
%15 = and i32 %EFLAGS_init, -2262
%16 = zext i1 %CF_0 to i32
%17 = xor i8 %14, 1
%18 = zext i8 %17 to i32
%19 = shl nuw nsw i32 %18, 2
%20 = zext i1 %ZF_0 to i32
%21 = shl nuw nsw i32 %20, 6
%22 = lshr i64 %RSP_2, 56
%.tr = trunc i64 %22 to i32
%23 = and i32 %.tr, 128
%24 = zext i1 %OF_0 to i32
%25 = shl nuw nsw i32 %24, 11
%26 = or i32 %21, %15
%27 = or i32 %26, %23
%28 = or i32 %27, %16
```

```
%29 = or i32 %28, %25
%EFLAGS_1 = or i32 %29, %19
store i32 %EFLAGS_1, i32* %EFLAGS_ptr, align 4
store i64 0, i64* %RAX_ptr, align 4
store i64 %1, i64* %RBP_ptr, align 4
store i64 %RCX_init, i64* %RCX_ptr, align 4
store i64 15, i64* %RDI_ptr, align 4
store i64 39, i64* %RIP_ptr, align 4
store i64 %RSI_init, i64* %RSI_ptr, align 4
store i64 %RSP_3, i64* %RSP_ptr, align 4
tail call void @fn_27(%regset* %0)
%EFLAGS_4 = load i32, i32* %EFLAGS_ptr, align 4
%RAX_3 = load i64, i64* %RAX_ptr, align 4
%EAX_4 = trunc i64 %RAX_3 to i32
%RBP_3 = load i64, i64* %RBP_ptr, align 4
%RDI_1 = load i64, i64* %RDI_ptr, align 4
%RSI_1 = load i64, i64* %RSI_ptr, align 4
%RSP_8 = load i64, i64* %RSP_ptr, align 4
%30 = add i64 %RBP_3, -20
%31 = inttoptr i64 %30 to i32*
store i32 %EAX_4, i32* %31, align 4
%RSP_5 = add i64 %RSP_8, 32
%RSP_6 = add i64 %RSP_8, 40
%32 = inttoptr i64 %RSP_5 to i64*
%RBP_2 = load i64, i64* %32, align 4
%RSP_7 = add i64 %RSP_8, 48
%33 = inttoptr i64 %RSP_6 to i64*
%RIP_18 = load i64, i64* %33, align 4
%ZF_1 = icmp eq i64 %RSP_5, 0
%34 = tail call { i64, i1 } @llvm.sadd.with.overflow.i64(i64 %RSP_8,↩
    i64 32)
%OF_1 = extractvalue { i64, i1 } %34, 1
%CF_1 = icmp ugt i64 %RSP_8, -33
%35 = trunc i64 %RSP_5 to i8
%36 = tail call i8 @llvm.ctpop.i8(i8 %35)
%37 = and i8 %36, 1
%38 = and i32 %EFLAGS_4, -2262
%39 = zext i1 %CF_1 to i32
%40 = or i32 %39, %38
%41 = xor i8 %37, 1
%42 = zext i8 %41 to i32
%43 = shl nuw nsw i32 %42, 2
%44 = zext i1 %ZF_1 to i32
%45 = shl nuw nsw i32 %44, 6
%46 = lshr i64 %RSP_5, 56
%.tr20 = trunc i64 %46 to i32
%47 = and i32 %.tr20, 128
%48 = zext i1 %OF_1 to i32
%49 = shl nuw nsw i32 %48, 11
%50 = or i32 %40, %45
%51 = or i32 %50, %47
%52 = or i32 %51, %49
%EFLAGS_3 = or i32 %52, %43
store i32 %EFLAGS_3, i32* %EFLAGS_ptr, align 4
store i64 0, i64* %RAX_ptr, align 4
store i64 %RBP_2, i64* %RBP_ptr, align 4
store i64 0, i64* %RCX_ptr, align 4
store i64 %RDI_1, i64* %RDI_ptr, align 4
```

```
  store i64 %RIP_18, i64* %RIP_ptr, align 4
  store i64 %RSI_1, i64* %RSI_ptr, align 4
  store i64 %RSP_7, i64* %RSP_ptr, align 4
  ret void
}

declare void @fn_27(%regset*)

; Function Attrs: nounwind readnone
declare { i64, i1 } @llvm.ssub.with.overflow.i64(i64, i64) #0

; Function Attrs: nounwind readnone
declare { i64, i1 } @llvm.usub.with.overflow.i64(i64, i64) #0

; Function Attrs: nounwind readnone
declare i8 @llvm.ctpop.i8(i8) #0

; Function Attrs: nounwind readnone
declare { i64, i1 } @llvm.sadd.with.overflow.i64(i64, i64) #0

define i32 @main(i32, i8**) {
  %3 = alloca %regset, align 8
  %4 = alloca [8192 x i8], align 1
  %5 = ptrtoint [8192 x i8]* %4 to i64
  %6 = add i64 %5, 8184
  %7 = inttoptr i64 %6 to i64*
  store i64 -1, i64* %7, align 4
  %8 = getelementptr inbounds %regset, %regset* %3, i64 0, i32 15
  store i64 %6, i64* %8, align 8
  %9 = getelementptr inbounds %regset, %regset* %3, i64 0, i32 11
  %10 = zext i32 %0 to i64
  store i64 %10, i64* %9, align 8
  %11 = getelementptr inbounds %regset, %regset* %3, i64 0, i32 14
  %12 = ptrtoint i8** %1 to i64
  store i64 %12, i64* %11, align 8
  %13 = getelementptr inbounds %regset, %regset* %3, i64 0, i32 2
  store i32 514, i32* %13, align 4
  call void @fn_0(%regset* %3)
  %14 = getelementptr inbounds %regset, %regset* %3, i64 0, i32 7
  %15 = load i64, i64* %14, align 8
  %16 = trunc i64 %15 to i32
  ret i32 %16
}

; Function Attrs: nounwind
define void @main_init_regset(%regset* nocapture, i8*, i32, i32, i8**)↩
    #1 {
  %6 = ptrtoint i8* %1 to i64
  %7 = zext i32 %2 to i64
  %8 = add i64 %6, -8
  %9 = add i64 %8, %7
  %10 = inttoptr i64 %9 to i64*
  store i64 -1, i64* %10, align 4
  %11 = getelementptr inbounds %regset, %regset* %0, i64 0, i32 15
  store i64 %9, i64* %11, align 4
  %12 = getelementptr inbounds %regset, %regset* %0, i64 0, i32 11
  %13 = zext i32 %3 to i64
  store i64 %13, i64* %12, align 4
  %14 = getelementptr inbounds %regset, %regset* %0, i64 0, i32 14
  %15 = ptrtoint i8** %4 to i64
  store i64 %15, i64* %14, align 4
  %16 = getelementptr inbounds %regset, %regset* %0, i64 0, i32 2
  store i32 514, i32* %16, align 4
  ret void
}

; Function Attrs: nounwind readonly
define i32 @main_fini_regset(%regset* nocapture readonly) #2 {
  %2 = getelementptr inbounds %regset, %regset* %0, i64 0, i32 7
  %3 = load i64, i64* %2, align 4
  %4 = trunc i64 %3 to i32
  ret i32 %4
}

attributes #0 = { nounwind readnone }
attributes #1 = { nounwind }
attributes #2 = { nounwind readonly }
```

D MC-Semantics Example

As described in section 3.1.2, MC-Semantics consists of two components which collectively decompile native code into LLVM IR. To demonstrate the capabilities of MC-Semantics, the relocatable object code of a simple C program (see listing 10) was analysed. For reference, the disassembly of the relocatable object code is presented in listing 7. Firstly, the control flow recovery component parsed the relocatable object code (i.e. ELF file) and disassembled its machine instructions (i.e. x86 assembly) to produce a serialized CFG (in the Google Protocol Buffer format), an extract of which is presented in listing 8. Secondly, the instruction translation component converted the native instruction of the serialized CFG into semantically equivalent, unoptimised LLVM IR; the listing of which was omitted for the sake of brevity. Lastly, the optimiser of the LLVM compiler framework analysed the unoptimised LLVM IR to produce an optimised version, which is presented in listing 9.

Listing 7: Disassembly of the relocatable object code which was produced from the C source code presented in listing 10.

```
main:
        push    ebp
        mov     ebp, esp
        sub     esp, 20
        mov     eax, [ebp+12]
        mov     ecx, [ebp+8]
        mov     dword [ebp-4], 0
        mov     [ebp-8], ecx
        mov     [ebp-12], eax
        mov     dword [ebp-20], 0
        mov     dword [ebp-16], 0

  loc_loop_cond:
        cmp     dword [ebp-16], 10
        jge     loc_ret
        cmp     dword [ebp-20], 100
        jge     loc_skip_if_body
        imul    eax, [ebp-16], 3
        add     eax, [ebp-20]
        mov     [ebp-20], eax

  loc_skip_if_body:
        jmp     loc_loop_post

  loc_loop_post:
        mov     eax, [ebp-16]
        add     eax, 1
        mov     [ebp-16], eax
        jmp     loc_loop_cond

  loc_ret:
        mov     eax, [ebp-20]
        add     esp, 20
        pop     ebp
        ret
```

Listing 8: An extract of the textual representation of a serialized CFG (in Google Protocol Buffer format), which was generated by the control flow recovery component of MC-Semantics when analysing the relocatable object code which contained the disassembly presented in listing 7.

```
internal_funcs {
  blocks {
    insts {
      inst_bytes: "U"
      inst_addr: 0
      true_target: -1
      false_target: -1
      inst_len: 1
      reloc_offset: 0
    }
    // ...
    insts {
      inst_bytes: "\017\215/\000\000\000"
      inst_addr: 46
      true_target: 99
      false_target: 52
      inst_len: 6
      reloc_offset: 0
    }
    base_address: 0
    block_follows: 99
    block_follows: 52
  }
  // ...
  blocks {
    insts {
      inst_bytes: "\351\000\000\000\000"
      inst_addr: 78
      true_target: 83
      false_target: -1
      inst_len: 5
      reloc_offset: 0
    }
    base_address: 78
    block_follows: 83
  }
  entry_address: 0
}
module_name: "example1"
entries {
  entry_name: "main"
  entry_address: 0
}
```

Listing 9: Decompiled LLVM IR, which was produced by MC-Semantics when analysing the relocatable object code which contained the disassembly presented in listing 7.

```
; ModuleID = 'example1.ll'
target datalayout = "e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"
target triple = "i686-pc-linux-gnu"

%struct.rlimit = type { i32, i32 }

```

```
define i32 @main(i32, i32) {
driverBlock:
  %rl = alloca %struct.rlimit, align 8
  %2 = bitcast %struct.rlimit* %rl to i64*
  store i64 0, i64* %2, align 8
  %3 = ptrtoint %struct.rlimit* %rl to i32
  %4 = call i32 @getrlimit(i32 3, i32 %3)
  %5 = getelementptr %struct.rlimit* %rl, i32 0, i32 0
  %6 = load i32* %5, align 8
  %7 = call i32 @mmap(i32 0, i32 %6, i32 3, i32 131106, i32 -1, i32 0)
  %8 = add i32 %7, %6
  %9 = add i32 %8, -52
  %10 = inttoptr i32 %9 to i32*
  store i32 %1, i32* %10, align 4
  %11 = add i32 %8, -56
  %12 = inttoptr i32 %11 to i32*
  store i32 %0, i32* %12, align 4
  %13 = add i32 %8, -84
  %14 = load i32* %10, align 4
  %15 = add i32 %8, -68
  %16 = inttoptr i32 %15 to i32*
  store i32 0, i32* %16, align 4
  %17 = add i32 %8, -72
  %18 = inttoptr i32 %17 to i32*
  store i32 %0, i32* %18, align 4
  %19 = add i32 %8, -76
  %20 = inttoptr i32 %19 to i32*
  store i32 %14, i32* %20, align 4
  %21 = inttoptr i32 %13 to i32*
  store i32 0, i32* %21, align 4
  %22 = add i32 %8, -80
  %23 = inttoptr i32 %22 to i32*
  store i32 0, i32* %23, align 4
  br label %block_0x34.i

block_0x34.i:                                       ; preds = ↩
    %block_0x53.i, %driverBlock
  %24 = phi i32 [ 0, %driverBlock ], [ %34, %block_0x53.i ]
  %25 = load i32* %21, align 4
  %26 = add i32 %25, -100
  %27 = icmp slt i32 %26, 0
  %28 = sub i32 99, %25
  %29 = and i32 %28, %25
  %30 = icmp slt i32 %29, 0
  %tmp57.i = xor i1 %27, %30
  br i1 %tmp57.i, label %block_0x41.i, label %block_0x53.i

block_0x41.i:                                       ; preds = ↩
    %block_0x34.i
  %31 = mul i32 %24, 3
  %32 = add i32 %31, %25
  store i32 %32, i32* %21, align 4
  %.pre.i = load i32* %23, align 4
  br label %block_0x53.i

block_0x53.i:                                       ; preds = ↩
    %block_0x41.i, %block_0x34.i
  %33 = phi i32 [ %.pre.i, %block_0x41.i ], [ %24, %block_0x34.i ]
  %34 = add i32 %33, 1
  store i32 %34, i32* %23, align 4
  %35 = add i32 %33, -9
  %36 = icmp slt i32 %35, 0
  %37 = sub i32 8, %33
  %38 = and i32 %34, %37
  %39 = icmp slt i32 %38, 0
  %tmp59.i = xor i1 %36, %39
  br i1 %tmp59.i, label %block_0x34.i, label %sub_0.exit

sub_0.exit:                                       ; preds = ↩
   %block_0x53.i
  %40 = load i32* %21, align 4
  %rl1 = alloca %struct.rlimit, align 8
  %41 = bitcast %struct.rlimit* %rl1 to i64*
  store i64 0, i64* %41, align 8
  %42 = ptrtoint %struct.rlimit* %rl1 to i32
  %43 = call i32 @getrlimit(i32 3, i32 %42)
  %44 = getelementptr %struct.rlimit* %rl1, i32 0, i32 0
  %45 = load i32* %44, align 8
  %46 = tail call i32 @munmap(i32 %7, i32 %45)
  ret i32 %40
}

declare i32 @getrlimit(i32, i32)

declare i32 @mmap(i32, i32, i32, i32, i32, i32)

declare i32 @munmap(i32, i32)
```

E Clang Example

The Clang compiler supports emitting LLVM IR from C source code. Using the Clang compiler and the LLVM IR optimiser of the LLVM compiler framework, the source code of the simple C program presented in listing 10 was translated into the LLVM IR assembly presented in listing 11.

Listing 10: The source code of a simple C program which iterates over a pre-test loop to conditionally increment an accumulator. The final value of the accumulator x determines the status code of the program.

```c
int main(int argc, char **argv) {
   int i, x;
   x = 0;
   for (i = 0; i < 10; i++) {
      if (x < 100) {
         x += 3*i;
      }
   }
   return x;
}
```

Listing 11: An optimised version of the LLVM IR assembly, which was emitted by Clang when compiling the C source code of listing 10.

```llvm
define i32 @main(i32 %argc, i8** %argv) {
  br label %1

; <label>:1                                       ; preds = %9, %0
  %i.0 = phi i32 [ 0, %0 ], [ %10, %9 ]
  %x.0 = phi i32 [ 0, %0 ], [ %x.1, %9 ]
  %2 = icmp slt i32 %i.0, 10
  br i1 %2, label %3, label %11

; <label>:3                                       ; preds = %1
  %4 = icmp slt i32 %x.0, 100
  br i1 %4, label %5, label %8

; <label>:5                                       ; preds = %3
  %6 = mul nsw i32 3, %i.0
  %7 = add nsw i32 %x.0, %6
  br label %8

; <label>:8                                       ; preds = %5, %3
  %x.1 = phi i32 [ %7, %5 ], [ %x.0, %3 ]
  br label %9

; <label>:9                                       ; preds = %8
  %10 = add nsw i32 %i.0, 1
  br label %1

; <label>:11                                      ; preds = %1
  ret i32 %x.0
}
```

F Control Flow Graph Generation Example

The `ll2dot` tool generates CFGs (in the DOT file format) from LLVM IR assembly files, as described in section 6.3.2. Using the `ll2dot` tool, a CFG was generated from the `main` function of the LLVM IR assembly in listing 11. A textual representation and an image representation of the generated CFG are presented on the left and the right side of figure 28 respectively. The image representation was generated using the `dot`[40] tool of the Graphviz project, by invoking the command `dot -Tpng main.png main.dot`.

```
digraph main {
    0->1;
    1->11 [label="false"];
    1->3 [label="true"];
    3->8 [label="false"];
    3->5 [label="true"];
    5->8;
    8->9;
    9->1;
    0 [label="entry"];
    1;
    11;
    3;
    5;
    8;
    9;
}
```

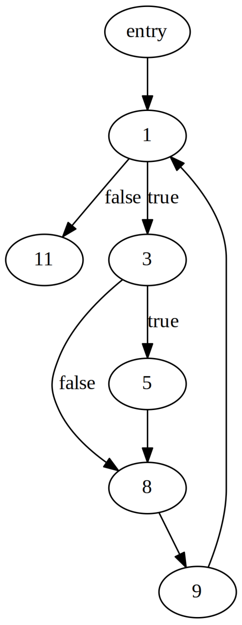


Figure 28: A textual representation in the DOT file format (left) and an image representation (right) of the CFG which was generated from the `main` function of the LLVM IR assembly in listing 11, using the `ll2dot` tool.

[40]Drawing Graphs with dot: `http://www.graphviz.org/pdf/dotguide.pdf`

G Control Flow Analysis Example

This section provides a step-by-step demonstration of how the control flow analysis is conducted by analysing the `stmt` function of the c4[41] compiler. For a detailed description of the control flow analysis stage, please refer to section 6.3.2. The control flow analysis operates exclusively on CFGs, which are generated by a set of components prior to the control flow analysis stage. Firstly, the C source code of the c4 compiler is translated into LLVM IR by the Clang compiler of the front-end. Secondly, the LLVM IR is optionally optimised by the `opt` tool of the LLVM compiler framework. Lastly, a CFG is generated for each function of the LLVM IR using the `ll2dot` tool. For this demonstration, the CFG of the `stmt` function is the starting point of the control flow analysis stage.

The first step of the control flow analysis recursively locates subgraph isomorphisms of the graph representation of pre-test loops (see figure 7d) in the original CFG of the `stmt` function, and replaces these subgraphs with single nodes as illustrated in figure 29.

The second step further simplifies the CFG of **step 1** by recursively replacing the subgraph isomorphisms of the graph representation of consecutive statements (see figure 7f) with single nodes, as illustrated in figure 30.

The third step further simplifies the CFG of **step 2** by recursively replacing the subgraph isomorphisms of the graph representation of 1-way conditionals (see figure 7a) with single nodes, as illustrated in figure 31.

The fourth step further simplifies the CFG of **step 3** by recursively replacing the subgraph isomorphisms of the graph representation of 1-way conditionals with body return statements (see figure 7c) with single nodes, as illustrated in figure 32.

The last step of the control flow analysis stage reduces the CFG of **step 4** into a single node by recursively replacing the subgraph isomorphisms of the graph representation of 2-way conditionals (see figure 7b) with single nodes, as illustrated in figure 33.

When the CFG has been reduced into a single node, a structured CFG may be generated which maps each node to a high-level control flow primitive; as further described in appendix H. Should the control flow analysis fail to reduce the CFG into a single node, the CFG is considered irreducible with regards to the supported high-level control flow primitives, a summary of which are presented in figure 7 of section 2.2.3.

[41] C in four functions: `https://github.com/rswier/c4`

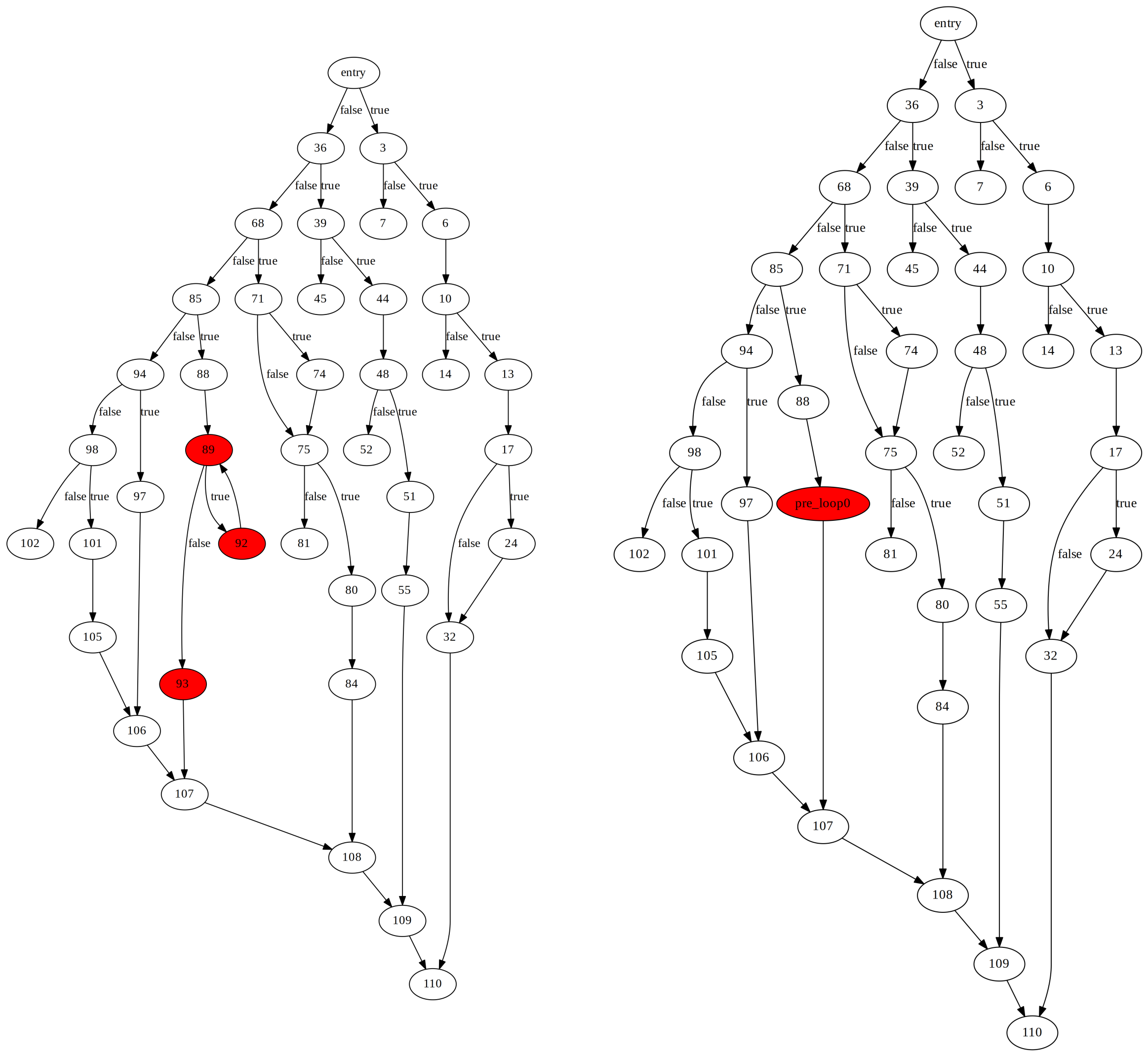


Figure 29: **Step 1**. The original CFG of the `stmt` function (left) and a simplified CFG (right) after identifying pre-test loops (see figure 7d).

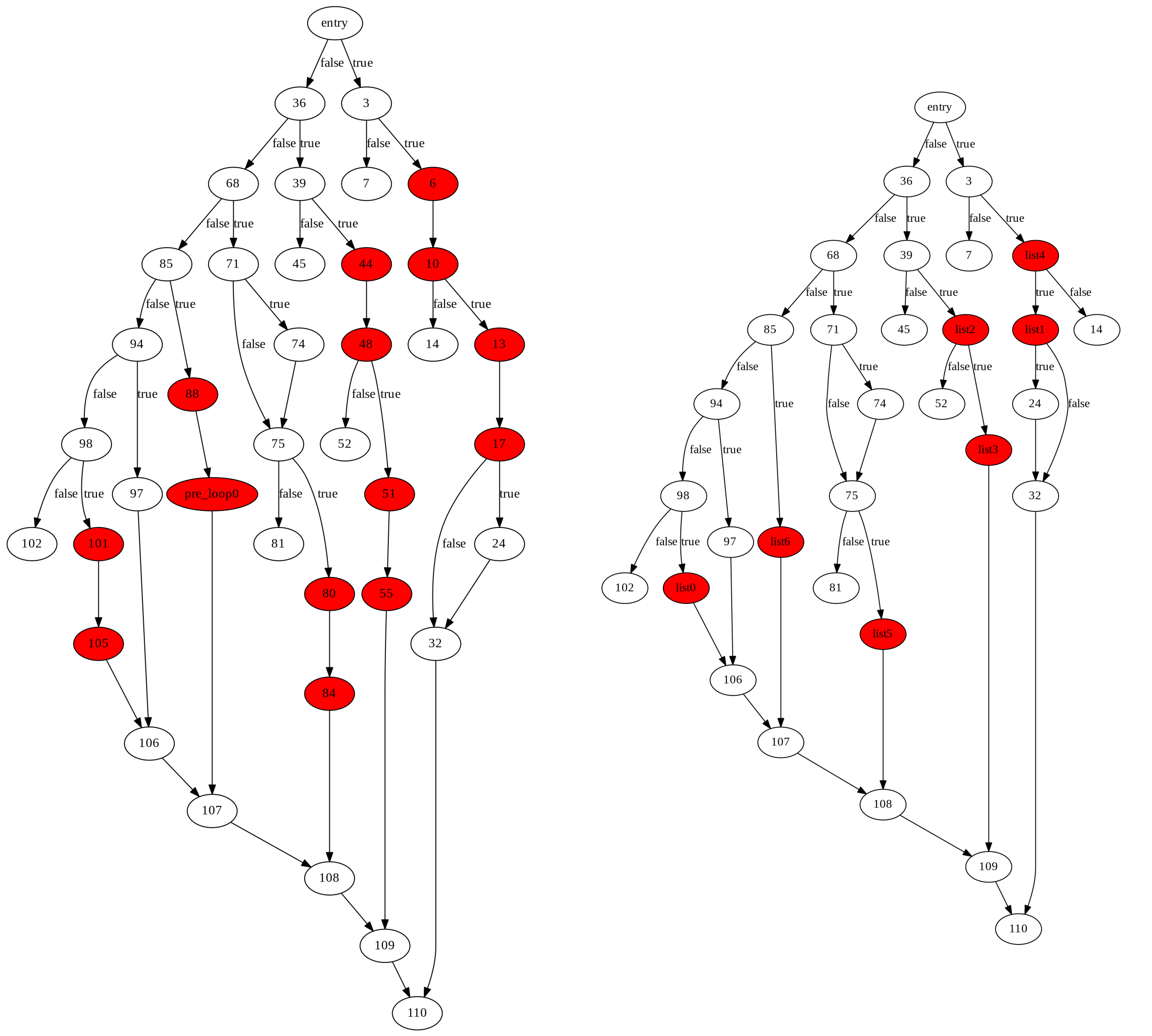


Figure 30: **Step 2**. The CFG from **step 1** (left) and a simplified CFG (right) after identifying consecutive statements (see figure 7f).

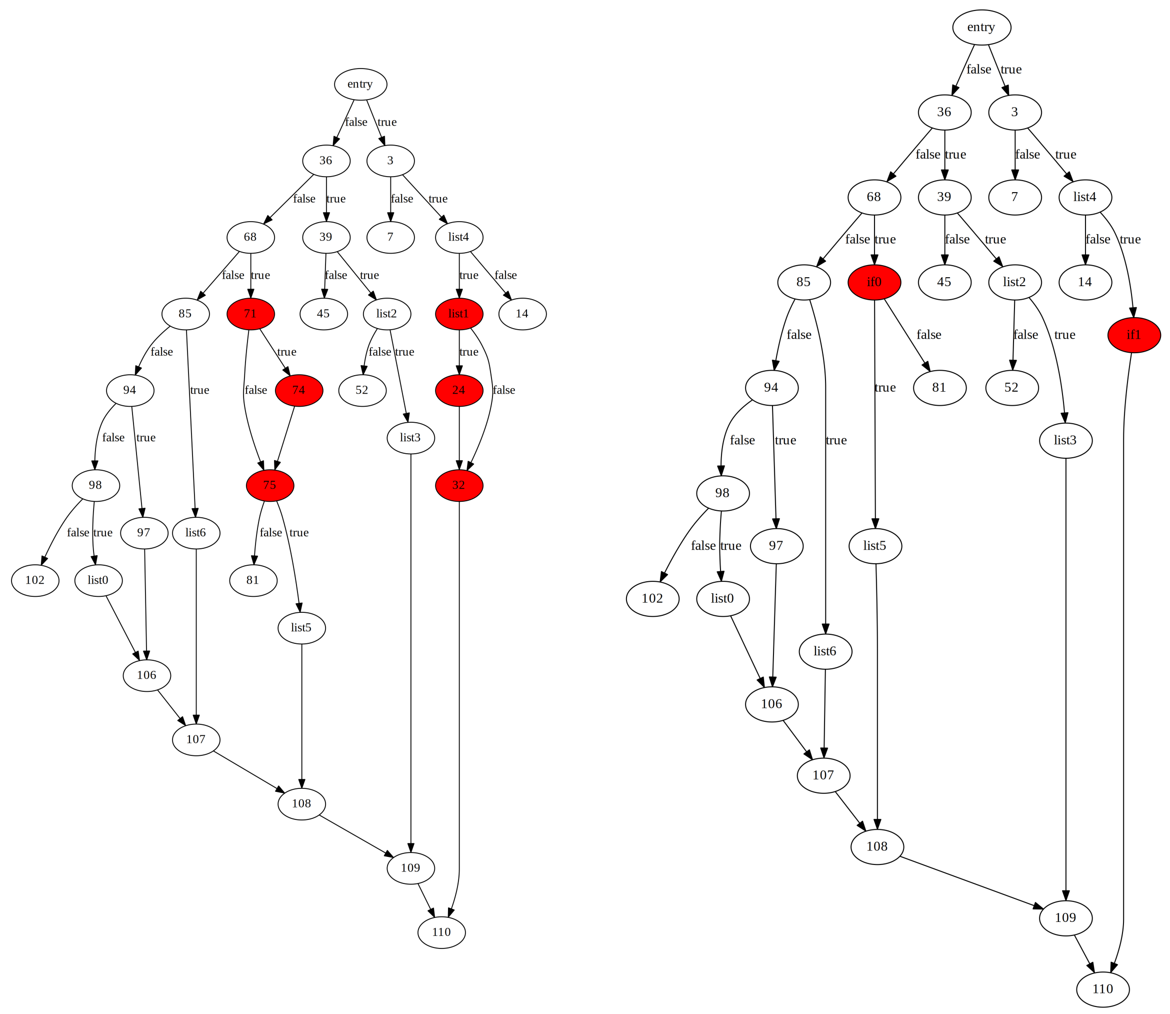


Figure 31: **Step 3**. The CFG from **step 2** (left) and a simplified CFG (right) after identifying 1-way conditionals (see figure 7a).

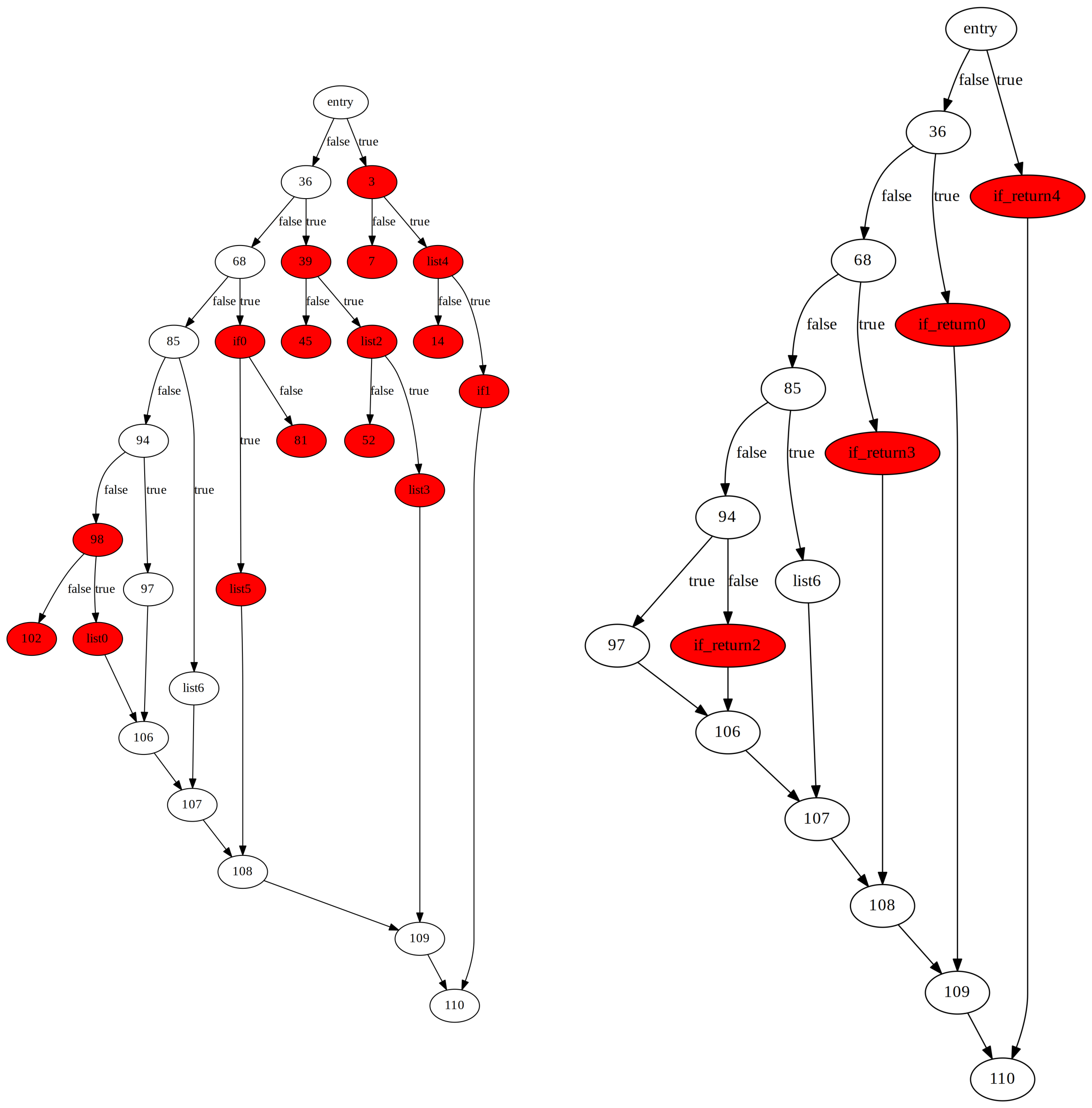
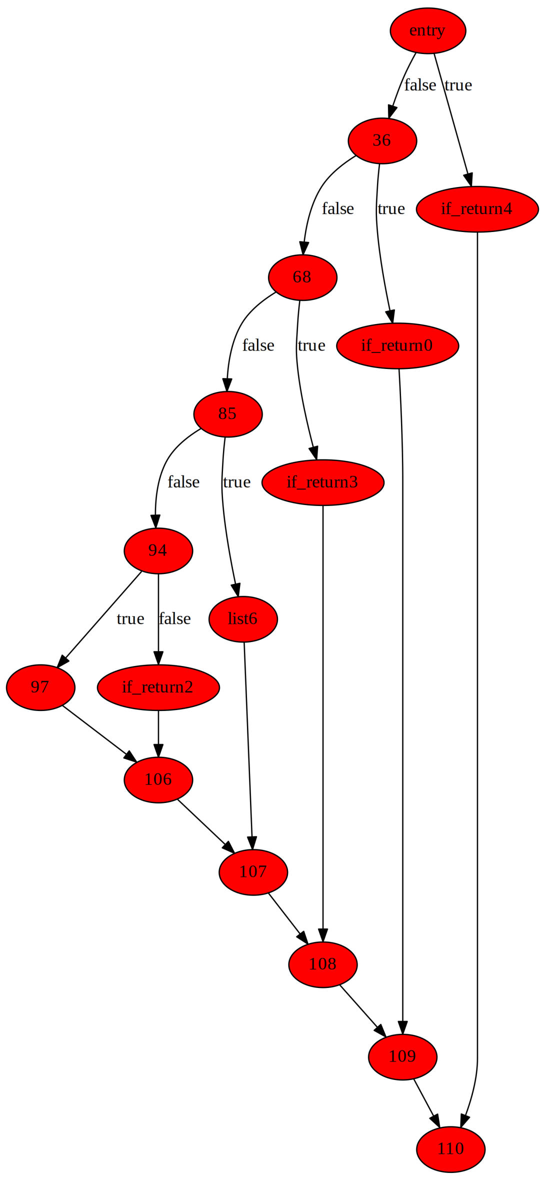


Figure 32: **Step 4**. The CFG from **step 3** (left) and a simplified CFG (right) after identifying 1-way conditionals with body return statements (see figure 7c).



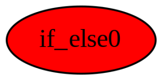


Figure 33: **Step 5**. The CFG from **step 4** (left) and a simplified CFG (right) after identifying 2-way conditionals (see figure 7b).

H Restructure Example

The `restructure` tool produces structured CFGs (in JSON format) from unstructured CFGs (in the DOT file format), as described in section 6.3.2. Listing 12 demonstrates the output of the restructure tool when analysing the CFG of the `main` function presented in figure 28.

Listing 12: The structured CFG (in JSON format) produced by the `restructure` tool when analysing the CFG of the `main` function presented in figure 28.

```
[
    {
        "prim": "list",
        "node": "list0",
        "nodes": {
            "A": "8",
            "B": "9"
        }
    },
    {
        "prim": "if",
        "node": "if0",
        "nodes": {
            "A": "3",
            "B": "5",
            "C": "list0"
        }
    },
    {
        "prim": "pre_loop",
        "node": "pre_loop0",
        "nodes": {
            "A": "1",
            "B": "if0",
            "C": "11"
        }
    },
    {
        "prim": "list",
        "node": "list0",
        "nodes": {
            "A": "0",
            "B": "pre_loop0"
        }
    }
]
```

I Code Generation Example

The ll2go tool translates LLVM IR assembly into unpolished Go source code, as described in section 6.4. Using the ll2go tool, the LLVM IR assembly of listing 11 was translated into the unpolished Go source code presented in listing 13. Please note that the ll2go tool produces unpolished Go source code which may not compile, as it does not follow Go conventions for program status codes and may include undeclared identifiers. Appendix J demonstrates how the post-processing stage may improve the quality of the unpolished Go source code.

Listing 13: Unpolished Go source code, which was produced by the ll2go tool when translating the LLVM IR of listing 11 into Go.

```go
package main

func main() {
	i = 0
	x = 0
	for i < 10 {
		_4 := x < 100
		x = x
		if _4 {
			_6 := 3 * i
			_7 := x + _6
			x = _7
		}
		_10 := i + 1
		i = _10
		x = x
	}
	return x
}
```

J Post-processing Example

This section demonstrates the rewriting capabilities of the `go-post` tool, by successively simplifying the unpolished Go source code presented in the left side of figure 34, through a series of six rewrites which are illustrated in figure 34, 35, 36, 37, 38 and 39 respectively.

For comparison, the original C source code is presented alongside of the decompiled Go output from **rewrite 6** in figure 40. Please note that the middle-end and back-end modules of the decompilation pipeline are only given access to the LLVM IR (see listing 11) produced by the front-end (as described in appendix E), and are completely unaware of the original C source code. When decompiling LLVM IR generated from native code, the original names of identifiers may be missing.

```
1  package main
2
3  func main() {
4      i = 0
5      x = 0
6      for i < 10 {
7          _4 := x < 100
8          x = x
9          if _4 {
10             _6 := 3 * i
11             _7 := x + _6
12             x = _7
13         }
14         _10 := i + 1
15         i = _10
16         x = x
17     }
18     return x
19 }
```

```
1  package main
2
3  func main() {
4      i := 0
5      x := 0
6      for i < 10 {
7          _4 := x < 100
8          x = x
9          if _4 {
10             _6 := 3 * i
11             _7 := x + _6
12             x = _7
13         }
14         _10 := i + 1
15         i = _10
16         x = x
17     }
18     return x
19 }
```

Figure 34: **Rewrite 1**. The original unpolished Go source code (left) and the simplified Go source code (right) after declaring unresolved identifiers. The assignment statements of line 4 and 5 have been rewritten into declare-and-initialise statements. This transformation was applied by invoking `go-post -r unresolved`.

```
 1 package main
 2
 3 func main() {
 4     i := 0
 5     x := 0
 6     for i < 10 {
 7         _4 := x < 100
 8         x = x
 9         if _4 {
10             _6 := 3 * i
11             _7 := x + _6
12             x = _7
13         }
14         _10 := i + 1
15         i = _10
16         x = x
17     }
18     return x
19 }
```

```
 1 package main
 2
 3 import "os"
 4
 5 func main() {
 6     i := 0
 7     x := 0
 8     for i < 10 {
 9         _4 := x < 100
10         x = x
11         if _4 {
12             _6 := 3 * i
13             _7 := x + _6
14             x = _7
15         }
16         _10 := i + 1
17         i = _10
18         x = x
19     }
20     os.Exit(x)
21 }
```

Figure 35: **Rewrite 2**. The Go source code from **rewrite 1** (left) and the simplified Go source code (right) after applying Go conventions for exit status codes. The return statement of line 18 have been rewritten into an `os.Exit` function call and the *"os"* package have been imported on line 3. This transformation was applied by invoking `go-post -r mainret`

```
 1 package main
 2
 3 import "os"
 4
 5 func main() {
 6     i := 0
 7     x := 0
 8     for i < 10 {
 9         _4 := x < 100
10         x = x
11         if _4 {
12             _6 := 3 * i
13             _7 := x + _6
14             x = _7
15         }
16         _10 := i + 1
17         i = _10
18         x = x
19     }
20     os.Exit(x)
21 }
```

```
 1 package main
 2
 3 import "os"
 4
 5 func main() {
 6     i := 0
 7     x := 0
 8     for i < 10 {
 9         x = x
10         if x < 100 {
11             x = x + 3*i
12         }
13         i = i + 1
14         x = x
15     }
16     os.Exit(x)
17 }
```

Figure 36: **Rewrite 3**. The Go source code from **rewrite 2** (left) and the simplified Go source code (right) after propagating temporary variables into expressions. The temporary variables declared at line 9, 12, 13 and 16 have been propagated into the expressions at line 10, 11 and 13. This transformation was applied by invoking `go-post -r localid`

```go
1  package main
2
3  import "os"
4
5  func main() {
6      i := 0
7      x := 0
8      for i < 10 {
9          x = x
10         if x < 100 {
11             x = x + 3*i
12         }
13         i = i + 1
14         x = x
15     }
16     os.Exit(x)
17 }
```

```go
1  package main
2
3  import "os"
4
5  func main() {
6      i := 0
7      x := 0
8      for i < 10 {
9          x = x
10         if x < 100 {
11             x += 3 * i
12         }
13         i++
14         x = x
15     }
16     os.Exit(x)
17 }
```

Figure 37: **Rewrite 4**. The Go source code from **rewrite 3** (left) and the simplified Go source code (right) after simplifying binary assignment statements. The assignment statements on line 11 and 13 have been rewritten into an addition assignment operation and an increment statement respectively. This transformation was applied by invoking `go-post -r assignbinop`

```go
1  package main
2
3  import "os"
4
5  func main() {
6      i := 0
7      x := 0
8      for i < 10 {
9          x = x
10         if x < 100 {
11             x += 3 * i
12         }
13         i++
14         x = x
15     }
16     os.Exit(x)
17 }
```

```go
1  package main
2
3  import "os"
4
5  func main() {
6      i := 0
7      x := 0
8      for i < 10 {
9          if x < 100 {
10             x += 3 * i
11         }
12         i++
13     }
14     os.Exit(x)
15 }
```

Figure 38: **Rewrite 5**. The Go source code from **rewrite 4** (left) and the simplified Go source code (right) after removing dead assignment statements. The assignment statements on line 9 and 14 have been removed. This transformation was applied by invoking `go-post -r deadassign`

```
1  package main
2
3  import "os"
4
5  func main() {
6      i := 0
7      x := 0
8      for i < 10 {
9          if x < 100 {
10             x += 3 * i
11         }
12         i++
13     }
14     os.Exit(x)
15 }
```

```
1  package main
2
3  import "os"
4
5  func main() {
6      x := 0
7      for i := 0; i < 10; i++ {
8          if x < 100 {
9              x += 3 * i
10         }
11     }
12     os.Exit(x)
13 }
```

Figure 39: **Rewrite 6**. The Go source code from **rewrite 5** (left) and the simplified Go source code (right) after propagating the initialisation statement on line 6 and the post-statement on line 12 to the for-loop header on line 7. This transformation was applied by invoking `go-post -r forloop`

```
int main(int argc, char **argv) {
  int i, x;
  x = 0;
  for (i = 0; i < 10; i++) {
    if (x < 100) {
      x += 3*i;
    }
  }
  return x;
}
```

```
package main

import "os"

func main() {
    x := 0
    for i := 0; i < 10; i++ {
        if x < 100 {
            x += 3 * i
        }
    }
    os.Exit(x)
}
```

Figure 40: The original C source code (left) and the decompiled Go output from **rewrite 6** (right).

K Decompilation of Nested Primitives

The following section demonstrates the decompilation of a source program which contains nested primitives. For comparison, the original C source code is presented alongside of the decompiled Go output in figure 41. Please note that the middle-end and back-end modules of the decompilation pipeline are only given access to the LLVM IR (see listing 14) produced by the front-end (as described in appendix E), and are completely unaware of the original C source code. When decompiling LLVM IR generated from native code, the original names of identifiers may be missing.

```c
int main(int argc, char **argv) {
  int i, j, sum;
  sum = 0;
  for (i = 0; i < 5; i++) {
    for (j = 0; j < 7; j++) {
      if (i < j) {
        sum += i;
      } else {
        sum += j;
      }
    }
  }
  return sum % 256;
}
```

```go
package main

import "os"

func main() {
    sum := 0
    for i := 0; i < 5; i++ {
        for j := 0; j < 7; j++ {
            if i < j {
                sum += i
            } else {
                sum += j
            }
        }
    }
    os.Exit(sum % 256)
}
```

Figure 41: The original C source code (left) and the decompiled Go output (right).

Listing 14: The LLVM IR assembly which was produced by Clang when compiling the C source code presented on the left side of listing 41.

```llvm
define i32 @main(i32 %argc, i8** %argv) {
  br label %1

; <label>:1                                       ; preds = %16, %0
  %i.0 = phi i32 [ 0, %0 ], [ %17, %16 ]
  %sum.0 = phi i32 [ 0, %0 ], [ %sum.1, %16 ]
  %2 = icmp slt i32 %i.0, 5
  br i1 %2, label %3, label %18

; <label>:3                                       ; preds = %1
  br label %4

; <label>:4                                       ; preds = %13, %3
  %j.0 = phi i32 [ 0, %3 ], [ %14, %13 ]
  %sum.1 = phi i32 [ %sum.0, %3 ], [ %sum.2, %13 ]
  %5 = icmp slt i32 %j.0, 7
  br i1 %5, label %6, label %15

; <label>:6                                       ; preds = %4
  %7 = icmp slt i32 %i.0, %j.0
  br i1 %7, label %8, label %10

; <label>:8                                       ; preds = %6
```

```
  %9 = add nsw i32 %sum.1, %i.0
  br label %12

; <label>:10                                      ; preds = %6
  %11 = add nsw i32 %sum.1, %j.0
  br label %12

; <label>:12                                      ; preds = %10, %8
  %sum.2 = phi i32 [ %9, %8 ], [ %11, %10 ]
  br label %13

; <label>:13                                      ; preds = %12
  %14 = add nsw i32 %j.0, 1
  br label %4

; <label>:15                                      ; preds = %4
  br label %16

; <label>:16                                      ; preds = %15
  %17 = add nsw i32 %i.0, 1
  br label %1

; <label>:18                                      ; preds = %1
  %19 = srem i32 %sum.0, 256
  ret i32 %19
}
```

L Decompilation of Post-test Loops

The following section demonstrates the decompilation of a source program which contains post-test loops. For comparison, the original C source code is presented alongside of the decompiled Go output in figure 42. Please note that the middle-end and back-end modules of the decompilation pipeline are only given access to the LLVM IR (see listing 15) produced by the front-end (as described in appendix E), and are completely unaware of the original C source code. When decompiling LLVM IR generated from native code, the original names of identifiers may be missing.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int i, n;

  i = 0;
  n = 1;
  do {
    if (i < 10) {
      i++;
      n *= 2;
    } else {
      i += 3;
      n *= 4;
    }
  } while(i < 15);
  return n%123;
}
```

```go
package main

import "os"

func main() {
	i := 0
	n := 1
	for {
		if i < 10 {
			i++
			n *= 2
		} else {
			i += 3
			n *= 4
		}
		if !(i < 15) {
			break
		}
	}
	os.Exit(n % 123)
}
```

Figure 42: The original C source code (left) and the decompiled Go output (right).

Listing 15: The LLVM IR assembly which was produced by Clang when compiling the C source code presented on the left side of listing 42.

```llvm
define i32 @main(i32 %argc, i8** %argv) {
  br label %1

; <label>:1                                       ; preds = %10, %0
  %i.0 = phi i32 [ 0, %0 ], [ %i.1, %10 ]
  %n.0 = phi i32 [ 1, %0 ], [ %n.1, %10 ]
  %2 = icmp slt i32 %i.0, 10
  br i1 %2, label %3, label %6

; <label>:3                                       ; preds = %1
  %4 = add nsw i32 %i.0, 1
  %5 = mul nsw i32 %n.0, 2
  br label %9

; <label>:6                                       ; preds = %1
  %7 = add nsw i32 %i.0, 3
  %8 = mul nsw i32 %n.0, 4
  br label %9

; <label>:9                                              ; preds = %6, %3
  %i.1 = phi i32 [ %4, %3 ], [ %7, %6 ]
  %n.1 = phi i32 [ %5, %3 ], [ %8, %6 ]
  br label %10

; <label>:10                                             ; preds = %9
  %11 = icmp slt i32 %i.1, 15
  br i1 %11, label %1, label %12

; <label>:12                                             ; preds = %10
  %13 = srem i32 %n.1, 123
  ret i32 %13
}
```