

## BACHELOR THESIS

of the course Computer Science with a focus on Artificial Intelligence at the Baden-Wuerttemberg Cooperative State University Mannheim

# Enabling Vulnerability Detection in RISC-V based Binaries through Lifting RISC-V ISA to LLVM

from

**Laura van Uffelt**

Submission date: 04.09.2026

|                                 |                                                |
|---------------------------------|------------------------------------------------|
| <b>Course:</b>                  | TINF23IKI1                                     |
| <b>Processing Period:</b>       | 02.06.2026 – 04.09.2026                        |
| <b>Apprenticing company:</b>    | Deutsches Zentrum für Luft- und Raumfahrt e.V. |
| <b>Department:</b>              | Institute of Software Technology               |
| <b>Company's Supervisor:</b>    | Dr. Ing. Zain A. H. Hammadeh                   |
| <b>University's Supervisor:</b> | Dr. Marcel C. Werner                           |

# Declaration

I hereby assure you that I have written my Bachelor Thesis on the subject

Enabling Vulnerability Detection in RISC-V based Binaries through Lifting RISC-V  
ISA to LLVM

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.

If both versions are required.

I used AI tools while writing this thesis. I have indicated this in the relevant sections of the thesis.

Braunschweig, 04.09.2026

---

Place, Date

---

Signature

# Abstract

Binary analysis techniques are commonly used to identify security vulnerabilities in software. However, many established analysis tools rely on intermediate representations that are primarily designed for specific architectures or require dedicated front ends. The growing relevance of the RISC-V instruction set architecture therefore motivates the development of approaches that enable the analysis of RISC-V binaries using existing analysis infrastructures.

This thesis investigates the lifting of RISC-V binaries to LLVM IR as a foundation for subsequent vulnerability detection. A prototype lifting pipeline was developed that processes RISC-V ELF binaries in multiple stages. First, executable code and function information are extracted from the input binary. The contained instructions are then identified using a custom decoder and transformed into an internal representation. Based on the decoded instructions, basic blocks and a control-flow graph are constructed. SSA form is subsequently generated using a dominator-based approach, including the placement of  $\phi$ -nodes and variable renaming. Finally, the resulting representation is translated into LLVM IR.

The implementation focuses on the RV64I base instruction set and supports a selected subset of instructions identified through an analysis of representative compiled programs. The generated LLVM IR was evaluated using several test programs and compared with the expected program structure and behavior.

The results demonstrate that the implemented pipeline can successfully reconstruct control flow and translate supported RISC-V instructions into valid LLVM IR for the considered examples.

The prototype also reveals limitations of the current approach, particularly regarding incomplete instruction coverage, indirect control-flow transfers, interprocedural analysis, and the simplified memory model. Consequently, the presented implementation does not perform vulnerability detection directly, but provides a basis for integrating RISC-V binaries into LLVM-based analysis frameworks. Future work can extend the instruction coverage, improve the modelling of memory and control flow, and connect the generated LLVM IR to established vulnerability detection techniques.

# Zusammenfassung

Analysen von Binärdateien werden häufig eingesetzt, um Sicherheitslücken in Software zu identifizieren. Viele etablierte Analysewerkzeuge stützen sich jedoch auf Zwischenrepräsentationen, die in erster Linie für bestimmte Architekturen konzipiert sind oder spezielle Frontends erfordern. Die wachsende Bedeutung der RISC-V-Befehlssatzarchitektur motiviert daher die Entwicklung von Ansätzen, die die Analyse von RISC-V-Binärdateien unter Verwendung bestehender Analyseinfrastrukturen ermöglichen.

Diese Arbeit untersucht die Umwandlung von RISC-V-Binärdateien in LLVM-IR als Grundlage für die anschließende Schwachstellenerkennung. Es wurde eine prototypische Umwandlungspipeline entwickelt, die RISC-V-ELF-Binärdateien in mehreren Schritten verarbeitet. Zunächst werden ausführbarer Code und Funktionsinformationen aus der Eingabebinarydatei extrahiert. Die enthaltenen Befehle werden dann mithilfe eines eigenen Decoders dekodiert und in eine interne Darstellung umgewandelt. Auf der Grundlage der dekodierten Befehle werden Basisblöcke und ein Kontrollflussgraph konstruiert. Anschließend wird mithilfe eines dominatorbasierten Ansatzes die SSA-Form generiert, einschließlich der Platzierung von  $\phi$ -Knoten und der Variablenumbenennung. Schließlich wird die resultierende Darstellung in LLVM-IR übersetzt.

Die Implementierung konzentriert sich auf den RV64I-Basisbefehlssatz und unterstützt eine ausgewählte Teilmenge von Befehlen, die durch eine Analyse repräsentativer kompilierter Programme ermittelt wurde. Die generierte LLVM-IR wurde anhand mehrerer Testprogramme evaluiert und mit der erwarteten Programmstruktur und dem erwarteten Programmverhalten verglichen. Die Ergebnisse zeigen, dass die implementierte Pipeline den Kontrollfluss erfolgreich rekonstruieren und unterstützte RISC-V-Befehle für die betrachteten Beispiele in gültige LLVM-IR übersetzen kann.

Der Prototyp deckt zudem Einschränkungen des aktuellen Ansatzes auf, insbesondere hinsichtlich unvollständiger Befehlsabdeckung, indirekter Kontrollflussübergänge, prozedurübergreifender Analyse und des vereinfachten Speichermodells. Folglich führt die vorgestellte Implementierung keine direkte Schwachstellenerkennung durch, bietet jedoch eine Grundlage für die Integration von RISC-V-Binärdateien in LLVM-basierte Analyse-Frameworks. Zukünftige Arbeiten können die Befehlsabdeckung erweitern, die

Modellierung von Speicher und Kontrollfluss verbessern und das generierte LLVM-IR mit etablierten Techniken zur Schwachstellenerkennung verknüpfen.

# Contents

|                                                    |            |
|----------------------------------------------------|------------|
| <b>List of Figures</b> .....                       | <b>iii</b> |
| <b>List of Tables</b> .....                        | <b>iv</b>  |
| <b>List of Listings</b> .....                      | <b>v</b>   |
| <b>Acronyms Index</b> .....                        | <b>vi</b>  |
| 1. Introduction .....                              | 1          |
| 1.1. Motivation .....                              | 1          |
| 1.2. Problem .....                                 | 2          |
| 1.3. Objective .....                               | 2          |
| 2. Background .....                                | 4          |
| 2.1. The RISC-V Instruction Set Architecture ..... | 4          |
| 2.1.1. Base ISAs and Extensions .....              | 4          |
| 2.1.2. Registers .....                             | 5          |
| 2.1.3. Instruction Formats .....                   | 6          |
| 2.1.4. Use Cases of RISC-V .....                   | 7          |
| 2.2. Binary Formats .....                          | 9          |
| 2.2.1. .text Section .....                         | 10         |
| 2.2.2. .symtab Section .....                       | 10         |
| 2.3. Binary Analysis .....                         | 10         |
| 2.3.1. Disassembly .....                           | 11         |
| 2.3.2. Control Flow Graph .....                    | 11         |
| 2.4. Intermediate Representation .....             | 12         |
| 2.4.1. Static Single Assignment .....              | 13         |
| 2.4.2. LLVM IR .....                               | 19         |
| 2.5. DEL (Dynamic Execution Lifter) .....          | 20         |
| 2.6. Related Work .....                            | 21         |
| 2.6.1. McSema .....                                | 22         |
| 2.6.2. RevGen .....                                | 22         |
| 2.6.3. Rellume .....                               | 23         |
| 2.6.4. Forklift Neural Lifter .....                | 23         |

|                                                            |           |
|------------------------------------------------------------|-----------|
| 2.6.5. Ghidra .....                                        | 24        |
| 3. Methodology .....                                       | 25        |
| 3.1. Input Binaries .....                                  | 25        |
| 3.2. Intended Workflow .....                               | 25        |
| 3.3. Binary Parsing .....                                  | 26        |
| 3.4. Instruction Decoding .....                            | 27        |
| 3.4.1. riscv-none-elf-objdump .....                        | 27        |
| 3.4.2. LLVM objdump .....                                  | 28        |
| 3.4.3. Capstone .....                                      | 28        |
| 3.4.4. Comparison and Decision .....                       | 29        |
| 3.5. Choice of SSA .....                                   | 30        |
| 3.6. Instruction Translation .....                         | 30        |
| 3.7. Memory Allocation .....                               | 32        |
| 4. Implementation .....                                    | 33        |
| 4.1. Decoding the Binary File .....                        | 35        |
| 4.2. CFG Reconstruction .....                              | 37        |
| 4.2.1. Identify Leader Instructions .....                  | 38        |
| 4.2.2. Building Basic Blocks .....                         | 39        |
| 4.2.3. Building the CFG .....                              | 40        |
| 4.3. SSA Implementation .....                              | 41        |
| 4.4. LLVM Generation .....                                 | 46        |
| 4.4.1. Memory Representation .....                         | 46        |
| 4.4.2. Function/Basic-Block and Instruction Emission ..... | 47        |
| 4.4.3. CFG Terminator Emission .....                       | 49        |
| 5. Evaluation .....                                        | 50        |
| 5.1. Security-Related Program Evaluation .....             | 52        |
| 5.2. Implementation Challenges and Limitations .....       | 55        |
| 6. Conclusion and Further Work .....                       | 57        |
| <b>Bibliography .....</b>                                  | <b>59</b> |
| <b>AI Acknowledgement .....</b>                            | <b>63</b> |

# List of Figures

|           |                                                                                              |    |
|-----------|----------------------------------------------------------------------------------------------|----|
| Figure 1  | RISC-V instruction encodings .....                                                           | 7  |
| Figure 2  | RISC-V Market Penetration for Revenues by Major Category .....                               | 8  |
| Figure 3  | Simplified ELF binary layout for the array reduction program example ....                    | 9  |
| Figure 4  | Structure of DEL [1] .....                                                                   | 21 |
| Figure 5  | intended workflow .....                                                                      | 26 |
| Figure 6  | Five most frequently occurring instructions in the analyzed binaries .....                   | 31 |
| Figure 7  | Memory Representation: RISC-V vs. LLVM IR .....                                              | 32 |
| Figure 8  | Workflow of the presented implementation .....                                               | 34 |
| Figure 9  | reconstructed CFG .....                                                                      | 41 |
| Figure 10 | Comparison of the lifted and original RISC-V binaries. ....                                  | 51 |
| Figure 11 | Control-flow graph recovered from the binary containing the PolyRhythm<br>test program. .... | 53 |

# List of Tables

|         |                                                                                                                 |    |
|---------|-----------------------------------------------------------------------------------------------------------------|----|
| Table 1 | General-purpose registers defined by the RISC-V ABI. Adapted from the RISC-V ELF psABI Specification. [2] ..... | 6  |
| Table 2 | Comparison of Binary disassembly tools .....                                                                    | 29 |
| Table 3 | Example of RISC-V instructions translated into LLVM IR .....                                                    | 48 |
| Table 4 | Description of Used AI Tools .....                                                                              | 63 |

# List of Listings

|           |                                                                    |    |
|-----------|--------------------------------------------------------------------|----|
| Listing 1 | Implementation of the decoder for RISC-V I-type instructions. .... | 36 |
| Listing 2 | Basic Block definition .....                                       | 39 |
| Listing 3 | example C file used for evaluation .....                           | 50 |
| Listing 4 | PolyRhythm attack implemented in C .....                           | 52 |
| Listing 5 | Extract of the generated LLVM file .....                           | 54 |

# Acronyms Index

|              |                                                |
|--------------|------------------------------------------------|
| <b>ABI:</b>  | Application Binary Interface                   |
| <b>API:</b>  | Application Programming Interface              |
| <b>ARM:</b>  | Advanced RISC Machines                         |
| <b>CFG:</b>  | Control Flow Graph                             |
| <b>CSR:</b>  | Control and Status Register                    |
| <b>DEL:</b>  | Dynamic Execution Lifter                       |
| <b>DLR:</b>  | Deutsches Zentrum für Luft- und Raumfahrt e.V. |
| <b>DSE:</b>  | Dynamic Symbolic Execution                     |
| <b>ELF:</b>  | Executable and Linking Format                  |
| <b>IDE:</b>  | Integrated Development Environment             |
| <b>IR:</b>   | Intermediate Representation                    |
| <b>ISA:</b>  | Instruction Set Architecture                   |
| <b>LLVM:</b> | Low Level Virtual Machine                      |
| <b>NSA:</b>  | National Security Agency                       |
| <b>RISC:</b> | Reduced Instructions Set Computer              |
| <b>SMT:</b>  | Satisfiability Modulo Theories                 |
| <b>SSA:</b>  | Single Static Assignments                      |
| <b>pc:</b>   | program counter                                |
| <b>ra:</b>   | return address                                 |
| <b>sp:</b>   | stack pointer                                  |

# 1. Introduction

The Flight Software Department at the Institute of Software Technology is part of the German Aerospace Center (DLR). The department's research focuses on developing reliable flight software, including control software for payloads and embedded systems. The department's Real-Time Software and Security group addresses safety-critical systems, aiming to ensure and improve important quality properties such as timing, safety and cybersecurity [3].

## 1.1. Motivation

Software obtained from external sources is often available only in the form of binary executables. When the corresponding source code is unavailable, the feasibility of traditional debugging, code review, and security verification techniques is limited. The lack of source code makes it more difficult to identify compromised or unintended functionality and increases the challenges associated with software supply-chain security. Analysis must therefore be performed directly on the executable. Furthermore, binaries are closely tied to their target hardware architecture and execution environment, making their analysis more challenging and less portable. Therefore, it is quite difficult to detect vulnerabilities in the binary itself.

One approach to overcoming these challenges is binary lifting, which involves translating a binary executable into an intermediate representation (IR). The IR provides a more structured, architecture-independent representation of the program, making it suitable for subsequent automated analysis. This makes it possible to perform analyses that would otherwise require access to the original source code.

In this work, the lifted representation is intended to serve as a basis for static analysis. Static analysis allows program properties and behavior to be examined without executing the program. While this is typically performed on source code, binary lifting allows static analysis techniques to be applied when the original source code is unavailable. The lifted representation can therefore serve as a foundation for analyses such as control-flow reconstruction, data-flow analysis and vulnerability detection.

## 1.2. Problem

To enable reliable analysis and vulnerability detection in binary executables, several challenges must be addressed. Since source-level information is unavailable, the analysis must recover sufficient program structure and semantics directly from the binary representation.

A major challenge arises from the loss of information introduced during compilation. Source-level concepts such as variables, data types, function boundaries, and high-level control-flow structures may no longer be explicitly represented in the executable. Recovering this information automatically is difficult, while manually providing annotations can be time-consuming and error-prone.

Another challenge is the reconstruction of program control flow. Direct branches and function calls generally contain explicit target addresses and can therefore be resolved relatively easy. In contrast, indirect control-flow transfers like indirect jumps do not explicitly encode their destinations. These can occur, for example, through switch statements implemented using jump tables, function pointers, or virtual method dispatch. Failing to resolve such targets correctly can result in incomplete or incorrect control-flow graphs, which subsequently affects analyses based on them.

Therefore, a reliable binary-lifting approach must do more than translate individual machine instructions. It must also recover sufficient control-flow and program-state information to preserve the semantics relevant for subsequent analysis.

## 1.3. Objective

Existing binary-lifting approaches demonstrate the feasibility of translating native binaries into Low Level Virtual Machine IR (LLVM IR), but support and implementation strategies differ across architectures and lifting objectives.

RISC-V is increasingly relevant for embedded and research applications due to its open instruction set architecture and extensibility. Therefore, extending binary-lifting capabilities to RISC-V can provide similar analysis capabilities for RISC-V-based software.

This thesis presents a prototype implementation of a binary-lifting approach for RISC-V binaries that reconstructs program control flow and translates the recovered instructions into LLVM IR. The resulting representation is intended to preserve the relevant program semantics while providing a suitable basis for subsequent static analysis.

## 2. Background

This chapter introduces the RISC-V Instruction Set Architecture (ISA) as well as DEL and other tools which are the foundation of this work.

### 2.1. The RISC-V Instruction Set Architecture

RISC-V originated at UC Berkeley as a research project in 2010, gaining traction with its first specification release in 2016 [4]. While it started for educational purposes, its goal quickly devolved to providing a free, open ISA that could bridge the gap between academia and industry [5].

At its core, RISC-V is built on a minimal integer instruction set designed to be modular. Instead of one closed system, it relies on a base set that can be extended via standard optional modules. These range from floating-point arithmetic and atomic operations to advanced vector processing. Because the architecture allows custom user-defined extensions, developers can add application-specific logic where needed.

A defining feature of RISC-V is its implementation independence. The ISA specification separates “what” the processor does from “how” it is built, allowing for massive flexibility ranging from tiny embedded chips to high-performance servers. It natively supports heterogeneous multicore environments and, depending on extensions like the Hypervisor extension, scales well for virtualization.

RISC-V typically offers three variants: RV32 (32-bit) and RV64 (64-bit), with a newer 128-bit version (RV128) that is still emerging. Like Advanced RISC Machines (ARM), it employs a load/store architecture. Meaning only load and store instructions can access memory, while arithmetic operations occur strictly within registers. This keeps the hardware simple and performance predictable [6].

#### 2.1.1. Base ISAs and Extensions

RISC-V defines base integer instruction sets, which differ in the width of their general-purpose registers, the size of the address space they can represent and the number of available registers. The main variants are RV32I and RV64I, which define 32-bit and 64-

bit integer architectures respectively. The specification also defines the RV32E variant, which has fewer registers and is intended for embedded systems, as well as the experimental RV128I 128-bit variant, which is not yet widely implemented.

RISC-V separates its ISA into a small base integer ISA and a set of optional extensions, allowing additional functionality to be added without expanding the base ISA [7]. Additional functionality is provided through optional standard extensions, including M (integer multiplication and division), A (atomic operations), F and D (floating-point arithmetic), C (compressed instructions), and V (vector processing). This modular design enables processor implementations to incorporate only the necessary functionality for their intended application, while ensuring compatibility with software targeting the same ISA profile.

The modular structure of RISC-V also directly influences binary lifting. A binary lifter must implement the decoding and semantics of every instruction it supports. Consequently, supporting additional ISA extensions requires corresponding additions to the decoder and semantic translation layer [7]. Due to the scope of this thesis the implementation presented here targets the RV64I base integer instruction set only.

### 2.1.2. Registers

The RISC-V base integer ISA defines 32 general-purpose registers, denoted as x0 through x31. In the RV32I base ISA, these registers are 32 bits wide, whereas in RV64I they are 64 bits wide. Register x0 is hardwired to the constant value zero, while registers x1–x31 are general-purpose registers that hold operands and results for arithmetic, logical, memory, and control-flow instructions. Their contents are untyped and may be interpreted by different instructions as signed integers, unsigned integers, memory addresses, or bit vectors.

In addition to the general-purpose registers, the architecture defines the program counter (pc), which stores the address of the instruction currently being executed. The ISA does not reserve any general-purpose register for a specific purpose. Nevertheless, the standard RISC-V Application Binary Interface (ABI) assigns conventional roles to several registers. In particular, register x1 serves as the return address register (ra), while register x2 is used as the stack pointer (sp). These conventions enable interoperability

between independently compiled programs and libraries [7]. Table 1 shows the 32 registers and their corresponding ABI names as well as their purpose and if they are preserved across function calls. Preserved indicates whether the value stored in the register must be preserved across a function call according to the RISC-V calling convention. Callee-saved registers must be restored by the called function before returning, whereas caller-saved registers may be modified by the called function [2].

| Register | ABI Name | Purpose                            | Preserved? |
|----------|----------|------------------------------------|------------|
| x0       | zero     | Constant zero                      | -          |
| x1       | ra       | Return address                     | No         |
| x2       | sp       | Stack pointer                      | Yes        |
| x3       | gp       | Global pointer                     | -          |
| x4       | tp       | Thread pointer                     | -          |
| x5–x7    | t0–t2    | Temporary registers                | No         |
| x8–x9    | s0–s1    | Saved registers (frame pointer)    | Yes        |
| x10–x17  | a0–a7    | Function arguments / return values | No         |
| x18–x27  | s2–s11   | Saved registers                    | Yes        |
| x28–x31  | t3–t6    | Temporary registers                | No         |

Table 1: General-purpose registers defined by the RISC-V ABI. Adapted from the RISC-V ELF psABI Specification. [2]

### 2.1.3. Instruction Formats

RISC-V instructions are divided into different instruction formats. They determine how an instruction is encoded and decoded. All instructions contain an *opcode* field, which identifies the instruction format and the general class of operation. Additional fields, such as *funct3* and, where applicable, *funct7*, further distinguish individual instructions that share the same opcode. [7]

The RV64I base ISA defines six instruction formats: R-type, I-type, S-type, B-type, U-type, and J-type. Except for the optional compressed extension, all instructions are fixed at 32 bits in length. To simplify instruction decoding, the source registers (rs1 and rs2) and the destination register (rd) are located at fixed bit positions across all instruction formats. Immediate operands are encoded differently depending on the instruction format but are generally arranged to reduce hardware complexity. Unless otherwise

specified, immediate values are sign-extended to the processor's register width. The only exception is the 5-bit immediate (zimm) used by certain Control and Status Register (CSR) instructions, which is zero-extended. [7]

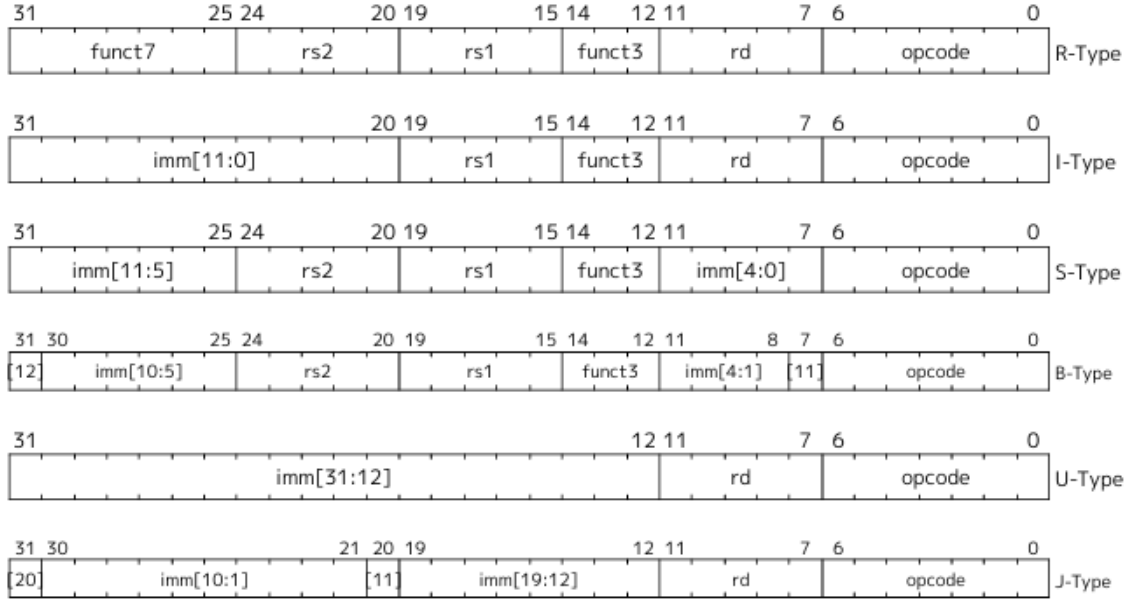


Figure 1: RISC-V instruction encodings

Figure 1 illustrates the six instruction formats and their corresponding encodings defined by the RV64I base ISA [7]. Understanding these formats is essential for instruction decoding and therefore forms the basis of the lifting process.

During lifting, each decoded instruction is translated into an equivalent sequence of LLVM IR instructions. Consequently, correctly identifying the instruction format and reconstructing its operands is a prerequisite for preserving the semantics of the original binary.

#### 2.1.4. Use Cases of RISC-V

Over the past five years, RISC-V has become increasingly relevant as an ISA. As shown in Figure 2 [8], revenues associated with RISC-V-based products have at least doubled across the major categories presented in the figure and are projected to increase further through 2031. RISC-V is particularly relevant to embedded and research applications due to its open ISA and extensible architecture.

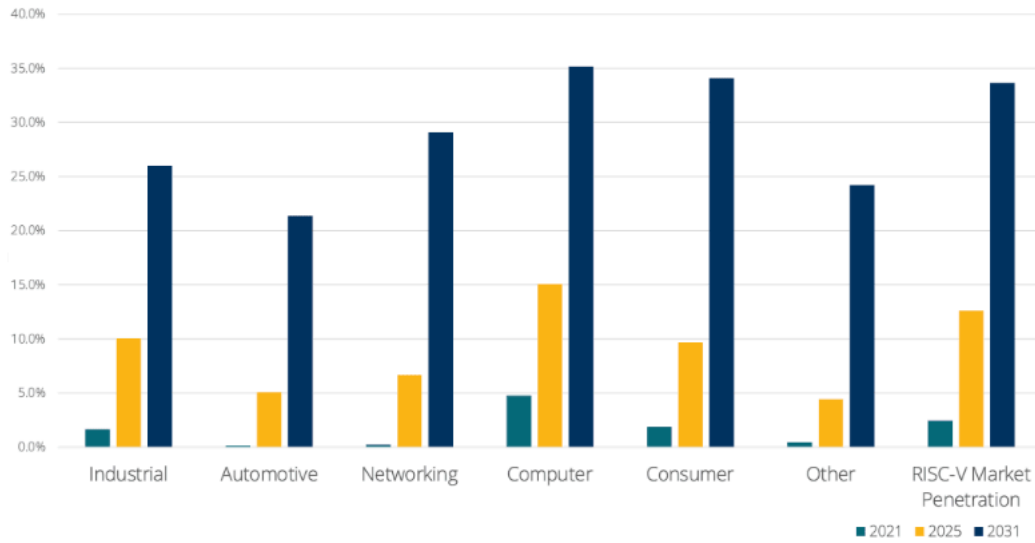


Figure 2: RISC-V Market Penetration for Revenues by Major Category

RISC-V is used across a range of applications, including embedded computing, security-sensitive workloads, and specialized systems. Its extensible ISA allows processor implementations to incorporate functionality tailored to specific application requirements.

Security and cryptography represent one example of this extensibility. The RISC-V specification defines dedicated scalar cryptography extensions designed for architectures ranging from embedded IoT-class processors to application-class cores [9].

A concrete example of RISC-V adoption in embedded and specialized computing is Microchip’s PolarFire® SoC family, which integrates RISC-V processor cores with FPGA fabric and supports Linux and real-time applications. Microchip has also developed a radiation-tolerant PolarFire® SoC based on RISC-V for mission-critical space applications, including payload and avionics development [10].

RISC-V is also being investigated for specialized software platforms in the space domain. The Stellar Apps project, run by DLR, provides a platform for the secure and isolated execution of software applications on spacecraft, including satellites, rovers and rockets. It supports applications such as spacecraft control, scientific experiments, sensor data processing and AI-based data analysis. Stellar Apps is designed to support multiple computing architectures and includes a RISC-V-based reference platform alongside ARM- and x86-based platforms [11].

These examples demonstrate that RISC-V is being deployed in applications where software can perform security-sensitive or mission-critical functions. When such software is available only as compiled binaries, techniques for analyzing the resulting machine code become increasingly relevant. Binary lifting provides one approach to this problem by translating architecture-specific machine code into an intermediate representation suitable for subsequent program analysis.

## 2.2. Binary Formats

The Executable and Linkable Format (ELF) is the standard binary file format used by many Unix-like operating systems for executables, object files, shared libraries, and core dumps [12]. Depending on their purpose, ELF files are classified into relocatable files, executable files, shared object files, and core files. Since this work focuses on lifting executable programs, executable ELF files are of primary interest.

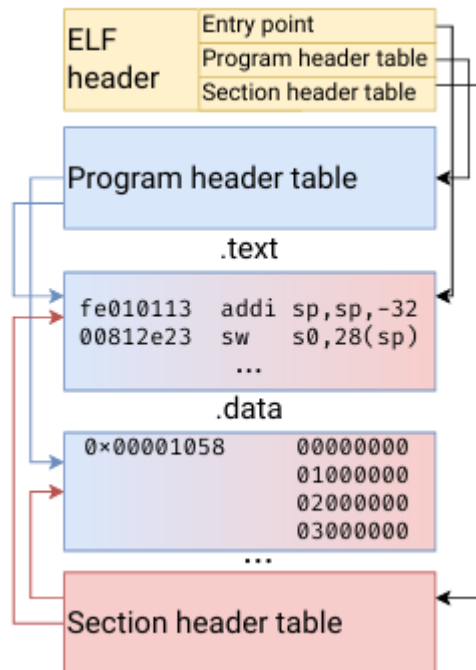


Figure 3: Simplified ELF binary layout for the array reduction program example

An ELF file consists of an ELF header followed by a program header table, a section header table, or both, as shown in Figure 3 [13]. The ELF header contains metadata

describing the file, including the location and size of these tables. The program header table specifies how the operating system loads the executable into memory, whereas the section header table describes the individual sections contained in the file, such as executable code, initialized data, and symbol information. These sections provide the information required for static analysis and binary lifting [14].

### **2.2.1. .text Section**

The .text section contains the executable machine instructions of the program. During execution, this section is typically loaded into memory with read and execute permissions. Since the instructions to be lifted are stored in this section, it serves as the primary input for the decoder. The decoder sequentially reads the instructions from the .text section and translates them into the internal instruction representation used throughout the lifting process [12].

### **2.2.2. .symtab Section**

The .symtab section contains the symbol table, which stores information about symbols defined within the binary, including functions and global variables. Each symbol entry includes metadata such as its name, address, size, and type. In this work, the symbol table is used to identify function entry points and their corresponding sizes. This information allows the lifting process to analyze each function independently and limits control-flow reconstruction to the boundaries of the current function. [12]

## **2.3. Binary Analysis**

Binary analysis comprises of techniques for recovering and analyzing information about a program directly from its executable representation. In contrast to source-code analysis, binary analysis must operate without access to much of the high-level information available during compilation, such as variable names, data types, and explicit control-flow structures. Two fundamental steps in binary analysis are the recovery of machine instructions through disassembly and the reconstruction of the program's control-flow graph. [15], [16]

### 2.3.1. Disassembly

Disassembly is the process of recovering a symbolic representation of machine-code instructions from their binary representation. Two general approaches can be distinguished: static and dynamic disassembly.

Static disassembly analyzes the binary without executing the program. It identifies and decodes machine-code instructions directly from the binary file. Dynamic disassembly, in contrast, observes instructions during program execution and decodes the instructions encountered at runtime [15].

Static disassembly was considered in this work because the objective is to analyze and lift the binary without executing the original program. Relying on dynamic disassembly would require executing the binary. This is not desirable since the binary could contain malicious code, which then would be executed.

### 2.3.2. Control Flow Graph

A control-flow graph (CFG) is a representation of the possible execution paths of a program and forms the basis for many program analysis techniques, such as data-flow analysis, optimization, and binary lifting [17]. In the context of this work, reconstructing the control flow is essential, as lifting individual instructions alone would not preserve the program's execution semantics.

A CFG is a directed graph in which each node represents a basic block and each edge represents a possible transfer of control between two basic blocks. A basic block consists of a maximum number of consecutive instructions with one entry point and one exit point. Control enters the block only at its first instruction and leaves it only at its last instruction, without branching into or out of the block in between.

The CFG is constructed by connecting the basic blocks according to the possible control-flow transfers between them. Consequently, the basic blocks form the nodes of the graph, while the edges represent the potential execution paths of the program [17].

A basic block may have multiple predecessors and successors depending on the program's control flow. For example, conditional branch instructions introduce two successor blocks corresponding to the taken and non-taken execution paths, while mul-

multiple branches may converge at the same basic block, resulting in multiple predecessors. Within a function-level CFG, the entry block has no predecessors, while a terminating block, such as one ending in a return instruction, has no successors. A function-level CFG describes only the control flow within a function but not between functions.

Each basic block begins with exactly one leader instruction and ends with exactly one tail instruction. The leader marks the unique entry point of the block, while the tail instruction is the last instruction executed before control is transferred to another basic block or the program continues sequentially with the following basic block.

Leader instructions are identified according to the possible control-flow transfers within the program. The first instruction of a function is always considered a leader. In addition, the target of every branch or jump instruction is a leader, as it represents a possible entry point into a basic block. For conditional branches, the instruction immediately following the branch is also considered a leader because execution may continue sequentially if the branch condition is not satisfied.

The tail instruction of a basic block is either a control-flow instruction, such as a branch, jump, or return, or the instruction immediately preceding the next leader. Consequently, every basic block contains exactly one leader and one tail instruction, with zero or more ordinary instructions located between them [1].

The reconstructed basic blocks provide a natural mapping to LLVM IR basic blocks, while the recovered control-flow edges can be represented using the corresponding LLVM terminator instructions. The accuracy of the recovered CFG therefore directly affects the semantic correctness and usefulness of the generated LLVM IR.

## 2.4. Intermediate Representation

IRs are used to manage the complexity of the compilation process by providing a common representation of a program between different compiler phases. An IR is a data structure that captures the essential semantics of a program required for subsequent compiler stages, enabling analysis, optimization, and code generation. It acts as an interface between different compiler components, allowing each stage to operate on

a well-defined representation independent of the original source language or target architecture.

An IR provides a bridge between high-level programming languages, such as C or C++, and low-level machine code. Intermediate representations can vary significantly in their abstraction level: some are closer to the structure of the source language, while others resemble machine instructions more closely and reveal the low-level operations necessary for optimization and target-specific code generation. Modern compiler frameworks commonly employ multiple IRs at different abstraction levels throughout the compilation pipeline.

To support multiple source languages and target architectures, an IR is typically designed to be largely independent of both. This separation reduces compiler complexity by allowing different front ends to translate source languages into a common representation and different back ends to generate code for various target architectures. Consequently, IRs improve compiler modularity, extensibility, and code reuse [18].

In the context of binary lifting, an IR provides an abstraction layer between architecture-specific machine instructions and higher-level programs. By translating binary code into an intermediate representation, architecture-dependent details can be separated from subsequent analysis stages. This enables techniques such as control-flow analysis, data-flow analysis, optimization and, potentially, recompilation to be performed on a more uniform representation.

#### **2.4.1. Static Single Assignment**

Static Single Assignment (SSA) is a form of intermediate representation used in compilers that facilitates many code optimizations. Its defining property is that each variable is assigned exactly once, with every assignment creating a new version of the variable. Consequently, variables are never reassigned. When different control-flow paths produce different versions of the same source-level variable, SSA uses  $\varphi$  (phi) functions at control-flow merge points to combine them into a single value [19].

SSA offers several important advantages for compiler analysis and optimization. Since each SSA variable has exactly one definition, data-flow relationships between definitions and uses are made explicit. This simplifies analyses and optimization algorithms,

including constant propagation, dead-code elimination, and value numbering, because the compiler no longer needs to reason about multiple assignments to the same variable.

Another property is that the SSA representation remains linear in the size of the original program. Converting a program into SSA form introduces additional definitions and may require the insertion of  $\varphi$ -functions at control-flow merge points. However, the number of additional SSA constructs is bounded by the structure of the original program and its control-flow graph. Although the construction of SSA may introduce additional variables and  $\varphi$ -functions, the number of these additional elements is bounded by the size of the control-flow graph. Consequently, SSA does not introduce an asymptotic increase in program size and can therefore be constructed and manipulated efficiently. This combination of explicit data dependencies and linear representation size is one of the main reasons SSA has become a fundamental intermediate representation in modern compilers [20].

The assignment of variable names is straightforward in linear code, where there is no branching. Each new assignment creates a new version of the variable. For example:

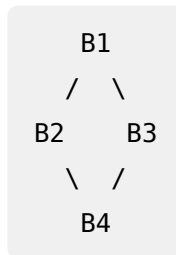
```
x = 1
x = x + 2
x = x * 3
```

can be transformed into SSA form as:

```
x_1 = 1
x_2 = x_1 + 2
x_3 = x_2 * 3
```

At every point in the program, there is therefore a unique definition of `x` that reaches the current instruction.

The situation becomes more complicated when the control flow branches and subsequently merges. Consider the following control-flow graph:



Suppose both branches assign a new value to `x` :

```
B2:
    x_2 = 10

B3:
    x_3 = 20
```

When execution reaches `B4` , either `x_2` or `x_3` may be the value of `x` , depending on which predecessor was executed. Consequently, there is no single variable version that can simply be used as the current definition of `x` .

SSA resolves this ambiguity by introducing a  $\varphi$ -function at the control-flow merge point:

```
B4:
    x_4 = phi(x_2, x_3)
```

The  $\varphi$  function conceptually selects the value corresponding to the predecessor through which control flow entered `B4` . If execution arrived from `B2` , `x_4` receives `x_2` ; if it arrived from `B3` , `x_4` receives `x_3` .

Thus, the purpose of the  $\varphi$ -function is not to perform a conventional runtime function call. Instead, it represents the merging of different definitions of the same source-level variable at a control-flow join point. This allows SSA to maintain its fundamental property that every SSA variable has exactly one definition [20].

The placement of  $\varphi$ -functions in SSA form is closely related to the concepts of dominators, dominator trees, and dominance frontiers. These concepts describe the structure of a control-flow graph and provide an efficient method for determining where different definitions of a variable must be merged.

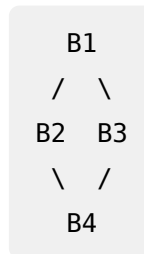
Given a CFG with an entry block, a basic block  $A$  is said to **dominate** another block  $B$  if every possible path from the entry block to  $B$  passes through  $A$ . In other words, execution cannot reach  $B$  without first executing  $A$ .

Formally, a block  $A$  dominates a block  $B$ , written as:

$$A \text{ dominates } B$$

if every path from the entry block to  $B$  contains  $A$ . Every block dominates itself, and the entry block dominates all blocks that are reachable from it.

For example, consider the following control-flow graph:



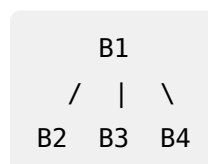
In this graph,  $B1$  dominates  $B2$ ,  $B3$ , and  $B4$ , because every path from the entry block to these blocks passes through  $B1$ . However,  $B2$  does not dominate  $B4$ , since  $B4$  can also be reached through  $B3$ . Similarly,  $B3$  does not dominate  $B4$  [21].

Dominators are particularly important in SSA construction because they provide a framework for determining which definitions can reach particular uses of a variable.

For every block, except the entry block, there is a unique immediate dominator. The immediate dominator of a block  $B$ , denoted by  $\text{idom}(B)$ , is the strict dominator of  $B$  that is closest to  $B$  in the control-flow graph.

The immediate dominator relationship can be represented as a tree called the dominator tree. The entry block forms the root of the tree, and every other block has exactly one parent corresponding to its immediate dominator.

For the previous example, the dominator tree is:



Here,  $B_1$  is the immediate dominator of  $B_2$ ,  $B_3$ , and  $B_4$ . The dominator tree provides an efficient hierarchical representation of dominance relationships and is commonly used during SSA construction and variable renaming.

Once  $\varphi$ -functions have been inserted, the dominator tree can be traversed to assign unique SSA versions to variables. During this traversal, a new version is created whenever a variable is defined, while uses of a variable refer to the currently active version [21].

While dominators describe where a definition is guaranteed to be encountered, the dominance frontier identifies locations where this guarantee ends. The dominance frontier of a block is particularly important because it identifies potential locations where definitions originating from different control-flow paths may need to be merged.

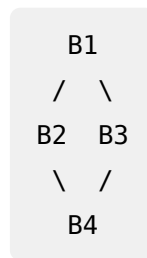
The dominance frontier of a block  $B$ , denoted by  $DF(B)$ , consists of all blocks  $Y$  such that  $B$  dominates at least one predecessor of  $Y$ , but  $B$  does not strictly dominate  $Y$ .

Formally:

$$DF(B) = \{Y \mid \exists P \in \text{pred}(Y), B \text{ dominates } P \wedge B \text{ does not strictly dominate } Y\}$$

Intuitively, the dominance frontier represents the points at which values defined in one region of the control-flow graph can meet values originating from another region.

Consider again the following CFG:



Neither  $B_2$  nor  $B_3$  dominates  $B_4$ , because  $B_4$  can be reached through either branch. However,  $B_2$  dominates one predecessor of  $B_4$ , and  $B_3$  dominates another predecessor. Therefore:

$$DF(B_2) = \{B_4\}$$

and:

$$DF(B3) = \{B4\}$$

This identifies `B4` as the location where definitions produced in `B2` and `B3` may need to be merged [21].

Dominance frontiers provide a systematic method for determining where  $\varphi$ -functions should be inserted during SSA construction. If a variable is defined in multiple blocks,  $\varphi$ -functions are inserted at the iterated dominance frontiers of those definition blocks.

For example, suppose the variable `x` is defined in both `B2` and `B3`:

```
B2:
    x_2 = 10

B3:
    x_3 = 20
```

Since the dominance frontiers of both `B2` and `B3` contain `B4`, a  $\varphi$ -function is required in `B4`:

```
B4:
    x_4 = phi(x_2, x_3)
```

The use of dominance frontiers ensures that  $\varphi$ -functions are inserted only at control-flow locations where independently generated definitions may meet. This avoids unnecessary insertion of  $\varphi$ -functions throughout the control-flow graph and enables efficient SSA construction [21].

In practice, SSA construction consists of three main steps. First, dominator relationships and the dominator tree are computed for the control-flow graph. Second, dominance frontiers are used to determine the appropriate locations for  $\varphi$ -functions. Finally, the dominator tree is traversed to rename variables and assign unique SSA versions to every definition.

In binary lifting, SSA provides a convenient representation for tracking the values of registers and temporary program state across the reconstructed control-flow graph. A RISC-V register may be assigned multiple times during program execution, but each

assignment can be represented by a distinct SSA version. At control-flow merge points,  $\varphi$ -functions can be used to represent the different possible result values. This makes data dependencies between instructions explicit and provides a suitable representation for subsequent translation into LLVM IR.

### 2.4.2. LLVM IR

LLVM IR is the intermediate representation used by the LLVM compiler framework. It uses SSA, where each variable is assigned exactly once and subsequent values are represented by new assignments. This design simplifies data-flow analysis and enables various compiler optimizations. In addition, LLVM IR provides explicit type information and low-level operations while remaining independent of both the original source language and the target architecture.

LLVM IR is available in three equivalent forms: as an in-memory representation used internally by the compiler framework, as a bitcode representation stored on disk, and as a human-readable assembly-like representation. This flexibility allows LLVM IR to be inspected, transformed, and debugged throughout the compilation pipeline.

The design goal of LLVM IR is to provide a representation that is sufficiently abstract to represent programs from different programming languages while remaining close enough to machine-level operations to support efficient optimization and code generation. Its typed instruction set, SSA-based structure, and architecture-independent design provide a foundation for analyses and transformations performed by the LLVM framework.

The architectural independence of LLVM IR makes it well suited as a target representation for binary lifting. Machine instructions from architectures such as RISC-V can be translated into LLVM IR while preserving their semantics. The resulting representation can then utilize LLVM's existing analysis and optimization infrastructure, enabling further processing such as static analysis, optimization, or later recompilation into target-specific machine code.

An LLVM program is organized hierarchically into modules, functions, basic blocks, and instructions. A module represents the top-level program structure and contains, among other elements, global variables and functions. Functions consist of basic blocks, each

of which contains a sequence of instructions. The final instruction of a basic block is a terminator instruction that defines the subsequent control flow [22].

## 2.5. DEL (Dynamic Execution Lifter)

DLR’s DEL is a binary-lifting tool designed to translate ARMv7 binaries into an enhanced LLVM IR, with particular emphasis on resolving indirect control-flow transfers. DEL combines static analysis with Dynamic Symbolic Execution (DSE) to improve the recovery of the program’s CFG. The approach was developed to address a fundamental limitation of purely static binary lifting: Indirect jumps may depend on runtime-computed values and therefore cannot always be resolved from the binary alone.

DEL consists of a static and a dynamic component as shown in Figure 4. The static component performs disassembly, CFG reconstruction, and translation of the recovered instructions into LLVM IR. This stage provides broad coverage of the executable code but may leave indirect control-flow targets unresolved. The dynamic component subsequently uses DSE to analyze these unresolved control-flow transfers and identify possible target addresses. The two components therefore complement each other: static analysis provides code coverage, while dynamic symbolic execution provides additional information required for resolving runtime-dependent control flow.

To perform the dynamic analysis, DEL translates LLVM IR instructions into expressions for a Satisfiability Modulo Theories (SMT) solver. The symbolic execution is supported by models of the processor’s registers and memory, allowing the execution state of the binary to be represented symbolically. DEL also includes a specialized mechanism for handling processor condition flags, which are relevant to the semantics of ARMv7 instructions and conditional execution.

The dynamic component uses the symbolic execution state to derive constraints on the targets of indirect control-flow transfers. These constraints are passed to an SMT solver, which determines whether potential target addresses are satisfiable. The resulting information can then be used to update the CFG and generate enhanced LLVM IR representations containing the recovered indirect control-flow edges.

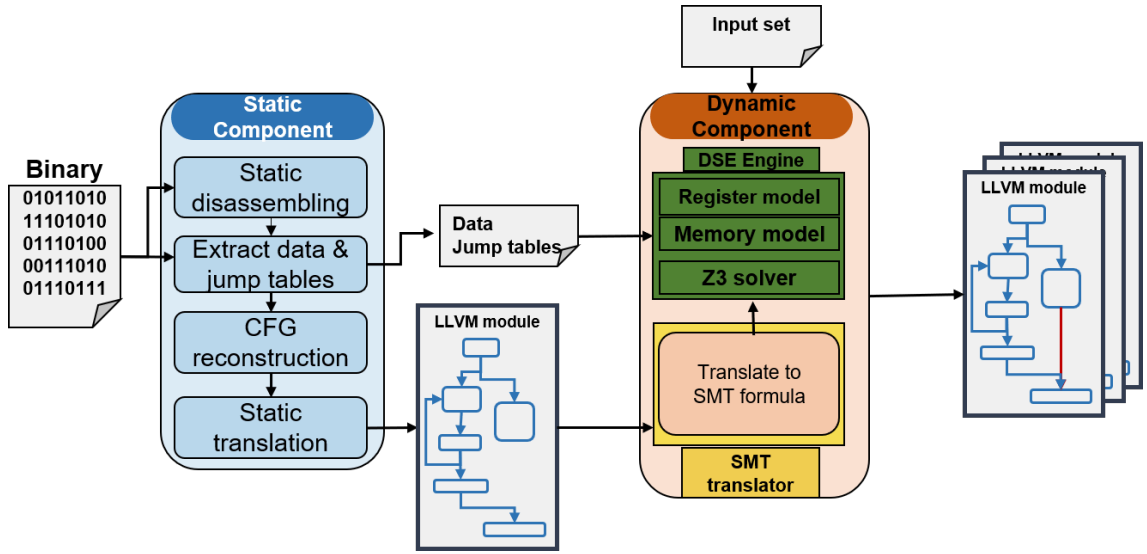


Figure 4: Structure of DEL [1]

An important characteristic of DEL is therefore the separation between code recovery and control-flow resolution. The static component can recover and translate code without requiring concrete execution inputs, while the dynamic component provides additional information about control-flow transfers whose targets cannot be determined statically. This combination is intended to avoid the limited code coverage associated with purely dynamic lifting while addressing the difficulty of resolving indirect jumps through purely static techniques [1].

## 2.6. Related Work

This section reviews existing approaches and tools related to binary lifting and reverse engineering and compares them to the presented work. The presented work includes traditional rule-based lifters, learning-based approaches, and general-purpose reverse engineering frameworks. Their concepts and design choices provide the context for the implementation presented in this thesis.

### 2.6.1. McSema

McSema<sup>1</sup> is an open-source binary lifter developed by Trail of Bits that translates native binaries into LLVM bitcode [23]. It was originally designed primarily for x86 and x86-64 binaries and separates control-flow recovery from instruction translation. This separation allows the control-flow structure of a binary to be established before individual instructions are translated into LLVM IR. McSema therefore provides a useful example of a modular lifting architecture in which binary analysis and instruction translation are treated as distinct stages [23].

A key aspect of McSema is that it operates on already compiled binaries and attempts to reconstruct sufficient program structure to generate LLVM IR. This makes it particularly relevant to the present work, where the original source code is similarly assumed to be unavailable. However, McSema primarily relies on static techniques for recovering the program structure and therefore faces the general difficulties associated with statically resolving indirect control-flow transfers.

### 2.6.2. RevGen

RevGen is a static binary translator that converts x86-binaries into equivalent LLVM modules [24]. Its primary objective is to make existing binary software accessible to LLVM-based analysis, instrumentation, and execution frameworks without requiring the original source code. By translating machine code into LLVM IR, RevGen enables the reuse of existing analysis tools that would otherwise require programs to be available in an LLVM-compatible representation.

An important challenge addressed by RevGen is the preservation of instruction semantics during translation. The approach uses an intermediate representation consisting of micro-operations to decompose complex machine instructions before translating them to LLVM IR. In addition, the framework can use emulation helpers for instructions whose behavior cannot be represented directly by the available intermediate operations. This enables the lifting of instructions that manipulate processor or system state in addition to ordinary arithmetic and memory operations [24].

---

<sup>1</sup><https://github.com/lifting-bits/mcsema>

RevGen is particularly relevant to the approach presented in this thesis because it also uses LLVM IR as the target representation and require the reconstruction of executable code and control-flow information before producing an LLVM representation. However, the present work targets RISC-V binaries rather than x86 binaries and focuses on generating an intermediate representation suitable for subsequent static analysis. In particular, the reconstruction of the control-flow graph and the handling of indirect control-flow transfers are central aspects of the proposed lifting approach.

### 2.6.3. Rellume

Rellume<sup>2</sup> is an LLVM-based binary lifting framework developed for dynamic binary translation. Its lifting pipeline consists of instruction decoding, control-flow graph reconstruction, and the translation of machine instructions into LLVM IR. To simplify the implementation of individual instruction semantics, Rellume provides a common infrastructure for register and memory accesses, allowing the instruction lifter to focus on mapping instruction semantics to LLVM IR. The generated IR is designed to integrate with LLVM’s optimization framework, enabling efficient execution of translated code [25]

This thesis follows a similar lifting pipeline by decoding instructions, reconstructing the control-flow graph, and translating instructions into LLVM IR. However, the primary objective differs. While Rellume is designed for dynamic binary translation and runtime optimization, the implementation proposed in this thesis focuses on generating LLVM IR suitable for static program analysis. Consequently, the reconstructed control-flow graph and intermediate representation are intended to support analyses such as control-flow and data-flow analysis, while also providing a foundation for future dynamic lifting.

### 2.6.4. Forklift Neural Lifter

Armengol-Estapé et al. introduced Forklift<sup>3</sup> [26], a neural binary lifting framework that translates assembly code into LLVM IR using a Transformer-based encoder-decoder architecture. In contrast to traditional lifters such as Lasagne [27], which rely on manu-

---

<sup>2</sup><https://github.com/aengelke/rellume>

<sup>3</sup><https://github.com/jordiae/forklift>

ally crafted translation rules, Forklift learns the mapping between assembly instructions and LLVM IR from large-scale parallel datasets. The framework supports multiple ISAs, including x86, ARM, and RISC-V, and employs an incremental learning strategy in which a shared LLVM IR decoder is reused while architecture-specific encoders are fine-tuned for new ISAs. Experimental results demonstrate that Forklift significantly outperforms both the rule-based Lasagne lifter and GPT-4 when lifting optimized assembly code. The work shows that neural approaches can reduce the engineering effort required to support additional architectures while maintaining competitive lifting accuracy [26] .

In contrast to Forklift, this work relies on rule-based instruction decoding and lifting rather than a learned translation model. This provides deterministic translations that preserve instruction semantics and are well suited for static program analysis, while avoiding the need for large training datasets.

### 2.6.5. Ghidra

Ghidra<sup>4</sup> is an open-source software reverse engineering framework developed by the National Security Agency (NSA) [28]. It provides a comprehensive environment for analyzing compiled binaries and supports numerous processor architectures, including RISC-V. Its functionality includes disassembly, decompilation, control-flow graph generation, symbol recovery, and scripting capabilities, making it a widely used platform for binary analysis and reverse engineering [28] .

Unlike the implementation presented in this thesis, Ghidra is a general-purpose reverse engineering framework targeting a broad range of architectures and analysis tasks. This work, in contrast, focuses specifically on implementing a custom RISC-V binary lifter, including instruction decoding, control-flow graph reconstruction and translation into an intermediate representation. Consequently, Ghidra serves primarily as a reference system and a point of comparison rather than as the basis for the proposed implementation.

---

<sup>4</sup><https://github.com/NationalSecurityAgency/ghidra>

### 3. Methodology

The following section introduces the tools and technologies selected for the project and explains the reasoning behind these choices. Furthermore, it discusses the design decisions made during the implementation and their relevance to the overall approach.

#### 3.1. Input Binaries

To maintain control over the binaries used for evaluation, all binaries in this work were compiled specifically for this purpose. The compiler and assembler were configured to avoid the use of compressed instructions and pseudo-instructions. This is done to focus only on the RV64I instructions. Consequently, the resulting binaries contain only instructions from the RV64I base instruction set within the scope of the evaluation. This controlled setup allows the custom decoder to focus on the RV64I instruction set without requiring support for additional RISC-V extensions.

#### 3.2. Intended Workflow

The proposed lifting process transforms an input ELF binary into LLVM IR through a sequence of five stages: reading the binary, decoding its instructions, constructing the CFG, implement SSA representation, and translating the resulting representation to LLVM IR as shown in Figure 5.

Each stage operates on the representation produced by the preceding stage. First, executable instructions are extracted from the input binary and decoded into an internal representation. The decoded instructions are then grouped into basic blocks and connected according to their possible execution flow. Based on the resulting CFG, SSA form is constructed to provide an explicit representation of variable definitions and uses. Finally, the SSA-based representation is translated into LLVM IR.

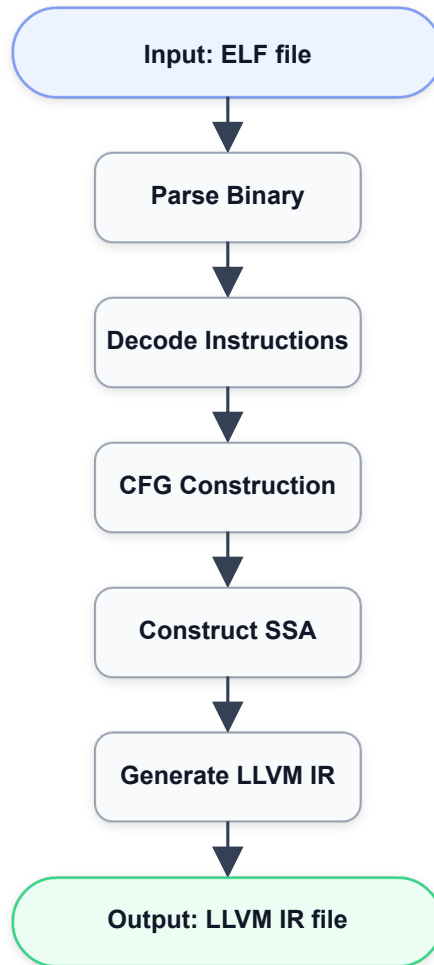


Figure 5: intended workflow

### 3.3. Binary Parsing

The first step of the lifting process is to read the input binary and extract the executable code. The implementation expects the input to be provided as an ELF binary. The Python library `pyelftools` is used to access the structure and contents of the ELF file. It provides functionality for parsing ELF files and accessing their headers, sections, segments, and associated metadata [29].

After parsing the ELF file, the section containing the executable code is identified. The contents of this section are extracted as a sequence of bytes. For debugging and subsequent instruction decoding, these bytes can be represented in hexadecimal notation. The

resulting byte sequence serves as the input to the RISC-V instruction decoder in the subsequent lifting stage.

The extraction process can therefore be summarized as follows:

1. Read the input ELF binary.
2. Parse the ELF structure using `pyelftools`.
3. Identify the relevant executable section.
4. Extract the section contents as raw bytes.
5. Pass the extracted bytes to the instruction-decoding component.

### 3.4. Instruction Decoding

Following the parsing, the machine-code instructions are decoded.

A disassembler performs this decoding by interpreting the machine-code bytes according to the instruction encodings defined by the target architecture. For RISC-V binaries, the resulting disassembly can provide information such as the instruction address, mnemonic, operands, and instruction size. This information is subsequently used by the binary-lifting process, for example during control-flow reconstruction and instruction translation.

Several tools can be used to disassemble RISC-V binaries. In this work, `riscv-none-elf-objdump`, `llvm-objdump`, and Capstone were considered and evaluated for their suitability. The selection of the disassembly tool is discussed in the following sections.

#### 3.4.1. `riscv-none-elf-objdump`

`riscv-none-elf-objdump` is provided as part of the GNU RISC-V toolchain and is based on the GNU Binutils `objdump` utility. GNU `objdump` provides functionality for examining object files and disassembling machine code. For RISC-V binaries, it can be used to display the decoded instructions together with information such as their addresses and instruction bytes [30].

### 3.4.2. LLVM *objdump*

`llvm-objdump` provides functionality similar to `riscv-none-elf-objdump`, but is part of the LLVM project. It can display information from object files and final linked images and can disassemble executable sections. In addition, `llvm-objdump` provides RISC-V-specific disassembly options. For example, the `no-aliases` option disables instruction aliases and displays the underlying instruction mnemonic, while the `numeric` option displays numerical register names instead of their ABI names [31].

### 3.4.3. Capstone

In contrast to the other presented tools, Capstone is a disassembly framework that provides an architecture-neutral Application Programming Interface (API). It supports multiple instruction set architectures, including RISC-V, and can be integrated directly into applications for programmatic disassembly.

For RISC-V binaries, Capstone can decode machine-code instructions and provide information such as the instruction address, mnemonic, operands, and instruction size. In addition, the decoded instruction information can be accessed directly through the API instead of parsing the textual output of an external disassembly tool. This makes Capstone particularly suitable for integration into the binary-lifting process [32].

### 3.4.4. Comparison and Decision

| Criterion                               | GNU<br><code>riscv-none-elf-objdump</code> | LLVM<br><code>llvm-objdump</code>       | Capstone                           |
|-----------------------------------------|--------------------------------------------|-----------------------------------------|------------------------------------|
| Type                                    | Command-line tool                          | Command-line tool                       | Disassembly library                |
| Integration                             | External process                           | External process or LLVM infrastructure | Programmatic API                   |
| Input                                   | Binary/object files                        | Binary/object files                     | Machine-code bytes                 |
| Output                                  | Textual disassembly                        | Textual disassembly                     | Structured instruction information |
| RISC-V support                          | Yes                                        | Yes                                     | Yes                                |
| External dependency                     | GNU RISC-V tool-chain                      | LLVM                                    | Capstone library                   |
| Control over decoding                   | Limited                                    | Limited                                 | Limited to API functionality       |
| Suitability for custom lifting pipeline | Moderate                                   | Moderate                                | High                               |

Table 2: Comparison of Binary disassembly tools

All three considered options presented in Table 2 provide established solutions for disassembling RISC-V machine code. However, using an existing disassembly solution would introduce a dependency on an additional software component. In the case of `riscv-none-elf-objdump` and `llvm-objdump`, integration would primarily rely on command-line interfaces and textual disassembly output. Capstone provides a more suitable programmatic interface, but would still introduce an additional third-party dependency and delegate the instruction decoding process to an external framework.

To retain full control over the decoding process, a custom RISC-V instruction decoder was implemented. Therefore, it does not rely on the textual output of for example `llvm-objdump`. For example Capstone uses pseudo-instructions even if the compiler is

configured not to use them. The decoder directly interprets machine-code bytes according to the RISC-V instruction encodings and extracts the instruction fields required by the subsequent stages of the lifting process.

Implementing a custom decoder increases the implementation effort and limits immediate extensibility to other instruction set architectures. However, the scope of this work is restricted to the RISC-V RV64I base instruction set. Therefore, implementing only the required subset of the instruction decoder is feasible. The increased control over the decoding process and the absence of additional external dependencies make a custom decoder the most suitable approach for this work.

### 3.5. Choice of SSA

For the construction of SSA form, this work follows the classical approach proposed by Cytron et al. [21]. This approach constructs SSA form based on the control-flow graph by first computing dominance information and dominance frontiers. The dominance frontiers are then used to determine the locations at which  $\varphi$ -functions are required. Finally, definitions are renamed according to the dominance relationships in the control-flow graph.

The Cytron et al. approach was selected because it provides a well-established and systematic method for constructing SSA form from a control-flow graph. It is particularly suitable for this work because the CFG is available before SSA construction, providing the required input for the algorithm. The implementation therefore follows the classical SSA construction algorithm rather than introducing a custom SSA construction method.

### 3.6. Instruction Translation

At the beginning a few instructions of the RV64I base instruction set were chosen to be implemented first. To determine which instructions to start with a set of representative programs was used to identify instructions occurring frequently in the generated binaries. The most often occurring instructions were then used to start the prototype implementation.

Four representative programs from a C lecture were compiled into RISC-V ELF binaries using `riscv-none-elf-gcc`. The compiler was configured to target the RV64I instruction set using the LP64 ABI. The binaries were generated using the following command:

```
riscv-none-elf-gcc -march=rv64i -mabi=lp64 -o <output>.elf <source>.c
```

No optimization level was explicitly specified. The executable code was extracted and the instructions were decoded according to the RV64I instruction encoding. Each decoded instruction was counted, and the instruction frequencies were aggregated across the four binaries.

The resulting frequencies were used to determine the initial set of instructions supported by the translator. Figure 6 shows the five instructions with the highest occurrence counts in the analyzed binaries. The *addi* instruction occurred most frequently, followed by the remaining four instructions *ld*, *sd*, *jal* and *beq* as shown in the figure. These five instructions were consequently selected as the initial instruction set implemented by the translator.

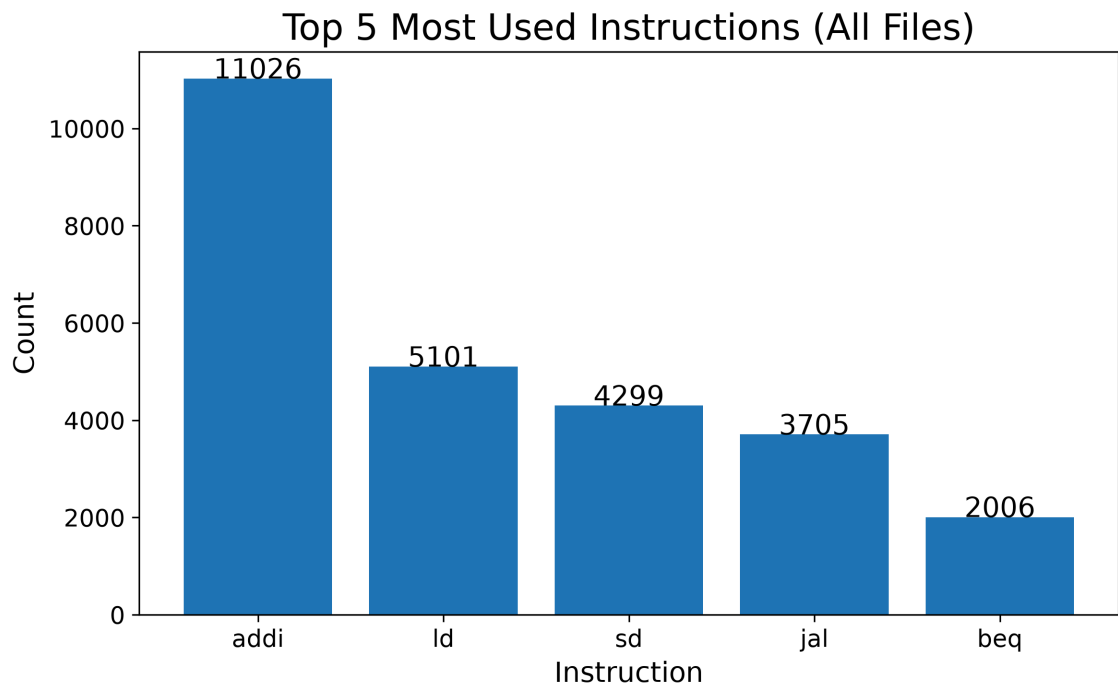


Figure 6: Five most frequently occurring instructions in the analyzed binaries

### 3.7. Memory Allocation

The memory model of the source architecture differs from the memory model used in LLVM IR. RISC-V provides a byte-addressable memory space that is accessed explicitly through load and store instructions, while registers are represented separately as architectural state. LLVM IR provides typed memory operations such as load and store but does not force a specific implementation of memory.

To represent the architectural state of the RISC-V program within LLVM IR, this implementation models memory as a contiguous byte array. The byte array represents the addressable memory of the lifted program, while the RISC-V registers are represented separately as LLVM values. Load and store instructions are translated into accesses to the corresponding locations in this byte array as shown in Figure 7.

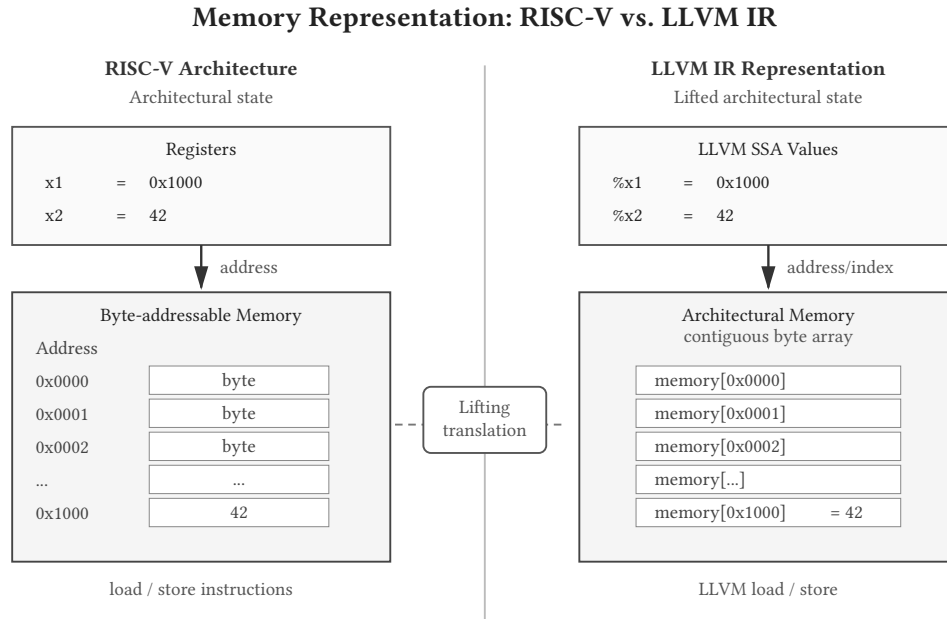


Figure 7: Memory Representation: RISC-V vs. LLVM IR

This representation provides a simple and uniform model for memory accesses while preserving the byte-addressable nature of RISC-V memory. It also allows the lifter to represent memory operations independently of LLVM’s native allocation mechanisms, such as stack allocations and global objects, rather than attempting to reconstruct the original binary’s memory organization using those mechanisms.

## 4. Implementation

The implementation follows a modular approach to facilitate further extension to other ISAs. The overall workflow, shown in Figure 8, consists of five main stages. First, the input ELF file is parsed and its instructions are decoded into an internal representation. Then, the decoded instructions are used to construct basic blocks and the corresponding CFG. Fourth, the SSA form is constructed by computing dominators, inserting  $\varphi$ -nodes, and renaming variables. Finally, the resulting representation is translated into LLVM IR.

The following sections describe the implementation of these stages in detail, including the design decisions and implementation details of each component.

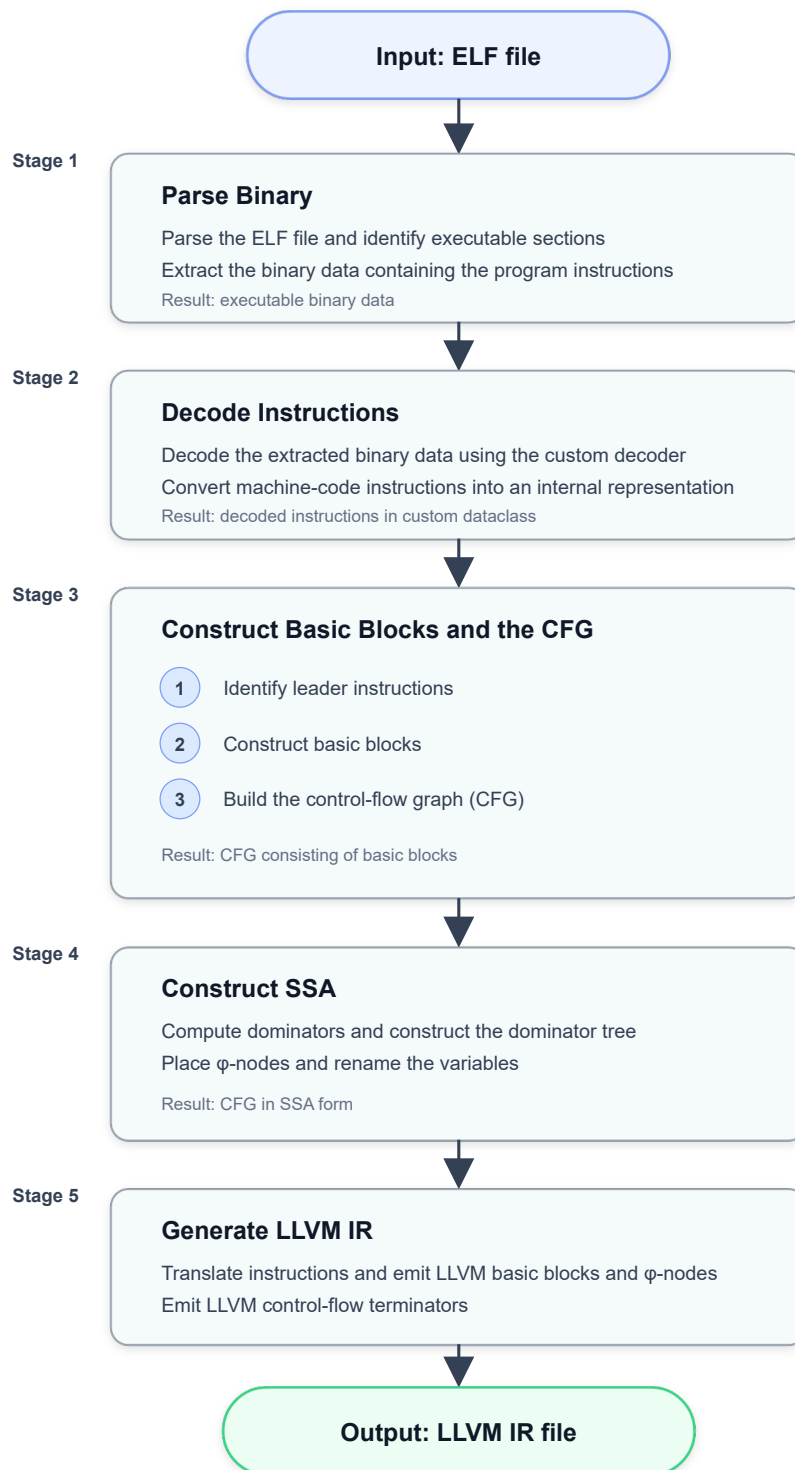


Figure 8: Workflow of the presented implementation

## 4.1. Decoding the Binary File

The binary is processed using the `pyelftools` framework which enables the extraction of relevant sections from the executable. Among these, the `.text` section is the most significant as it contains the program instructions. Additionally, the `.symtab` section stores the symbol table, including function definitions which are required throughout the lifting process.

Binary files are commonly displayed in hexadecimal notation. During decoding, however, instructions are treated as 32-bit integer values, allowing individual fields to be extracted using bitwise operations.

As discussed previously, instructions can be decoded using a disassembler. In this work, however, a custom decoder is employed instead. Each RISC-V instruction format required by the supported instruction set is represented by a dedicated data class. This includes the standard R-, I-, S-, B-, U-, and J-type formats, as well as specialized representations for CSR and JALR instructions. The decoder interprets each 32-bit instruction word by extracting the relevant bit fields and mapping them to the attributes of the corresponding instruction representation. Listing 1 shows the implementation of the decoder for I-type instructions. The immediate field is extracted from the instruction word and sign-extended before the remaining fields are decoded and stored in the corresponding `IType` object.

```
@dataclass
class IType:
    opcode: int
    rd: str
    rs1: str
    imm: int
    funct3: int

def decode_i(insn) -> IType:
    imm = (insn >> 20) & 0xFFF
    if imm & 0x800:
        imm -= 0x1000

    return IType(
        opcode=insn & 0x7F,
        rd=reg_name((insn >> 7) & 0x1F),
        funct3=(insn >> 12) & 0x7,
        rs1=reg_name((insn >> 15) & 0x1F),
        imm=imm,
    )
```

Listing 1: Implementation of the decoder for RISC-V I-type instructions.

The decoder assigns each instruction to a semantic category that corresponding to the underlying RISC-V encoding. These categories are introduced to simplify the handling of instructions during later lifting stages.

- Arithmetic instructions include register-register and immediate-based computations, such as additions, subtractions, logical operations, shifts, and comparisons. These instructions primarily modify register values and can therefore be translated independently of memory or control-flow state.
- Load and store instructions represent memory accesses. Load instructions transfer data from memory into a register, whereas store instructions write register values to memory. Since memory operations affect the program state outside of the register file, they require separate handling during lifting.

- Branch instructions represent conditional control-flow changes. They evaluate a condition between registers and either continue execution at the next instruction or transfer control to a target address. Therefore, they are vital when reconstructing the control-flow graph of the binary.
- Jump instructions perform unconditional control-flow transfers. In addition to regular jumps, the decoder distinguishes function calls and returns. A JAL instruction writing the return address register ( `ra` ) is classified as a call, while a JALR instruction with the pattern `rd = zero`, `rs1 = ra`, and an immediate value of zero is classified as a return. This additional classification supports later lifting stages to identify function boundaries and reconstruct higher-level program structure.
- System instructions represent interactions with the execution environment, including environment calls, breakpoints, and CSR accesses. These instructions require special handling because their behavior depends on the processor state and system configuration.
- Finally, instruction words with unsupported opcodes are assigned to the UNKNOWN category. Invalid or unsupported encodings within otherwise recognized instruction classes may instead raise an exception, preventing them from being processed as valid instructions.

## 4.2. CFG Reconstruction

Using the decoded instructions and the recovered functions from the `.symtab` section of the binary, the next step is to reconstruct the CFG. The reconstruction is separated into three stages:

1. identifying all leader instructions,
2. constructing the basic blocks,
3. determining the control-flow edges between the blocks.

#### **4.2.1. Identify Leader Instructions**

A leader is the first instruction of a basic block and is used as the starting point for constructing the control-flow graph. The first instruction of every function is always considered a leader.

Additional leaders are identified based on the semantic categories assigned during decoding. For branch instructions, both the branch target and the fallthrough instruction are considered leaders, as they represent the possible entry points of the two execution paths. For direct jumps with a statically known target inside the current function, the target address is added as a leader. Indirect jumps, whose targets cannot be determined statically during decoding, do not introduce additional leaders. Instead, their potential targets remain unknown until further analysis or runtime information is available.

Function calls normally introduce a new leader at the instruction following the call, representing the point at which execution resumes after the callee returns. Calls to known non-returning functions, such as `exit`, `abort`, and `_exit`, do not introduce a fallthrough as a leader. This represents the point at which execution continues after returning from the called function. System instructions are treated similarly by adding their fallthrough instruction as a leader, while return instructions do not introduce additional leaders because they terminate the current control-flow path.

Only targets within the boundaries of the current function are considered. This prevents instructions belonging to other functions from being incorrectly included when constructing the basic blocks of a single function.

### 4.2.2. Building Basic Blocks

After finding all leader instructions the next step is to build the basic blocks. All necessary information are stored as shown in Listing 2.

```
@dataclass
class BasicBlock:

    start: int
    end: int | None = None

    instructions: list[Instruction] = field(default_factory=list)

    succ: list[int] = field(default_factory=list)
    pred: list[int] = field(default_factory=list)

    calls: list[int] = field(default_factory=list)
    external_jumps: list[int] = field(default_factory=list)
    terminator: Instruction | None = None
```

Listing 2: Basic Block definition

While iterating over all instructions, for each leader, instructions are appended sequentially until the next leader is reached, an instruction is unavailable in the decoded instruction map, the end of the function or an instruction classified as terminating the block is encountered. Instructions that may alter the normal sequential control flow, including branches, jumps, calls, returns, and system instructions, terminate the current basic block.

In addition to storing the instructions belonging to the block, the BasicBlock representation contains information required by later lifting stages. The successor and predecessor lists are used during control-flow graph construction, while call and external jump preserve information required for later interfunctional analysis.

The separation between leader identification and basic block construction simplifies the control-flow analysis. The first one determines all possible entry points, while the second

one groups the instructions between these entry points into executable regions that can be analyzed independently.

### 4.2.3. Building the CFG

After constructing the basic blocks, the next step is to determine the control-flow relationships between them. These relationships form the CFG, where each basic block represents a node and each transfer of execution represents an edge.

The successors of each basic block are determined by examining the semantic category of its terminating instruction. For conditional branch instructions, two successor edges are possible: one to the branch target and one to the fallthrough block representing the instruction executed when the branch condition is not satisfied. This can be seen in Figure 9 as there are two blocks after the *bne* instruction. For unconditional jumps, only the jump target is added as a successor. If the target is outside the current function, it is stored separately as an external jump.

Function calls are handled differently from regular jumps. The called function is recorded separately while the instruction following the call remains the successor of the current block. This preserves the intra-procedural control flow of the current block while still maintaining information about interprocedural calls.

Return instructions terminate the control flow of the current function and therefore do not introduce successor edges. For example the *jalr* instruction as seen in Figure 9 terminates the control flow. System instructions also terminate the current basic block. During CFG construction, however, no successor edge is currently added for them, even though their fallthrough address may have been identified as a leader during block construction. This is done, since System instructions are not implemented yet and need a different translation logic.

For instructions that do not modify control flow, the successor is the basic block beginning at the next sequential instruction address. After all successor relationships have been determined, the predecessor lists are constructed by iterating over the successor edges and adding the originating block to the corresponding predecessor list.



Figure 9: reconstructed CFG

After constructing the CFG, unreachable basic blocks can be removed by performing a graph traversal beginning at the entry block. Only blocks reachable through successor edges are retained. The predecessor relationships are then rebuilt to ensure consistency with the reduced CFG.

The resulting CFG provides the structure required for later analysis stages, such as data-flow analysis, SSA construction, and translation into the intermediate representation.

### 4.3. SSA Implementation

To implement SSA, several dataclasses were created. `SSAVariable` which only contains the name and the version of a variable. The `Phi` dataclass encompasses the inputs of a  $\varphi$  node, while `SSABlock` is a wrapper around the `BasicBlock` class and adds additional SSA-related information to it.

The implementation has seven steps:

1. compute Dominators
2. compute immediate Dominators
3. build the dominator tree
4. compute the dominance frontiers
5. Find definition Blocks
6. Place  $\varphi$  nodes
7. rename existing variables

In the implementation, finding definition blocks is performed internally as part of the Phi-placement stage.

### **1. Compute Dominators**

First, the entry block needs to be identified. It is assumed that the entry block is always the first block in the CFG. It also collects the addresses of all blocks.

For the initial dominator sets, the algorithm assumes that every block dominates every block. This is an intentionally overly broad initial assumption that will be refined later. The entry block is an exception since it can only be dominated by itself.

After this initialization, the algorithm repeatedly refines the dominator sets until they no longer change. For each non-entry block, its predecessors are examined. If a block has no predecessors, the algorithm assumes that it is only dominated by itself. This can occur for unreachable blocks.

The CFG already contains the predecessors of each block, so the algorithm gathers the current dominator sets of all predecessors. The dominators common to all predecessor paths are obtained by intersecting the dominator sets of the block's predecessors. The block itself is then added to the resulting set. Finally, the current block itself is added to its dominator set.

This process is repeated until there are no more changes to the dominator set. At this point, the algorithm has reached a fixed point and returns the computed dominators for each block.

## 2. Compute Immediate Dominators

An immediate dominator is the closest block that dominates another block. In other words, it is the last dominator that must be passed before reaching the current block.

Since all dominators of each block were computed beforehand, these dominator sets can be used to determine the immediate dominator. The algorithm examines all possible dominators of the current block. A candidate cannot be the immediate dominator if another candidate is strictly dominated by it. In this case, the other candidate is closer to the current block in the dominator hierarchy.

The remaining candidate is therefore the closest strict dominator and is assigned as the immediate dominator of the current block.

## 3. Build Dominator Tree

To build the dominator tree, the previously computed immediate dominators are used. First, an empty list of children is created for every block. This provides a structure in which every block initially has no children. The immediate-dominator relationships are then used to establish the parent-child relationships of the dominator tree: If a block has an immediate dominator, this dominator becomes the parent of the block in the dominator tree, and the block is added to the parent's list of children. If there is no immediate dominator, as is the case for the entry block, the block simply does not get a parent.

## 4. Compute Dominance Frontier

The dominance frontier is computed using the previously calculated immediate dominators. For every block, an empty set is created. This set will contain the blocks that belong to that block's dominance frontier. The algorithm then iterates over all blocks. A block can have a local dominance frontier when it has multiple predecessors, meaning that it is a join point in the CFG. Therefore, blocks with fewer than two predecessors are skipped.

For every predecessor of a join block, the algorithm starts at that predecessor and follows the immediate-dominator chain upwards. This continues until the runner reaches the immediate dominator of the join block. During this process, the join block is added to the dominance frontier of each block encountered. The runner is then moved to its

own immediate dominator. In this way, the algorithm identifies blocks where different control-flow paths merge. These locations are relevant for determining where  $\varphi$ -functions may be required during SSA construction.

The function finally returns a dictionary containing the dominance frontier of every block.

## 5. Find Definition Blocks

Definition blocks are blocks that contain the definition of a register. The algorithm iterates over every block and every instruction within that block. For each instruction, the algorithm checks whether it contains any register definitions. The “zero” register is ignored because it always contains the value 0 and cannot be redefined. For every other register, the block’s address is added to that register’s set of definition blocks. This results in a mapping where each register is associated with all blocks in which it is assigned a value.

## 6. Place $\varphi$ Nodes

First, it needs to be determined where  $\varphi$ -functions need to be inserted for each register when constructing SSA form. For this purpose the definition blocks and the dominance frontier are used. The algorithm then processes each register separately. For every register, a worklist is initialized with all blocks in which that register is defined. To prevent the same  $\varphi$ -function from being inserted multiple times, the algorithm keeps track of the blocks where a  $\varphi$ -function for the current register has already been inserted.

While the worklist is not empty, a definition block is removed and its dominance frontier is examined. For every block in the dominance frontier, the algorithm checks whether a  $\varphi$ -function for the register has already been placed there. If not, a new  $\varphi$ -function is inserted.

Note that the newly inserted  $\varphi$ -function is treated as a new definition of the register. If the block containing the newly inserted  $\varphi$ -function was not one of the original definition blocks, it is added to the worklist so that its dominance frontier can also be processed. This implements the iterated dominance frontier. A newly inserted  $\varphi$ -function can create another definition that requires additional  $\varphi$ -functions further downstream. The algorithm therefore continues until no new  $\varphi$ -functions need to be placed.

## 7. SSA Renaming

After the  $\varphi$ -functions have been placed, the registers are renamed according to the SSA form. The goal of this step is to ensure that every definition of a register receives a unique version and that every use of a register refers to the correct definition.

To keep track of the currently visible version of each register, the implementation uses a stack for every register except the “zero” register. Each stack initially contains version 0 of the corresponding register. Whenever a new definition of a register is encountered, a new SSA version is created and added to the corresponding stack. The top element of a stack therefore always represents the currently visible version of that register. The “zero” register is treated separately because it always represents the constant value 0 and cannot be redefined.

The renaming process is performed by traversing the dominator tree. This traversal ensures that definitions made in a dominating block remain intact while all blocks dominated by it are processed.

At the beginning of each block, the  $\varphi$ -functions are processed first. Since a  $\varphi$ -function represents a new definition of a value, it receives a new SSA version, which is added to the corresponding register stack.

Then the instructions of the block are processed. For every register use, the currently visible SSA version is obtained from the top of the corresponding stack. Whenever an instruction defines a register, a new SSA version is created and added to the stack. This ensures that every definition has a unique name while each use refers to the most recent visible definition.

After all instructions in a block have been processed, the operands of  $\varphi$ -functions in successor blocks are determined. For each outgoing control-flow edge, the current version of a register is used as the corresponding input of the  $\varphi$ -function. Therefore, each  $\varphi$ -function receives the SSA values that are visible along its incoming control-flow paths.

The SSA values visible at the exit of each block are stored explicitly in the block’s `exit_values` mapping. Afterwards, the blocks dominated by the current block are

processed recursively. Because the definitions of the current block remain on the stacks during this traversal, they are visible to all dominated blocks.

Once all dominated blocks have been processed, the SSA versions defined in the current block are removed from their respective stacks. This restores the state that was present before entering the block and ensures that definitions from one branch do not incorrectly remain visible when processing another branch.

This approach ensures that every register definition receives a unique SSA version and that every register use and  $\varphi$ -function operand refers to the correct definition according to the control-flow and dominance relationships.

## 4.4. LLVM Generation

After recovering the CFG and applying SSA the instructions can be emitted as LLVM IR. Therefore this work implements the following pipeline:

1. Emit a global memory array.
2. Generate an LLVM function for each recovered function.
3. Emit an LLVM label for each CFG basic block.
4. Emit Phi nodes at the beginning of each block.
5. Translate non-terminator instructions using SSA-renamed operands.
6. Emit LLVM control-flow terminators based on CFG

### 4.4.1. Memory Representation

Before emitting any instructions related to the actual program code, the memory needs to be represented in the generated LLVM IR. Memory is modelled as a byte-addressable global array. The implementation assumes a memory size of  $1024 \times 1024$  bytes. The memory is emitted as a global array in which each element is represented as an 8-bit integer (i8). The array is initialized using the values stored in the program's memory representation. Other approaches to modelling memory are discussed in Section 3.7..

#### 4.4.2. Function/Basic-Block and Instruction Emission

Before the actual program instructions are emitted, the recovered functions and their corresponding control-flow graphs are processed. For each function recovered from the `.symtab` section of the ELF file, an LLVM function is generated. The function parameters represent the initial values of the registers, excluding the “zero” register.

After the SSA construction, the LLVM functions are translated. Each basic block in the CFG receives a corresponding LLVM label. These labels represent the control flow of the original program and provide targets for translated branch and jump instructions.

Before the instructions of a basic block are emitted, its  $\varphi$ -functions are generated. Each  $\varphi$ -function combines the SSA values originating from the predecessor blocks and is therefore placed at the beginning of the corresponding LLVM basic block. The instructions of each basic block are then processed. The previously generated SSA representation provides the renamed operands and results for each instruction.

The recovered instructions are translated into LLVM IR according to their mnemonic. The translation functions operate on the SSA-renamed operands generated during the SSA construction and emit the corresponding LLVM instructions. While some instructions can be translated directly, others require multiple LLVM instructions to preserve the semantics of the original instruction.

For arithmetic instructions operating on 64-bit values, the translation can be performed directly. For example, the `addi` instruction adds an immediate value to the value of a source register and stores the result in the destination SSA variable. It can therefore be represented directly by an LLVM add instruction.

Instructions operating on 32-bit values require additional conversions because the registers in the generated LLVM representation are represented as 64-bit integers. For example, `addiw` and `addw` first truncate their operands to 32 bits, perform the addition using an LLVM `i32` value, and then sign-extend the result back to 64 bits. This preserves the 32-bit semantics of the original instructions.

Memory instructions require the address of the accessed memory location to be calculated first. The implementation uses the base register together with the immediate offset to obtain a pointer into the global byte-array representation of memory. Load

and store instructions perform typed accesses through the calculated pointer, using the corresponding LLVM integer type. For example, *lw* loads a 32-bit value and sign-extends it to 64 bits, whereas *sw* truncates its 64-bit input to 32 bits before storing it.

Control-flow instructions require a different translation approach. Conditional branch instructions such as *beq* and *bne* first compare their two operands using an LLVM *icmp* instruction. The resulting Boolean value is then used by the control-flow emission to select the appropriate successor block.

Function calls and jumps are handled separately. For a *jal* instruction, the target address is matched against the recovered functions. If the target corresponds to a known function, an LLVM call instruction is emitted. In the current implementation, the return value of a function is represented using the *a0* register. Jumps that use the “zero” register as their destination do not produce a return value, since the “zero” register cannot be modified.

| RISC-V instruction               | translated LLVM                                                                                                                                                                                                            |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>addi sp,sp, -32</code>     | <code>%sp_1 = add i64 %sp_0, -288</code>                                                                                                                                                                                   |
| <code>ld ra, 24(sp)</code>       | <code>%ld_10274_addr = add i64 %sp_1, 16</code><br><code>%ld_10274_ptr = getelementptr [1048576 x i8], ptr @memory, i64 0, i64 %ld_10274_addr</code><br><code>%s0_2 = load volatile i64, ptr %ld_10274_ptr, align 8</code> |
| <code>bne a4, a5, 0x10260</code> | <code>%cmp_10250 = icmp ne i64 %a4_1, %a5_3</code><br><code>br i1 %cmp_10250, label %BB_10260, label %BB_10254</code>                                                                                                      |

Table 3: Example of RISC-V instructions translated into LLVM IR

Table 3 illustrates how individual RISC-V instructions are lifted into LLVM IR. The translation preserves the semantics of the original instructions while representing program state using SSA form.

The SSA representation is visible in the translated register values, where each new definition receives a unique version number. For example, registers such as *sp*, *a4*, and *a5* are represented by SSA variables with appended version numbers, ensuring that every SSA variable has exactly one definition.

This approach allows the instruction translator to preserve the semantics of the recovered instructions while expressing them using LLVM's instruction set and type system.

#### 4.4.3. CFG Terminator Emission

After the instructions of a basic block have been emitted, its control-flow behavior is translated into an LLVM terminator instruction. The type of terminator depends on the structure of the corresponding CFG block.

Depending on the type of terminator, the translation differs. For conditional branch instructions, the operands are first translated into an LLVM comparison. For example, *beq* is translated into an *icmp eq* instruction. The resulting Boolean value is then used as the condition of an LLVM conditional *br* instruction. The two targets of the branch correspond to the branch target and the fallthrough successor of the original instruction.

If a block has only one successor, an unconditional LLVM *br* instruction is emitted. The successor is represented by the LLVM label corresponding to the target basic block.

External jumps are handled separately. If a block contains a single external jump whose target corresponds to a recovered function, the target function is emitted as an LLVM call. The returned value is then returned from the current function. If the target cannot be resolved to a recovered function, the generated code uses LLVM's unreachable instruction.

Finally, return instructions are translated into LLVM *ret* instructions. In the current implementation, the return value is obtained from the SSA value of the *a0* register at the exit of the corresponding block.

This ensures that the control-flow structure of the recovered program is explicitly represented in LLVM IR and that every generated basic block is terminated with an appropriate LLVM control-flow instruction.

## 5. Evaluation

The generated LLVM IR was evaluated at several levels. First, the LLVM compiler infrastructure is used to verify whether the generated IR is syntactically valid and can be processed by LLVM. Second, the generated IR was compiled back into an executable binary. Finally, the resulting binary was disassembled and compared with the original binary to identify differences in the generated instructions, control flow, and memory accesses.

For the evaluation, several representative programs with simple and easily interpretable control-flow structures were used. The programs were compiled into RISC-V ELF binaries, which were subsequently processed by the binary-lifting pipeline. The generated LLVM IR was then compiled back into RISC-V machine code. The resulting binaries were inspected using `llvm-objdump` and compared with the corresponding original binaries.

A simple conditional program was used as an initial test case, as shown in Listing 3:

```
int main(){
    int a = 3;
    int b;
    if (a == 3){
        b = a;
    } else {
        b = 4;
    }
    return b;
}
```

Listing 3: example C file used for evaluation

Listing 3 contains a conditional branch and assignments along both control-flow paths, making it suitable for evaluating instruction translation and control-flow reconstruction. The generated LLVM IR was first processed by LLVM to verify its syntactic and structural validity. Successful processing demonstrates that the lifting process produces LLVM IR that conforms to the requirements of the LLVM compiler infrastructure. The LLVM IR was subsequently compiled into a RISC-V executable. The resulting binary

and the original were disassembled using `llvm-objdump` and compared. As shown in Figure 10, the lifted binary does not contain the same instruction sequence as the original binary. Differences can be observed in the function prologue and epilogue, register usage, memory accesses, and control-flow instructions. These differences are expected because the lifted program is translated into LLVM IR before being processed by the compiler. During recompilation, the compiler may choose different registers, instruction sequences, and memory layouts while representing the same program structure. Consequently, binary equality is not used as the primary correctness criterion. Instead, the comparison of the binaries is used to determine whether the generated binary reflects the relevant structure and operations of the original program.

```
000000000001022c <main>:
1022c: fe010113      addi sp, sp, -32
10230: 00113c23      sd ra, 24(sp)
10234: 00813823      sd s0, 16(sp)
10238: 02010413      addi s0, sp, 32
1023c: 00300793      addi a5, zero, 3
10240: fef42423      sw a5, -24(s0)
10244: fe842783      lw a5, -24(s0)
10248: 0007871b      addiw a4, a5, 0
1024c: 00300793      addi a5, zero, 3
10250: 00f71863      bne a4, a5, 0x10260
10254: fe842783      lw a5, -24(s0)
10258: fef42623      sw a5, -20(s0)
1025c: 00c0006f      jal zero, 0x10268
10260: 00400793      addi a5, zero, 4
10264: fef42623      sw a5, -20(s0)
10268: fec42783      lw a5, -20(s0)
1026c: 00078513      addi a0, a5, 0
10270: 01813083      ld ra, 24(sp)
10274: 01013403      ld s0, 16(sp)
10278: 02010113      addi sp, sp, 32
1027c: 00008067      jalr zero, 0(ra)
```

(a) Original RISC-V binary.

```
00000000000111bc <main>:
111bc: fe058613      addi a2, a1, -32
111c0: 000126b7      lui a3, 18
111c4: 21068693      addi a3, a3, 528
111c8: 00d60633      add a2, a2, a3
111cc: 00a63c23      sd a0, 24(a2)
111d0: 01163823      sd a7, 16(a2)
111d4: 00d585b3      add a1, a1, a3
111d8: 00300513      addi a0, zero, 3
111dc: fea5a423      sw a0, -24(a1)
111e0: fe85a683      lw a3, -24(a1)
111e4: 00a68663      beq a3, a0, 0x111f0
111e8: 00400513      addi a0, zero, 4
111ec: 0080006f      jal zero, 0x111f4
111f0: fe85a503      lw a0, -24(a1)
111f4: fea5a623      sw a0, -20(a1)
111f8: fec5a503      lw a0, -20(a1)
111fc: 01863003      ld zero, 24(a2)
11200: 01063003      ld zero, 16(a2)
11204: 00008067      jalr zero, 0(ra)
```

(b) RISC-V binary generated from the lifted LLVM IR.

Figure 10: Comparison of the lifted and original RISC-V binaries.

No optimization level was explicitly specified during compilation. This was intentional, as compiler optimization can substantially transform the generated code and remove, combine, or replace instructions that are relevant to evaluating the implemented instruction translation. Although optimization could reduce or introduce differences

between the generated binaries, it would also make it more difficult to trace individual instructions through the lifting process. The evaluation therefore prioritizes traceability between the original binary, the generated LLVM IR, and the recompiled binary over compiler optimization.

## 5.1. Security-Related Program Evaluation

In addition to small programs with simple control-flow structures, a security-related program was used to evaluate the lifting process. The selected program implements the PolyRhythm attack, which exploits timing interference caused by contention for shared hardware resources. The attack uses specific memory-access patterns to interfere with the execution of a victim program [33].

The purpose of this evaluation is not to detect the PolyRhythm attack automatically. Rather, the example is used to investigate whether the binary-lifting process can reconstruct and represent a program containing security-related memory-access behavior. This provides an additional test case that is more representative of security-sensitive software than the simple programs used in the previous evaluation.

Listing 4 shows the C implementation used to generate the test binary.

```
int main (){
    int mem_size = 64;
    int CACHE_LINE = 16;
    int local_attack_array[64];
    int sum = 0;
    for (int i = 0; i < mem_size / sizeof(int); i += CACHE_LINE / 4) {
        /* Write attack */
        local_attack_array[i] = 0xff;
    }
}
```

Listing 4: PolyRhythm attack implemented in C

The resulting binary was processed by the binary-lifting pipeline. The control-flow graph recovered from the `main` function is shown in Figure 11. The recovered graph

represents the initialization and loop structure of the original program, including the loop body and its back edge. This demonstrates that the lifting process can recover control-flow structures from a binary containing repeated memory accesses.

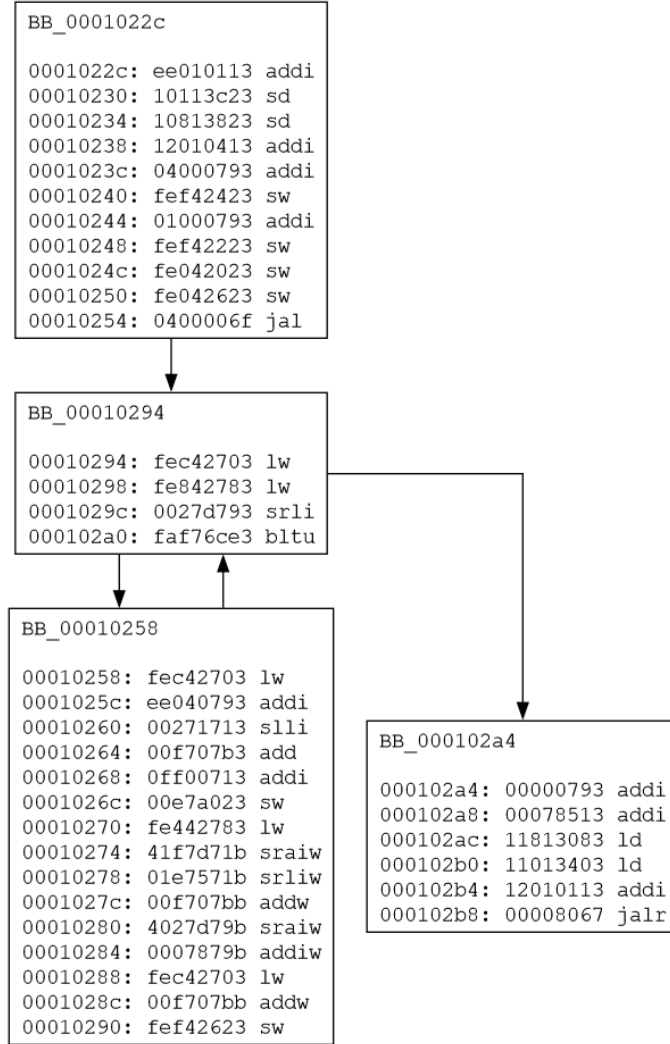


Figure 11: Control-flow graph recovered from the binary containing the PolyRhythm test program.

The corresponding LLVM IR generated by the lifter is shown in Listing 5. The IR contains explicit memory accesses through the global byte-array representation used by the implementation. For example, the `getelementptr` instructions calculate addresses within the modeled memory, while the subsequent `store` instructions represent memory writes performed by the original RISC-V instructions. The loop structure is represented

through LLVM basic blocks and a back edge, while  $\varphi$ -functions merge values originating from different control-flow paths.

```
define i64 @main(i64 %ra_0, i64 %sp_0, i64 %gp_0, i64 %tp_0, i64
%t0_0, i64 %t1_0, i64 %t2_0, i64 %s0_0, i64 %s1_0, ...) {
BB_1022c:
    %sp_1 = add i64 %sp_0, -288
    %sd_10230_addr = add i64 %sp_1, 280
    ...
    %sw_10250_ptr = getelementptr [1048576 x i8], ptr @memory, i64 0,
i64 %sw_10250_addr
    %sw_10250 = trunc i64 0 to i32
    store volatile i32 %sw_10250, ptr %sw_10250_ptr, align 4
    br label %BB_10294
BB_10258:
    %lw_10258_addr = add i64 %s0_1, -20
    ...
    %sw_10290_ptr = getelementptr [1048576 x i8], ptr @memory, i64 0,
i64 %sw_10290_addr
    %sw_10290 = trunc i64 %a5_12 to i32
    store volatile i32 %sw_10290, ptr %sw_10290_ptr, align 4
    br label %BB_10294
BB_10294:
    %a5_3 = phi i64 [%a5_2, %BB_1022c], [%a5_12, %BB_10258]
    %a4_1 = phi i64 [%a4_0, %BB_1022c], [%a4_8, %BB_10258]
    ...
    %lw_10298 = load i32, ptr %lw_10298_ptr, align 4
    %a5_4 = sext i32 %lw_10298 to i64
    %a5_5 = lshr i64 %a5_4, 2
    %cmp_102a0 = icmp ult i64 %a4_2, %a5_5
    br i1 %cmp_102a0, label %BB_10258, label %BB_102a4
BB_102a4:
    %a5_13 = add i64 0, 0
    ...
    ret i64 %a0_1
}
```

Listing 5: Extract of the generated LLVM file

The generated LLVM IR preserves the main structural elements of the program, including the loop and the associated memory accesses. The memory operations are represented explicitly through accesses to the byte-array memory model described in Section 3.7.. This demonstrates that the lifter is able to translate memory-access instructions occurring in a security-related program into the selected LLVM IR representation. However, this evaluation does not establish that the PolyRhythm attack is detected or that its security properties are preserved at the microarchitectural level. Such an evaluation would require additional analysis of timing behavior, shared-resource contention, and the interaction between the attacker and a victim program. The purpose of this test case is therefore limited to evaluating the ability of the binary-lifting pipeline to represent the relevant program structure and memory operations.

## 5.2. Implementation Challenges and Limitations

During the implementation and evaluation of the binary-lifting prototype, several challenges were identified that affected the completeness and reliability of the generated LLVM IR.

The instruction decoder represents a single point of failure within the lifting pipeline. All subsequent stages operate on the instruction representation produced by the decoder. Consequently, an error during decoding can propagate through the CFG construction, SSA construction, and instruction translation stages and ultimately result in incorrect LLVM IR. This is particularly relevant because a custom decoder was implemented instead of relying on an existing disassembly framework. Although restricting the implementation to RV64I reduced the number of required instruction encodings, the decoder still had to correctly identify instruction formats, operands, immediates, and control-flow targets. Errors in control-flow instructions were especially problematic because an incorrectly decoded target can result in an incorrect CFG in addition to an incorrect instruction translation.

Another challenge was control-flow reconstruction when control transfers were not limited to the currently analyzed function. The implementation reconstructs function-level CFGs, making ordinary intra-function branches relatively straightforward to represent. Transfers involving other functions require additional handling because their

targets lie outside the current CFG. Such jumps to other functions are logged at the moment, but not correctly resolved.

Indirect control-flow transfers present an even greater challenge because their target addresses are not necessarily encoded directly in the instruction. The target may instead depend on values computed at runtime. Consequently, such transfers cannot always be resolved using the static information available to the current lifting stage. This limitation is particularly relevant in the context of DEL, discussed in Section 2.5., where dynamic symbolic execution is used to resolve control-flow transfers that cannot be recovered statically. The current prototype does not implement an equivalent dynamic resolution mechanism and therefore cannot guarantee complete recovery of all indirect control-flow targets.

Memory representation was another significant implementation challenge. RISC-V exposes a byte-addressable memory space, whereas LLVM IR provides a typed memory model based on pointers and memory objects. The implementation therefore represents the binary’s memory as a large byte array and translates RISC-V memory operations as accesses to this array.

This approach provides a uniform representation for loads and stores and allows memory addresses calculated by RISC-V instructions to be represented explicitly. However, it also introduces additional complexity when translating different access widths and address calculations. The binary-round-trip evaluation showed that the resulting binaries can differ from the original binaries in their memory-management instructions and memory layout, even when the relevant program behavior is preserved. This represents a limitation of using the LLVM memory model as an intermediate representation of the original architecture-level memory state.

## 6. Conclusion and Further Work

The primary objective of this work was to investigate the feasibility of lifting RISC-V binaries into LLVM IR. A prototype implementation was developed that demonstrates the fundamental lifting process. The implementation includes a custom decoder for RV64I instructions, reconstructs the control flow of the binary, and constructs an SSA-based representation before translating the recovered instructions into LLVM IR. The results demonstrate that RISC-V binaries can be translated into LLVM IR using the proposed approach.

Memory handling is a critical part of this work and needs further development. Although the translation of the supported instructions works, the current memory representation does not yet provide a fully functional mapping between the memory model of the RISC-V binary and the corresponding LLVM representation. Addressing this problem is key for further development. Future work should establish a consistent, maybe even custom, memory model like DEL [1]. It should account for the RISC-V address space, stack and heap memory, global data, and load and store operations.

One major direction for future work is the integration of the developed RISC-V lifting functionality into DLR's DEL. DEL currently provides a binary-lifting and analysis infrastructure for ARMv7. Integrating RISC-V support would allow this existing infrastructure to be reused for a second target architecture and could provide a basis for extending DEL's dynamic analysis capabilities to RISC-V binaries.

For such an integration, the current Python prototype should be reimplemented in C++, as DEL is implemented entirely in C++. A common implementation language would simplify integration with the existing codebase and avoid additional interfaces between components implemented in different languages. Furthermore, a C++ implementation could reduce the runtime overhead of the current prototype and become more suitable for processing larger binaries.

The implementation tries to separate all the ISA specific information from the translation and SSA logic. This was only achieved partially. To enable future development with more ISAs and their extensions, the implementation should become completely modular, for more ISA independence

The current implementation focuses on a subset of the RV64I instruction set. Future work could extend the translator to support additional instructions and RISC-V ISA extensions, such as the M extension for integer multiplication and division, the A extension for atomic operations, and the C extension for compressed instructions. Supporting additional extensions would increase the range of real-world RISC-V binaries that can be processed by the lifter.

Future work could also include optimization of the generated LLVM IR like Rellume [25]. The current prototype primarily focuses on correct instruction translation rather than producing an optimized representation. LLVM's existing optimization infrastructure could potentially be used to simplify the generated IR and improve its suitability for subsequent analysis or execution.

# Bibliography

- [1] H. Abdelmaksoud, Z. A. H. Hammadeh, G. Fey, and D. Lüdtkke, “DEL: Dynamic Symbolic Execution-based Lifter for Enhanced Low-Level Intermediate Representation,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–2. doi: [10.23919/DAT56975.2023.10137253](https://doi.org/10.23919/DAT56975.2023.10137253).
- [2] RISC-V, “RISC-V ABIs Specification Version 1.1.” Accessed: Jul. 20, 2026. [Online]. Available: <https://riscv-non-isa.github.io/riscv-elf-psabi-doc/>
- [3] DLR, “Flight Software Department.” Accessed: Sep. 04, 2026. [Online]. Available: <https://www.dlr.de/en/sc/about-us/departments/flight-software>
- [4] “About RISC-V International.” Accessed: Aug. 26, 2026. [Online]. Available: <https://riscv.org/about/>
- [5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1,” technical report UCB/EECS-2016–118, May 2016. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html>
- [6] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.
- [7] RISC-V, “The RISC-V Instruction Set Manual, Volume I Unprivileged Architecture.” Accessed: Jul. 20, 2026. [Online]. Available: [https://docs.riscv.org/reference/isa/\\_attachments/riscv-unprivileged.pdf](https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf)
- [8] J. D. Vile, “Behind The Scenes of SHD Group’s 2026 RISC-V Market Forecast.” Accessed: Aug. 27, 2026. [Online]. Available: <https://riscv.org/blog/shd-forecast-2026/>
- [9] RISC-V, “Cryptography Extensions: Scalar & Entropy Source Instructions, Version 1.0.1.” Accessed: Aug. 27, 2026. [Online]. Available: <https://docs.riscv.org/reference/isa/v20240411/unpriv/scalar-crypto.html>

- [10] Microchip, “Radiation-Tolerant PolarFire® SoC FPGAs.” Accessed: Aug. 27, 2026. [Online]. Available: <https://www.microchip.com/en-us/products/fpgas-and-plds/radiation-tolerant-fpgas/rt-polarfire-soc>
- [11] H. Otte *et al.*, “Preliminary Design of the Stellar Apps Software Platform for Developing and Executing On-board Applications,” in *2025 European Data Handling & Data Processing Conference (EDHPC)*, 2025, pp. 1–8.
- [12] T. Committee, “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2.” Accessed: Aug. 05, 2026. [Online]. Available: <https://refspecs.linuxfoundation.org/elf/elf.pdf>
- [13] D. S. Gracia, A. Valero, R. G. Tejero, M. Villarroja, and V. Viñals, “peRISCVcope: A Tiny Teaching-Oriented RISC-V Interpreter,” in *2022 37th Conference on Design of Circuits and Integrated Circuits (DCIS)*, 2022, pp. 1–6.
- [14] Linux, “elf(5) — Linux manual page.” Accessed: Aug. 05, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man5/elf.5.html?>
- [15] C. Kruegel, W. Robertson, F. Vaur, and G. Vigna, “Static disassembly of obfuscated binaries,” in *USENIX security Symposium*, 2004, p. 18.
- [16] C. Cifuentes and K. J. Gough, “Decompilation of binary programs,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995, doi: <https://doi.org/10.1002/spe.4380250706>.
- [17] F. E. Allen, “Control flow analysis,” *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, doi: [10.1145/390013.808479](https://doi.org/10.1145/390013.808479).
- [18] F. Chow, “Intermediate representation,” *Communications of the ACM*, vol. 56, no. 12, pp. 57–62, 2013.
- [19] V. Alfred, S. Monica, S. Ravi, U. Jeffrey D, and others, *Compilers Principles, Techniques*. Pearson, 2007.
- [20] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge University Press, 1997.
- [21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,”

- ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, doi: [10.1145/115372.115320](https://doi.org/10.1145/115372.115320).
- [22] LLVM, “LLVM Language Reference Manual.” Accessed: Jun. 19, 2026. [Online]. Available: <https://llvm.org/docs/LangRef.html>
  - [23] A. Dinaburg and A. Ruef, “Mcsema: Static translation of x86 instructions to llvm,” in *ReCon 2014 Conference, Montreal, Canada*, 2014.
  - [24] V. Chipounov and G. Candea, “Enabling sophisticated analyses of x86 binaries with RevGen,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 211–216. doi: [10.1109/DSNW.2011.5958815](https://doi.org/10.1109/DSNW.2011.5958815).
  - [25] A. F. Engelke, “Optimizing Performance Using Dynamic Code Generation,” Doctoral dissertation, 2021. [Online]. Available: <https://mediatum.ub.tum.de/1614897>
  - [26] J. Armengol-Estapé, R. C. O. Rocha, J. Woodruff, P. Minervini, and M. F. P. O’Boyle, “Forklift: An Extensible Neural Lifter.” [Online]. Available: <https://arxiv.org/abs/2404.16041>
  - [27] R. C. O. Rocha *et al.*, “Lasagne: a static binary translator for weak memory model architectures,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, in PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 888–902. doi: [10.1145/3519939.3523719](https://doi.org/10.1145/3519939.3523719).
  - [28] N. S. Agency, “Ghidra Software Reverse Engineering Framework.” Accessed: Aug. 07, 2026. [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra>
  - [29] E. B. et al., “pyelftools.” Accessed: Aug. 26, 2026. [Online]. Available: <https://github.com/eliben/pyelftools>
  - [30] “4 objdump.” Accessed: Aug. 12, 2026. [Online]. Available: <https://sourceware.org/binutils/docs/binutils/objdump.html>

- [31] LLVM, “llvm-objdump - LLVM’s object file dumper.” Accessed: Aug. 12, 2026.  
[Online]. Available: <https://llvm.org/docs/CommandGuide/llvm-objdump.html>
- [32] Capstone, “Capstone The Ultimate Disassembler.” Accessed: Aug. 21, 2026.  
[Online]. Available: <https://www.capstone-engine.org/>
- [33] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, “PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference,” in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 225–239. doi: [10.1109/RTSS55097.2022.00028](https://doi.org/10.1109/RTSS55097.2022.00028).

# AI Acknowledgement

| Tool        | Description of Usage                                                   |
|-------------|------------------------------------------------------------------------|
| ChatGPT     | Used for language corrections and stylistic improvements (full thesis) |
| DeepL Write | Used for language corrections and stylistic improvements (full thesis) |
| DeepL       | Used for translating text passages (full thesis)                       |

Table 4: Description of Used AI Tools