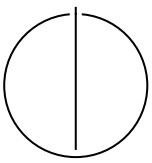
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Airlift: A Binary Lifter Based on a Machine-Readable Architecture Specification

Wonkeun Choi



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

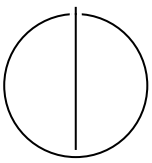
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Airlift: A Binary Lifter Based on a Machine-Readable Architecture Specification

Airlift: Ein Binärlifter auf Basis einer maschinenlesbaren Architekturspezifikation

Author:	Wonkeun Choi
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Martin Fink
Submission Date:	30.05.2025



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 30.05.2025 Wonkeun Choi

Acknowledgments

I'd like to thank Martin for what has been easily the most enjoyable project topic I've had the honor to be a part of. I'm also grateful to Prof. Bhatotia for his trust and generous support. Without him, I wouldn't have found my footing in this area or completed the thesis with such fulfillment.

Thanks to my parents for always being there—and consistently giving the wrong advice. And thanks to Fabi for moral support.

Abstract

Binary lifting is the process of translating machine code into an intermediate representation suitable for program analysis and transformation. It is a foundational technique in both academic research and industrial tooling, but building and maintaining binary lifters remains complex and error-prone. This thesis presents **Airlift**, a binary lifter automatically generated from a machine-readable architecture specification. By eliminating the need for manual lifter development, Airlift significantly reduces implementation effort and ensures consistency with the formal semantics of the target architecture.

Airlift is integrated into the TrustNoJit system, which applies the proof-carrying code framework to increase the safety of just-in-time compilation. The lifter translates AArch64 machine code into AIR, a custom lightweight IR tailored for formal verification, and serves as a crucial step in enabling static safety checks.

Airlift leverages partial evaluation of instruction semantics via hybrid transpilation: computations involving only values known at lifter runtime, such as instruction encoding fields, are transpiled into executable code, while semantics dependent on machine state, such as register values, are emitted as residual AIR code. This approach significantly reduces the volume and complexity of generated AIR, streamlining the verification process.

The practical applicability of Airlift is demonstrated by decoding and successfully lifting all 114 programs in the Sightglass benchmarking suite. Focusing primarily on correctness, this work compares Airlift's performance and structural efficiency with a manually written lifter and identifies potential optimizations for future work.

Contents

1 Introduction

Binary lifting is a foundational technique in binary analysis and verification, used to translate machine code into an intermediate representation (IR) for purposes such as program analysis [29, 8], symbolic execution [25, 16], binary translation [24, 11, 5], decompilation [4], and formal verification [31]. It plays a central role in industry tools for reverse engineering [12] and malware analysis [17], as well as in production-grade binary instrumentation [7] and verified system deployment [13]. Despite its utility, writing a binary lifter manually is complex and error-prone, requiring deep knowledge of the target architecture's semantics and a significant engineering effort to ensure correctness and maintainability [8].

Manually implemented lifters such as McSema [6] exemplify this difficulty, relying on extensive hand-written decoding and translation logic. Other approaches have attempted to automate aspects of lifter generation: Cross-Architecture Lifter Synthesis [27] and Forklift [2] present novel frameworks for deriving lifters via synthesis or transformation from existing binaries or IR. However, these systems introduce complexity and potential for incompleteness due to reliance on symbolic reasoning or transformation heuristics.

This thesis explores an alternative: automatically generating a binary lifter from a machine-readable architecture specification (MRAS), and applying it in a real-world setting. Specifically, we target the TNJ system (Section 2.2), which increases the safety of just-in-time (JIT) compilation through machine code verification. We introduce Airlift, a binary lifter generated directly from a formal specification of the AArch64 instruction set architecture (ISA) [15].

All components of Airlift that depend on the ISA, including instruction decoding and semantic translation, are generated from the specification. The instruction and operand decoder is directly transpiled into executable Rust code. Instruction semantics are partially transpiled into executable code and partially into AIR, the IR custom designed for use in TNJ. Semantics that do not depend on register values are executed during lifting time, while register-dependent logic is emitted as IR. This hybrid approach reduces IR size and simplifies lifting runtime behavior.

We use Arm's ASL, a formal, machine-readable description of AArch64 instruction semantics [28]. Decoder logic and computations with values known at lifter runtime, meaning those that depend only on the instruction encoding, are transpiled into Rust

and executed during translation. Semantics that depend on runtime state, such as register or memory values, are translated into AIR. We reuse the lexer and parser from the existing ASL interpreter (ASLi) project [23]. All other lifting logic and code generation are implemented from scratch in Rust using standard metaprogramming libraries. This design makes the system self-contained and easy to integrate into TNJ. Airlift currently supports integer, memory, and branch instructions. Floating point (FP) and single instruction, multiple data (SIMD) instructions, which are not relevant to TNJ, are assigned opaque behavior.

Airlift is evaluated by comparing its output and performance against a manually implemented lifter developed in a previous thesis for TNJ [10]. Although Airlift does not aim to outperform hand-written lifters in speed, this comparison helps identify optimization opportunities. Additionally, we run Airlift on all Sightglass benchmark programs [1] to validate instruction coverage and semantic correctness in a practical context.

The core contribution of this work lies in demonstrating that a binary lifter can be automatically generated from a formal specification and deployed within a verified execution pipeline. Airlift was developed independently and prior to our awareness of Lift-Offline [9], a recent system that formalizes a similar approach. While Lift-Offline presents a theoretical framework and tooling for instruction lifter synthesis, Airlift serves as a concrete application of this idea in a real-world security-focused system.

The source code for Airlift is available as open source on GitHub.[1]

[1] `https://github.com/TUM-DSE/airlift`

2 Background

2.1 Binary Lifting

Binary lifting is the process of translating machine code into an IR at a higher level of abstraction, making it suitable for program analysis or transformation. Analyzing raw binary instructions directly is difficult because they are architecture-specific, densely encoded, and often rely on implicit side effects. Lifting exposes control flow, data flow, and memory effects explicitly, allowing higher-level tools to operate on a more abstract and uniform representation.

While the introduction discussed the broader relevance of binary lifting, this section presents its technical foundations and use cases in a more structured way. Binary lifting is widely used in both academic and industrial systems, especially in binary analysis frameworks [29, 8, 17, 31], symbolic execution engines [25, 16], binary translation systems [24, 11, 5], binary instrumentation tools [7], decompilers [4, 12], and formally verified kernels [13].

2.1.1 Intermediate Representations

An intermediate representation (IR) is a machine-independent language that abstracts away the low-level details of machine code. It serves as a bridge between raw binary instructions and the tools that analyze or modify them. A well-designed IR captures the essential semantics of the input code while remaining tractable for automated reasoning and transformation.

LLVM IR is one of the most widely adopted IR because of its maturity, comprehensive tooling, and documentation [14]. Its static single-assignment (SSA) form and widespread adoption make it particularly useful for compiler optimization and symbolic execution, as it comes with extensive tooling, optimizations, and analysis features available out of the box. However, LLVM IR was not designed to model certain low-level behaviors, such as partial register updates, architecturally undefined operations, or precise memory side effects. This limitation can result in incorrect behavior when lifting machine code into LLVM IR, especially in security-critical or formally verified contexts.

To address these limitations, many systems use custom IRs that are tailored to

the semantics of machine instructions. Examples include VEX (used in Valgrind and angr) [19, 25] and BIL (used in BAP) [8], which are explicitly designed to model low-level behavior in a way that is both accurate and analyzable. This is critical for use cases such as verification and dynamic analysis. In contrast, AIR (used in TNJ) serves a different goal: it is designed to be lightweight and simple, with just enough expressiveness to support machine code verification while avoiding unnecessary complexity.

2.1.2 Use-Cases

Binary lifting enables a wide range of applications. While many of these are outlined in documentation for tools like McSema [6], they apply more generally across various platforms:

- **Binary Modification and Patching:** After lifting a binary into an IR, it becomes easier to apply transformations such as restructuring control flow, optimizing performance, inserting runtime checks, or removing unwanted functionality. The transformed IR can then be compiled back into a new binary.
- **Symbolic Execution:** Engines like KLEE operate on LLVM IR. Lifting a binary into this format allows symbolic execution of programs even when the original source code is not available.
- **Reuse of IR-based Tooling:** Once a binary is lifted into a general-purpose IR, it becomes compatible with existing tools such as libFuzzer, sanitizers, and optimization passes. This makes it easier to integrate analysis and testing into existing workflows.
- **Binary-Only Analysis:** When source code is unavailable, or when compiler optimizations significantly alter the code's behavior, analyzing the binary itself is the only reliable way to understand the program's true runtime behavior.
- **Unified Tooling:** Using a shared IR for both source and binary code allows developers to maintain a single suite of analysis tools, reducing development effort and improving consistency across different types of input.

2.2 TrustNoJit (TNJ)

JIT compilers improve runtime performance by converting high-level code such as JavaScript or WebAssembly into machine code during execution. However, this optimization introduces significant security risks. The output machine code is typically not

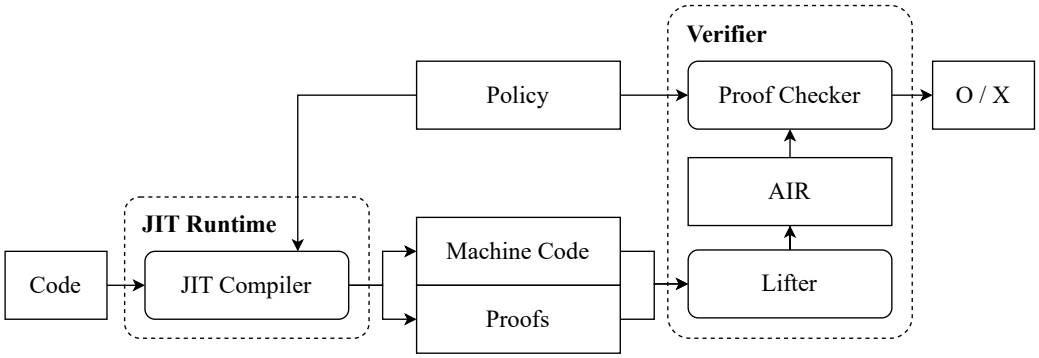


Figure 2.1: Workflow of TNJ.

verified, and the JIT compilation process can introduce vulnerabilities that are difficult to detect at runtime.

This problem is well-documented. In 2023, 9 of the 19 zero-day vulnerabilities exploited in the wild against web browsers originated from JavaScript engines [26]. Since 2019, approximately 45% of known vulnerabilities in Google's V8 JavaScript engine have involved the JIT compiler [20]. These statistics indicate that JIT compilation remains one of the main attack surfaces in modern browser environments.

TNJ complements existing runtime defenses by applying the proof-carrying code (PCC) framework [18] to enforce security guarantees for JIT-compiled code. The JIT compiler must produce a formal proof that its output satisfies a predefined security policy, which is verified before execution. Only the verifier and the lifter are trusted; the JIT compiler is treated as untrusted.

Figure 2.1 shows the architecture of TNJ. The JIT compiler receives a high-level program and a security policy. It compiles the program into machine code and generates a corresponding safety proof. The verifier receives the machine code and the proof. It uses a lifter to translate the machine code into AIR, an intermediate representation designed for verification. The verifier then checks whether the proof establishes that the lifted code satisfies the given policy. If the verification succeeds, the machine code is allowed to execute.

This project focuses on the development of the lifter. The lifter translates machine code into AIR, which is designed to be lightweight and platform-agnostic. AIR exposes enough semantic information to support verification, while avoiding the complexity of architecture-specific details. By operating on AIR, the verifier can check that the machine code adheres to the policy before execution.

2.2.1 Assembly Intermediate Representation (AIR)

AIR is a minimal intermediate representation designed specifically for verifying the safety of machine code generated by JIT compilers. Unlike general-purpose IRs such as LLVM IR or VEX, AIR is tailored to support formal verification of instruction-level behavior with minimal complexity. Its design is constrained by the need to statically check safety properties over low-level code without introducing unnecessary overhead or architectural dependencies.

AIR is in SSA form, meaning each variable is assigned exactly once and every variable is defined before use. SSA form facilitates a wide range of optimizations and program analyses by making data dependencies explicit. This property simplifies tasks such as constant propagation, dead-code elimination, and control-flow simplification, which in turn reduces the complexity of code that undergoes verification or instrumentation.

AIR includes three data types:

- `FixedSizeInt`, a bounded integer type used to represent register values and arithmetic results of machine instructions.
- `Bool`, used to represent condition codes and control-flow decisions.
- `Int`, an unbounded, signed integer type used for expressing operations whose semantics, as described in the ASL MRAS, cannot be fully captured with fixed-width integers alone.

Only `FixedSizeInt` and `Bool` were strictly necessary for the purposes of verifying safety properties in TNJ. However, `Int` was introduced to ensure that the semantics described in ASL could be fully represented during the lifting process, even if those semantics involved arbitrary-precision values that do not appear directly in the target machine code.

Table 2.1 lists all AIR instructions required to represent the subset of ASL semantics used in this work. This table illustrates the minimality of the AIR instruction set. The small size of this instruction set not only simplifies the verifier but also makes AIR more amenable to formal reasoning.

Figure 2.2 provides an example of AIR code generated from the ASL semantics of the `csel` instruction. This example demonstrates how AIR captures the data flow and control flow of an instruction while maintaining verifiability and semantic transparency. From the listing, it is also evident how basic blocks are defined and how data is passed explicitly as block parameters. This design allows control-flow joins to be expressed cleanly in SSA form and is sufficient to represent all control-flow structures encountered in ASL instruction semantics.

Table 2.1: Overview of AIR instructions (`FSI` = `FixedSizeInt` and `BB` = `BasicBlock`).

Instruction	Description	Input	Output
`add`	Add unbounded integers	`Int`, `Int`	`Int`
`sub`	Subtract unbounded integers	`Int`, `Int`	`Int`
`mul`	Multiply unbounded integers	`Int`, `Int`	`Int`
`div`	Divide ubounded integers	`Int`, `Int`	`Int`
`modulo`	Modulo on unbounded integers	`Int`, `Int`	`Int`
`wrapping_add`	Add `FSI`s	`FSI`, `FSI`	`FSI`
`wrapping_sub`	Subtract `FSI`s	`FSI`, `FSI`	`FSI`
`umul`	Unsigned multiply `FSI`s	`FSI`, `FSI`	`FSI`
`imul`	Signed multiply `FSI`s	`FSI`, `FSI`	`FSI`
`udiv`	Unsigned divide `FSI`s	`FSI`, `FSI`	`FSI`
`idiv`	Signed divide `FSI`s	`FSI`, `FSI`	`FSI`
`modulo`	Modulo on `FSI`s	`FSI`, `FSI`	`FSI`
`lshl`	Logical shift left	`FSI`, `Int`	`FSI`
`lshr`	Logical shift right	`FSI`, `Int`	`FSI`
`ashl`	Arithmetic shift left	`FSI`, `Int`	`FSI`
`ashr`	Arithmetic shift right	`FSI`, `Int`	`FSI`
`and`	Logical AND	`FSI`, `FSI`	`FSI`
`or`	Logical OR	`FSI`, `FSI`	`FSI`
`xor`	Logical XOR	`FSI`, `FSI`	`FSI`
`bitwise_not`	Bitwise NOT	`FSI`	`FSI`
`zext`	Zero-extend value	`FSI`	`FSI`
`sext`	Sign-extend value	`FSI`	`FSI`
`from_bool`	Convert `Bool` to `FSI` (i1)	`Bool`	`FSI`
`from_i1`	Convert `FSI` (i1) to `Bool`	`FSI`	`Bool`
`signed_from_bits`	Convert `FSI` as signed to `Int`	`FSI`	`Int`
`unsigned_from_bits`	Convert `FSI` as unsigned to `Int`	`FSI`	`Int`
`to_bits`	Convert `Int` to `FSI`	`Int`	`FSI`
`load`	Load from memory	`FSI`	`FSI`
`store`	Store to memory	`FSI`, `FSI`	-
`read_reg`	Read from register	`Reg`	`FSI`
`write_reg`	Write to register	`FSI`, `Reg`	-
`icmp`	Compare values	`FSI`, `FSI`, `CmpTy`	`Bool`
`jump`	Jump unconditionally	`BB`	-
`jumpif`	Jump conditionally	`BB`, `BB`, `Bool`	-
`dynamic_jump`	Jump to an address	`FSI`	-
`opaque`	Create an opaque value	-	`FSI`
`trap`	Trap unconditionally	-	-

```
entry:
    v0 = i64.read_reg "x1"
    v1 = i64.read_reg "x2"
    v2 = i1.read_reg "n"
    v3 = bool.icmp.i1.eq v2, 0x1
    jumpif v3, block_0, block_1
block_0:
    jump block_2(v0)
block_1:
    jump block_2(v1)
block_2(v4: i64):
    write_reg.i64 v4, "x0"
```

Figure 2.2: An example AIR code generated for the `csel` instruction.

2.3 AArch64 Architecture

AArch64 is the 64-bit execution mode of the Arm architecture [15]. It has become widely adopted in mobile, embedded, and increasingly in server environments, owing to its regular instruction encoding, straightforward register model, and reduced instruction set design. These architectural choices facilitate efficient hardware implementations and simplify tasks such as code generation, static analysis, and formal verification.

All instructions in AArch64 are 32 bits wide. The architecture defines 31 general-purpose registers (`x0` $\cdots$ `x30`). The encoding value 31 is used to represent either the zero register or the stack pointer, depending on context. A separate set of 32 FP and SIMD registers (`v0` $\cdots$ `v31`), introduced by the Neon architecture extension, are not relevant to this project.

The ISA supports several addressing modes, including immediate offsets, register offsets, and scaled register-based indexing. Instruction classes include arithmetic and logical operations, load/store operations (including exclusive memory accesses and paired loads/stores), and a variety of control-flow instructions such as conditional branches, indirect jumps, and exception-generating instructions.

Figure 2.3 shows a short sequence of AArch64 assembly to demonstrate the syntax and structure of the ISA. This code demonstrates basic data movement, arithmetic, condition flag usage, and branching. Despite the visual simplicity, AArch64 instructions often carry nontrivial semantics, especially when interacting with condition flags and control flow.

Many instructions in AArch64 have multiple encodings or aliases. Some instructions

```
start:
    mov x0, #5
    mov x1, #10
    add x2, x0, x1
    cmp x2, #15 // sets condition flags
    b.eq match // reads condition flags
    mov x3, #0
    b end
match:
    mov x3, #1
end:
```

Figure 2.3: An example AArch64 code snippet.

affect condition flags stored in the PSTATE register (N, Z, C, V), which are then consumed by conditional branches and comparisons. These implicit dependencies introduce additional complexity when analyzing instruction sequences.

AArch64 was chosen as the target architecture for this project for several reasons. It is one of the most widely used 64-bit ISAs in practice and one of the few with a complete, publicly available MRAS. Another example of such a specification is RISC-V's Sail-based formal model [3], while an x86 specification is still under active development [22]. The focus on a single ISA is a deliberate design decision: building a functional binary lifter for just one ISA is already a significant workload, particularly within the six-month timeframe of a master's thesis and without extensive expertise in compiler development. However, the approach presented in this work is general and can be extended to other ISAs with MRASs, including RISC-V and x86, with appropriate modifications to the parser and translation pipeline.

2.3.1 Architecture Specification Language (ASL)

ASL is a formal language used to define the behavior of instructions in an MRAS [21]. It is designed to serve multiple purposes:

- **Documentation:** ASL is intended to be human-readable and serves as the official specification of the ISA.
- **Test Generation:** It assists in writing test programs to check actual hardware behavior against expected semantics.

- **ISA Design:** It helps in modeling and testing the behavior of proposed ISA features during the design process.
- **Formal Verification:** It provides a formal basis for verifying the correctness of hardware and binary analysis tools.

For many of these use cases, it is important that the specification is executable. Subsection 2.3.1 introduces the ASLi, a tool for executing ASL-defined instruction semantics.

ASL is written as an imperative language to align with the mental model of most programmers. It includes support for unbounded integers, infinite-precision reals, fixed-size bitvectors, booleans, enumerations, and record types (similar to structs). These data types are sufficient to describe complex instruction behavior in a precise and analyzable way.

To demonstrate the structure of ASL and the layout of its machine-readable specification, we use the `csel` instruction as a running example. We selected `csel` because it is semantically simple while still showcasing key language features such as field extraction, conditional logic, and subroutine calls.

The ASL MRAS is split across several files, each with a different role:

- `arch_decode.asl` defines the top-level instruction decoding logic. It consists of nested match/case statements that determine which instruction is being decoded based on specific bit fields in the 32-bit instruction. Figure 2.4 shows how the opcode of a `csel` instruction is extracted. Once the instruction is identified, decoding continues into the next file.
- `arch_instrs.asl` defines operand decoding logic and instruction semantics. Each instruction definition includes a decode block that extracts operand fields from the binary encoding and an execute block that describes its behavior during execution. Figure 2.5 shows the decode and execute blocks for `csel`. In this example, register reads and writes are modeled as array accesses to `X[]`, and control logic such as `ConditionHolds()` and `NOT()` is abstracted into subroutines for readability.
- `arch.asl` contains subroutine definitions. It is the largest file in the specification by content, as it centralizes most of the reusable logic, including functions like `ConditionHolds()` that are referenced by decode blocks and execute blocks in `arch_decode.asl` and `arch_instrs.asl`.
- `prelude.asl` defines the lowest-level built-in operations such as `NOT()`. These subroutines are dependent on the simulation or verification backend and are often transpiled directly to equivalent operations in the target language.

- `regs.asl` lists architectural registers and describes their structure, including field-level breakdowns of bit segments.
- `memory.asl` defines logic related to memory reads and writes.
- Other files, such as `debug.asl` and `interrupts.asl`, describe other parts of the architecture, but they were not relevant to the implementation of Airlift and had minimal or no interaction during development.

```
__decode A64
case (29 +: 3, 24 +: 5, 0 +: 24) of
    ...
    when (_, 'x101x', _) =>
        case (
            31 +: 1, 30 +: 1, 29 +: 1, 28 +: 1, 25 +: 3,
            21 +: 4, 16 +: 5, 10 +: 6, 0 +: 10
        ) of
            ...
            when (_, _, _, '1', _, '0100', _, _, _) => // csel
                __field sf 31 +: 1
                __field op 30 +: 1
                __field S 29 +: 1
                __field Rm 16 +: 5
                __field cond 12 +: 4
                __field op2 10 +: 2
                __field Rn 5 +: 5
                __field Rd 0 +: 5
                case (sf, op, S, op2) of
                    when ('0', '0', '0', '00')
                        => __encoding aarch64_integer_conditional_select
                    ...
```

Figure 2.4: Instruction decoding logic of `csel` in `aarch_decode.asl`.

ASL interpreter (ASLi)

ASL was designed to be an executable specification, and Arm provides an open-source implementation of an interpreter, referred to as ASLi, that can lex, parse, and evaluate ASL code [23]. ASLi supports both interactive interpretation and ahead-of-time

```
__instruction aarch64_integer_conditional_select
    __encoding aarch64_integer_conditional_select
        __instruction_set A64
        __field sf 31 +: 1
        __field op 30 +: 1
        __field Rm 16 +: 5
        __field cond 12 +: 4
        __field o2 10 +: 1
        __field Rn 5 +: 5
        __field Rd 0 +: 5
        __opcode 'xx011010␣100xxxxx␣xxxx0xxx␣xxxxxxxx'
        __guard TRUE
        __decode
            integer d = UInt(Rd);
            integer n = UInt(Rn);
            integer m = UInt(Rm);
            integer datasize = if sf == '1' then 64 else 32;
            bits(4) condition = cond;
            boolean else_inv = (op == '1');
            boolean else_inc = (o2 == '1');

    __execute
        bits(datasize) result;
        bits(datasize) operand1 = X[n];
        bits(datasize) operand2 = X[m];

        if ConditionHolds(condition) then
            result = operand1;
        else
            result = operand2;
            if else_inv then result = NOT(result);
            if else_inc then result = result + 1;

        X[d] = result;
```

Figure 2.5: Operand decoding logic and instruction semantics of `csel` in `aarch_instrs.asl`.

compilation into C. This makes it a valuable tool for simulating instruction behavior and prototyping ASL-based workflows.

ASLi is implemented in OCaml and relies on the Ott tool for defining ASL's grammar and type system. Ott enables the definition of programming languages in a high-level, machine-readable format and generates parser and type-checker code accordingly. In the case of ASL, it produces the OCaml modules needed to lex, parse, and type-check the specification.

While ASLi provides a foundation for executing ASL code, it is currently in alpha and lacks full coverage of the ASL specification. Several known bugs and missing features limit its usability as a general-purpose simulator or conformance tool. It is not intended to serve as a fully validated architectural reference simulator.

In this project, ASLi was used selectively. Although the interpreter as a whole is incomplete and not thoroughly tested, its low-level components such as the lexer, parser, and type checker have proven reliable. These components were reused in the early stages of Airlift's transpilation pipeline to parse and type-check ASL code before generating Rust code and AIR.

3 Overview

Binary lifting is a central technique in many binary analysis frameworks, including decompilers, symbolic execution engines, and formal verification systems, as discussed in Section 2.1. However, writing a binary lifter by hand remains a labor-intensive and error-prone process. It requires in-depth understanding of instruction semantics, decoding logic, and IR generation. Even small implementation errors can lead to incorrect analysis results downstream, which is particularly problematic in security- and verification-sensitive applications.

Despite these challenges, binary lifters are a critical component of many widely used systems. However, their manual construction does not scale across ISAs. Each new ISA or ISA extension typically requires a fresh reimplementation of decoding logic and instruction semantics, introducing opportunities for bugs and increasing long-term maintenance costs.

At the same time, MRASs are becoming more common. AArch64 has ASL, a formally specified and parsable representation of its instruction semantics [28]. RISC-V uses Sail [3], and an MRAS for x86 is also under development [22]. These specifications encode the same information lifters require, but are underused in practice for lifter generation. There is a clear opportunity to reduce manual engineering effort by automatically generating lifters directly from an MRAS.

This gap is particularly visible in the TNJ project. TNJ requires a formally verified lifting pipeline that converts machine code into a verification-friendly IR, enabling proof-carrying code checks. Unlike most systems using binary lifters, TNJ cannot rely on runtime instrumentation or informal semantics. Instead, it needs a statically checkable and trustworthy translation from binary code to AIR.

Airlift addresses this need. It demonstrates how a lifter can be automatically generated from ASL, eliminating the need to reimplement decoding and semantics manually. The resulting lifter emits AIR, a minimal IR designed for proof-carrying verification. This pipeline shows that MRAS-driven lifting is not only possible, but practical for use in formally secure systems.

The design of Airlift was guided by the following goals:

- **High Instruction Coverage:** The system aims to support a broad subset of AArch64 instructions, prioritized based on the needs of TNJ. Focus is placed

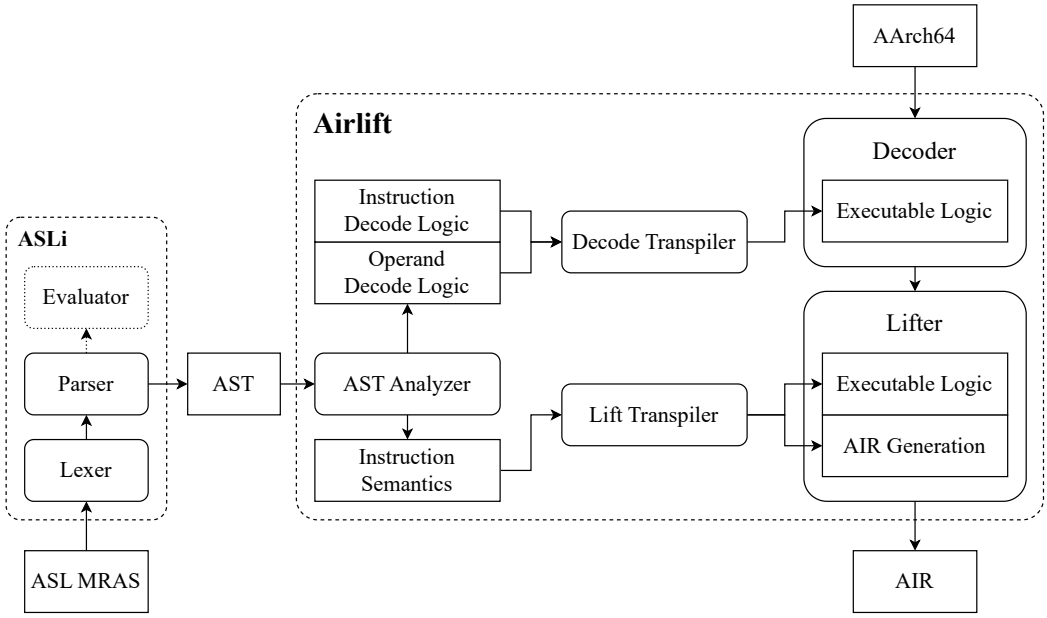


Figure 3.1: Workflow of Airlift.

on integer, memory, and control-flow instructions that are most relevant to the verification pipeline.

- **Correctness as the Primary Objective:** The main goal of this project is to ensure semantic correctness. Lifted AIR code must accurately preserve the behavior of AArch64 instructions as described in the MRAS. This priority guides all design and implementation decisions.
- **Performance and Optimization Out of Scope:** Runtime performance and systematic optimizations were not in scope for this thesis. Correctness was prioritized due to the limited timeframe and complexity of the task. However, the main design decision for the code generator already reduces the number of generated AIR instructions significantly. Additional possible optimization strategies are documented in Chapter 7.

Figure 3.1 shows a simplified model of the main components of Airlift. While many implementation details are omitted, the overall pipeline can be abstracted into the following stages:

- **MRAS AST Extraction:** As described in Subsection 2.3.1, Airlift reuses the lexer,

parser, and typechecker provided by ASLi to extract the abstract syntax tree (AST) of the entire AArch64 MRAS. This parsed and typed AST serves as the input to the rest of the pipeline.

- **Code Generation:** The goal of this stage is to generate all parts of the decoder and lifter that depend on the architecture specification. This comprises most of the decoder and lifter logic, with the exception of some manually written skeleton code. The main components of the MRAS AST used here include the instruction decode logic, operand decoding logic, and instruction semantics, examples of which were presented in Subsection 2.3.1. Other AST components are also processed by the AST analyzer but are not shown in the figure for simplicity.

  Because decoding and lifting require fundamentally different translation strategies, Airlift uses two separate transpilers for these tasks:

  - *Decoder transpilation:* The decode transpiler converts instruction decode and operand decode logic into executable Rust code that becomes the core of the decoder.
  - *Lifter transpilation:* The lift transpiler translates instruction semantics into code that emits AIR. Not all semantics need to be lifted directly; Airlift supports partial evaluation, allowing semantics independent of the runtime state to be executed by the lifter itself. As a result, the transpiler outputs both executable logic and AIR generation code.

- **Lifting:** Once the decoder and the lifter have been generated, Airlift can translate AArch64 machine code into AIR in the same manner as a manually written binary lifter.

The design and logic behind each of these stages are described in detail in Chapter 4. Chapter 5 presents the corresponding implementation, including technical decisions and code-level mechanisms that realize this pipeline.

4 Design

4.1 Lexing and Parsing

To avoid reimplementing the extensive and evolving parsing infrastructure for ASL, we reuse the lexer, parser, and typechecker from the upstream ASLi, Arm's reference implementation. This decision ensures compatibility with the original specification and eliminates a major source of potential bugs, allowing us to focus on the lifter design itself. We treat the parser as a black-box frontend that produces a typed ASL AST, which serves as the sole input to subsequent analysis and generation stages.

4.2 AST Analysis

The AST analyzer traverses the typed ASL AST and extracts all information relevant for decoder and lifter generation. This includes constants, enumerations, structured record types, subroutines, instruction decode logic, operand decode logic, and instruction semantics. These components are placed into internal data structures tailored for efficient downstream processing.

- **Constants and Enums**: used to resolve symbolic expressions and control flow conditions.
- **Records**: represent structured types such as memory descriptors or fault metadata.
- **Subroutines**: reusable code blocks that may be invoked from decode logic, instruction semantics, or both. There are two kinds of subroutines: *functions*, which return a value, and *procedures*, which do not. Each subroutine is analyzed to determine its dependencies and transpiled by either the decode transpiler, the lift transpiler, or both accordingly.
- **Instruction Decode Logic and Operand Decode Logic**: provide the necessary information to generate executable code for instruction decoding.
- **Instruction Semantics**: represent the core behavior of each instruction and serve as the input to the lifter code generator.

The output of this stage is a collection of relevant AST nodes and minimal associated metadata, organized into internal data structures for later use by the code generation stages. Most elements are stored without transformation, preserving their original structure for deferred handling. Limited processing is applied only where necessary, such as indexing enumerations by type or variant name, to enable efficient access. The analysis phase itself remains lightweight and deliberately decoupled from code generation logic, ensuring a clean separation of concerns.

4.3 Decoder Code Generation

The generated decoder is essentially a direct transpilation of the instruction and operand decode logic from the MRAS into executable code. Here, "executable code" refers specifically to logic that is fully evaluated during lifter runtime, without generating any AIR code. This absence of generated AIR code is what sets the decode logic transpilation process apart from the transpilation of instruction semantics in lifter generation. In decoder generation, all logic is resolved up front, and no part of the decoding process is lifted into AIR. In contrast, lifter generation does not simply transpile all logic to AIR code. Instead, some lifting logic is transpiled to executable code, as in decoder generation, while the rest is transpiled to AIR code. The details of this hybrid transpilation process are described in Subsection 4.4.3.

Figure 2.4 illustrates the general structure of instruction decode logic in the MRAS, while Figure 2.5 shows the operand decode logic for the `csel` instruction as a concrete example.

To generate the decoder, we recursively traverse the AST of the instruction and operand decode logic in semantic order. For each construct encountered, we emit a corresponding construct in the target language, resulting in a one-to-one semantic transpilation. This approach ensures that the generated code closely mirrors the logic of the original specification, preserving its structure and behavior.

4.3.1 Lazy Subroutine Transpilation

As shown in the decode block of the `csel` instruction in Figure 2.5, subroutine calls, such as the call to `ConditionHolds()`, may appear within the decode logic. Unlike constants, enumerations, and records, which are transpiled in full at the beginning of the pipeline, subroutines are transpiled on demand.

When the decode transpiler encounters a subroutine call in the decode logic, it first checks whether the corresponding implementation has already been generated. If not, it recursively transpiles the subroutine definition, including any nested subroutine calls

it may contain. In the case of `ConditionHolds()`, this means resolving and generating the function body before continuing with the outer decode logic.

This lazy transpilation strategy avoids the need to implement full ASL coverage upfront. Only subroutines that are reachable from the decode logic of supported instructions are transpiled. This allows the system to ignore unsupported constructs gracefully while remaining modular and extensible for future extensions of the instruction set.

4.4 Lifter Code Generation

The lifter is responsible for translating decoded instructions into a verification-friendly IR. While the decoder's task is to extract the instruction opcode and field values from machine code, the lifter takes these decoded fields and produces AIR code that explicitly models the instruction's semantics. In other words, the decoder outputs concrete information about the instruction, such as which operation it represents and its operand values, while the lifter generates a sequence of AIR statements that describe how the instruction affects the machine state. An example of the instruction semantics handled by the lifter is shown in Figure 2.5 for the `csel` instruction.

In a naive but fully functional approach, the lifter could transpile all instruction semantics directly into AIR code, representing every computation as part of the output IR. While correct, this strategy is inefficient, as it forces later verification and analysis tools to reason about logic that could have been resolved earlier. To improve efficiency, Airlift uses a hybrid approach. Any logic that can be determined during lifter runtime, meaning logic that does not depend on machine state such as register or memory contents, is eagerly evaluated and transpiled into executable code. Logic that does depend on machine state is transpiled into AIR generation code, which emits explicit AIR instructions for symbolic execution and verification. This selective strategy enables partial evaluation of the ASL semantics, reducing the size and complexity of the resulting IR while ensuring correctness. The hybrid transpilation model is described in more detail in Subsection 4.4.3.

4.4.1 Extension of AIR to Accommodate ASL Semantics

To support the semantics expressed in ASL, the AIR language was extended in several ways. First, AIR was augmented with the new type `Int`, which represents unbounded integers, along with basic arithmetic operations over this type. These include addition, subtraction, multiplication, and division. Additionally, conversion operations were introduced between `Int` and `FixedSizeInt`, allowing both types to interoperate as needed.

Second, support for arbitrary fixed-size bitvectors was introduced. Originally, AIR supported only a limited set of bit sizes: 8, 16, 32, 64, and 128 bits. To accommodate ASL's use of bitvectors of any size, the `FixedSizeInt` type was generalized to support arbitrary widths from 1 to 128 bits. This enhancement allows AIR to directly represent the width-precise semantics specified in the ASL source, without requiring further approximation or re-encoding.

```
(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x[shift+N-1:shift];
    carry_out = extended_x[shift-1];
    return (result, carry_out);
```

(a) Original `ashr` implementation.

```
(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    if shift >= N then
        if x[N-1] == '1' then
            result = Replicate('1', N);
            carry_out = '1';
        else
            result = Replicate('0', N);
            carry_out = '0';
    else
        result = x >> shift; // arithmetic shift
        carry_out = x[shift-1];
    return (result, carry_out);
```

(b) Overwritten `ashr` implementation.

Figure 4.1: Example replacement of a subroutine with dynamic typing.

4.4.2 Implementation Overwrites of ASL Subroutines

Some subroutines defined in the MRAS cannot be used directly in Airlift for two main reasons. First, some subroutines are not relevant in the context of TNJ. For example, exception-related subroutines are unnecessary because exception handling does not

affect correctness for this use case; these were replaced with stubs that do not modify architectural state. Second, AIR does not support dynamic typing, so all bit widths must be known at lifter generation time. However, some ASL subroutines rely on dynamic typing by computing bitvector lengths at runtime. This is incompatible with the requirements of Airlift.

To address these issues, we selectively overwrite or replace the problematic subroutines with hand-written implementations. For exception handling, we use no-op stubs. For subroutines that require dynamic typing, we provide new implementations that explicitly handle all relevant edge cases using fixed-width logic. This applies only to arithmetic shift right, logical shift left, and logical shift right. The replacements preserve the original semantics while ensuring compatibility with AIR.

Figure 4.1 shows the original and replacement versions of the `ASR_C` subroutine. The replacement version avoids dynamic sizing by covering the full range of shift values and using only fixed-width operations, which are fully compatible with AIR.

4.4.3 Hybrid Transpilation Approach

This subsection explains how Airlift distinguishes between instruction semantics that can be evaluated during lifter runtime and those that must be lifted into AIR. The lift transpiler uses this separation to perform partial evaluation wherever possible, while preserving correctness by delegating machine-state-dependent logic to AIR.

We begin by introducing a few terms used in this thesis to classify variables and control-flow scopes during transpilation:

- **Residual Variables**: Variables whose values depend on the machine state, such as register or memory contents. These variables must be lifted into AIR.
- **Static Variables**: Variables whose values can be fully resolved during lifter runtime. These can be evaluated directly and are not lifted.
- **Residual Scopes**: Control-flow scopes whose execution depends on machine state. Whether or not these scopes are executed cannot be determined during lifter runtime.
- **Static Scopes**: Control-flow scopes whose execution is fully determined by values available during lifter runtime. Note that static scopes may still contain residual variables. The distinction is based solely on whether the control-flow decision to enter the scope depends on machine state, not on the types of variables used within it.

Algorithm 1 Variable promotion rule for binary operations

```
function BinaryOp(∘, a, b)
    if IsResidualVar(a) or IsResidualVar(b) then
        if IsStaticVar(a) then
            a ← PromoteToResidualVar(a)
        end if
        if IsStaticVar(b) then
            b ← PromoteToResidualVar(b)
        end if
        return EmitAirOp(∘, a, b)
    else
        return a ∘ b                          ▷ Evaluate during lifter runtime
    end if
end function
```

Residual Variables

The definition of residual variables is straightforward, but we introduce a step-by-step approach for promotion in Algorithm 1 to clarify the decision process. Whenever a static variable is used in an operation with a residual variable, the result of that operation is also treated as residual. This is conceptually similar to constant propagation in compilers, where expressions that depend solely on constants are evaluated at compile time. In Airlift, partial evaluation is performed during lifter execution for any logic that depends only on known values, leaving only machine-dependent logic to be lifted.

Figure 4.2 shows the implementation of the `ConditionHolds()` function, which is invoked by the instruction semantics of `csel`. The variable `result` in this example illustrates the concept of variable promotion. As shown in Figure 2.5, the argument `condition` passed to this function is a static variable because its value is not derived from machine state. Therefore, the parameter `cond` in `ConditionHolds()` is also a static variable.

Within the function body, if `cond[3:1]` equals `'111'`, then `result` is assigned the constant value `TRUE`, and thus remains a static variable. However, for all other cases, `result` is defined in terms of the `PSTATE` register, which is part of the architectural state. As a result, `result` becomes a residual variable, and any operation that uses it is also treated as residual. Furthermore, since this variable is returned from the function, any use of its return value outside the function will likewise be marked as residual.

```
boolean ConditionHolds(bits(4) cond)
    case cond[3:1] of
        when '000' result = (PSTATE.Z == '1');
        when '001' result = (PSTATE.C == '1');
        when '010' result = (PSTATE.N == '1');
        when '011' result = (PSTATE.V == '1');
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');
        when '101' result = (PSTATE.N == PSTATE.V);
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');
        when '111' result = TRUE;

    if cond[0] == '1' && cond != '1111' then
        result = !result;

    return result;
```

Figure 4.2: Example static and residual variables.

Residual Scopes

Airlift handles most statements in residual scopes and static scopes in a uniform manner. However, two types of statements require special treatment when they occur within a residual scope: variable assignments and return statements.

Figure 4.3a revisits the instruction semantics of the `csel` instruction, which illustrates why variable assignments in residual scopes must be translated to AIR rather than executed directly. As discussed earlier, the return value of the `ConditionHolds()` function is likely a residual variable because it depends on machine state. Assuming this is the case, the predicate of the outer `if`-statement becomes residual, which in turn means that both the `then` and `else` blocks are residual scopes.

Since the lifter cannot determine at runtime which branch will be taken, it must handle both. Executing both branches would result in assignments to the `result` variable overwriting each other, which compromises correctness. Alternatively, lifting the entire conditional into AIR avoids this issue but sacrifices the benefits of partial evaluation. For example, the subconditions `else_inv` and `else_inc` are static variables and could be evaluated during lifter execution, effectively collapsing the branches of the inner `if`-statements. Promoting them unnecessarily would increase the complexity of the generated AIR.

To handle such cases, we present the rules for variable assignments in residual and static scopes as shown in Algorithm 2. In static scopes, assigning and reading

Algorithm 2 Handling variable assignments in residual scope

```
procedure AssignVarInStaticScope(var_name, value)
    Var(var_name) ← value                          ▷ Assign to the lifter runtime variable
end procedure

function GetVarInStaticScope(var_name) return Var(var_name)
end function

procedure AssignVarInResidualScope(var_name, value, assigns)
    if IsStaticVar(value) then
        value ← PromoteToResidualVar(value)
    end if
    assigns[var_name] ← value
end procedure

function GetVarInResidualScope(var_name, assigns_stack)
    for all assigns in assigns_stack (innermost to outermost) do
        if var_name ∈ assigns then
            return assigns[var_name]
        end if
    end for
    return Var(var_name)
end function

procedure ExitResidualScope(branches)
    assigned ← UnionOfAssignedVarsAcrossBranches(branches)
    declared ← VarsDeclaredInBranches(branches)
    params ← assigned \ declared
    OrderConsistentlyAcrossBranches(params)
    EmitBlockParams(params)
end procedure

procedure EnterNextBlock(params, outer_scope_residual, assigns)
    if Not(outer_scope_residual) then
        for all (var_name, value) ∈ params do
            Var(var_name) ← value
        end for
    else
        for all (var_name, value) ∈ params do
            assigns[var_name] ← value
        end for
    end if
end procedure
```

variables is straightforward: the lifter directly writes to and reads from runtime variables during execution. In contrast, assignments in residual scopes are emulated rather than executed immediately. Here, "emulated" means the lifter keeps track of variable-value pairs without directly modifying the variables during lifter runtime. The actual assignment only materializes in the generated AIR, where assignment values from different branches can be merged as control flow converges.

```
__execute
    bits(datasize) result;
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];
    if ConditionHolds(condition) then
        result = operand1;
    else
        result = operand2;
        if else_inv then result = NOT(result);
        if else_inc then result = result + 1;
    X[d] = result;
```

(a) ASL instruction semantics for `csel`.

```
entry:
    v0 = i64.read_reg "x1"
    v1 = i64.read_reg "x2"
    ...
    jumpif v3, addr_0_block_0, addr_0_block_1
addr_0_block_0:
    jump addr_0_block_2(v0)
addr_0_block_1:
    jump addr_0_block_2(v1)
addr_0_block_2(v4: i64):
    write_reg.i64 v4, "x0"
```

(b) Generated AIR code for `csel x0, x1, x2, <cond>`.

Figure 4.3: Example variable assignments in residual scope.

To emulate variable assignments, Airlift maintains an `assigns` map for each residual scope. When a variable is assigned a new value inside a residual scope, the assignment

is recorded in the corresponding `assigns` map instead of being applied to the variable itself. When a variable is read from within a residual scope, the system first consults the `assigns` maps, beginning with the innermost scope and proceeding outward. If no assignment is found in any scope, the value of the variable from the runtime environment is used.

At the end of a residual scope, all variables that were assigned new values across any control-flow branch are passed to the next basic block as block parameters. At the beginning of this block, the way these parameters are handled depends on the nature of the surrounding scope. If the outer scope is a static scope, the block parameters are directly assigned back to the corresponding static variables. If the outer scope is also residual, the block parameters are instead recorded in that scope's `assigns` map as new emulated assignments.

Figure 4.3b shows the simplest example of a residual AIR code produced from emulated variable assignments in a residual scope.

```
integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x[i] == '1' then return i;
    return -1;
```

Figure 4.4: Example return statement in a residual scope.

The only other type of statement that requires differentiated handling between residual and static scopes is the return statement. Figure 4.4 shows an example function, `HighestSetBit()`, in which a return statement is located inside a conditional that may depend on machine state.

If the parameter `x` is a residual variable, then the condition `x[i] == '1'` becomes machine-state-dependent. As a result, the `then` block containing the return statement must be treated as a residual scope. Furthermore, since the outer `for`-loop iterates over `N` different indices, each with the potential to take a distinct return path, the lifter must be able to handle multiple possible return values and control-flow exits.

To manage this complexity, the lifter must generate AIR that converges all these return paths into a single return point. This return point receives the potential return values as block parameters and ensures that only the correct value, based on the machine state, is ultimately used.

Algorithm 3 presents the logic for handling return statements that arise in residual scopes. When a return is encountered in such a scope, it signifies that the decision to return is not known at lifter runtime. As a result, a dedicated return block is created in AIR, and all subsequent return statements, whether they occur in static scopes

or in residual scopes, must also jump to this block. If no residual return has been encountered, return statements in static scopes are handled normally by returning directly from the lifter runtime function.

Algorithm 3 Handling return statements in residual scope

```
procedure HandleReturnInResidualScope(return_value, has_return_block)
    if Not(has_return_block) then
        CreateReturnBlock()
        has_return_block ← True
    end if
    if IsStaticVar(return_value) then
        return_value ← PromoteToResidualVar(return_value)
    end if
    EmitJumpToReturnBlock(return_value)
end procedure

procedure HandleReturnInStaticScope(return_value, has_return_block)
    if Not(has_return_block) then
        Return(return_value)              ▷ Return from the lifter runtime subroutine
    else
        if IsStaticVar(return_value) then
            return_value ← PromoteToResidualVar(return_value)
        end if
        EmitJumpToReturnBlock(return_value)
    end if
end procedure

procedure ProcessReturnBlock(return_params)
    merged_return ← MergeReturnValues(return_params)
    Return(merged_return)                 ▷ Return from the lifter runtime subroutine
end procedure
```

To ensure consistent control-flow handling, Airlift creates a dedicated AIR block corresponding to a particular function the first time a return is encountered within a residual scope in that function. This block serves as a target for all subsequent return statements. Once this return block has been created, every following return, regardless of the type of scope in which it appears, jumps to this block instead of returning directly. In cases where a return statement is located in a static scope and no residual return has been encountered earlier, the lifter simply returns the value in the standard way.

At the end of the function, after all appropriate return statements have jumped to

the dedicated return block, the lifter merges all block parameters (which represent the possible return values across different control-flow paths). The merged residual variable is then returned from the lifter runtime function.

Variable assignments and return statements are the only kinds of statements that require distinct handling between static and residual scopes. All other statements, including subroutine calls, are treated uniformly. This uniformity is possible because any side effects that occur within subroutines, such as register writes, memory writes, assertions, or exceptions, are always lifted into AIR and never executed during lifter runtime. As a result, subroutines behave as if they are free of side effects from the perspective of the lifter and can be handled in the same manner regardless of the surrounding scope.

4.4.4 Basic Block Structuring

Airlift uses a two-pass strategy to resolve block labels during the lifting process. This approach is inspired by the manually written lifter, which used a similar analysis for instruction-level control flow [10]. However, while the manual lifter could hardcode all intra-instruction jump labels in advance, Airlift generalizes this logic to handle not only inter-instruction jumps but also jumps within instruction semantics, which are discovered dynamically during lifting.

During lifting, jumps can target both forward and backward locations. For backward jumps, the lifter may encounter a jump to a location that has not yet been processed, making it unclear whether a block label is needed there. A naive solution would be to insert a block label before every instruction, but this would clutter the generated AIR with unnecessary blocks and degrade analysis performance.

To avoid this, Airlift performs a preliminary pass through the instruction and subroutine semantics to collect all jump targets. These include:

- Instruction entry points referenced by inter-instruction jumps
- Branch targets within the instruction semantics, such as the start of if branches or loop bodies

In the second pass, the lifter generates AIR using only the necessary block labels. This includes both instruction-level jump targets and dynamically determined branch targets within the instruction semantics. The latter are ordered to match the evaluation order of the original ASL code, ensuring that control flow proceeds naturally and avoids jumping into the middle of unevaluated logic.

For example:

- For an `if` statement, the generated AIR consists of several blocks arranged in order: first, a basic block (BB) evaluating the `if` condition; next, a BB for the `then` branch; then, if present, a BB for the `else` branch; and finally, a convergence block where control flow from both branches rejoins. This order ensures that control flow always enters a branch only after the condition has been evaluated, and prevents jumps into the middle of a branch before the condition.
- For return statements inside residual scopes, a unified return block is placed at the end of the function's generated AIR. All return paths jump to this BB, ensuring convergence and preventing mid-function termination from corrupting control flow.

These ordering choices ensure that control-flow paths reconverge at the appropriate point in the semantics, so that subsequent AIR generation can proceed naturally to the next instruction without requiring backward jumps or reordering.

5 Implementation

5.1 Project Structure

The project relies on a small number of external repositories. The official MRAS release from Arm includes instruction encodings, decode trees, and ASL pseudocode embedded within XML metadata. However, this metadata is not directly usable due to its fragmented and deeply nested structure. The `mra_tools` project extracts and restructures this data into standalone ASL files, such as `arch_decode.asl`, `arch_instrs.asl`, and `arch.asl`, which serve as clean entry points for external analysis and code generation.

We also integrate ASLi, from which only the lexer, parser, and typechecker components are used, as discussed in Subsection 2.3.1.

There is no other significant dependency besides Rust's standard metaprogramming libraries, `syn` and `quote`.

The project itself is divided into two main library crates:

- `codegen`: Responsible for analyzing the MRAS AST and generating the architecture-dependent parts of both the decoder and the lifter. This crate is invoked only at build time to generate Rust source code for the `lifter` crate.
- `lifter`: Contains a minimal amount of manually written skeleton code; the rest of the crate is generated by the `codegen` crate. The `lifter` code is invoked at lift time, i.e., when translating input binaries.

This separation reflects Airlift's overall workflow: `codegen` and its dependencies run once at build time to generate the architecture-specific lifter code, while the resulting `lifter` code is invoked for each lift task at runtime. This division ensures that architecture analysis and code generation are decoupled from the performance-critical lifting process.

The manually written and automatically generated parts of the `lifter` crate are tightly coupled and cannot function independently. To maintain this integration, code generation is triggered from within `build.rs`, ensuring that generated code is updated automatically whenever relevant source files are modified.

The precise delegation of responsibilities between `codegen` and `lifter` is discussed in Section 5.4.

5.2 ASLi Integration and Interface Boundary

ASLi is implemented in OCaml, a statically typed functional programming language often used in compiler development. OCaml's strong type system, pattern matching, and functional design make it effective for implementing language semantics and transformations. Its ecosystem also includes tools like Ott, which simplifies the definition and implementation of language grammars.

Ott is a framework for specifying grammars, type systems, and pretty-printers. In ASLi, Ott is used to define the grammar and abstract syntax of ASL, generating the lexer, parser, and typechecker automatically.

To integrate ASLi with our Rust-based pipeline, we forked the ASLi repository and introduced the following changes:

- Added a script to run the parser and typechecker in isolation, without executing AST traversal or evaluation.
- Defined a custom JSON serialization format using Ott's pretty-print homomorphisms.
- Fixed minor versioning issues that prevented ASLi from building or running in our environment.

5.2.1 Parser-Only Invocation

ASLi provides scripts for running the lexer and for full interpretation (including evaluation of the AST). We required a way to extract the typed AST without triggering evaluation or traversal logic. To support this, we added a script that invokes only the parser and the typechecker. This enabled reuse of ASLi's robust parsing infrastructure while keeping downstream logic entirely in Rust.

5.2.2 Interface Design and JSON Serialization

We chose to use JSON serialization as the interface between ASLi and Airlift. Although it would have been possible to reuse traversal logic from ASLi, doing so would require implementing our lifting logic in OCaml. We initially avoided this approach due to the availability of existing Rust libraries for TNJ and AIR code generation, as well as the opportunity cost of learning OCaml and Ott.

In retrospect, handling AST traversal ourselves provided valuable flexibility. It allowed us to explicitly manage context-sensitive information such as scoped symbol tables and function-level environments, which would have been difficult to retrofit into ASLi's evaluation-oriented architecture.

Ott's pretty-print homomorphisms (pp homs) were originally intended for generating human-readable representations of AST nodes, typically for documentation, debugging, or producing concrete syntax from abstract terms. In our case, however, pp homs worked perfectly well as a tool for JSON serialization. This approach proved more flexible than alternatives like `yojson`, as it allowed us to precisely control the output format. For example, we were able to inject a custom `serde_tag` field into each node to facilitate seamless deserialization into Rust structs using `serde_json`.

5.3 AST Structure and Serialization Pipeline

Each AST node is represented as a generic Rust struct with three main fields: `node_type`, `node_subtype`, and `node_data`. The `node_data` field is an enum whose variant depends on the values of `node_type` and `node_subtype`. This design was chosen instead of defining separate structs for each distinct AST node type, in order to maximize flexibility during development.

In most cases, nodes of different types are handled distinctly. However, certain operations, such as register read and write handling, require traversing and processing nodes without prior knowledge of their exact type. The unified structure simplifies this kind of generic handling. Additionally, it improves code reusability by accommodating alternate AST formats without rewriting the surrounding logic.

This design sacrifices some type safety, since invalid combinations of `node_type` and `node_subtype` cannot be statically ruled out. To compensate for this, explicit checks are performed wherever nodes are matched or handled to ensure the validity of the type-subtype combination.

To deserialize the JSON representation of the AST, we use Rust's `serde_json` library. The `serde` ecosystem supports tagged enums, which allow an enum to be deserialized by matching a tag field in the serialized JSON object. For this purpose, each `node_data` object includes a `serde_tag` field that specifies the intended variant of the enum.

This approach allows `serde` to automatically deserialize the `node_data` field into its corresponding Rust struct based on the value of the `serde_tag`, without requiring manual dispatch logic. This mechanism provides a robust and extensible interface between the JSON-serialized output of ASLi and the statically typed structure of the Airlift frontend.

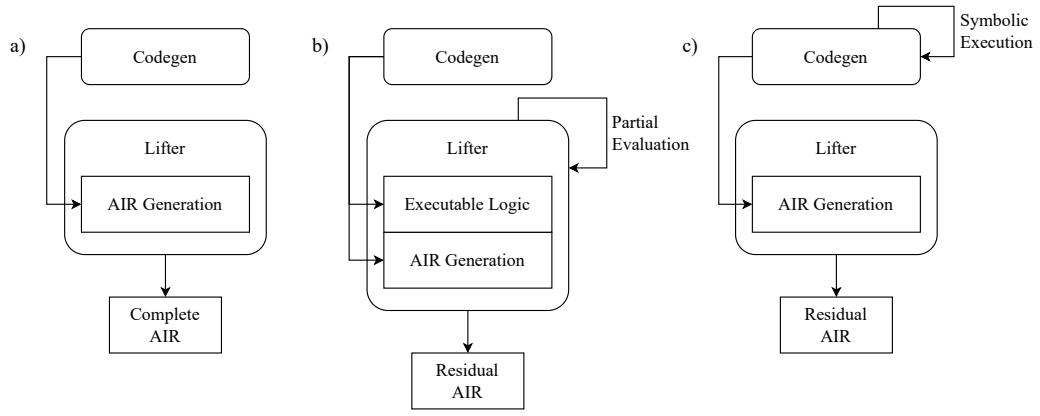


Figure 5.1: Partial evaluation and alternatives.

5.4 Runtime Placement of Partial Evaluation

Instruction semantics from the MRAS can be interpreted or transformed into AIR at different stages in the pipeline: during code generation, during lifter runtime, or through direct translation into AIR without any evaluation. Figure 5.1 illustrates these three strategies.

- **Naive Translation:** In this approach, shown in Figure 5.1a, the full instruction semantics are transpiled into AIR without any attempt to resolve expressions based on known values. This leads to unnecessarily large AIR programs that include logic which could have been resolved earlier, had evaluation occurred. While simple to implement, this strategy produces verbose and inefficient output.
- **Partial Evaluation at Lifter Runtime:** This is the strategy used by Airlift, as shown in Figure 5.1b. During lifter execution, all instruction field values are available since they are extracted from the instruction encoding. The lifter evaluates all expressions that can be resolved using these values and emits AIR for any remaining computations. The result is residual AIR, which is smaller and more efficient than the naive approach. This strategy balances implementation effort and output performance.
- **Symbolic Execution During Code Generation:** In this alternative, shown in Figure 5.1c, the code generator symbolically evaluates instruction semantics by treating field values as symbolic inputs. The resulting AIR is residual in the same sense as with partial evaluation at lifter runtime. The key difference is that all

simplification occurs during code generation. While this eliminates the need for partial evaluation during lifter execution and produces the same output, it requires exploring all symbolic execution paths, which makes code generation significantly more complex and time-consuming.

Although symbolic execution could have reduced lifter complexity and runtime cost, we chose partial evaluation at lifter runtime for this project. Our goal was to keep the code generator simple while ensuring correctness and reasonably efficient AIR output. This also allowed us to isolate the logic responsible for evaluation and maintain modularity between the lifter and the generated code.

To support partial evaluation during lifter execution, code generation performs several preparatory steps. For each variable operation or scope creation, the generator emits logic that handles both promoted and unpromoted cases, depending on values determined at runtime. Static analyses identify variable reassignments in promoted scopes and detect early return paths, which inform the shape of block parameters and return blocks. These decisions are guided by symbol tables, scope contexts, and subroutine contexts tracked during the recursive AST traversal.

5.5 Subroutine Transpilation

As briefly mentioned in Subsection 4.3.1, subroutine transpilation is performed on demand, unlike constants, enumerations, and records, which are transpiled in their entirety at the beginning of the pipeline. This on-demand approach enables clean and incremental development of the lifter and avoids generating code for subroutines that are not used by any supported instruction.

There are two categories of subroutines in ASL: those whose implementation is provided by ASL itself, and those whose semantics are left unspecified and must be implemented externally. Airlift maintains a separation between automatically translated subroutines and manually implemented ones. Transpiled subroutines are generated into a dedicated space, while manually written subroutines reside in a separate file that can override or complement the generated logic.

Automatically transpiled subroutines are allowed to call manually written ones. To accommodate this, the build script (`build.rs`) is rerun whenever the manually written subroutine set changes, such as when a subroutine is added or removed. This ensures consistency between generated and handwritten code. Additionally, if a subroutine is already defined manually, the code generator will skip its translation and instead use the custom implementation. This provides a convenient mechanism to override or specialize the behavior of ASL-defined subroutines when necessary.

```
bits(M*N) replicate_bits(
    bits(M) x,
    integer N
);
```

(a) Original ASL code.

```
fn replicate_bits(
    x: Bits,
    n: Integer,
    m: Integer,
    _n: Integer
) -> Result<Bits, Error>;
```

(b) Transpiled Rust code.

Figure 5.2: Implicit parameters example.

Many ASL subroutines include implicit parameters, such as bit widths of arguments or return types, which are not explicitly declared in the subroutine signature. These values are injected into the AST during parsing by ASLi. Since they are frequently used within subroutine bodies, Airlift translates them into explicit parameters in the generated Rust code, placed alongside the original subroutine arguments.

The ordering of these implicit parameters is inferred from examples found in the AST: first, any bit widths of the explicit parameters are extracted from left to right, followed by the bit width of the return type. In some cases, an implicit parameter shares the same name as an explicit parameter. These represent the same value semantically, but since both appear in the AST node for the function call, both are included in the translated Rust signature for simplicity. When this occurs, one of the parameters is renamed to avoid duplication, which is safe to do since both parameters represent the same value and only one is actually used in practice. An example of this pattern is shown in Figure 5.2.

5.6 Representation of ASL Data Types in AIR

ASL defines a range of high-level data types that must be lowered into the more minimal set of types supported by AIR. The translation preserves semantics while aligning with the restricted and verifiable type system of AIR. Each ASL type is mapped as follows:

- **Integers** are represented as `Int`.
- **Booleans** are represented as `Bool`.
- **Bits** are represented as `FixedSizeInt`.
- **Enumerations** are also represented as `FixedSizeInt`. ASL enums are only used in comparison operations, and both the ASLi parser and the lifter enforce that values from different enum types are never compared. This makes it sufficient to encode each enum variant as an integer index, where the nth variant of an enum type is converted into a `FixedSizeInt` with value n.
- **Records** are translated by recursively flattening their fields. When a record is passed to a BB, this can happen either because one or more of its fields are updated within a promoted scope and the updated record must be propagated to a converging block, or because the record is returned by an early return statement. In these cases, the record is decomposed into a list of primitive values such as integers, booleans, bits, or enums. These are passed as separate block parameters in a fixed, deterministic order. At the start of the destination block, the record is reconstructed using the received block parameters.

  For example, consider the following nested record definition:

  ```
  type AddressDescriptor is (
      FaultRecord fault,
      MemoryAttributes memattrs,
      FullAddress paddress,
      bits(64) vaddress
  )
  ```

  In this case, flattening proceeds by visiting the fields of the record in the order they are declared. The primitive fields of `fault` are extracted first, followed by those of `memattrs`, then `paddress`, and finally the `vaddress` field. All nested records are recursively unpacked in the same way. The resulting list of primitive values is passed to the destination block as individual block parameters. Because the traversal and reconstruction order is consistent, the original `AddressDescriptor` can be accurately reassembled at the entry of the receiving block.

5.7 Uncertainty with ASL Decoding Match-Case Logic

ASL decoding logic is structured entirely as nested match-case expressions that define the dispatch of instruction encodings. These expressions map tuples of instruction field

values to the corresponding semantic implementation of each instruction. Each case specifies a pattern, which may include exact bit values or wildcards, and an associated instruction name.

```
case (op1, op2, Rt) of
    when (_, _, !'11111') => __UNALLOCATED
    when (_, _, '11111')
        => __encoding aarch64_system_register_cpsr
    when ('000', '000', '11111')
        => __encoding aarch64_integer_flags_cfinv
    when ('000', '001', '11111')
        => __encoding aarch64_integer_flags_xaflag
    when ('000', '010', '11111')
        => __encoding aarch64_integer_flags_axflag
    ...
case (CRm, op2) of
    when (_, _) => __encoding aarch64_system_hints
    when ('0000', '000') => __encoding aarch64_system_hints
    when ('0000', '001') => __encoding aarch64_system_hints
    when ('0000', '010') => __encoding aarch64_system_hints
    ...
```

Figure 5.3: `arch_decode.asl` match-case branch ordering examples.

As shown in Figure 5.3, broader and more general patterns, such as those using wildcards, frequently appear before more specific ones. In a conventional top-down evaluation model, this ordering would cause general patterns to override the specific cases listed afterward, making the latter unreachable. However, when implementing the decoder, we found that evaluating match-case expressions in reverse order produced the expected behavior, with specific branches matched correctly. This suggests that ASL decoding logic is designed with bottom-up pattern matching in mind.

This bottom-up interpretation is consistent with all decoding patterns observed so far in `arch_decode.asl`. Accordingly, the current implementation assumes reverse-order branch checking during decoding. If future examples are encountered where a specific case appears above a broader one and still takes precedence, this assumption would need to be revised. In such a case, a mechanism for partially ordering branches based on their pattern specificity would be required to ensure correct decoding behavior.

5.8 Transpilation Challenges from Rust's Type Safety Features

5.8.1 Exhaustiveness in Match-Case Logic

ASL match-case constructs may or may not be exhaustive, and the specification does not explicitly differentiate between the two. Since Rust requires all `match` expressions to be exhaustive, we instead translate ASL match-case expressions into chains of `if` and `else if` statements.

```
ShiftType decode_shift(Bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;
```

(a) Original ASL code.

```
fn decode_shift(op: Bits) -> Result<ShiftType, Error>
    {
    if (op.match_with_pattern("00")) {
        return Ok(ShiftType::ShiftType_LSL);
    } else if (op.match_with_pattern("01")) {
        return Ok(ShiftType::ShiftType_LSR);
    } else if (op.match_with_pattern("10")) {
        return Ok(ShiftType::ShiftType_ASR);
    } else if (op.match_with_pattern("11")) {
        return Ok(ShiftType::ShiftType_ROR);
    }
    unreachable!("decode_shift");
}
```

(b) Transpiled Rust code.

Figure 5.4: Exhaustive match-case transpilation example.

In cases where the original ASL logic is indeed exhaustive, the resulting Rust code is not statically recognized as such by the compiler. To indicate that the code path should never be reached, we append a trailing `unreachable!()` macro at the end of each subroutine. This serves both as a compiler hint and as explicit documentation for

unreachable code paths; it will panic if triggered at runtime, but such a path should be impossible under correct semantics.

Additionally, for ASL variable declarations that are syntactically uninitialized but are guaranteed to be assigned a value through exhaustive match-case logic, we initialize the corresponding Rust variables with default values. This avoids compiler errors related to potentially uninitialized use, even though the logic is sound in the original ASL.

An example is shown in Figure 5.4, where an ASL function using exhaustive match-case dispatch is transpiled into a sequence of conditional checks with a trailing `unreachable!` statement to satisfy Rust's exhaustiveness constraints.

5.8.2 Nested Mutable Borrow and InstructionBuilder Access

One of the central components in AIR generation is the `InstructionBuilder` object, which is passed as a mutable reference to all subroutines that emit AIR code. However, nested calls involving mutable access to the same `InstructionBuilder` are forbidden by Rust's borrow checker. This restriction poses a challenge during the generation of deeply nested AIR expressions, which are common in transpiled ASL arithmetic and logic constructs.

To resolve this, any function call that accepts a mutable reference to the builder object is refactored to evaluate its arguments first and store them in intermediate variables. These intermediate values are then passed as arguments to the subroutine. This not only satisfies the borrow checker but also avoids the need for unsafe code. To reduce naming conflicts, Rust's variable shadowing is used so that intermediate results can reuse the original variable names where appropriate.

Figure 5.5 shows how a nested function call is rewritten into sequential variable bindings to avoid illegal nested mutable borrows.

5.9 Registers and Flags Configuration

The AIR model explicitly includes only a minimal subset of architectural state: general-purpose registers `X[0]` $\cdots$ `X[30]`, the stack pointer `SP`, and the condition flags `PSTATE.N`, `PSTATE.Z`, `PSTATE.C`, and `PSTATE.V`. These are the only registers and flags that are accessed or modified directly by the instruction semantics for the subset of AArch64 instructions supported in this project.

The program counter (PC) is not included in AIR as an explicit register. Instead, its value is managed separately in the lifter's runtime memory. This is because the lifter is responsible for resolving block labels and tracking control flow, as discussed in Subsection 4.4.4, and because TNJ does not require the PC to be represented directly in the AIR output.

```
let unsigned_sum: TypedValue = add_int(
    builder, // outer mutable borrow
    add_int(
        builder, // nested mutable borrow 1
        uint(
            builder, // nested mutable borrow 2
            x, Integer::from(n.clone())
        )?,
        uint(
            builder, // nested mutable borrow 3
            y, Integer::from(n.clone())
        )?
    )?,
    uint(
        builder, // nested mutable borrow 4
        carry_in, Integer::from(1)
    )?
)?;
```

(a) Nested mutable borrow.

```
let unsigned_sum: TypedValue = {
    let arg_0 = {
        let arg_0 = uint(builder, x, Integer::from(n.clone()))?;
        let arg_1 = uint(builder, y, Integer::from(n.clone()))?;
        add_int(builder, arg_0, arg_1)?
    }
    let arg_1 = uint(builder, carry_in, Integer::from(1))?;
    add_int(builder, arg_0, arg_1)?
};
```

(b) Nested mutable borrow fix.

Figure 5.5: Nested mutable borrow example.

For all other architectural state, such as system registers, control flags, and exception-related fields, we assume that programs run in user-level privilege mode, with most optional architectural features considered inactive.

The one exception to this is the pointer authentication code (PAC) extension. We assume this hardware extension is enabled, as it is featured in virtually all Apple CPUs for more than five years and is available in many other manufacturers' cores. Certain instructions in the Sightglass benchmark suite require PAC support; if it were disabled, these instructions would fail to decode or behave incorrectly during execution.

5.10 FP/SIMD and Other Unsupported Instructions

Airlift fully supports the lifting of integer arithmetic, branch, and memory instructions using their complete semantics as defined in the MRAS. These categories constitute the majority of instructions in the AArch64 base ISA, and are sufficient for the purposes of TNJ, which focuses primarily on memory safety.

A large portion of the remaining, unsupported instructions consists of FP and SIMD operations. These instructions fall under the Neon extension of the AArch64 ISA, and are included in the MRAS. However, they operate on vector registers and are generally orthogonal to memory semantics, making them largely irrelevant in the TNJ context.

To ensure that Airlift can still lift complete real-world binaries, we implement a robust fallback mechanism: every instruction, whether supported or not, can be decoded. For unsupported instructions, Airlift inserts fallback AIR code that preserves memory soundness. Specifically, whenever an unsupported instruction writes to a general-purpose register (`X[0]` $\cdots$ `X[30]`), an `opaque` value is assigned to that register. This conservative approximation maintains correctness guarantees without requiring full semantic coverage.

6 Evaluation

This chapter evaluates the correctness, robustness, and structural efficiency of Airlift, a binary lifter that translates AArch64 instructions into AIR. Airlift was not designed with performance as a primary goal; instead, it prioritizes semantic correctness by adhering closely to the MRAS and supports a generalized transpilation pipeline. As a result, performance comparable to a manually written lifter is neither expected nor pursued. However, evaluating its output against the hand-written baseline allows us to quantify the inefficiencies introduced by this design and to identify specific bottlenecks that may be optimized in future work. The evaluation is guided by the following research questions:

- **RQ1.** Can Airlift lift real-world benchmark programs completely, and how robust are its decoding and lifting pipeline?
- **RQ2.** Does Airlift's performance on real-world benchmark programs suggest it is suitable for practical use?
- **RQ3.** What are the lifting time, AIR instruction count, and block count characteristics of Airlift's output across supported instruction categories (integer arithmetic, branches, memory)?
- **RQ4.** Which instruction-level patterns lead to the greatest structural inefficiencies in Airlift's output, and what qualitative insights can be drawn to guide future improvements?

6.1 Practical Usability

This section evaluates Airlift's practical usability in two dimensions: its ability to lift real-world binaries completely and correctly (RQ1), and its suitability for use in practice from a performance standpoint (RQ2). Airlift is not solely a research prototype; it also serves a practical purpose by enabling TNJ to verify memory safety on real-world binaries. To be viable in this context, the lifter must decode and lift code generated in actual environments without crashing, producing unsound output, or requiring impractical runtimes.

6.1.1 Decoding Coverage and Lifting Robustness (RQ1)

A key requirement for practical usability is robustness. As discussed in Section 5.10, Airlift does not fully lift the instruction semantics of AArch64 instructions that are not relevant to memory safety, such as FP and SIMD instructions. These are instead treated as unsupported: the lifting logic conservatively approximates their behavior by assigning `opaque` values to any general-purpose registers (`X[0]` $\cdots$ `X[30]`) they write. This fallback mechanism prevents unsafe behavior while preserving correctness for memory safety verification.

We evaluated Airlift using the `.text` segments of 114 applications from the Sightglass benchmark suite, compiled from WebAssembly (WASM) to AArch64 using Cranelift with `O2` optimization. We used Cranelift's default settings for AArch64 MacOS.

All lifting was performed on a machine with the following specifications:

- CPU: AMD EPYC 7713P 64-Core Processor
- RAM: 991 GiB
- OS: NixOS 24.11

All binaries were successfully decoded and lifted by Airlift, demonstrating complete decoding coverage and robust lifting behavior. By comparison, the manually written lifter was only tested on 31 applications, and it failed on 21 of them. These failures were caused by decoding errors in the `yaxpeax-arch` crate, which does not support instructions from the AArch64 ISA's Scalable Vector Extension (SVE) and Scalable Matrix Extension (SME) extensions. This result highlights a key advantage of the MRAS-based approach: decoding coverage is guaranteed by adherence to the specification, and the flexibility of Airlift's generalized transpilation pipeline allows arbitrary fallback instructions to be emitted in a low-effort, error-resistant way.

Answer to RQ1. Airlift achieved full decoding coverage and successfully lifted all 114 WASM applications in the Sightglass benchmark suite, demonstrating robustness and specification-compliance in both decoding and lifting stages.

6.1.2 Performance Bottleneck in Practical Use (RQ2)

Although Airlift successfully lifted all 114 binaries in the Sightglass suite, its performance remains too slow for most practical use cases. It was 1111 times slower than `llvm-objdump` on median, making it unsuitable for latency-sensitive workflows or large-scale batch processing in its current form.

`perf` analysis revealed that memory allocation and deallocation were the primary bottlenecks, consuming over 20% of total CPU time. This was largely due to the large volume of lifter code transpiled from the MRAS, which includes promotion logic that handles both static and residual semantics, as explained in Subsection 4.4.3. To avoid stack overflows, stack usage had to be minimized, and many data structures were moved to the heap. We expect that this bottleneck could be alleviated with more efficient memory management.

In addition, the emulation of variable assignments, as discussed in Subsection 4.4.3, frequently involves adding to and removing from heap-stored data structures. Delegating this responsibility to the code generation stage may reduce memory pressure and improve overall performance.

Further analysis of output structure and lifting cost is provided in Section 6.2, where Airlift is compared against a manually written baseline lifter that generates equivalent AIR.

Answer to RQ2. Airlift is 1111 times slower than `llvm-objdump` on median, making it unsuitable for performance-critical use in its current form. Profiling identified memory allocation and deallocation as the main bottlenecks, suggesting that better memory management could significantly reduce runtime.

6.2 Performance and Structural Inefficiencies

This section investigates Airlift's efficiency and sources of structural overhead. While the previous chapter evaluated its correctness and viability on real-world programs, here we quantify how much lifting cost is introduced across instruction categories (RQ3) and identify the instruction-level patterns that cause these inefficiencies (RQ4).

6.2.1 Lifting Output Metrics (RQ3)

To evaluate Airlift's efficiency, we measured lifting time, AIR instruction count, and block count across the three supported instruction categories: integer arithmetic, branch, and memory operations.

All benchmarks were conducted by running both Airlift and a manually written baseline lifter on individual instructions, five times each, and averaging the results. For instructions with multiple semantic blocks in the MRAS, such as `add` and `sub`, we included all relevant variants (`add_sub_carry`, `add_sub_shiftedreg`, and `add_sub_extendedreg`) and computed the average over them.

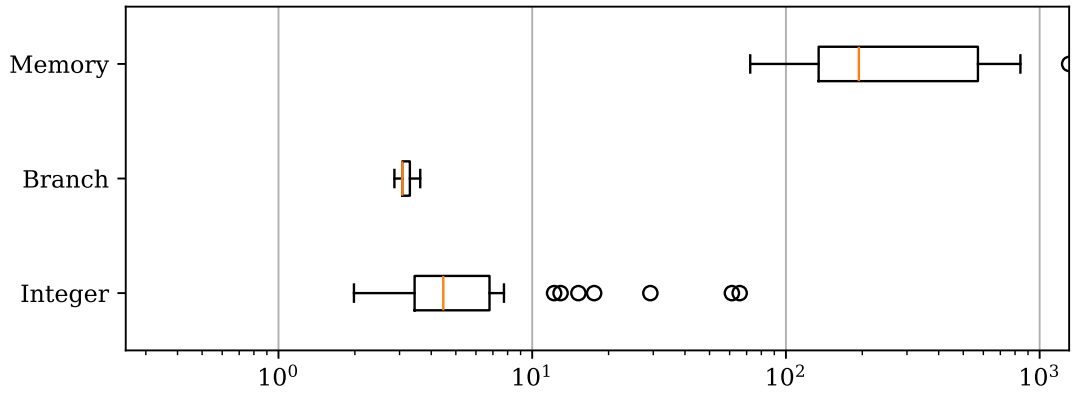
Both lifters were executed on the same machine:

- CPU: 12th Gen Intel i7-1255U (12) @ 4.70GHz
- RAM: 15 GiB
- OS: Arch Linux

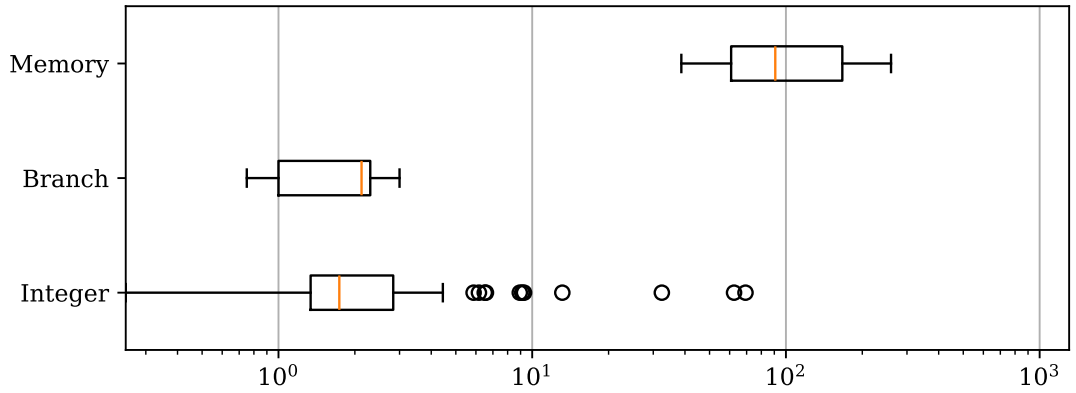
Figure 6.1 presents the comparative results. On a logarithmic scale, the lifting time (6.1a), instruction count (6.1b), and block count (6.1c) are plotted as the ratio of Airlift to the manual baseline across instruction categories.

Airlift performs significantly worse for memory operations in all three metrics. However, this is not a flaw in the system. The inefficiency results from Airlift's complete adherence to architectural semantics, which are largely skipped by the manual baseline. While the manually written lifter simply loads and stores values, Airlift handles the full spectrum of memory-related semantics, including:

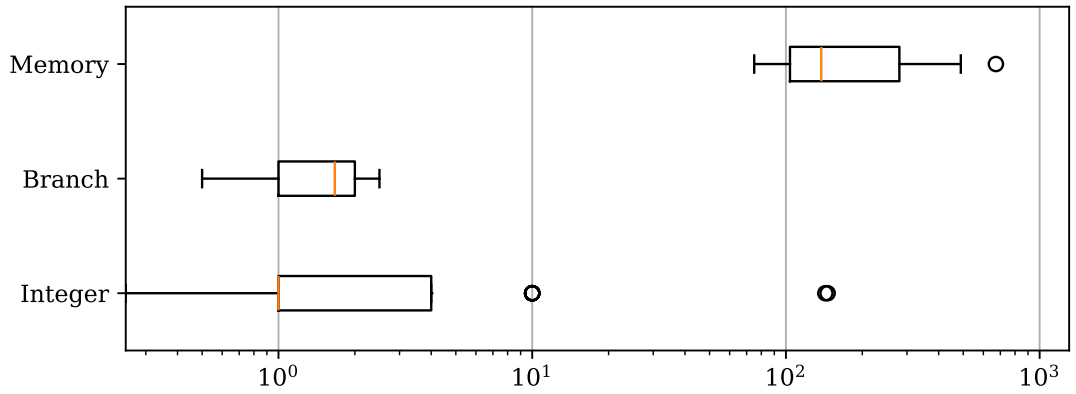
- **Memory Ordering Semantics**: Memory accesses are modeled to follow architectural ordering guarantees, ensuring correct interaction with concurrent and atomic operations.
- **Alignment Enforcement**: Proper alignment is enforced based on access size and memory type. In cases of misalignment, fallback mechanisms are employed to maintain architectural correctness.
- **Unaligned Access Handling**: When alignment cannot be guaranteed, memory operations are performed byte-wise to preserve compliance with architectural semantics, even under constrained or unpredictable conditions.
- **Endianness Handling**: Byte-order transformations are applied as necessary to support both little-endian and big-endian execution modes.
- **Exclusivity Semantics**: Exclusive access behavior is maintained to support atomic read-modify-write sequences, including tracking of exclusivity state based on address and memory region properties.
- **Memory Attribute Awareness**: Memory accesses are influenced by architectural attributes such as cacheability, shareability, and memory type, which affect both visibility and access behavior.
- **Address Translation and Permissions**: Virtual-to-physical address translation and permission enforcement are incorporated to determine effective addresses and access legitimacy under architectural rules.



(a) Airlift / manual ratio: lifting time.



(b) Airlift / manual ratio: instruction count.



(c) Airlift / manual ratio: block count.

Figure 6.1: Comparison of lifting efficiency metrics across instruction categories.

Note: Memory instruction ratios appear disproportionately high due to Airlift's full adherence to architectural semantics, which the manual baseline omits.

- **Atomicity Conditions**: Guarantees of atomicity are preserved for memory operations, particularly for wide accesses. When necessary, such operations are decomposed into smaller, atomic sub-accesses to uphold semantic correctness.

When memory operations are excluded, we find that among integer arithmetic and branch instructions, the runtime difference is more pronounced than the difference in AIR size. On average, Airlift is 4.3 times slower, while the generated AIR code is only 2.1 times bigger. This suggests that partial evaluation helps reduce the size and complexity of the generated AIR by avoiding code generation for inactive control paths. In contrast, a naive lifting strategy that transpiles the full instruction semantics would generate AIR for all branches, leading to bloated control flow and data flow graphs.

However, this comparison between lifting time and AIR instruction count only hints at the effectiveness of partial evaluation. The larger runtime difference may also stem from unrelated factors such as poor memory management, as discussed in Subsection 6.1.2.

Answer to RQ3. Airlift took a median of 4.3 times longer than the manually written baseline lifter, while the generated AIR code was only 2.1 times bigger. This suggests partial evaluation helps reduce code size, though the runtime gap likely also reflects other inefficiencies, such as memory management. Directly translating MRAS logic also introduces structural and performance inefficiencies, leaving room for further optimization.

```
__execute
    bits(datasize) operand = X[n];
    bits(datasize) result;
    for i = 0 to datasize-1
        result[datasize-1-i] = operand[i];
```

(a) Loop example in `rbit`.

```
integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x[i] == '1' then return i;
    return -1;
```

(b) Loop example in `cls` and `clz`.

Figure 6.2: Loop examples in ASL instruction semantics.

6.2.2 Sources of Inefficiency (RQ4)

Despite the benefits of partial evaluation, it also introduces performance drawbacks. Loops are the clearest example. The three outlier points in lifting time for integer instructions (shown in Figure 6.1.a) can be traced to Airlift executing loops like those in Figure 6.2. In the case of `rbit` (6.2a), `datasize` is a static variable and `operand` is residual. This causes the lifter to execute the loop and emit AIR for every iteration. The same applies to `cls` and `clz` (6.2b), where the loop executes 32 or 64 times depending on operand width, resulting in increased lifting time and AIR code size.

```
__execute
    ...
    (result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    X[d] = result;

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum[N-1:0];
    bit n = result[N-1];
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Figure 6.3: Unnecessary flag value calculation

Other inefficiencies, while less severe, are still noteworthy. Figure 6.3 shows the semantics for addition and subtraction instructions. The `AddWithCarry` subroutine calculates flag values regardless of whether the instruction sets flags. This is wasteful for variants where `setflags` is not set. Optimizations of this type could be applied to the AST of the MRAS before generating lifter code.

Answer to RQ4. Loops are the primary weakness of Airlift's partial evaluation strategy. In some cases, they result in worse performance than naive lifting. Additionally, generic ASL code causes unnecessary computation, highlighting opportunities for optimizations in the MRAS AST before code generation.

7 Future Work

7.1 Additional Implementations

Airlift can be extended to support other ISAs that already provide a complete MRAS, such as RISC-V. Since ASL is currently only used to describe the Arm ISA, supporting other MRAS languages like Sail (used by RISC-V) would require developing a new frontend. Another direction is generalizing the transpilation framework to allow easy integration of custom IRs. This would require broader instruction coverage. Although FP and SIMD instructions were not essential for the current use case of TNJ, supporting them would broaden the applicability of Airlift.

A key advantage of basing the lifter on MRAS is that extending support to additional instructions within the same ISA generally requires little effort. The generic mechanisms for instruction transpilation are already in place, so supporting new instructions would often only require adding support for a few additional ASL constructs.

7.2 Optimizations

As discussed in Chapter 3, correctness was the primary objective of this project. Apart from partial evaluation, no other optimizations were applied. This section outlines potential optimizations that can be applied to different stages of the Airlift workflow.

7.2.1 Loop Optimizations

In Section 6.2 we mentioned that loops, specifically ones whose body is a runtime scope but contains operations with promoted variables, were a major source of inefficiency for our approach. In the examples presented in Figure 6.2, there are not many runtime operations included in the body of the loop. In the first loop, only `datasize-1-i` is a runtime operation, while in the second loop, all operations involve promoted variables. This means that the reduction in the volume of the generated AIR code from iterating over the loops was minimal. A more efficient approach in this case would be to transpile the entire loop without any partial evaluation, possibly excluding `datasize-1` and `N-1` in the loop condition, which can still benefit from being precomputed. An effective optimization would be to design a heuristic that determines whether or not the

loop should be partially evaluated based on how much runtime workload in the loop body could be precomputed. The examples in Figure 6.2 would benefit from complete transpilation, both in lifting time and generated IR volume. On the other hand, a loop with a large number of runtime operations and a single promoted operation would benefit from partial evaluation.

7.2.2 AIR Code Post-Processing

From a qualitative analysis of the generated AIR codes, we observed that dead-code elimination would be one of the most effective optimizations to reduce AIR instruction count. This is due to how many ASL instruction semantics are designed. A representative example is shown in Figure 6.3. In the example, the `NZCV` flags are calculated before checking the `setflags` instruction field, resulting in flag calculation logic even for instructions like `add` and `sub` where the flags are not needed. Eliminating such dead code would significantly reduce the AIR code volume. Other optimizations, like peephole optimizations, could yield further improvements. However, these would increase lifting time, as they must be performed during lifter runtime.

7.2.3 ASL Code Pre-Processing

Unlike optimizations on the generated AIR code, optimizations performed on the input ASL code would reduce lifting time and instruction count without adding runtime overhead. In fact, most such optimizations would improve both metrics. Considering again the example in Figure 6.3, modifying the ASL to calculate flag values only when the `setflags` field is set allows the partial evaluation strategy to skip this segment entirely.

```
integer HighestSetBit(bits(N) x)
    return x != Zeros(N) ? FloorLog2(UInt(x)) : -1;

integer FloorLog2(integer x)
    integer result = -1;
    while x > 0 do
        x = x >> 1;
        result = result + 1;
    return result;
```

Figure 7.1: Possible ASL optimization of Figure 6.2b

Figure 7.1 shows another possible optimization on ASL code: this time on a loop

that we have already discussed in Section 6.2. Because of the `if` statement in the original loop body, every iteration generates three blocks in the output AIR code. In the modified implementation, which has no `if` in the loop body, the block count drops by at least 100. The challenge here is automating detection and patching of such opportunities.

7.2.4 Delegating Partial Evaluation to Codegen

In Section 5.4 we discussed the possibility of performing partial evaluation during the code generation stage instead of during lifter runtime. This would drastically increase complexity and workload at the code generation stage but significantly reduce lifting time, which is a much more relevant performance metric. This idea has been explored by Lift-Offline [9].

8 Related Work

8.1 Manual Lifter Implementations

McSema is an open-source, manually written lifter that translates machine code from various ISAs into the widely used LLVM IR [6]. Developed by the security-focused company Trail of Bits, McSema targets use cases such as symbolic execution, formal verification, and reverse engineering. It provides full control-flow recovery and a memory model suitable for program analysis, but is not optimized for seamless recompilation or direct integration into compiler pipelines.

In contrast, MCtoLL is another manually developed LLVM IR lifter, created by Microsoft [30]. It focuses on clean integration with the LLVM toolchain, enabling lifted binaries to directly benefit from existing compiler infrastructure such as static analysis passes, optimization pipelines, and backend support for recompilation to other architectures. MCtoLL assumes well-formed, compiler-generated binaries with symbol information, making it less suitable for stripped or obfuscated binaries but more suitable for practical reuse in toolchains.

While many tools lift machine code into popular IRs such as LLVM IR, other systems define and use custom IRs tailored to specific use cases. For example, QEMU uses TCG IR, a low-level, architecture-independent representation optimized for fast dynamic binary translation [5]. Risotto, a QEMU-based system, improves QEMU's support for weak memory model architectures by formalizing the concurrency semantics of TCG IR and inserting fences at runtime to preserve correctness [11]. Similarly, Lasagne targets static binary translation from x86 to Arm and introduces a concurrency-aware IR with a formally defined memory model, allowing it to statically insert fences while proving correctness with respect to the source and target ISAs [24].

8.2 Automatic Lifter Generation

Although there are numerous tools for lifting machine code into existing IRs, writing lifters for specialized IRs or constrained use cases remains a manual and error-prone task. Recent work has explored automated approaches. Cross-Architecture Lifter Synthesis proposes learning a lifter for one ISA by analyzing an existing lifter for

another ISA [27]. Forklift uses a token-level encoder-decoder transformer model to lift assembly code to LLVM IR [2]. While promising, these approaches do not offer the level of semantic precision required for formally sound system-level lifting.

8.2.1 MRAS-Based Lifting

Lift-Offline takes a more precise approach by generating a lifter directly from the ASL machine-readable specification [9]. It applies partial evaluation to simplify instruction semantics before generating the lifter, a technique that shares the same underlying motivation as Airlift. However, the stage at which partial evaluation occurs differs. Lift-Offline performs it during the code generation phase, whereas Airlift defers it to lifter runtime, as illustrated in Figure 5.1 and discussed in Section 5.4. Additionally, Airlift was developed independently and prior to the publication of Lift-Offline, with its implementation motivated by integration requirements in real-world systems where both semantic correctness and practical constraints must be balanced.

9 Conclusion

In this thesis, we presented Airlift, an automatically generated Rust-based lifter built from the AArch64 ISA's ASL MRAS. Airlift was developed with a focus on real-world integration into TNJ, which uses the generated AIR code to verify code safety in JIT compilation. The lifter is generated via hybrid transpilation of the MRAS, emitting executable Rust code for operations involving runtime-known values, and residual AIR code for machine state-dependent semantics. Prioritizing correctness in its design, we also compared Airlift's performance against a manually written baseline lifter to identify optimization opportunities for future work.

Abbreviations

AIR Assembly Intermediate Representation

ASL Architecture Specification Language

ASLi ASL interpreter

AST abstract syntax tree

BB basic block

FP floating point

IR intermediate representation

ISA instruction set architecture

JIT just-in-time

MRAS machine-readable architecture specification

PAC pointer authentication code

PCC proof-carrying code

PC program counter

SIMD single instruction, multiple data

SME Scalable Matrix Extension

SP stack pointer

SSA static single-assignment

SVE Scalable Vector Extension

TNJ TrustNoJit

WASM WebAssembly

List of Figures

Bibliography

[1] B. Alliance. *Sightglass: A Benchmarking Framework for Wasmtime and Cranelift*. `https://github.com/bytecodealliance/sightglass`. Accessed: 2024-05-09. 2024.

[2] J. Armengol-Estapé, R. C. O. Rocha, J. Woodruff, P. Minervini, and M. F. P. O'Boyle. *Forklift: An Extensible Neural Lifter*. 2024. arXiv: `2404.16041 [cs.PL]`.

[3] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: `10.1145/3290384`.

[4] Avast Software. *RetDec: A Retargetable Machine-Code Decompiler*. `https://retdec.com/`.

[5] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[6] T. of Bits. *McSema: Lift-based Static Binary Translation*. `https://github.com/lifting-bits/mcsema`. Accessed: 2024-05-09. 2024.

[7] D. L. Bruening and S. Amarasinghe. "Efficient, transparent, and comprehensive runtime code manipulation." AAI0807735. PhD thesis. USA, 2004.

[8] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform." In: *Computer Aided Verification*. Ed. by G. Gopalakrishnan and S. Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1.

[9] N. Coughlin, A. Michael, and K. Lam. "Lift-Offline: Instruction Lifter Generators." In: *Static Analysis*. Ed. by R. Giacobazzi and A. Gorla. Cham: Springer Nature Switzerland, 2025, pp. 86–119. ISBN: 978-3-031-74776-2.

[10] K. Garbers. "Design and Implementation of a Binary Translator from AArch64 to a Custom Intermediate Representation." Bachelor's thesis. Munich, Germany: Department of Informatics, Technical University of Munich, 2025.

[11] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia. "Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2022, pp. 107–122. ISBN: 9781450399159. DOI: 10.1145/3567955.3567962.

[12] Hex-Rays. *Hex-Rays Decompiler*. https://hex-rays.com/decompiler. https://hex-rays.com/decompiler. 2024.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596.

[14] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[15] A. Ltd. *A64 Instruction Set Architecture*. https://developer.arm.com/Architectures/A64%20Instruction%20Set%20Architecture. Accessed: 2025-05-13. 2024.

[16] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. arXiv: 1907.03890 [cs.SE].

[17] National Security Agency. *Ghidra Software Reverse Engineering Framework*. https://ghidra-sre.org/. Accessed: 2024-05-09. 2024.

[18] G. C. Necula. "Proof-carrying code." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. ISBN: 0897918533. DOI: 10.1145/263699.263712.

[19] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[20] J. Norman. *Super Duper Secure Mode*. https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/. Microsoft Browser Vulnerability Research, Accessed: 2024-05-09. 2021.

[21] A. Reid. *ARM's Architecture Specification Language*. https://alastairreid.github.io/specification_languages/. Accessed: 2024-05-10. 2016.

[22] A. Reid. *Towards a Formal Specification of Intel's x86 Architecture*. Presented at the Novel Architecture and Novel Design Automation (NANDA) Workshop. Imperial College London. Sept. 2022.

[23] A. Reid. *Using ASLi with Arm's v8.6-A ISA specification*. https://alastairreid.github.io/using-asli/. Accessed: 2024-05-10. 2020.

[24] R. C. O. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia. "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*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 888–902. ISBN: 9781450392655. DOI: 10.1145/3519939.3523719.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *IEEE Symposium on Security and Privacy*. 2016.

[26] M. Stone, J. Semrau, and J. Sadowski. *We're All in This Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023*. Tech. rep. Accessed: 2024-05-09. Google, 2024.

[27] R. van Tonder and C. Le Goues. "Cross-Architecture Lifter Synthesis." In: *Software Engineering and Formal Methods*. Ed. by E. B. Johnsen and I. Schaefer. Cham: Springer International Publishing, 2018, pp. 155–170. ISBN: 978-3-319-92970-5.

[28] "Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture." In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2016)*. Mountain View, CA, USA, Oct. 2016, pp. 161–168. ISBN: 978-0-9835678-6-8.

[29] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis. "Egalito: Layout-Agnostic Binary Recompilation." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 133–147. ISBN: 9781450371025. DOI: 10.1145/3373376.3378470.

[30] S. B. Yadavalli and A. Smith. "Raising Binaries to LLVM IR with MCTOLL (WIP Paper)." In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019.

Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 213–218. ISBN: 9781450367240. DOI: 10.1145/3316482.3326354.

[31] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. "Formal verification of SSA-based optimizations for LLVM." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 175–186. ISBN: 9781450320146. DOI: 10.1145/2491956.2462164.