Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Using LLMs to Rank Decompiled Code Variants

Luigi Timossi

Master's Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35 16146 Genova, Italy
https://www.dibris.unige.it/

Computer Science MSc
Computer Security and Engineering Curriculum

Using LLMs to Rank Decompiled Code Variants

Luigi Timossi

Advisors: Matteo Dell'Amico, Giovanni Lagorio
Examiner: Marina Ribaudo

January, 2026

Abstract

This space will be occupied by the abstract, a summary of all the things that i have done in this thesis

Table of Contents

Chapter 1

Introduction

This is a template for the Computer Science Master's Theses at *dipartimento di informatica, bioingegneria, robotica e ingegneria dei sistemi* (DIBRIS), University of Genoa.[1] It uses the LaTeX class `masterthesis.cls` by Davide Ancona. We will use this template to discuss what is generally expected in the structure of a Master's Thesis. While reading academic documents, you will find that they generally follow a similar structure.

This structure is generally good for research documents, where your main goal is to do something new. In other cases (e.g., the goal of your thesis is to review a topic and provide an understanding of the field), a different structure may be more desirable. In any case, discuss the structure of your thesis with your advisor.

I have put hints and links through the document for tools you can use; some may become stale over time, and something better may appear. If you find some that are not covered here, consider sending me an email or a pull request to integrate them!

1.1 Style and Language

We are writing a scientific document, and we should consider that skepticism is at the core of the scientific method. We should be skeptical of our work, and we can see the thesis as an effort to persuade a skeptical reader that we have done a piece of work that is both useful and solid. Whenever you state something, you should give proof: either by referencing one or more authoritative documents or proving it yourself. Avoid marketing-speak ("our work is poised to revolutionize...").[2]

[1] https://dibris.unige.it

[2] Unless you can prove what you say. But these statements are mostly impossible to prove... By the way, notice that footnotes should be put after punctuation as in this case.

Do not try to reach a given number of pages. Instead, try to make it *complete*, *clear* and *concise*. Try to give all the information that will convince the reader that your work is solid and that you have done a good job; use simple language and consistent terminology. If you can say something more simply, do it. Avoid weasel words[3] and, when you can, the passive voice. "We consider this to be not a security risk" is much clearer than "This is not considered a security risk" because it makes it explicit who makes the claim, and takes the responsibility for it.

Consider that your reader will be a fellow computer scientist, but not necessarily an expert on your topic. Try to give them the information they need to understand your work, but do not lose your reader's time and attention with unnecessary details.

Documents like this generally follow a structure like this one. It is not a strict requirement, but if you follow it readers will likely find it easier to navigate. Consider that in scientific documents some readers will not read the whole document, but will look for specific pieces of information. A clear structure will help them.

Chapters generally start with a short introduction to their topic. This is followed by the main content of the chapter, and then by a conclusion that summarizes the main points. Note that both chapter and section titles are capitalized.[4]

Introduce the expanded form of acronyms the first time you use them, excepting only those so common that you can reasonably take for granted for any reader (e.g., CPU or PDF).

For this document, you should use formal writing. Do not use contractions[5] such as *don't* or *it's*.

Be extremely wary of large language models (LLMs) like Gemini, ChatGPT, or Copilot.[6] You take responsibility for the content of your thesis and declare you have written it yourself. If you want a tool that helps you to write in better English, consider using something like Grammarly[7]; it is also supported by the Overleaf online LaTeX editor.[8] Note that Overleaf now has a generative AI option: do not use it without declaring it, as it would violate its terms of service[9] which, among other things, forbids "to mislead any third party that any output from Grammarly's generative AI was solely human generated".

[3] https://en.wikipedia.org/wiki/Weasel_word

[4] https://www.grammarly.com/blog/capitalization-in-the-titles/

[5] https://www.grammarly.com/blog/contractions/

[6] Note that we used the serial comma (https://en.wikipedia.org/wiki/Serial_comma) here. It is a good idea to use it as it can remove ambiguity from your text.

[7] https://grammarly.com

[8] https://www.overleaf.com/learn/how-to/Use_Grammarly_with_Overleaf

[9] https://www.grammarly.com/terms

1.2 Motivation

This is the first important thing you should write in your Introduction. You can see it as the effort to convince your reader that the problem you are addressing *matters*. Mention that the problem is unsolved, but you will discuss that at you're facing a skeptical reader asking "is this useful?".

Sometimes it will be obvious why the problem matters, but in other cases, this will be one of the most delicate parts of your work. Of course, your advisor will help you with this.

1.3 Content of the Thesis

The other main goal of the Introduction is to give the reader an overview of what they will find in the document. There is no concept of "spoiler" in scientific documents: summarize what the reader will find in every chapter. Avoid obvious statements that help nobody such as, for example, *"In the Related Work chapter we discuss relevant pieces of work in the state of the art"*; provide synthetic but useful information instead (e.g., *"In Chapter 3 we discuss other density-based clustering algorithms and highlight the differences to our approach")*. This will help them to navigate it (and to find what they're interested in, if they are reading the whole document).

I am not a fan of those *"This document is structured as follows"* paragraphs. My advice is to expose the content of the thesis and give references, organically, to the parts of it that treat them while you are at it.

The Introduction is arguably the most important chapter of an academic text because, after reading it, one should understand the key concepts of your work. The rest of the document "only" explains the details; you can expect somebody to stop at the Introduction alone.[10]

1.4 LaTeX Tricks

If you're using LaTeX to write your document, you can use some tricks to make your life easier. This document uses the `\Cref`[11] command to conveniently reference sections, figures, and tables, and `\textcite` and `\authorcite` to cite paper authors. You can also

[10] Likewise, while you are going through somebody else's work, it makes sense to stop at the Introduction (and maybe the Conclusion) if you're interested only in a high-level overview.

[11] I also created a `\fullref` to generate text like "Introduction (Chapter 1)". Check the source if you are interested.

use `\xspace` to define your macros and avoid spacing problems. Check the source code of this document to see how they are used.

`rubber` is a good tool to compile LaTeX documents. You can run the command `rubber -d main` to compile this document to PDF, while running `pdflatex` and `biber` (or `bibtex`) the right number of times.

The Overleaf[12] online editor is a good tool to synchronize with your advisor; it exports a git repository so you can edit your work online and offline.

To handle acronyms, the `acronym`[13] package is a good resource to automatically introduce an acronym's expanded form the first time you use it. It is also possible to create an acronyms table; consider using it if you use many acronyms in your paper and you think it may help the reader. Once again, you can have a look at how it's used in the source of this document.

You are likely going to cite several websites in your bibliography. Here is documentation[14] on how to do that in both BibTeX and BibLaTeX. You may want to use a reference management software like Zotero[15] or Mendeley[16] to handle your bibliography database.

Your supervisors and reviewers will likely go over your document in several revisions; I'm sure they would really be thankful to you if you used a way to clearly show what changed since their previous pass. Latexdiff[17] is a tool to do just that; you can also use it on Overleaf[18]. Another possibility is tracking the changes on the PDF output, using tools like Draftable[19].

Advisors may have very different preferences in how to share feedback. A possibility is the `todonotes`[20] package. This document's `preamble.tex` source file shows an example of using it to include comments; note that they can be removed from the PDF output by passing the `final` option to the document class (i.e., `\documentclass[final]{masterthesis}` in the main `tex` file.

MD: This is an example of a note using `todonotes`.

IMPORTANT: Make sure the curriculum on the cover pages and the department address matches your own. Currently, you'll have to edit the `masterthesis.cls`

[12] https://www.overleaf.com
[13] https://ctan.org/pkg/acronym
[14] https://bibtex.eu/faq/how-can-i-use-bibtex-to-cite-a-website/
[15] https://www.zotero.org/
[16] https://www.mendeley.com/
[17] https://ctan.org/pkg/latexdiff
[18] https://www.overleaf.com/learn/latex/Articles/How_to_use_latexdiff_on_Overleaf
[19] https://www.draftable.com/
[20] https://tug.ctan.org/macros/latex/contrib/todonotes/todonotes.pdf

file accordingly; I plan to eventually provide a class option to facilitate that without the need to edit the `.cls` file. If you feel like you want to help me with that, please let me know! You'll get gratitude and an acknowledgment here.

Chapter 2

Background

summary of all the things ghidra + llm

2.1 The Ghidra Architecture

Ghidra, released by the National Security Agency (NSA) in 2019, employs a bifurcated design that separates the user-facing interaction layer from the core analysis engine. This separation is not merely an implementation detail but a fundamental architectural constraint that dictates how data flows during the reverse engineering process.

The framework operates across two distinct memory spaces: a frontend implemented in Java and a backend analysis engine written in C++. The Java frontend is responsible for the Graphical User Interface (GUI), project database management, and plugin orchestration. It provides the high-level API exposed to users and scripts (e.g., Python or Java scripts via the GhidraScript framework). However, the computationally intensive tasks of data-flow analysis, variable inference, and control flow structuring are offloaded to a native C++ executable, typically named `decomp` or `decomp_dbg` (for debugging). These executables and the code are located at `Ghidra/Features/Decompiler/src/decompile/cpp`.

Communication is mediated by the `ghidra.app.decompiler.DecompInterface`. This interface manages a dedicated input/output stream to the native process, utilizing an XML-based protocol to exchange data. When a function decompilation is requested, the Java client does not simply invoke a library function; it serializes the request into an XML command (e.g., `<decompile_at>`) and transmits it to the backend. The C++ process, holding its own representation of the function's data flow in `Funcdata` objects, performs the analysis and returns the results as a serialized XML stream describing the high-level code structure and syntax tokens.

2.2 SLEIGH and P-code

As written in the documentation created by running `<make doc>` [NSA] The decompiler provides its own Register Transfer Language (RTL), referred to internally as p-code, which is designed specifically for reverse engineering applications. The disassembly of processor specific machine-code languages, and subsequent translation into p-code, forms a major sub-system of the decompiler. There is a processor specification language, referred to as SLEIGH, which is dedicated to this translation task, this piece of the code can be built as a standalone binary translation library, for use by other applications.

2.2.1 P-code Semantics and Varnodes

Unlike intermediate languages in compilers, P-code is designed specifically for reverse engineering, prioritizing the explicit representation of memory and register modifications.

The fundamental unit of data in P-code is the **Varnode**. A Varnode is defined by the triple $(Space, Offset, Size)$, representing a contiguous sequence of bytes in a specific address space.

Table 2.1: Some P-code Operations and Semantics `opcodes.hh`[NSA][Pco]

Opcode	**Operands**	**Semantics**
`CPUI_COPY`	$in_0 \rightarrow out$	Copy one operand to another.
`CPUI_LOAD`	$space, ptr \rightarrow out$	Load from a pointer into a specific address.
`CPUI_STORE`	$space, ptr, val$	Store at a pointer into a specified address space.
`CPUI_INT_ADD`	$in_0, in_1 \rightarrow out$	Integer addition, signed or unsigned.
`CPUI_CBRANCH`	$dest, cond$	Conditional jump to *dest* if *cond* is non-zero.

We must distinguish between two forms of P-code used during analysis:

1. **Raw P-code:** The direct, unoptimized output of the SLEIGH translation. It is represented by the class **PcodeOpRaw** (or by unprocessed PcodeOp), and contains the bare essentials: an opcode, a sequence number (address), and the input/output Varnodes.

2. **High P-code:** The result of the analysis pipeline. In this form, the code has been converted to Static Single Assignment (SSA) form (a form where every varnode

is defined exactly once for each function, if a variable is assigned multiple times, each assignment is given a new instance called low-level variable), dead code has been eliminated, and high-level concepts like function calls (replacing jump-and-link semantics) have been recovered. It is represented by the class **HighVariable**; this is an abstraction that groups multiple low-level Varnodes (which may reside in different registers or stack locations during execution) into a single logical variable, similar to a variable in C code.

The transformation from Raw to High P-code is where the majority of the decompilation logic resides. It is an inference process that attempts to raise the abstraction level of the code, often relying on heuristics that may fail in the presence of obfuscation or aggressive compiler optimizations.

2.3 The Decompilation Pipeline

The C++ decompiler engine processes a function at a time through a series of iterative passes. The architecture organizes these passes into **Actions** and **Rules**, managed by the `ActionDatabase`. inside the `ActionDatabase::universalAction` we have two main types of objects:

- `ActionGroup`: Represents a list of Actions that are applied sequentially. The group's properties (eg., rule_repeatapply) influence how the contained actions are executed.
- `ActionPool`: It is a pool of Rules that are applied simultaneously to every PcodeOp. Each Rule triggers on a specific localized data-flow configuration. The Rules are applied repeatedly until no Rule can make any additional transformations.

2.3.1 Actions and Rules

Actions represent large-scale transformations applied to the graph of varnodes and operations. They are the base class for objects that make modifications to a function's (Funcdata) syntax tree. Their purpose is to manage complex stages of the workflow, such as recovering the control-flow structure or generating SSA form.

Rules, on the other hand, are a class designed to perform a single specific transformation on a PcodeOp or a Varnode. A Rule triggers when it recognizes a particular local configuration in the data flow and specifies a sequence of modification operations to transform it.

2.3.2 DefaultGroups

Actions and Rules are selected and activated according to the type of **DefaultGroup** they belong to. These groups represent standardized workflows for different analysis phases and are built by the method `ActionDatabase::buildDefaultGroups()`. The main groups are:

- **decompile**: the standard workflow for full decompilation, composed of all of the phases.
- **jumptable**: optimized for analyzing jump tables.
- **normalize**: used for code normalization.
- **paramid**: for parameter identification.
- **register**: for register analysis.
- **firstpass**: a first fast analysis pass.

Each DefaultGroup is a list of names that refer to specific `ActionGroup`, `ActionPool` or individual `Action` to execute in that configuration. These lists define subsets of all the Actions.

The decompiler can be customized by selecting different DefaultGroups in java with the method `setSimplificationStyle()` of the decompiler interface but Only the group named **decompile** return C code to ghidra, since in `ghidra_process.cc` we have:

Listing 2.1: ghidra_process.cc

```
[...]
  fd->encode(encoder,0,ghidra->getSendSyntaxTree());
  if (ghidra->getSendCCode()&&
 (ghidra->allacts.getCurrentName() == "decompile")) //HERE WE HAVE THE CHECK
    ghidra->print->docFunction(fd);
[...]
```

2.4 Logic of Control Flow Structuring

Recovering high-level control structures (loops, conditionals) from the unstructured Control Flow Graph (CFG) is arguably the most challenging phase of decompilation. It is effectively a pattern-matching problem on a directed graph, aimed at finding subgraphs that correspond to structured programming constructs.

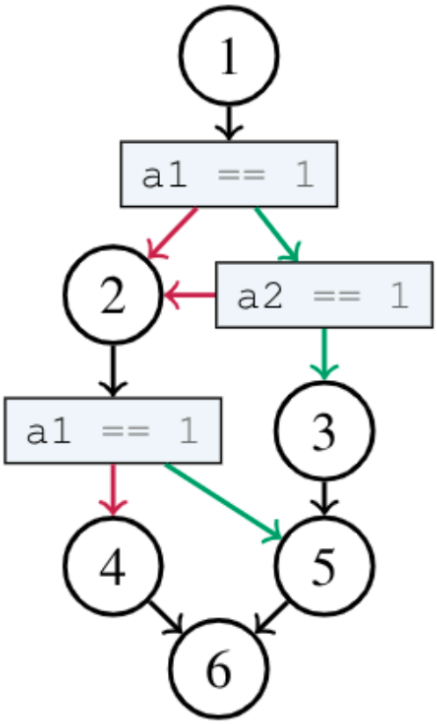

Figure 2.1: Control Flow Graph of a function

2.4.1 Basic Block Formulation

The decompiler first aggregates P-code operations into **BasicBlocks** sequences of instructions with a single entry point and a single exit point (excluding internal calls). The CFG is formed by the edges representing jumps and branches between these blocks. Ghidra normalizes this graph to ensure a unique entry block, often inserting empty placeholder blocks to handle re-entrant loops or complex function entries.

In this example we have the C code of the function described in 2.1 and its corresponding P-code representation [1].

[1] P-codes varies during all phases of the decompilation process; due to optimization rules, dead code elimination, and other transformations, the P-code shown here are taken from the `collapseInternal()` method using `printRaw()` of the FlowBlock class. The BasicBlock order may not correspond directly to the original source code order

Source Code (C)	P-Code / Basic Blocks

`int a2_local;`
`int a1_local;`
`putchar(L'1');`
`if ((a1 == 1)`
	Basic Block 0
	`0x0010118d:1: RSP(0x0010118d:1) = RSP(i) + #0xfffffffffffffff8`
	`0x0010118d:2: *(ram,RSP(0x0010118d:1)) = RBP(i)`
	`0x00101195:d: u0x00004780(0x00101195:d) = RSP(i) + #0xfffffffffffffff4`
	`0x00101195:f: *(ram,u0x00004780(0x00101195:d)) = EDI(i)`
	`0x00101198:10: u0x00004780(0x00101198:10) = RSP(i) +`
	`#0xfffffffffffffff0`
	`0x00101198:12: *(ram,u0x00004780(0x00101198:10)) = ESI(i)`
	`0x001011a0:14: RSP(0x001011a0:14) = RSP(i) + #0xfffffffffffffffe0`
	`0x001011a0:15: *(ram,RSP(0x001011a0:14)) = #0x1011a5`
	`0x001011a0:67: u0x10000008:1(0x001011a0:67) =`
	`*(ram,RSP(0x001011a0:14))`
	`0x001011a0:16: call`
	`jputchar(free)(#0x31:4,u0x10000008:1(0x001011a0:67))`
	`0x001011a5:17: u0x00004780(0x001011a5:17) = RSP(i) +`
	`#0xfffffffffffffff4`
	`0x001011a5:18: u0x00011e80:4(0x001011a5:18) =`
	`*(ram,u0x00004780(0x001011a5:17))`
	`0x001011a5:1e: ZF(0x001011a5:1e) = u0x00011e80:4(0x001011a5:18) ==`
	`#0x1:4`
	`0x001011a9:23: goto Block_2:0x001011bd if (ZF(0x001011a5:1e) != 0)`
	`else Block_1:0x001011ab`

`|| (a2 != 2)){`	**Basic Block 1**
	`0x001011ab:24: u0x00004780(0x001011ab:24) = RSP(i) +`
	`#0xfffffffffffffff0`
	`0x001011ab:25: u0x00011e80:4(0x001011ab:25) =`
	`*(ram,u0x00004780(0x001011ab:24))`
	`0x001011ab:2b: ZF(0x001011ab:2b) = u0x00011e80:4(0x001011ab:25) !=`
	`#0x2:4`
	`0x001011af:31: goto Block_2:0x001011bd if (ZF(0x001011ab:2b) != 0)`
	`else Block_4:0x001011b1`

`putchar(L'2');`
`if (a1 != a2) {`
	Basic Block 2
	`0x001011c2:46: RSP(0x001011c2:46) = RSP(i) + #0xfffffffffffffffe0`
	`0x001011c2:47: *(ram,RSP(0x001011c2:46)) = #0x1011c7`
	`0x001011c2:69: u0x10000011:1(0x001011c2:69) =`
	`*(ram,RSP(0x001011c2:46))`
	`0x001011c2:48: call`
	`jputchar(free)(#0x32:4,u0x10000011:1(0x001011c2:69))`
	`0x001011c7:49: u0x00004780(0x001011c7:49) = RSP(i) +`
	`#0xfffffffffffffff4`
	`0x001011c7:4a: u0x00011e80:4(0x001011c7:4a) =`
	`*(ram,u0x00004780(0x001011c7:49))`
	`0x001011ca:4d: u0x00004780(0x001011ca:4d) = RSP(i) +`
	`#0xfffffffffffffff0`
	`0x001011ca:4e: u0x00006a00:4(0x001011ca:4e) =`
	`*(ram,u0x00004780(0x001011ca:4d))`
	`0x001011ca:54: ZF(0x001011ca:54) = u0x00011e80:4(0x001011c7:4a) ==`
	`u0x00006a00:4(0x001011ca:4e)`
	`0x001011cd:59: goto Block_3:0x001011cf if (ZF(0x001011ca:54) == 0)`
	`else Block_5:0x001011db`

Source Code (C)	P-Code / Basic Blocks

`putchar(L'4');`
`goto LAB_001011e5;`
	Basic Block 3
	`0x001011d4:5b: RSP(0x001011d4:5b) = RSP(i) + #0xfffffffffffffe0`
	`0x001011d4:5c: *(ram,RSP(0x001011d4:5b)) = #0x1011d9`
	`0x001011d4:6b: u0x1000001a:1(0x001011d4:6b) =`
	`  *(ram,RSP(0x001011d4:5b))`
	`0x001011d4:5d: call`
	`  jputchar(free)(#0x34:4,u0x1000001a:1(0x001011d4:6b))`
	`0x001011d9:5e: goto Block_6:0x001011e5`

`} else {`
`  putchar(L'3');`
`}`
	Basic Block 4
	`0x001011b6:33: RSP(0x001011b6:33) = RSP(i) + #0xfffffffffffffe0`
	`0x001011b6:34: *(ram,RSP(0x001011b6:33)) = #0x1011bb`
	`0x001011b6:6d: u0x10000023:1(0x001011b6:6d) =`
	`  *(ram,RSP(0x001011b6:33))`
	`0x001011b6:35: call`
	`  jputchar(free)(#0x33:4,u0x10000023:1(0x001011b6:6d))`
	`0x001011bb:36: goto Block_5:0x001011db`

`putchar(L'5');`
`}`
	Basic Block 5
	`0x001011e0:38: RSP(0x001011e0:38) = RSP(i) + #0xfffffffffffffe0`
	`0x001011e0:39: *(ram,RSP(0x001011e0:38)) = #0x1011e5`
	`0x001011e0:6f: u0x1000002c:1(0x001011e0:6f) =`
	`  *(ram,RSP(0x001011e0:38))`
	`0x001011e0:3a: call`
	`  jputchar(free)(#0x35:4,u0x1000002c:1(0x001011e0:6f))`

`LAB_001011e5:`
`putchar(L'6');`
`return;`
`}`
	Basic Block 6
	`0x001011ea:3c: RSP(0x001011ea:3c) = RSP(i) + #0xfffffffffffffe0`
	`0x001011ea:3d: *(ram,RSP(0x001011ea:3c)) = #0x1011ef`
	`0x001011ea:71: u0x10000035:1(0x001011ea:71) =`
	`  *(ram,RSP(0x001011ea:3c))`
	`0x001011ea:3e: call`
	`  jputchar(free)(#0x36:4,u0x10000035:1(0x001011ea:71))`
	`0x001011f1:44: return(#0x0)`

Basicblocks are created in `flow.cc` by the function `FlowInfo::splitBasic()`. The routine partitions the P-code instruction stream at control-flow boundaries: conditional and unconditional jumps, call sites that alter control flow, and return instructions. Each such instruction ends the current block and/or starts a new one (targets of jumps also begin blocks).

2.4.2 The Structuring Algorithm

To transform the CFG into C statements, Ghidra employs a structuring algorithm implemented in the `ActionBlockStructure` class. The process involves identifying regions of the graph that match known schemas (or patterns) of control flow:

inside the apply method of `ActionBlockStructure` we have a call to collapseAll() that is the main loop of the algorithm:

```
void CollapseStructure::collapseAll(void)
{
  int4 isolated_count;

  finaltrace = false;
  graph.clearVisitCount();
  orderLoopBodies();

  collapseConditions();

  isolated_count = collapseInternal((FlowBlock *)0);
  while(isolated_count < graph.getSize()) {
    FlowBlock *targetbl = selectGoto();
    isolated_count = collapseInternal(targetbl);
  }
}
```

The method implements a deterministic sequence of passes that progressively transform the BasicBlocks into structured FlowBlocks and performs the following steps:

1. **Preparation**
   The algorithm first clears previous visitation state (`graph.clearVisitCount()`) and invokes `orderLoopBodies()`. This pass discovers loop headers and back-edges, establishing a partial ordering among loop bodies. Detecting loops early is essential to prevent later structuring passes from erroneously breaking loop semantics.

2. **Conditional simplification**
   Next, `collapseConditions()` attempts to simplify complex boolean logic and fold adjacent blocks that form logical AND/OR patterns (for example, transforming sequences that represent `if (A && B)` or `if (A || B)` into single conditional constructs). This phase applies local rules such as `ruleBlockOr` to reduce predicate complexity before higher-level structuring.

3. **Initial collapse**

The engine then calls `collapseInternal((FlowBlock *)0)`, which scans the graph and applies standard structuring rules (e.g. `ruleBlockIfElse`, `ruleBlockWhileDo`, `ruleBlockSwitch`) to collapse perfectly structured regions. The routine returns an `isolated_count` indicating how many blocks have been fully resolved without introducing gotos.

4. **Unstructured flow handling**
   If the graph is not fully collapsed (`isolated_count < graph.getSize()`), the method iterates: it selects a problematic edge with `selectGoto()` and marks that edge as unstructured (to be emitted as a `goto`/`break`/`continue` in the final code). The selection is driven by heuristics to minimize disruption to surrounding structure. After marking the edge, `collapseInternal(targetbl)` is invoked again (often passing the target block of the newly created goto) so the structuring engine can resume collapsing other regions. This loop repeats until every block is resolved.

in the `collapseInetrnal()` method we have the main pattern recognition method, some patterns have precedence over others, since it may occur that a region matches multiple schemas. For example, a `switch` may also match an `if-else` pattern.

These are the preferred patterns, in order:

- `goto`
- `cat` (block concatenation)
- `proper if` (if without else)
- `if-else`
- `while-do`
- `do-while`
- `infinite loop`
- `switch`

This "rules" are implemented inside a loop that tryes every pattern till no more changes are possible.

in the `ruleBlockWhileDo()` method we can see how the pattern matching is done:

```
bool CollapseStructure::ruleBlockWhileDo(FlowBlock *bl)

{
  FlowBlock *clauseblock;
  int4 i;

  if (bl->sizeOut() != 2) return false; // Must be binary condition
  if (bl->isSwitchOut()) return false;
  if (bl->getOut(0) == bl) return false; // No loops at this point
  if (bl->getOut(1) == bl) return false;
  if (bl->isInteriorGotoTarget()) return false;
  if (bl->isGotoOut(0)) return false;
  if (bl->isGotoOut(1)) return false;
  for(i=0;i<2;++i) {
    clauseblock = bl->getOut(i);
    if (clauseblock->sizeIn() != 1) continue; // Nothing else must hit clause
    if (clauseblock->sizeOut() != 1) continue; // Only one way out of clause
    if (clauseblock->isSwitchOut()) continue;
    if (clauseblock->getOut(0) != bl) continue; // Clause must loop back to bl

    bool overflow = bl->isComplex(); // Check if we need to use overflow syntax
    if ((i==0)!=overflow) {      // clause must be true out of bl unless we use
        overflow syntax
      if (bl->negateCondition(true))
   dataflow_changecount += 1;
    }
    BlockWhileDo *newbl = graph.newBlockWhileDo(bl,clauseblock);
    if (overflow)
      newbl->setOverflowSyntax();
    return true;
  }
  return false;
}
```

Firstly is checked that the block has exactly two outgoing edges (a binary condition) and is not already part of a switch or a loop. Then, for each outgoing edge, it checks if the clauseblock (the potential loop body) has exactly one incoming edge (from the condition block) and one outgoing edge (back to the condition block). If these conditions are met, it confirms the presence of a while-do loop structure.

A condition is considered complex when the basic block that computes it contains too

many instructions to be cleanly represented within a single conditional expression. The method `BlockBasic::isComplex()` performs this check.

Criteria: the algorithm counts the number of **statements** in the block:

- A conditional jump (branch) counts as 1 statement.
- CALL instructions count as 1.
- Operations that produce outputs used only inside the block or marker instructions do not count, but if a variable is used many times or is tied to memory, it contributes to the count.
- Threshold: if the total number of statements in the block exceeds 2, the block is considered complex.

The overflow syntax (or `f_whiledo_overflow`) is a specific state assigned to a `BlockWhileDo` when its loop control condition is determined to be complex. It indicates that, although a logical `while` structure exists, the conditional block is too long or complicated to be emitted as a single boolean expression `while(condition)`. Instead of printing `while(<complex condition>){...}`, the decompiler emits an alternative form, typically an infinite loop with an internal `break` to preserve semantics.

2.4.3 The *for* special case

As can be seen in section 2.4.2, the Ghidra decompiler does not have an explicit rule to recognize `for` loops. Indeed, `for` loops in Ghidra are treated as special cases of `while-do` loops[2]: The check is performed in the method `BlockWhileDo::finalTransform`, this method proceeds only if the block is not marked with overflow syntax.

1. **Loop variable identification:** `findLoopVariable` is called to search for a variable controlling the iteration (e.g., `i` in `i < 10`). This variable must appear in the exit condition and be modified within the loop body.
2. **Initializer identification:** `findInitializer` searches for the instruction that initializes the variable (e.g., `i = 0`) in the block immediately preceding the loop.
3. **Opcode relocation:** If both an iterator (`iterateOp`) and an initializer (`initializeOp`) are found, the decompiler physically moves the P-code operations (using `opUninsert` / `opInsertAfter`) so they lie adjacent to the loop boundaries, preparing them for syntactic emission.

[2]The transformation is triggered only if the architecture option `analyze_for_loops` is enabled.

4. **Non-printing marking:** In `finalizePrinting` these operations are marked with `opMarkNonPrinting`. This instructs the emitter not to print them as separate statements inside the body or before the loop, but to include them in the `for(...)` header.

```
{
  // Simplification style
  BlockGraph::finalTransform(data);
  if (!data.getArch()->analyze_for_loops) return;
  if (hasOverflowSyntax())
      return; // Still too complex
  FlowBlock *copyBl = getFrontLeaf();
  if (copyBl == (FlowBlock *)0) return;
  BlockBasic *head = (BlockBasic *)copyBl->subBlock(0);
  if (head->getType() != t_basic) return;
  PcodeOp *lastOp = getBlock(1)->lastOp(); // There must be a last op in body,
      for there to be an iterator statement
  if (lastOp == (PcodeOp *)0) return;
  BlockBasic *tail = lastOp->getParent();
  if (tail->sizeOut() != 1) return;
  if (tail->getOut(0) != head) return;
  PcodeOp *cbranch = getBlock(0)->lastOp();
  if (cbranch == (PcodeOp *)0 || cbranch->code() != CPUI_CBRANCH) return;
  if (lastOp->isBranch()) {      // Convert lastOp to -point- iterateOp must
      appear after
    lastOp = lastOp->previousOp();
    if (lastOp == (PcodeOp *)0) return;
  }

  findLoopVariable(cbranch, head, tail, lastOp);
  if (iterateOp == (PcodeOp *)0) return;

  if (iterateOp != lastOp) {
    data.opUninsert(iterateOp);
    data.opInsertAfter(iterateOp, lastOp);
  }

  // Try to set up initializer statement
  lastOp = findInitializer(head, tail->getOutRevIndex(0));
  if (lastOp == (PcodeOp *)0) return;
  if (!initializeOp->isMoveable(lastOp)) {
    initializeOp = (PcodeOp *)0;  // Turn it off
    return;
```

```
  }
  if (initializeOp != lastOp) {
    data.opUninsert(initializeOp);
    data.opInsertAfter(initializeOp, lastOp);
  }
}
```

If all conditions are met, the decompiler effectively transforms the `while-do` structure into a `for` loop by relocating and marking the relevant P-code operations.

2.4.4 The *Goto* Problem

A significant limitation of this approach arises when the CFG contains irreducible control flow that does not match any predefined schema. (This is common in binaries optimized with aggressive compiler techniques or those containing manual assembly optimizations).

When `ActionBlockStructure` fails to find a matching pattern, the jump inside the Flow-Block remains and it will be represented as a **goto** [3]. statement to preserve semantic correctness, this phenomenon significantly degrades the readability of the output.

2.5 Code Emission

The final phase of the pipeline is the translation of the structured High P-code into C syntax. This is not a simple text dump but a structured generation of an Abstract Syntax Tree (AST) represented by ClangToken objects.

Before emission, the **ActionNameVars** pass attempts to assign meaningful names to the recovered `HighVariable` objects. If debug symbols (DWARF, PDB) are available, they are utilized. In their absence, Ghidra relies on heuristics based on variable usage (e.g., loop counters named `i`, `j`) or storage location (e.g., `iVar1`, `uVar2`). This process is highly stochastic and often results in generic, non-descriptive identifiers.

The C++ backend generates a stream of `ClangToken` objects representing the code structure. This tokenized representation is sent to the Java frontend via the XML protocol. This structured data allows the Ghidra GUI to provide interactive features—such as cross-referencing and dynamic renaming—since the UI elements remain linked to the underlying `Varnode` and `HighVariable` objects.

[3] Or a `break`/`continue` if it jumps out of/into a loop structure

2.6 LLM

2.6.1 perplexity

Chapter 3

Related Work

This chapter is dedicated to the work that has been done by others in the field of your thesis. Don't write a mere list of papers with a summary of each. Instead, the goal is to give a picture of how your work is situated in the context of the work of others. In this case, you should think your skeptical reader is asking "hasn't this been solved already?".

There will be two main kinds of work that you will need to discuss: similar work to yours, and work that is complementary to yours.

Similar work will try to solve the same problem you are addressing, or a similar one. Here, focus on the differences: for example, you may write something along the lines of "Turing [Tur50] provides a naive definition of 'intelligent machine', which has this and that limitation. Our work is similar to that of Turing, but we address these problems by doing this and that."[1]

Complementary work does not try to solve the same problem you are addressing, but it is important to get a full picture of your contribution. For example, if you are *not* attempting to solve a particular problem (it is "out of scope" for your work), but the solution to that problem is necessary for your work to be useful, you should explain that here.

The Related Work chapter may end up in other places of the document. Consider that it does two pieces of work at the same time: *(i)* substantiating the claim you made in the Introduction (Chapter 1) that your problem is relevant, and *(ii)* describe the context around your work to help understand its impact. If the latter part is more important, it may make sense to move this section to the end of the document, just before the Conclusion (Chapter 6). If the Background (Chapter 2) is not necessary to understand this chapter, it is also a good idea to move it just after the Introduction (Chapter 1).

[1] Do not be arrogant! I called the work of Turing naive as a joke. For sure your work will have limitations that other work may not have, and you should be honest about them.

3.1 Sources

While doing your review of related work, Google Scholar[2] is probably the easiest search engine for academic papers. You can be mindful of some bibliometric tricks to understand which papers are likely to be most influential in the field. You can see on Scholar the number of citations of a paper; CORE[3] and Scimago[4] provide rankings of conferences and journals respectively. Non-peer-reviewed articles like those published only on arXiv[5] may, of course, be important, but you should be more careful and skeptical about them. In case of doubt, ask for guidance from your advisor.

Another tool that you should *use with caution* is Semantic Scholar.[6] It provides you with LLM-based tools to synthesize and ask questions about papers. Always double-check, because LLMs hallucinate and, as usual, you take responsibility for whatever you write in your document: treat LLMs as a useful tool you shouldn't trust.

Wikipedia is a great source of information, but you shouldn't consider it as reliable. Wikipedia articles should always provide the sources of the information they contain; you should check it yourself, judge its reliability, and cite the source.

In general, be careful about citing websites: make sure that they're authoritative, and when it's not obvious, justify why you consider them to be worth citing. Unlike this document, which I don't expect readers to print, your thesis must be readable in printed form: hence, make URLs readable in the text.

If a paper is very important for your own piece of work, have a look at the papers that cited it, to see if there are more recent works that are relevant to your thesis.

3.2 Style

Here are a few tricks about how to cite other pieces of work:

- If a paper is available in a peer-reviewed venue (journal or conference), cite that version rather than the non-reviewed one. Sometimes papers are published in two similar versions: one in a conference and an extended one in a journal. In that case, cite the journal.

[2] https://scholar.google.com
[3] https://portal.core.edu.au/conf-ranks/
[4] https://www.scimagojr.com
[5] https://arxiv.org
[6] https://www.semanticscholar.org/

- Use citations such as "The Hamiltonian cycle problem is NP-complete [Kar72]" or "Karp [Kar72] showed that the Hamiltonian cycle problem is NP-complete". Do not write "[Kar72] showed that the Hamiltonian cycle problem is NP-complete". Note, like in this case, that academics pretty much always refer to papers by citing authors' names rather than their titles.[7]
- DBLP[8] is a good source of quality Bibtex records for papers you may want to cite.
- You may have a doubt about whether to put a link in the text or use a reference in the bibliography. Use a reference if it is a document you want to refer to (even if it's not a text document, like a video or audio), use a link if you just want to refer to something but do not use it as a reference document.
- If you decide to leave links, make it so that they're usable even in printed form. The `\myhref` command of this document can be an example of how to do it.

[7] If you use LaTeX, check the source code for how to cite paper authors using the `\textcite` command.
[8] https://dblp.org

Chapter 4

Methodology

This chapter describes the core of your work: the methodology you used to solve the problem you described in the Introduction (Chapter 1). It is quite likely to have a different name (e.g., the name of the tool you wrote), and/or to be split into multiple chapters. *Methodology* suggests that you will spend space discussing your overall approach and justification for it: it is a good idea for a Master's Thesis, as opposed to just a list of things that you have done (which could be more correctly called *Method* or *Methods*, and would risk being boring and not very useful for the reader).

This chapter will likely be quite densely connected with things you have written in the Background (Chapter 2), and the results that you will show in the Results (Chapter 5). Put abundant references to help the reader navigate the document.

Try to keep a level of detail such that a skilled reader will be able to reproduce your result, but do not include unnecessary details that the reader can find out for themselves (e.g., the list of commands to perform a standard task). That is indeed valuable information that can help reproducibility, but if you do not think it is needed to understand your work, its place is likely to be in an appendix or in the documentation of the code you wrote, if you decide to release it.

This chapter should include work that has been done by you. If that is not the case, consider moving it to the Background (Chapter 2), or make it very clear that you are not the author. The same thing applies to images: if they aren't yours, you should cite the source.

The results of your work should generally not be included in this chapter. In some cases, preliminary results can be included to explain why you took a certain direction in your work (e.g., you took a choice rather than another because you measured that it was more efficient), but the overall evaluation of your work should be in the Results (Chapter 5).

You are likely to include algorithms in pseudocode. Check relevant LaTeX packages[1] for that.

[1] https://www.overleaf.com/learn/latex/Algorithms

Chapter 5

Results

Here you will present the results of your work. Your skeptical reader is now asking themself "is this working?". You will show here to what extent it does. Be honest about the limitations: if your system doesn't work in some cases, you should say so (and explain where and why, if you can). Like the Methodology chapter, this one may end up being split into multiple chapters.

Keep in mind that this shouldn't be a mere dump of experimental results; you're rather *teaching* your reader how and to what extent you managed to solve the problem you described in the Introduction (Chapter 1). If you have additional results that may be useful but are not necessary to understand the points you're making (e.g., you evaluated your system on multiple datasets and the results all tell the same story), the place for them is in an appendix.

This chapter should have a lot of links to the methodology chapter(s), because you're evaluating the choices you made there. If you developed a system made of multiple parts, make sure that you test them separately and together, so that the reader can understand how important each part is.

This chapter is likely to be quite full of figures and tables. Try to make them as informative as possible (e.g., use multiple lines in the same plot if possible). Showing graphs effectively is a complex art; try to spend some time on it and ask for guidance from your advisor. Whenever you have a figure or a table, make sure that you refer to it in the text (after all, if it's not referred to in the text it means it has no part in the story you're telling, so it has no place in this Chapter). Always use the text (and the caption, if you can) to explain what you want the reader to understand from the figure.

If you are using LaTeX, it will automatically place figures, tables, algorithms, etc. somewhere in the text. These parts of the documents are called *floats* because their position

Column 1	Column 2	Column 3
aaa	2	3.42
bbb	505	6.00
ccc	8	901.02

Table 5.1: A table using the LaTeX `booktabs` package. Note there are no vertical rules. In case you want to do more fancy stuff (e.g., merging cells), check also the `multirow` and `multicol` packages. It's generally a good idea for readability to align text to the left and numbers to the right (use the same number of significant digits).

floats around depending on LaTeX's will. There are options to advise LaTeX about where floats should be placed. My advice is not to waste too much time on this and trust that they will end up in a good place. Generally, write the floats' code before you refer to them: they will never be placed before the place they appear in.

For Figures, generally use vectorial formats (e.g., PDF, SVG) where it makes sense if you can. They will result in higher-quality images that are in most cases also smaller, leading to shorter compiling times and a smaller resulting PDF.

Tables look better without vertical rules: an example is in Table 5.1.

5.1 Discussion

At the end of this chapter, take a step back and summarize all the results so that the reader can understand the big picture. Sometimes, this becomes large and important enough to be worth a chapter on its own.

Chapter 6

Conclusion

You are finally done! Here you will summarize for the reader what you have taught them through the document, and the main takeaways of your work you would like them to remember.

It is also a good idea to discuss the limitations of your work and your views about what can be possible future work.

Bibliography

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2%5C_9.

[NSA] NSA. *Ghidra Decompiler Analysis Engine: Decompiler Analysis Engine — share.google*. https://share.google/Q9gHjuuTY3ZlFlm4n. [Accessed 16-01-2026].

[Pco] Pcode-doc. *P-Code Reference Manual — spinsel.dev*. https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html. [Accessed 19-01-2026].

[Tur50] Alan M. Turing. "Computing machinery and intelligence". In: *Mind* LIX.236 (1950), pp. 433–460. DOI: 10.1093/MIND/LIX.236.433. URL: https://doi.org/10.1093/mind/LIX.236.433.