Lift-*offline*: Instruction Lifter Generators

Nicholas Coughlin[1,2][0000−0001−8758−0666], A. Michael[1,2][0009−0000−7316−9112], and Kait Lam[1,2][0009−0001−2599−2259]

[1] Defence Science and Technology Group, Australia
[2] School of Electrical Engineering and Computer Science, The University of Queensland, Australia
n.coughlin@uq.edu.au

Abstract. Binary analysis techniques depend on instruction lifters to map instruction encodings to their semantic effects. Existing work has demonstrated automatic methods to extract such semantics from trustworthy architecture specifications, on a per-instruction basis. We extend these results to extract the semantics of *all* instructions at once, effectively generating an instruction lifter. We attain this result through the offline partial evaluation of formal architecture specifications, along with their analysis via *instruction opcode sensitive* abstract domains. To illustrate the approach, we generate a generic instruction lifter for ARMv8 and specialise it to a series of use cases. In addition to the static analysis of architecture specifications, this approach permits the static analysis of the generated lifter. We exploit this to establish bounds on lifter behaviours and its produced semantics.

1 Introduction

Binary analysis is crucial for applications where reasoning over source program representations is impossible or insufficient. For instance, the source program may simply be unavailable or the desired analysis outcomes may be invalidated by compilation [13,19,4,5]. Direct analysis of binary objects is often infeasible due to their complex and concise encoding. Consequently, most approaches first reconstruct internal representations (IRs) of their effects, designed to be more amenable to software analysis [7,46,3]. A common component of this translation is an *instruction lifter*, capable of mapping encoded machine instructions, i.e., instruction opcodes, into IR snippets. Through the analysis of other aspects of the binary, such as memory initialisation and control flow [32,14], these snippets are composed into an IR program, representing a binary's full behaviour.

Evidently, the soundness of any subsequent analysis is dependent on the correctness of this initial reconstruction. However, correct instruction lifters are notoriously difficult to develop [2,11]. Hardware vendors often only specify instruction behaviours as informal prose and pseudocode [1,18], leading to a laborious encoding process to capture their semantics. Given the breadth of instructions supported by modern architectures and the minor variations in behaviours across processor generations, it is unsurprising that a manual encoding process may introduce errors and/or only implement a subset of instructions. These concerns have motivated a series of research outcomes exploring and comparing independently derived instruction lifters, identifying a plethora of encoding mistakes [22,25].

To address these issues, existing work has explored the extraction of instruction semantics from formal architecture specifications [33,41,25]. These formal specifications, made available either by hardware vendors [38] or developed independently [15,17], encode the operational semantics of an architecture's instructions in a high-level language. They have been leveraged for various verification tasks concerned with hardware behaviours [24,27]. Evidently, their wide use lends credibility to their encoding of hardware behaviour [39]. Existing semantic extraction techniques reduce these formal specifications given a concrete instruction opcode and, potentially, additional constraints on the architecture state. Through aggressive simplification, based on SMT solving [41] and partial evaluation [25], the residual specification concisely represents the instruction's effects.

Despite the soundness benefits of these results, their integration into existing binary analysis projects remains a significant challenge. For instance, it is likely necessary to translate the extracted representation into the IR anticipated by the analysis project, possibly crossing language runtimes in the process. Given the semantics of these representations are rarely formalised [35], it is difficult to establish the validity of this translation. Moreover, the performance overhead of reducing the formal specification and its subsequent translation may be prohibitive for certain applications.

A further issue with existing semantic extraction techniques concerns their dependence on a concrete instruction opcode for simplification. As a consequence, it is difficult to reason over the results of their simplifications for *all* instruction opcodes. For instance, it may not be statically obvious what language structures and architecture state could be present in the residual, simplified representations. Additionally, certain instruction specifications may be beyond the capabilities of the extraction process, leading to failures or soundness issues when such instruction opcodes are processed. While these issues can be partially resolved through testing, i.e., applying the extraction process to a series of instruction opcodes, full coverage is difficult to establish.

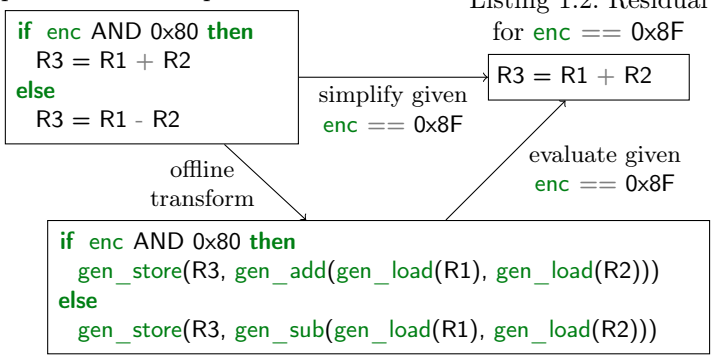
We contribute the novel notion of instruction lifter *generators* to address these shortcomings. Under this approach, an instruction lifter is automatically extracted from the formal architecture specification, rather than simply the semantics for a single instruction opcode. We attain this result through the appli-

Listing 1.1: Architecture Specification for Opcode enc

```
if enc AND 0x80 then
  R3 = R1 + R2
else
  R3 = R1 - R2
```

Listing 1.2: Residual for enc == 0x8F

```
R3 = R1 + R2
```



Listing 1.3: Generated Lifter

```
if enc AND 0x80 then
  gen_store(R3, gen_add(gen_load(R1), gen_load(R2)))
else
  gen_store(R3, gen_sub(gen_load(R1), gen_load(R2)))
```

cation of *offline* partial evaluation techniques [21]. Note that this work aims to produce semantics for a single instruction opcode, with the complexities of composing these to obtain semantics for the program as a whole and reconstructing language abstractions left to subsequent analysis [32,14].

To illustrate, consider the example architecture specification in Listing 1.1, in which a bitwise test of the instruction opcode enc determines whether an addition or subtraction operation should be performed over registers R1, R2 and R3. Existing extraction methods simplify based on a concrete value for enc to attain a residual form, as illustrated by Listing 1.2 when enc == 0x8F. Under the offline approach the architecture specification is instead transformed into a program that constructs this residual form for an arbitrary value of enc, as demonstrated in Listing 1.3. Construction of the residual form is implemented via a set of IR building primitives, illustrated here as gen_ prefixed operations. Evaluation of this transformed program for a value of enc will return a corresponding residual representation, e.g., also producing Listing 1.2 when enc == 0x8F.

The result of the offline transform corresponds to an instruction lifter, returning IR for concrete values of the instruction opcode. This is attained by statically delineating Listing 1.1 into notions of *lifting* and *residual effects*, where the former can be fully evaluated given an instruction opcode (e.g. enc AND 0x80) but the latter must appear in the residual output (e.g. R1 + R2). This static delineation is convenient for subsequent analysis and transformation, addressing the shortcomings of existing techniques. For instance, the generated lifter may be transpiled to new languages and transformed to construct alternative IRs, eliminating the need to cross language runtimes and translate IR representations per-instruction.

However, the static delineation may also lead to more complex residual representation relative to existing extraction approaches. Observe that existing approaches simplify given a concrete instruction opcode, leading to effective reductions for simple transforms such as constant propagation. Such reductions aren't as trivially apparent for the offline transform, as it must determine residual representations for arbitrary instruction opcodes.

Given such trade-offs, this work evaluates the feasibility of instruction lifter generators relative to existing extraction techniques. To attain this, we implement the necessary analyses and transforms for an encoding of the ARMv8 architecture [38] and compare with an existing extraction method [25] for said encoding. To address shortcomings with the approach, we contribute analyses that are *instruction opcode sensitive* to improve reasoning and simplification in the presence of arbitrary instruction opcodes. Moreover, we evaluate the versatility of the approach by generating lifters in three distinct languages, each producing outputs in distinct IRs. We further illustrate the static nature of the approach by analysing the produced lifters, establishing properties such as the absence of violated invariants, lower bounds on supported instructions and upper bounds on the complexity of the output representation.

We detail the formalisation of ARMv8 in Section 3, along with the existing semantic extraction technique, which we build on and compare with, in Section 4. In Section 5 we detail our offline partial evaluation approach, and illustrate its versatility through three distinct specialisations in Section 6. Following this, we evaluate the produced lifters across a series of metrics and detail bounds on their behaviour in Section 7. Finally, we explore related work and conclude in Sections 8 and 9.

2 Preliminaries

We consider a standard type system throughout this work, with judgements of the form $x : A$, representing x of type A. Moreover, we represent a program as $f : A \to B$, consuming arguments of type A and producing results of type B, with program application represented as $f(x) : B$. We additionally consider the encoding of programs, such that $f_{\mathcal{L}} : \{A \to B\}_{\mathcal{L}}$ corresponds to the encoding of a program of type $A \to B$ in language $\mathcal{L}$. Therefore, a program g that *generates* programs of type $B \to C$ in language $\mathcal{L}$, given some A, would have a type $g : A \to \{B \to C\}_{\mathcal{L}}$. The conversion of such a representation into an executable form is denoted as $\langle f_{\mathcal{L}} \rangle : A \to B$. For simplicity, we assume all language representations operate over equivalent types.

3 ARM Specification Language (ASL)

The ARM Specification Language (ASL) is a language for formally specifying the semantics of the ARM architecture. It grew from a formalisation of the pseudocode within the Architecture Reference Manual, and now its scope has grown to include description of almost the entire ARM system architecture. Since v8.2, ARM has used ASL within their machine-readable specifications which are published alongside the traditional natural language descriptions. The availability of such a trustworthy and authoritative architecture model, in a form intelligible by computer programs, has prompted development of numerous tools which use ASL as a foundation for verification and analysis.

In terms of language design, ASL borrows from both high-level and low-level languages and makes design decisions with its domain-specific use in mind. In addition to ordinary procedural programming features, some of its more unique features are:

- constructs for defining decode trees and instruction semantics,
- a selection of types catering to hardware specification: integers, real numbers, dependently-sized bitvectors, registers, arrays, tuples, and records are available for use,
- operator and function overloading, as well as overloaded array read/write syntax,
- bit mask values for use in pattern matching and decoding,
- formal reference parameters, and
- exception handling.

The architecture specifications are organised into instruction sets. Each instruction set (e.g. `A64`, `A32`) has a decoder which examines the opcode pattern, then dispatches to a particular instruction encoding family. An instruction family describes the semantics for a number of related instructions. Instructions within a family share roughly the same ASL semantics, with minor differences handled by conditionals.

Consider the example of `mov x1, x2` (with bytecode `E1 03 02 AA`). In the architecture, this encoding is an alias for `orr x1, xzr, x2` so the A64 decoder maps this to `aarch64_integer_logical_shiftedreg`. This instruction family, shown in Figure 1, describes the `and`/`or`/`eor` instructions with shifted register operands. We will use this opcode and its instruction family as a running example through this paper. The instruction's functionality may be simple, but this makes it useful test to demonstrate simplification and reduction of ASL.

In Figure 1's **`__decode`** and **`__execute`** sections, we see some of the difficulties which might arise when using these semantics for binary analysis work. The machine-readable specification is extensive and this instruction is only a simple bitwise operation, but we can already notice overloaded array syntax in `X[]`, parsing of enum types and arbitrary-precision integers, and several subroutine calls with their own internal complexity. The encoding also handles permutations of data size, operation, shift types, and flag behaviour.

Within the specification, this generality is advantageous, enabling the specification to define a breadth of instruction mnemonics without excessive repetition. However, when we require the semantics for a *particular* opcode, this becomes a burden. With these factors in mind, directly translating ASL into an analysis tool's input (e.g. theorem prover definitions or intermediate representation) would be unwieldy and ineffective, quickly ballooning the amount of code which the tool must reason about.

3.1 Formalisation

The ASL specification is sufficiently detailed to evaluate binary programs [38], primarily consisting of an outer evaluation loop that loads the next instruction

```
__encoding aarch64_integer_logical_shiftedreg
    __instruction_set A64
    __field sf 31 +: 1, opc 29 +: 2, shift 22 +: 2, N 21 +: 1,
            Rm 16 +: 5, imm6 10 +: 6, Rn 5 +: 5, Rd 0 +: 5
    __opcode 'xxx01010 xxxxxxxx xxxxxxxx xxxxxxxx'
    __decode
        (* parse bit fields into integers *)
        integer d = UInt(Rd); integer n = UInt(Rn); integer m = UInt(Rm);
        (* determine data size and operation *)
        integer datasize = if sf == '1' then 64 else 32;
        boolean setflags; LogicalOp op;
        case opc of
            when '00' => op = LogicalOp_AND; setflags = FALSE;
            when '01' => op = LogicalOp_ORR; setflags = FALSE;
            when '10' => op = LogicalOp_EOR; setflags = FALSE;
            when '11' => op = LogicalOp_AND; setflags = TRUE;

        if sf == '0' && imm6[5] == '1' then UNDEFINED;
        (* decode shift parameters, using helper method *)
        ShiftType shift_type = DecodeShift(shift);
        integer shift_amount = UInt(imm6);
        boolean invert = (N == '1');

    __execute
        (* load registers, X[n] is overloaded to return zeros if n == 31 *)
        bits(datasize) operand1 = X[n];
        bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

        if invert then operand2 = NOT(operand2);
        (* perform bitwise operation *)
        case op of
            when LogicalOp_AND => result = operand1 AND operand2;
            when LogicalOp_ORR => result = operand1 OR  operand2;
            when LogicalOp_EOR => result = operand1 EOR operand2;
        (* return results through output register and flag registers *)
        if setflags then
            PSTATE.[N,Z,C,V] = result[datasize-1]:IsZeroBit(result):'00';
        X[d] = result;
```

Fig. 1: Example ASL for the logical (shifted register) instruction family, including the mov <Xd>, <Xm> instructions.

opcode given the current program counter PC, decodes it to an instruction family, and then evaluates the instruction family. The evaluation of an instruction family consists of the evaluation of a series of ASL statements, as illustrated in Figure 1, culminating in a series of modifications to the architecture state, such as updating registers, flags or the PC in the event of a branch instruction. As this paper focuses on the semantics of an individual instruction, we ignore the outer evaluation loop and simply consider the decoding and evaluation stages for a given instruction opcode. Abstractly, we refer to the composition of these structures as:

$$Spec_{\text{ASL}} = \{(Opcode \times State) \rightarrow State\}_{\text{ASL}} \tag{1}$$

corresponding to a program, represented in ASL, that consumes an *Opcode* and a *State*, the architecture's notion of an instruction opcode and its execution state respectively, producing a new *State*, representing the state after evaluating the instruction.

We introduce $Eff_{\mathcal{L}} = \{State \rightarrow State\}_{\mathcal{L}}$, to concisely represent an instruction's effects as a program in language $\mathcal{L}$. We define an instruction lifter, that produces semantics in language $\mathcal{L}$, as follows:

$$Lifter_{\mathcal{L}} = Opcode \rightarrow Eff_{\mathcal{L}} \tag{2}$$

This paper describes an approach of deriving $Lifter_{\mathcal{L}}$ from $Spec_{\text{ASL}}$, over the full ASL language. This is achieved through a variety of static analyses and transforms operating over ASL statements, of type *Stmt*. For illustrative purposes, we describe these operations in terms of assignment, conditional and assertion statements. For instance, `x = e` assigns the result of expression `e` to `x` and **assert** `b` asserts that the expression `b` evaluates to true. Additionally, **if** `b` **then** `t` **else** `f` evaluates the expression `b` and performs either statement `t` or `f` accordingly.

The program operates over a standard state, mapping variables to values. Expressions are assumed to be pure, i.e., their evaluation does not alter state. We assume a function vars(`e`), that returns the free variables of expression `e`.

4 Online Partial Evaluation

Partial evaluation is the program transformation technique of specialising a program to a set of statically known values ahead-of-time. It propagates these static values as much as possible, allowing it to eliminate branches and unreachable code where possible. For values not known ahead-of-time, partial evaluation will emit a *residual* program to perform the remaining computation. The behaviour of the residual program and the original program should be identical when given the same set of inputs.

In prior work, we have developed *ASLp* [25], an online partial evaluator for ASL. Here, partial evaluation allows reducing the full architecture specification—necessarily large due to its fidelity and breadth—into a simpler form suitable

for binary analysis purposes. ASLp takes as input the architecture specification files and an instruction opcode, then partially evaluates the specification to return a concise summary of that instruction's semantics. The semantics are returned in *reduced ASL*, a subset of ASL which aims to be trivially usable with analysis tools. We encode this process formally as $\text{ASLp} : (Spec_{\text{ASL}}, Opcode) \to Eff_{\text{rASL}}$, where rASL represents reduced ASL. Consequently, correctness of ASLp is phrased as follows, equating the final states for any given opcode and initial state:

$$\forall spec : Spec_{\text{ASL}},\ op : Opcode,\ st : State \cdot \\ \langle \text{ASLp}(spec, op) \rangle (st) = \langle spec \rangle (op, st) \tag{3}$$

4.1 Implementation

At a high level, the online partial evaluation is built by mirroring the evaluation functions of an existing interpreter for ASL, *ASLi* [40]. It simply modifies the methods to operate on symbolic values instead of concrete bitvector values. A symbolic value is a disjoint union of three possible states:

(1) *Known* literal values,
(2) *Pure* expressions, and
(3) *Unknown*.

Known represents quantities that are known at *lift-time* (during the execution of the lifter). *Unknown* represents values which cannot be known until *run-time* (the context where the resulting semantics are given meaning, e.g. within an analysis tool). These expressions are emitted into the residual program as a computation followed by an assignment into a fresh variable. Hence, *Pure* encodes pure expressions of these reified unknown values.

These symbolic values enable significant simplification of the architecture specification. If all values within an expression are known, the expression can be computed ahead-of-time using ASLi's evaluation methods. However, the bulk of the simplification occurs with pure expressions. Although these cannot be fully evaluated, they can be simplified by applying algebraic rules. This includes ordinary mathematical identities on integers and booleans, as well as more sophisticated bitvector transformations (e.g. identifying definitely set/unset bits and coalescing repeated slices of the same expression).

After partial evaluation, post-processing passes are run to further simplify the residual program. Copy propagation, dead-code elimination, and common subexpression elimination passes are used, as in conventional compiler optimisation. Specialised passes translate particular ASL structures into more primitive operations, reducing the implementation burden on downstream applications. These include converting arbitrary-precision integers to sufficiently wide bitvectors, lowering ASL enum and register types to ordinary bitvectors, and, where slices are lift-time unknown, translating them to shift/truncate.

Given these transforms, ASLp is able to produce output which is natural and concise—similar in style and complexity to a hand-written lifter but with the

trustworthy foundations of the official specification. For the example mov x1, x2, ASLp reduces the code of Figure 1 into only: _R[1] = _R[2]. [3]

The correctness of the partial evaluator has also been validated against the ASLi interpreter with differential testing, and against other lifters by translation validation [25]. These efforts ensure ASLp's output can be trusted as a faithful translation of the ARM specification.

4.2 Limitations

Unfortunately, limitations of ASLp reveal themself when attempting to integrate it within other tools. The current partial evaluation is built with the core assumption that it processes one opcode at a time—the resulting semantics are specialised to describe only that instruction.

Therefore, downstream projects handling arbitrary opcodes are forced into a tight coupling with the ASLp program; users must call the partial evaluation for each instruction, and ASLp (with the complete ASL specifications) must be distributed alongside the user's program. This complicates the development processes and inflicts a considerable runtime cost onto end-users. Moreover, the single-opcode limitation renders caching, the typical mitigation for runtime costs, almost entirely ineffective.

Separately, it is also difficult to communicate the reduced semantics from ASLp to the program using it. ASLp returns its reduced ASL semantics as a textual encoding of the abstract syntax tree and projects using ASLp are required to implement and maintain their own parser. After parsing, the result must be manually translated into a useful intermediate representation. The downstream projects must ensure, manually, that their parser and translator can handle the range of possible reduced ASL statements.

At present, this is hindered by the lack of a comprehensive description of ASLp's output and the inherent complexity of developing such a description. In general, reasoning about the behaviour of ASLp across the whole instruction set is not straightforward. Due to its on-demand partial evaluation, testing is the only possible path towards such a goal. While the differential testing of ASLp includes $\sim 250,000$ opcodes, this remains a small subset of the instruction set. A more robust approach would need to ensure path coverage, possibly by use of symbolic execution to discover candidate test cases. Despite the possibility, the work required would be substantial.

5 Offline Partial Evaluation

Offline partial evaluation [21] represents an alternative specialisation technique, capable of addressing limitations encountered with the online approach. Given a program and a classification of its arguments as either *static* or *residual*, the offline approach statically transforms this program, *staging* it such that computations based on static arguments may be evaluated first. For instance, given a

[3] The _R array is the register file, referenced indirectly by the X[] accessor in Figure 1.

program f where $f : \{(A \times B) \to C\}_{\mathcal{L}}$ and A is marked as static, this process produces a new program f', such that $f' : \{A \to \{B \to C\}_{\mathcal{L}}\}_{\mathcal{L}}$. The resulting program f' can be considered a *specialiser* for f, producing specialised variants of f given some concrete value for the argument A. Let *offline* denote the process of transforming programs of the form f to f'.

In contrast to the online approach, where static arguments must be instantiated to derive residual programs, this approach delineates between static and residual programs purely based on the initial argument classification via a binding-time analysis [36], introduced later in Section 5.1. While more complex to implement [9], the resulting specialiser removes the need to simplify and reduce the original program for each static argument instantiation. Moreover, the specialiser is amenable to static analysis, allowing for reasoning over and modification of its specialising effects.

To illustrate, recall Listing 1.3 which may be considered a specialiser for Listing 1.1. In this setting, the instruction opcode `enc` is marked as static, resulting in any operations based purely on `enc` also being considered static, e.g., `enc AND 0x80`. Computations based on other state, such as `R1` and `R2`, must instead be residual. Such residual operations are transformed into IR building primitives to effectively defer their evaluation, discussed later in Section 5.2.

We consider the application of offline partial evaluation to architecture specifications with the instruction opcode marked as static:

$$\begin{aligned} \mathit{offline}(\mathit{Spec}_{\mathcal{L}}) &= \mathit{offline}(\{(\mathit{Opcode} \times \mathit{State}) \to \mathit{State}\}_{\mathcal{L}}) && \text{from Eq. 1} \\ &= \{\mathit{Opcode} \to \{\mathit{State} \to \mathit{State}\}_{\mathcal{L}}\}_{\mathcal{L}} && \text{by dfn. of } \mathit{offline} \\ &= \{\mathit{Lifter}_{\mathcal{L}}\}_{\mathcal{L}} && \text{by Eq. 2 \& dfn. of } \mathit{Eff}_{\mathcal{L}} \end{aligned}$$

Therefore, offline partial evaluation applied to $\mathit{Spec}_{\text{ASL}}$ will obtain a lifter program of type $\{\mathit{Lifter}_{\text{ASL}}\}_{\text{ASL}}$. We implement *aslgen*, an instance of *offline* for ASL specifications, to evaluate the feasibility of such a transform[4]. In practice, this process consumes an ASL specification and generates a lifter program in reduced ASL, such that $\mathit{aslgen} : \mathit{Spec}_{\text{ASL}} \to \{\mathit{Lifter}_{\text{rASL}}\}_{\text{rASL}}$. Similar to Equation 3, we phrase correctness in terms of the final states relative to the interpretation of the original specification:

$$\forall \mathit{spec},\ \mathit{op},\ \mathit{state} \cdot \langle\langle \mathit{aslgen}(\mathit{spec})\rangle(\mathit{op})\rangle(\mathit{state}) = \langle \mathit{spec}\rangle(\mathit{op}, \mathit{state}) \tag{4}$$

The lifter generated by $\mathit{aslgen}(\mathit{spec})$ produces simplified representations of an instruction's semantics given its opcode. Note that the generated lifter itself and its emitted semantics are encoded as reduced ASL programs. As discussed in Section 4.2, this representation presents a series of difficulties. Therefore, we convert this representation into forms more suited to integration through a series of backend generators, $\mathit{gen}^{\mathcal{L}}_{\text{IR}} : \{\mathit{Lifter}_{\text{rASL}}\}_{\text{rASL}} \to \{\mathit{Lifter}_{\text{IR}}\}_{\mathcal{L}}$. The function $\mathit{gen}^{\mathcal{L}}_{\text{IR}}$ transpiles the output of *aslgen* to a program in language $\mathcal{L}$, capable of constructing a semantically equivalent representation of instruction behaviour

[4] Available within the main ASLp repository, at https://github.com/UQ-PAC/aslp.

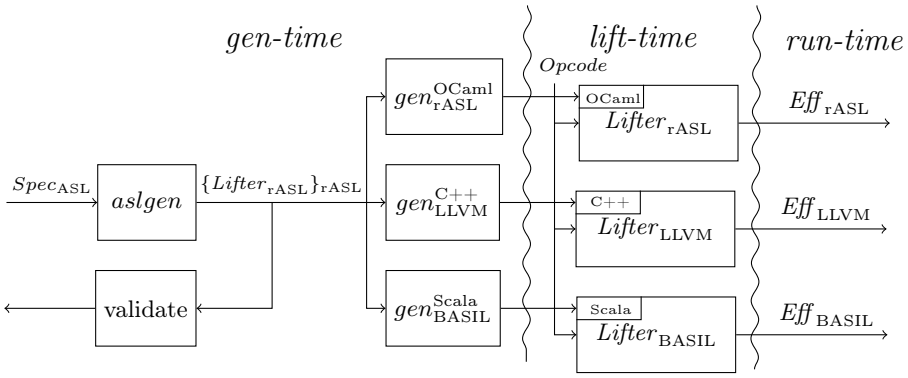


Fig. 2: Overview of the lifter generator process and its uses.

in language IR. Formally, we phrase its correctness as:

$$\forall \mathit{lifter},\ \mathit{op},\ \mathit{state} \cdot \langle\langle gen^{\mathcal{L}}_{\text{IR}}(\mathit{lifter})\rangle(\mathit{op})\rangle(\mathit{state}) = \langle\langle \mathit{lifter}\rangle(\mathit{op})\rangle(\mathit{state}) \tag{5}$$

Note the use of two, possibly distinct, languages: $\mathcal{L}$ and IR, the former describing the lifter and the latter describing instruction behaviour. Offline partial evaluation, through its delineation of static and residual computations, enables these two to be distinguished and transformed independently. For instance, it is possible to transform solely the lifter language $\mathcal{L}$, producing a OCaml program that generates reduced ASL representations of instruction opcode semantics. Alternatively, one can transform both, producing a C++ program that constructs LLVM IR to represent instruction semantics, which may be compiled and linked directly into some larger C++ binary analysis project. Additionally, this approach enables static analyses to be applied to the results of the lifter generator *aslgen*, establishing properties over its produced lifters.

The overall process can be seen in Figure 2. We refer to the lifter generation process as *gen-time*, beginning with lifter generation through *aslgen*, followed by analysis and translation of this lifter via some $gen^{\mathcal{L}}_{\text{IR}}$. This lifter is later executed, given some language runtime for $\mathcal{L}$ and concrete instruction opcode. This stage is considered *lift-time*, with the produced semantics considered *run-time*, corresponding to concepts from Section 4.

This section details the *aslgen* process, first covering the transforms required to perform offline partial evaluation, followed with additional stages to improve qualities of the produced lifter. Moreover, we cover static validation of the produced lifter, bounding its behaviours, in Section 5.5. The eventual conversion of these results into executable lifters is covered in Section 6.

5.1 Binding-time Analysis

In the context of instruction lifter generation, binding-time analysis is tasked with delineating computations between lift-time and run-time, i.e., identifying

which expressions could be evaluated if the instruction opcode was known. These results are used to construct the lifter, detailed later in Section 5.2. The analysis considers the syntactic dependencies between expressions, similar to taint analysis, such that operations syntactically independent of run-time values are considered lift-time.

We consider a two-value lattice, $Time$, for the abstract value domain, consisting of points *lift* and *run*, such that $\mathit{lift} \sqsubset \mathit{run}$. Informally, *lift* denotes the value will be known at lift-time, whereas *run* denotes it may not be. Consequently, the transfer function for an expression in this domain must produce a *lift* result only if all arguments to the expression are *lift*. The abstract state st tracks such a value for each variable, i.e., $st : Var \to Time$.

The abstract state must additionally consider the influence of run-time values on control flow. For instance, given a branch condition based on run-time state, it is not possible to determine at lift-time whether any assignments guarded by the branch should be evaluated. The abstract state tracks an additional value, $cond : Time$, capturing whether control flow to a statement is influenced by run-time values. The analysis operates over the product of these two abstractions, i.e., $(cond, st) : Time \times (Var \to Time)$. These behaviours are evident in the transfer function, tf_{bta}, for assignment and **if** statements:

$$
\begin{aligned}
\mathrm{tf}_{bta}(\mathsf{x = e})(cond, st) = \;& (cond, st[\mathsf{x} := cond \sqcup \mathrm{time}(\mathsf{e})(st)]) \\
\mathrm{tf}_{bta}(\textbf{if } \mathsf{b} \textbf{ then } \mathsf{t} \textbf{ else } \mathsf{f})(cond, st) = \;& \textbf{let } cond' = cond \sqcup \mathrm{time}(\mathsf{b})(st) \textbf{ in} \\
& \textbf{let } (\_, st_t) = \mathrm{tf}_{bta}(\mathsf{t})(cond', st) \textbf{ in} \\
& \textbf{let } (\_, st_f) = \mathrm{tf}_{bta}(\mathsf{f})(cond', st) \textbf{ in} \\
& (cond, st_t \sqcup st_f)
\end{aligned}
$$

where $f[x := e]$ is a function update, and $\mathrm{time}(e)(st)$ returns *run* if there exists a variable v in $\mathrm{vars}(e)$ such that $st(v) = \mathit{run}$.

Note that, once control flow is no longer dependent on a condition, i.e, at the join after an **if** statement, the value of *cond* is reverted to remove the condition's influence. This is trivially implemented over ASL, due to its limited control flow.

This analysis is applied inter-procedurally, with specialisation for calls based on the classification of arguments and control flow dependence. In this setting, calls are specification constructs in ASL and do not correspond to the procedural abstractions of a binary program.

Evidently, this analysis will under-approximate values known at lift-time, due to its reliance on syntactic dependencies. For instance, consider the expression `x - x`. The analysis, as described, will consider this expression as *run* given `x` is *run*. However, the expression is trivially `0` at lift-time, regardless of the value of `x`. Rather than embedding such cases into the binding-time analysis, we re-use the existing online partial evaluation pass (Section 4) before this analysis, but *without* a concrete instruction opcode. Recall that the online partial evaluator tracks and simplifies pure, symbolic expressions. Consequently, it may reduce `x - x` to `0` before binding-time analysis, along with various other cases.

5.2 Run-time Conversion

The results of binding-time analysis determine which program constructs, e.g., variables, control flow and expressions, must be placed in the residual program. Given this information, it is possible to transform the analysed program into a specialiser, such that these residual constructs are collected into a residual program as the specialiser executes. In the context of instruction lifters, this corresponds to the lifter building some representation of an instruction's semantic effects.

To attain this, we convert all program constructs that are influenced by *run*, i.e., run-time constructs, into calls against an *IR-building interface* (IBI). A series of assumptions and observations are made to simplify this transformation process and the IBI design. First, the IBI is assumed to construct IR with equivalent semantics to reduced ASL. For instance, the addition in Listing 1.1 is converted into a call to build a semantically equivalent addition IR construct in Listing 1.3. Hence most run-time constructs, such as primitive operations and variables, are simply transformed into their corresponding IBI calls. Second, the run-time constructs appear in the analysed program in evaluation order. Consequently, their in-place transformation will result in calls to the IBI also in evaluation order. To leverage this observation, the IBI is assumed to implicitly maintain a *writing* reference point, generally referring to the end of the actively constructed IR. A new construct is implicitly introduced at this writing reference point, with the reference point then updated to point after it.

```
if b then t else f
        ⇝
(t_branch,f_branch,join) =
      gen_if_stmt(runtime_b);
switch_to(t_branch);
runtime_t;
switch_to(f_branch);
runtime_f;
switch_to(join);
```

Fig. 3: Conversion of run-time **if** statement, where runtime_e refers to the run-time conversion of e.

Therefore, most IBI calls do not need to consider their placement in the IR, as this is handled implicitly. However, this approach fails when considering run-time control flow, where newly built constructs may need to be placed on a specific path. To address this, the IBI provides the means to build empty control flow structures and refer to their internal locations. To illustrate, consider the transformation of an **if** statement in Figure 3, where b is influenced by run-time state, requiring a run-time branch. The call to build such a structure, gen_if_stmt(), returns reference points to its internal locations: the true branch, the false branch and the join. The IBI provides the means to switch between these locations, ensuring structures are placed on appropriate branches and returning to the join once finished.

The process is further illustrated in Figures 4a and 4b. Through binding-time analysis, op__1 and setflags__1 are considered *lift* as they are derived from enc, the instruction opcode, which is trivially *lift*. Consequently, the **if** statements can all be resolved at lift-time. These constructs are therefore not

modified, implying their evaluation at lift-time. However, result__1 is considered *run*, as X.read52__2 is derived from a register read, corresponding to run-time global state. Therefore, the expressions written to result__1 are transformed into a series of semantically equivalent IBI calls, prefixed with gen_. The final register write, modifying an element of _R[], is also considered *run* and updated accordingly.

```
if eq_bits(op__1, '00') then
  result__1 = and_bits(X.read52__2, result__2);
else
  if eq_bits(op__1, '10') then
    result__1 = or_bits(X.read52__2, result__2);
  else
    if eq_bits(op__1, '01') then
      result__1 = eor_bits(X.read52__2, result__2);
    else
      throw UNSUPPORTED;
if setflags__1 then
  ...
else
  if ne_bits(enc[0+:5], '11111') then
    _R[cvt_bits_uint(enc[0+:5])] = ZeroExtend(result__1, 64);
```

(a) Representation of mov instruction semantics after initial partial evaluation.

```
if eq_bits(op__1, '00') then
  result__1 = gen_and_bits(X.read52__2, result__2);
else
  if eq_bits(op__1, '10') then
    result__1 = gen_or_bits(X.read52__2, result__2);
  else
    result__1 = gen_eor_bits(X.read52__2, result__2);
if setflags__1 then
  ...
else
  if ne_bits(enc[0+:5], '11111') then
    gen_array_store(_R, cvt_bits_uint(enc[0 +:5]), gen_ZeroExtend(result__1, ...));
```

(b) After *aslgen* analyses and run-time transforms. Expressions that must be run-time are converted to IBI calls (italicised). Trivial assertions and unreachable code are removed through opcode-sensitive dead-code elimination and value analysis.

Fig. 4: Comparison of instruction family aarch64_integer_logical_shiftedreg.

5.3 Opcode-Sensitive Abstract Domains

The application of binding-time analysis and run-time conversion are sufficient to generate a lifter, however, the semantics produced by this lifter will likely

be overly verbose and complex. This work aims to generate lifters that produce semantics as simple as those derived from the online lifter discussed in Section 4. This is challenging, given the online approach is capable of specialising given a concrete instruction opcode. Therefore, trivial optimisations obtain a significant specialising effect.

To illustrate, consider Figure 5, where enc corresponds to the instruction opcode, encoded as a bitvector, and X[] an array of registers. This is a generalisation of a common pattern found in the ARMv8 ASL specification where arguments to operations select between a register or a zero constant based on the instruction opcode. The subsequent use of temp for some calculation is implied here.

```
if enc[20] == enc[10] then
  temp = X[0];
else
  temp = 0;
```

Fig. 5: Zero register example.

Given a concrete value for enc, online partial evaluation can trivially reduce this branch. In the event of a false outcome, temp will be fixed to 0. As temp will be subsequently used as the argument to some series of operations, a 0 value will likely lead to substantial simplifying effects via transforms such as constant propagation. For offline partial evaluation, however, no immediate reduction of the branch is possible given enc is not known. Therefore, any reasoning over temp's value subsequent to this branch must consider the possibility of either a register value or a zero constant.

The difference between online and offline can be purely attributed to enc, as knowledge of this value distinguishes the two by definition. We address this difference by hoisting the opcode into the abstract domains of analyses applied in online partial evaluation, obtaining *opcode-sensitive* variants.

To illustrate our solution, consider an abstract domain $(A, \sqsubset)$ with a transfer function $\mathrm{tf} : \mathit{Stmt} \to A \to A$. We derive an opcode-sensitive variant via a function lattice $(\mathit{Opcode} \to A, \sqsubset_{op})$ and transfer function tf_{op}, with standard point-wise generalisations, such that, for $x, y : \mathit{Opcode} \to A$,

$$x \sqsubset_{op} y = \forall i \in \mathit{Opcode} \cdot x(i) \sqsubset y(i) \tag{6}$$

$$\mathrm{tf}_{op}(s)(x) = \lambda op \cdot \mathrm{tf}(s[op/\mathsf{enc}])(x(op)) \tag{7}$$

where $s[x/y]$ corresponds to the statement s with variable y replaced by expression x.

Application of the resulting domain corresponds to the analysis of the specification for all instruction opcodes simultaneously. While simply expressed, its practical implementation is non-trivial given the breadth of instructions supported by an architecture.

To obtain practical analyses, we make the following observations. First, instructions are grouped into families, based on similarities between their behaviours. Within these families only a limited set of instruction opcode bits influence their behaviour. Second, the instruction opcode generally influences behaviour via boolean operations over its bits, as seen in Figure 5. Consequently, we consider the analysis of individual instruction families to eliminate depen-

dence on a majority of instruction bits. Moreover, we encode functions of type $Opcode \to A$ as multi-terminal binary decision diagrams (MTBDDs) [8,16], given such representations efficiently and canonically encode boolean expressions of the style seen in instruction families.

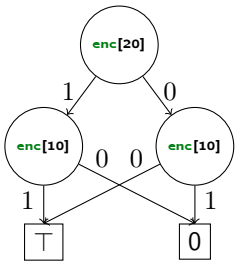


Fig. 6: MTBDD encoding of temp.

MTBDD-based encodings are well known in the context of abstract interpretation [28,31], with efficient implementations [20]. For our application, we consider a direct encoding of instruction opcode bits as decision nodes and lattice elements as terminal nodes. Operations over these terminal nodes are implemented in polynomial time, effectively considering their point-wise application. For instance, $x \sqsubset_{op} y$ is obtained via the application of $\sqsubset$ to terminal nodes of x and y, given the decision nodes leading to these terminal nodes are mutually feasible. This corresponds to Equation 6. Similar approaches are applied for other lattice operations.

This generalisation can also be applied to obtain the transfer function tf_{op}. In some applications, however, the resulting MTBDDs are overly precise, leading to costly representations that do not benefit the overall analysis. Consequently, some alternative, less precise transfer function tf'_{op} is implemented, such that:

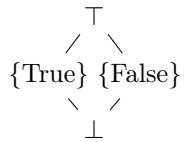
$$\forall x, x' \cdot x \sqsubseteq_{op} x' \implies \mathrm{tf}_{op}(s)(x) \sqsubseteq_{op} \mathrm{tf}'_{op}(s)(x')$$

For simplicity, a static order is applied to MTBDD decision nodes. In general, the highest bits of an opcode appear to have significant influence over the instruction behaviour, with the least significant bits corresponding to arguments. We order bits accordingly, with the highest bits placed earlier. This encoding is illustrated in Figure 6, corresponding to a constant propagation abstraction for temp in Figure 5, with $\top$ representing to an unknown value.

An additional concern with matching the online partial evaluator is its interprocedural behaviour. The existing implementation of the online approach inlines all functions up to some set of primitive functions. Naturally, this leads to precise reasoning across interprocedural boundaries. We match this behaviour in the offline approach, inlining all function calls into an instruction family definition. This benefits all analyses, including binding-time analysis, without prohibitively increasing program size.

We illustrate some of the implemented analyses and the encoding details to obtain opcode-sensitivity.

Value Analysis We consider a type-sensitive value analysis, to bound the possible values of variables and expressions throughout an instruction family. Abstractly, this lattice considers the powerset of a type's possible values as a function of the opcode, i.e., $Opcode \to \mathcal{P}(T)$ where T is all possible values of some type. Concretely, we consider distinct encodings of this lattice based on the given type.



For boolean values, the powerset lattice is directly considered, hoisted into a MTBDD-based encoding. Transfer functions over this domain are trivial, corresponding to the standard transfer functions over the powerset lattice applied to mutually feasible terminal nodes of the MTBDD. This corresponds directly to Equations 6 and 7.

This approach is extended to represent bitvector values, encoded as an array of boolean values. Bitwise and trivial operations, such as bitvector addition, are obtained via the corresponding boolean operations applied to array elements. Beyond these, over-approximations are considered, reducing to an array of $\top$ in most cases. Of additional concern is ASL's support for dynamic bitvector widths, requiring the representation of bitvectors of multiple widths. We address this through the analysis of integer values, detailed next, to derive the set of possible widths. A single array is therefore used, sized to the maximum possible width, to effectively represent all possible bitvectors. Operations dependent on the width, e.g., bitvector slicing and appending, extract the required sub-array.

An interval domain is implemented for integer values, i.e., (lo, hi) to represent all values v such that $lo \leq v < hi$, again with a trivial hoisting to obtain opcode-sensitivity via MTBDDs. Transfer functions are carefully considered to retain precision for operations frequently applied to instruction opcode bits, while collapsing intervals in other cases. Notably, this domain is structured to *not* represent infinite values, deviating from standard interval analysis. This may seem infeasible, as such a decision prevents the application of widening. However, we observe that the analysed semantics corresponds to the behaviours of a single hardware instruction. Consequently, its state and operations are trivially finite.

We leverage this observation and the precision of the value analysis to bound and unroll iterative behaviours, via repeated applications of the transfer function, avoiding the need for widening. This may not seem robust, given iterative behaviours based on values outside the capabilities of the analysis, i.e., values represented as $\top$, will result impractical analysis outcomes. In evaluating this approach for the ARMv8 semantics, however, we find all iterative behaviours of interest are appropriately bound. Admittedly, this may not hold for other architecture specifications. Such situations would, evidently, demand manual simplification of the specifications or a more aggressive analysis. We defer such complexities to these hypothetical specifications.

The resulting abstract state for the value analysis corresponds to $st : Var \rightarrow Opcode \rightarrow \mathcal{P}(T)$. Additionally, we encode a notion of reachability, $reach : Opcode \rightarrow Bool$, to identify statements that will not be evaluated for a given opcode. Abstractly, this corresponds to the empty set of states at a particular statement. We encode this notion as a binary decision diagram (BDD) directly, i.e., a MTBDD with only true and false terminal nodes, such that false represents unreachable. Control flow statements, such as conditions on **if** and **assert**, determine the abstract value of their branching expressions and update $reach$ accordingly. We illustrate the transfer function tf_{val} for assignments and asserts,

assuming a transfer function for expression tf_e:

$$\begin{aligned}\mathrm{tf}_{val}(\mathsf{x} = \mathsf{e})(reach, st) &= (reach, st[\mathsf{x} := \mathrm{tf}_e(\mathsf{e})(v)])\\ \mathrm{tf}_{val}(\textbf{assert } \mathsf{b})(reach, st) &= \textbf{let } b' : Opcode \to \mathcal{P}(Bool) = \mathrm{tf}_e(\mathsf{b})(st) \textbf{ in}\\ &\quad \textbf{let } r : Opcode \to Bool = (\lambda v \cdot \mathrm{True} \in v) \circ b' \textbf{ in}\\ &\quad (reach \wedge r, st)\end{aligned}$$

where boolean operations over a BDD correspond to boolean operations over its terminal nodes, and $f \circ g$ is function composition, i.e., $(f \circ g)(x) = f(g(x))$, implemented here as the application of function f to all terminal nodes of the MTBDD g.

To benefit from $reach$, we implicitly consider all values $\bot$ for an opcode where $reach$ is false, i.e., $st(v)(op) = \bot$ for all variables v when $\neg reach(op)$. This encoding avoids the duplication of $reach$'s conditions throughout all of the state's values. Consequently, we define join over the state abstraction $(Opcode \to Bool) \times (Var \to Opcode \to \mathcal{P}(T))$ as follows:

$$\begin{aligned}&(reach_x, st_x) \sqcup (reach_y, st_y) =\\ &\qquad (reach_x \vee reach_y, \lambda v \cdot (reach_x \ ?\ st_x(v) : \bot) \sqcup (reach_y \ ?\ st_y(v) : \bot))\end{aligned}$$

where $c \ ?\ x : y = \lambda op \cdot \textbf{ if } c(op) \textbf{ then } x(op) \textbf{ else } y(op)$.

Optimisation Analyses The online approach implements a series of standard program optimisations after its specialisation process, such as copy propagation and dead-code elimination. To match these simplifications, their analyses are also made opcode-sensitive via MTBDD-based hoisting. Moreover, their transfer functions are structured to leverage information derived in the value analysis.

For instance, dead-code elimination is dependent on liveness analysis to identify dead variables. We encode the liveness analysis domain as $Var \to Opcode \to Bool$, corresponding to a BDD, encoding $Opcode \to Bool$, for each variable such that a true value denotes the variable is live. To obtain a path-sensitive result, the transfer function considers the $reach$ value for each program statement, altering the abstract state only for the conditions under which the statement will evaluate:

$$\begin{aligned}\mathrm{tf}_{live}(\mathsf{x} = \mathsf{e})(st) = \lambda v \ \cdot\ &\textbf{ if } v \in \mathrm{vars}(\mathsf{e}) \textbf{ then } st(v) \vee reach\\ &\textbf{ else if } v = \mathsf{x} \textbf{ then } st(v) \wedge \neg reach\\ &\textbf{ else } st(v)\end{aligned}$$

where $reach$ is the reachability of this statement given the value analysis.

A similar strategy is implemented for copy propagation, tracking redundant variables and their replacements conditionally on the opcode.

5.4 Opcode-Sensitive Transforms

Given these opcode-sensitive analysis results, we apply a series of transforms to the lifter with the intention of simplifying its produced semantics, potentially

at the cost of additional lift-time complexity. To handle opcode-sensitivity in the general case, consider a program transform $\mathrm{tr} : \mathit{Stmt} \to A \to \mathit{Stmt}$. This function is invoked over all program statements, each being transformed in-place given some abstract domain, A, representing analysis results for the statement. If this analysis result is made opcode-sensitive, i.e., $\mathit{Opcode} \to A$, the corresponding opcode-sensitive transform, tr_{op}, would be the following:

$$\begin{aligned} \mathrm{tr}_{op}(s)(x) = \ & \textbf{if } x(\mathsf{enc}) == l_1 \textbf{ then } \mathrm{tr}(s)(l_1) \\ & \textbf{else if } x(\mathsf{enc}) == l_2 \textbf{ then } \mathrm{tr}(s)(l_2) \\ & \ldots \\ & \textbf{else if } x(\mathsf{enc}) == l_n \textbf{ then } \mathrm{tr}(s)(l_n) \end{aligned} \tag{8}$$

where $l_1, l_2, ..., l_n$ correspond to the range of x. Note that the **if** statements represent the *transformed* statement, i.e, the outcome of applying tr_{op}, and are not evaluated.

This encodes the specialisation of statement s for different classes of instruction opcode, based on the complexity of the abstract domain. In practice, further reductions are necessary, e.g., collapsing equivalent branches outcomes, or succinct and semantically equivalent statements may be directly generated. While this technique may be applied generally, such a specialising transform is only beneficial when s is costly, i.e., the potential of a simplified variant of s outweighs the overhead of the additional branching. Evidently this is the case when s is a run-time statement, as the additional reasoning at lift-time is a minor cost relative to producing simplified run-time semantics.

We detail three applications of this technique, implemented for the offline partial evaluation pipeline.

Dead-Code Elimination Variables may be conditionally dead based on the instruction opcode, i.e., their use after definition may be conditional on bits of the instruction opcode. In such cases, if these variables hold run-time values, the lifter will emit their definitions even if they aren't subsequently needed. Given the opcode-sensitive liveness analysis and the results of binding-time analysis, such situations can be trivially identified. We apply the generalisation detailed in Equation 8 to capture these situations.

```
temp = X[0];
if enc[0] == 0 then
  X[1] = temp;
else
  X[1] = 0;
```

(a) Pre-transform

```
temp = X[0];
X[1] = 0;
```

(b) Residual of (a), when `enc[0] != 0`

```
if enc[0] == 0 then
  temp = X[0];
if enc[0] == 0 then
  X[1] = temp;
else
  X[1] = 0;
```

(c) After transform

```
X[1] = 0;
```

(d) Residual of (c), when `enc[0] != 0`

Fig. 7: Opcode-sensitive dead-code elimination.

We illustrate this approach in Figure 7, where X[0] is a run-time value assigned to the conditionally dead variable temp. Given its use is conditional on enc, the liveness analysis will construct a BDD capturing such conditions. This BDD is transformed back into the corresponding guard of the branching pattern in Equation 8, wrapping the conditionally-live variable.

Copy Propagation A similar strategy is applied to obtain opcode-sensitive copy propagation, tracking the possibility of conditional clobbers. For instance, consider temp = X[0]; X[UInt(enc [5:0])] = 0; ... temp ..., corresponding to a load of X[0] followed a write to some element of X[] based on opcode bits. If it were possible to show the written and read elements of X[] were distinct, i.e. 0 != UInt(enc [5:0]), the subsequent uses of temp could be replaced with X[0]. An opcode-sensitive copy propagation analysis attains exactly this result, encoding this conditional clobbering of X[0] as a BDD. Similar to the dead-code illustration, this BDD is converted back into an equivalent guard over enc and transforms are conditionally applied to references to temp.

Constant Propagation Given the value analysis, it is possible to identify run-time expressions that evaluate to some constant c for all instruction opcodes *cond*, formally, $\forall op \cdot cond(op) \implies \mathrm{tf}_e(e)(st)(op) \sqsubseteq \{c\}$ where st is the value analysis at expression e. Therefore, a simpler representation of run-time behaviours can be obtained by replacing e with **if** *cond*(enc) **then** c **else** e, producing only the constant for suitable opcodes.

5.5 Lifter Validation

Given these stages, *aslgen* successfully derives an instruction lifter, represented as a reduced ASL program. We exploit this static result to establish bounds on its behaviour.

For instance, it is possible to establish reachability conditions for all failing statements, such as **assert**, to reason over which instruction opcodes may lead to failure. These assertions are specified in the ASL encoding, validating shallow conditions for correctness. For instance, assertions are introduced to perform bounds checking operations over array accesses and validate particular flag combinations for instructions. The abstract domain necessary for such reasoning is already present given the value analysis and its reachability outcomes. Formally, we over-approximate the failing conditions for a statement s as a BDD via $\mathrm{fail}(s)(st) : Opcode \to Bool$, where st is the result of the value analysis at statement s. For instance, $\mathrm{fail}(\textbf{assert}\ \mathsf{b})(st) = (\lambda v \cdot \mathrm{False} \in v) \circ \mathrm{tf}_e(b)(st)$, such that an **assert** over an expression that could be False is considered a failure. Given this outcome, we construct the BDD for $reach \wedge \mathrm{fail}(s)(st)$, where $reach$ is the reachability of s.

If the resulting BDD reduces to false, then the failure conditions are not possible. Otherwise, the resulting BDD identifies instruction opcodes that may reach failure. By taking the disjunction of such BDDs across all statements, we

determine the possible failure conditions across an entire instruction family. In practice, we track multiple BDDs, encoding distinct failure modes.

A further property of interest is the complexity of the semantics produced by the lifter. We first consider this complexity in terms of the language constructs necessary to represent instruction semantics. This can be trivially checked, simply via the collection of all run-time language structures in the produced lifter, as each will appear in the produced semantics. This addresses the issues discussed in Section 4.2, as all the required IR constructs are immediately known. Moreover, if some of these constructs are problematic, their reachability can be approximated via the prior discussed technique, identifying any instruction opcodes that would become unsupported without such a construct.

Additionally, we implement simple analyses to establish high-water marks on the complexity of the produced IR, given some model of the cost of its constructs. Intuitively, this analysis walks the produced lifter and maintains counts on the various run-time constructs that may be required. To obtain greater precision and capture the influence of opcode-sensitive transforms, these counts are also made opcode-sensitive.

The outcomes of these validation analyses for the ARMv8 specification are discussed in Section 7.

6 Backends

Following the offline transform and optimisation of the resulting lifter, the lifter-generator backend $gen^{\mathcal{L}}_{\mathrm{IR}} : \{\mathit{Lifter}_{\mathrm{rASL}}\}_{\mathrm{rASL}} \to \{\mathit{Lifter}_{\mathrm{IR}}\}_{\mathcal{L}}$ defines the transpilation of the lifter to a new *host language* $\mathcal{L}$, which can be compiled and executed. An implementation of the IBI in the host language for the *target* IR is provided so that the resulting lifter generates the target IR.

Concretely, implementing a backend means implementing the transpiler from the reduced ASL statements and expressions describing the generated lifter (e.g. Figure 4b) to the host language, then implementing the IBI functions to construct the target IR.

To enable the transpilation of the generated lifter to a wide range of host languages, the offline transform assumes minimal features from the host language. It only makes use of boolean, integer and bitvector types, mutable and immutable variables, assignments, function calls, conditional branches and fixed-length for loops. These features must all be given an equivalent representation in the host language.

The choice of host language is primarily driven by software engineering requirements, such as integration with existing tools and intermediate representations. For example, to emit ASL we reuse the data structure defined by ASLi's OCaml implementation, motivating the choice of OCaml as the host language. By generating a lifter in the implementation-language of the target IR we obtain the closest possible integration, and avoid the performance costs of serialisation and the associated I/O and parsing, or crossing language boundaries through a foreign function interface.

Implementation freedom afforded by the IBI allows the concrete type of both lift-time and run-time language constructs to be selected at gen-time, the compile-time of the generated lifter, or even at lift-time. Furthermore, this means the generated lifter may be parametric in lift-time effects—a lifter in one host language may be able emit multiple IRs, or perform additional analysis and transformation when executed.

We now discuss three generated lifters, hosted in OCaml, C++, and Scala, and targeting rASL, LLVM IR, and BASIL IR respectively.

6.1 rASL Offline Lifter Hosted by OCaml

The $gen_{\text{rASL}}^{\text{OCaml}}$ backend is the most straightforward of all implemented lifters, as the necessary functionality to evaluate and construct ASL IR is already present within ASLi, the ASL interpreter underlying ASLp. Therefore, the translation $\{Lifter_{\text{rASL}}\}_{\text{rASL}} \rightarrow \{Lifter_{\text{IR}}\}_{\mathcal{L}}$ is achieved by a simple walk over the lifter's structure, implementing lift-time behaviour with equivalent OCaml operations and calling ASLi functions to construct the reduced ASL representation. The residual programs produced by this lifter correspond closely with the existing online implementation, producing similar semantic representations of instructions. We leverage this for a series of comparisons in Section 7.

A fragment of the OCaml-ASL lifter produced by the $gen_{\text{rASL}}^{\text{OCaml}}$ backend is shown in Figure 8. This is obtained by transforming the instruction family of Figure 4b through the offline generation process. Figure 9a shows the rASL result produced from this lifter with the mov x1, x2 opcode.

```
if (f_eq_bits (f_and_bits (v_enc) (bits "0110...0")) (bits "0110...0"))
    || (f_eq_bits (f_and_bits (v_enc) (bits "0110...0")) (bits "0000...0")) then begin
  v_result__1 := f_gen_and_bits (!v_X_read52__2) (!v_result__2)
end else begin
  (* ... omitted cases corresponding to other op values ... *)
end;
if f_eq_bits (f_and_bits (v_enc) (bits "0110...0")) (bits "0110...0") then begin
  (* ... omitted case corresponding to setflags == true *)
end else begin
  if not (f_eq_bits (f_and_bits (v_enc) (bits "000...00011111"))
                    (bits "000...00011111")) then begin
    f_gen_array_store (v__R)
      (f_cvt_bits_uint (extract_bits (v_enc) 0 5))
      (f_gen_ZeroExtend (!v_result__1) (f_gen_int_lit (Z "64")))
  end
end
```

Fig. 8: Fragment of the generated OCaml lifter for the mov instruction family. Conditions have been transformed into tests of the instruction opcode v_enc.

```
_R[1] = or_bits('0', lsl_bits (_R[2],'0') )
```

(a) rASL from the OCaml lifter.

```
%0 = load i64, ptr @R2, align 4
%1 = shl i64 %0, i7 0
%2 = select i1 true, i64 %1, i64 0
%3 = or i64 0, %2
store i64 %3, ptr @R1, align 4
```

(b) LLVM IR from the C++ lifter.

```
R1 := bvor64(0bv64, bvshl64(R2, ZeroExtend(57, 0bv7)))
```

(c) BASIL IR from the Scala lifter.

Fig. 9: Lifter results when executed with mov x1, x2.

6.2 LLVM Offline Lifter Hosted by C++

A C++ backend enables direct integration with C++ frameworks like LLVM and Alive2 [30]. We describe the implementation of such a backend, $gen^{\text{C++}}_{\text{LLVM}}$, demonstrating that the generated lifter is not constrained to languages implementing a specific (functional) programming idiom.

This backend implementation is designed to be parametric. The lift-time and run-time types are specified through template parameters and may be varied at C++ compile-time without re-generating the lifter. Furthermore, this is implemented in a type-safe way, so we can utilise C++'s type checking to guide the implementation of the IBI. However, the restriction of the types at compile-time might be undesirable if, for example, a generic shared library for arbitrary target IRs is needed. In these cases, specifying the run-time and lift-time types as std :: any will enable this flexibility but lessen the strength of compile-time type checking.

A shared library is is also beneficial to avoid the impact of the C++ lifter's long compile time, discussed later in Section 7.4. We additionally improve compile times by enabling parallel compilation, pre-compiling template header files, and reducing the necessity of frequent recompilation. This includes separating type definitions from the generated lifter, and separating template implementations from declarations.

The C++ lifter was instantiated with an IBI emitting LLVM IR. The LLVM produced by this lifter can be seen in Figure 9b.

6.3 BASIL IR Offline Lifter Hosted by Scala

BASIL is a program verification tool for AArch64 binaries implemented in Scala [44]. Although it lacks an external-facing interface to construct its IR, the offline lifter allows us to inject a lifter into its code base and generate BASIL IR through its internal methods.

BASIL uses a disassembler, ddisasm [14], to reconstruct a binary's control flow, producing a control flow graph (CFG) with blocks consisting of instruction opcodes. The previous online lifter required a separate OCaml program to invoke ASLp and serialise the resulting semantics of each instruction opcode, which BASIL subsequently deserialised and translated into its IR.

The generated offline lifter no longer needs to cross this language boundary. It loads the CFG and basic blocks from ddisasm and calls its lifting operation on each opcode. This removes BASIL's run-time dependency on ASLp and the ASL specification, avoiding the burden of maintaining the parser and translator.

The primary challenge faced when implementing this backend was the JVM's 64 KB method bytecode size limit. Since *aslgen* outputs one (potentially very large) function for each instruction family, the generated lifter fails to compile if translated directly. To address this, the backend replaces some branch bodies and sub-expressions with function calls which evaluate their contents, i.e. they are outlined. Outlining is performed before translation to Scala by a forwards analysis over the rASL representation. It is applied recursively, depth-first, to any statement list larger than the size threshold, making it possible to produce a deep call stack.

The residual program generated by the lifter represents control flow between instructions in the form of assignments to the program counter (PC), with conditional control flow performing this assignment within an **if** statement. However, in BASIL IL, such control flow is instead modelled as calls and goto commands, corresponding to edges in the CFG. To match PC assignments to control-flow graph edges, we instrument the IBI for BASIL IR to track assignments to the program counter, along with any **if** statements guarding the assignment. For each control-flow edge identified by ddisasm, we identify PC assignments in the edge's source block, verify they occur at the end of the block, and replace the PC assignment with an appropriate goto (or call when the target block is a procedure entry). In the case of conditional jumps, the *join* point implied by the guarding **if** statement is removed, and the jumps to the *join* are replaced with jumps to the respective target blocks of each outgoing CFG edge.

7 Evaluation

To evaluate *aslgen*, we consider its application to an existing ASL encoding of the ARMv8 architecture. The implementation and results are available at [10]: https://zenodo.org/records/13219113. We first detail the outcomes of the validation analyses, as described in Section 5.5. Next, we leverage the OCaml backend to implement differential testing, similar to prior work [25], validating the correctness of the produced semantics with respect the ASL interpreter, ASLi. Moreover, we use the OCaml backend to compare the semantics produced by offline partial evaluation with the online approach, at a syntactic level. Following this, we detail the generation and compile times for *aslgen* and each of the implemented backends. Finally, we discuss the relative effects of the online and offline approaches on verification performance with a real binary analysis tool.

	Exprs		Branches		Declarations	
Class	mean	max	mean	max	mean	max
Branch	9.43	15	0.43	1	0.14	1
Float	14.40	38	0.13	1	0.13	1
Integer	60.88	1,713	2.14	64	0.22	4
Memory	60.57	747	0.09	1	0.51	2
Vector	185.12	1,929	7.50	160	5.30	65

Table 1: Maximum & mean counts of expressions, branches, and declarations emitted for a given instruction.

7.1 Static Validation

We validate the offline lifter generated from the ASL specification of ARMv8, i.e., the output of *aslgen*(ARMv8), through a series of static analyses.

First, we consider the reachability of failing statements throughout the specification. Through aggressive precision in the value analysis, we successfully establish lifting support for all instruction opcodes, except for those that go beyond the enforced assumptions of the lifter model[5].

Second, we identify all IR primitives required to express the instruction semantics, given the IBI calls required during the offline transform. The user-mode behaviours of the ARMv8 model can be expressed in terms of standard bitvector primitive operations, along with a series of primitives for floating point operations. Additionally, required control flow is limited to **if** branching statements, as all other control flow, i.e., calls and iteration, are successfully reduced.

Third, we establish bounds on the complexity of produced semantics, by tallying the maximum number of IBI calls that may be invoked for a given instruction family. The results of this analysis are detailed in Table 1, grouped by classes of instruction families and the generated program constructs.

Evidently, vector instructions are more expensive. Vector instructions commonly contain loops, and even nested loops, which the generated lifter fully unrolls to simplify reasoning in subsequent binary analyses, at the cost of larger representations. The floating point instructions do not suffer from this because they abstract the complex floating point operations into primitive function calls, corresponding to IEEE 754 operations where possible. Given a standardized notion of vector primitive operations, i.e., addition over a n-element vector of bitvectors, similar benefits could be observed for the vector instructions.

7.2 Differential Testing

We evaluate the correctness of the lifter produced by $gen_{\text{rASL}}^{\text{OCaml}}$ with respect to the composition of Equations 4 and 5, i.e, are the produced semantics for a given instruction equivalent to those described in the specification. Building on infrastructure from prior work [25], we generate a series of instruction opcodes by

[5] For instance, to produce simple semantics, the lifter enforces the global constraint that it is operating in user-mode. Therefore, any instructions that require alternative modes to evaluate will fail.

enumerating arguments to instruction families. Given randomised initial states, we then evaluate both the lifted semantics and the original specification for each instruction opcode using the ASLi interpreter, followed by a comparison of the final states. This corresponds to evaluating the following:

$$\forall op \in \text{enum} \cdot \forall st \in \text{random} \cdot \langle\langle \mathit{Lifter}_{\text{rASL}} \rangle(op)\rangle(st) = \langle \text{ARMv8} \rangle(op, st) \quad (9)$$

where enum corresponds to a finite enumeration of instruction opcodes and random corresponds to finite enumeration of random states.

We find no issues with the generated lifter after evaluating its behaviours across a sample of 152,703 instruction opcodes. The selected opcodes cover behaviours across all instruction families, varing their flags and arguments to obtain greater coverage of their behaviours. While this does not formally establish correctness of the produced lifter, it provides a greater confidence in its results.

7.3 Comparison with Online Approach

We compare the lifter produced by $gen_{\text{rASL}}^{\text{OCaml}}$ with the online lifter (as introduced in Section 4) since the two are interchangeable: both are OCaml programs that consume an instruction opcode and produce its semantics in reduced ASL representations. We first evaluate their performance and then consider a manual comparison of the syntactic differences in their results, with respect to the instruction enumeration used in Section 7.2.

Performance For performance, we compare the execution times required to extract the semantics for each instruction opcode. Table 2 shows that the offline lifter is multiple orders-of-magnitude faster at lift-time. This is expected since the lifter is now compiled rather than interpreted, and we have performed the computationally-expensive partial evaluation ahead-of-time.

Output Comparison We compare the reduced ASL outputs generated from the online and offline lifters across the 152,703 enumerated instructions. We find 32% of this set produce textually identical ASL code after normalising variable names. The remaining differences are explained by over-approximations in the offline partial evaluation process.

		Avg. time (ms)	
Class	Opcode count	Online	Offline
Branch	189	1.342	0.011
Float	3,846	1.400	0.023
Integer	12,667	1.360	0.023
Memory	29,397	3.762	0.039
Vector	106,863	2.745	0.052

Table 2: Average lift-time execution time per instruction, for each instruction family class in the offline and online lifters implemented in OCaml.

	Exprs				Branches				Decls			
	Online		Offline		Online		Offline		Online		Offline	
	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
Branch	15.40	19	18.20	27	0.60	1	0.60	1	0.00	0	0.20	1
Float	31.53	53	33.13	80	0.13	1	0.13	1	1.27	2	0.13	1
Integer	245.10	5,183	273.69	5,558	3.41	64	3.41	64	0.15	3	0.23	4
Memory	146.64	1,855	147.06	1,855	0.09	1	0.09	1	0.27	2	0.55	2
Vector	345.43	4,447	391.78	3,879	7.56	192	7.29	160	5.83	64	4.95	65

Table 3: Online and offline instruction complexity by instruction class.

Table 3 gives summaries of the IR constructs produced for this set of enumerated instructions, grouped by instruction family classes, for the online and offline lifters. While the offline lifter often introduces extra temporary variables and sub-expressions, we find that it produces equivalent or simpler control flow. Given this and the minor differences in expression complexity, we hypothesize that it is unlikely the offline results will have a detrimental effect on any subsequent analysis. We now consider some specific examples where the offline lifter produces more complex residual programs.

The main source of increased expression complexity can be attributed to algebraic reductions obvious to the online lifter, but hidden by over-approximations in the offline approach. Consider, for example, the instruction `b #0` which is a branch to offset `0` from the current program counter. Abstractly, this instruction corresponds to an operation `PC = PC + offset`, where `offset` is the offset encoded in the instruction opcode, i.e., `0` here. As `offset` is encoded within the instruction opcode, the online lifter can immediately exploit its concrete value, whereas the offline approach must consider all possible values of `offset`. Consequently, the online approach can trivially reduce `PC = PC + 0` to only `PC = PC` via standard pattern matching on its constructed expressions, whereas the offline will produce the non-reduced form. This is shown in the diff below, comparing the online residual program (`(-)` lines) against the offline residual program (`(+)` lines). The two outputs align for all other literal arguments to this branch instruction, as there are no other reduction the online approach can exploit.

```
(-) PC = PC;
(+) PC = add_bits(PC, '0...00000000000000000000');
```

Fixing the offline output is clearly possible, but it would mean identifying the zero argument as a special case when generating the addition of run-time and lift-time values, specialising when the lift-time value is zero.

Similar issues are observed in other algebraic reductions, particularly for nested bitvector slicing and shifting operations. For instance, consider the case of *test bit and branch* instructions. In these instructions, a bit is extracted from a register and compared with zero, jumping to a given address if this is the case.

```
(-) if eq_bits(_R[0][32 +: 1], '0') then {
(+) if eq_bits(lsr_bits(_R[0], '0100000')[0 +: 1][0 +: 1], '0') then {
(-)   PC = PC;
(+)   PC = add_bits(PC, '0...00000000000000000000');
(=) }
```

For such an instruction, the online lifter simplifies the bit extraction to a single slice expression ([32 +: 1]), while the offline lifter's result encodes this extraction as three operations. Specifically a logical shift right by 32 bits (lsr_bits), and a slice extracting of the lowest bit ([0 +: 1]) which is redundantly repeated. We also observe a redundant addition of zero, as discussed prior.

For these instructions, the bit index (32) and register (_R[0]) are both decoded from the instruction opcode. As a result, these values are concrete literals during online partial evaluation, permitting trivial reductions over shifting and slicing operations. Evidently, these values will not be known to the offline lifter, requiring it to generalise the reductions implemented by the online to arbitrary, non-literal expressions. Such a generalisation should be possible, however the offline approach relies on the existing set of reductions that have been fine-tuned for online partial evaluation. These fine-tuned approaches are geared specifically towards cases with literal arguments, as such information is readily available to the online approach. Therefore, these issues are addressable, given additional effort to migrate online reductions to the offline.

These examples demonstrate the pathological cases that constitute the majority of differences in the complexity of expressions generated by the online and the offline lifter. Addressing such differences would likely require more sophisticated static analysis in the offline transform. Alternatively, standard reductions over the offline lifter's output are sufficient to clean up the resulting semantics, However, this has the limitation that it must be implemented for every backend.

7.4 Compile Times of Generated Lifters

While the offline approach provides considerable benefits, particularly for integration purposes, replicating the output of the online approach demands aggressive analysis and specialisation, as discussed in Section 5. These costs are apparent in the time taken to generate and compile a lifter. For instance, the current implementation of *aslgen* and the subsequent backend transforms take approximately 10 seconds to produce a lifter for the ARMv8 specification. Moreover, the subsequent compilation of this program into an executable takes significant time, varying based on language implementation. We list these in Table 4, along with the compile time of the online lifter for reference.

Evidently, these compile times are significant, in addition to the 10 seconds required to produce a lifter. Nevertheless, these stages are only required once per hardware architecture specification. As these specifications do not frequently change, such overheads are not significant.

Lifter	Compile time (s)
Online OCaml 4.14	2.60
Offline OCaml 4.14	63.22
Offline Scala 3.3.1	95.11
Offline C++	73.35

Table 4: Compile-time for lifter code using 16 threads.

	verification time (s)			resource count		
	mean	max	std dev	mean	max	std dev
offline	0.05	8.12	0.17	29,078.94	5,380,601	191,142.23
online	0.04	1.44	0.09	21,989.63	4,867,142	181,745.24

Table 5: Average solving time by SMT solver using online vs offline lifter.

7.5 Verification Performance

The BASIL analysis tool implements the information flow security logic of Winter et. al [47], through the annotation and translation of its IR into the Boogie verification language [26]. Boogie, subsequently, verifies the IR using weakest-precondition reasoning with an SMT solver.

The tool has been integrated with the existing online approach, via a custom translator from reduced ASL to BASIL, as well as the offline approach, via the $gen^{\text{Scala}}_{\text{BASIL}}$ backend. We evaluate all of BASIL's provided test programs for both lifters, to identify possible differences in analysis outcomes due to IR complexity. We find that all test program produce equivalent outcomes for the given lifters. Table 5 summarizes these findings, representing outcomes for the SMT queries ultimately produced by the tool, in terms of their termination times and resource count, averaged across 10 runs. Resource count is a figure produced by the solver to give some relative measure of its resource usage. Evidently, the offline approach incurs slightly higher resource usage relative to the online. We attribute this to the slight increase in residual program complexity. Note that Table 5 measures the solving time for the produced SMT queries, excluding the processing time to lift the binary and construct these queries.

8 Related Work

Given the wide variety of hardware architectures and their complexity, existing work has explored the development of reusable instruction lifters [34,43]. Such encodings are generally manually derived, potentially along with testing to validate their behaviours. Despite this, mistakes in their encodings continue to be found [25]. Additionally, the semantics of their encodings are rarely formalised, presenting soundness concerns for any subsequent analysis [35].

This paper is closely related to our prior work, in which online partial evaluation is applied to ASL [25]. An alternative approach is proposed by Sammler et al., applying symbolic execution [23] to a formal hardware specification as a means to extract a symbolic trace of an instruction's effects [41]. This work leverages an SMT solver to reason over the symbolic trace, discharging branch conditions and simplifying expressions. The resulting trace is similar to the concept of residual programs found in partial evaluation, representing an instruction's influence on the hardware state. Moreover, by leveraging additional automation, the correctness of the produced trace is formally verified with respect to the original semantics through a translation validation technique [37]. Relative to our work, these approaches provide strong arguments for soundness, however, are non-trivial to integrate into subsequent analysis.

Given formal architecture specifications are often defined as inputs to interactive theorem provers [15], existing work has explored the derivation of instruction semantics within such tools. These approaches generally leverage the prover's rewrite engine to soundly reduce representations [33,45,29]. Evidently, this approach provides strong formal guarantees of correctness and provides significant benefits when reasoning within the prover. However, similar to other automatic techniques, the extraction and integration of these reduced semantics for analysis in other settings is non-trivial.

Existing work has explored the application of binary decision diagrams to abstract interpretation. These approaches consider more general applications than the trivial notion of opcode-sensitivity discussed here. For instance, decision nodes may operate over arbitrary linear constraints [12]. Moreover, decisions over the bits of a bitvector, similar to our approach, have been used to encode integer sets [31] and points-to relations [28].

Differences between offline and online partial evaluation have been extensively discussed in existing studies [42]. While it is generally viewed that online partial evaluation will obtain more accurate results, given concrete static values, Christensen et al. [9] demonstrate equivalent results can be obtained with a *maximally polyvariant* offline partial evaluator. In essence, this approach avoids over-approximation during binding-time analysis at control flow joins by splitting the join and specialising for each outcome. While such a technique could be applied in our setting to match specialising effects, our primary challenge in matching the two approaches is due to subsequent simplifying transforms.

9 Conclusion

We have demonstrated the application of offline partial evaluation to a formal architecture specification, as a means to statically derive a generic instruction lifter and the specialisation of that lifter to a series of IRs. Moreover, we have illustrated the benefits of analysing the produced lifter, validating the breadth of supported instructions and the complexity of their representations. We believe this work paves the way to widespread use of automatically derived instruction lifters for analysis, reducing integration overhead to relatively simple lifter transforms. Additionally, this work enables the use of such lifters for contexts where per-instruction extraction would be impractical, such as emulation [6].

Evidently, online partial evaluation can derive simpler semantic representations for complex instructions. In future work, we aim to integrate further simplifications into the *aslgen* process, bridging this gap. However, we hypothesize that this difference in representation does not significantly limit uses of the lifter, as the bulk of these correspond to relatively minor syntactic differences.

The differential testing approach provides confidence in the correctness of the produced lifter, that goes beyond the assurances made by off-the-shelf lifters currently in use. However, this testing approach falls short of formal verification. We hope to address this in future work, formally establishing Equation 4 for a produced lifter through a process akin to translation validation.

References

1. ARM: ARM Architecture Reference Manual for A-profile architecture (2023)
2. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. Proc. ACM Program. Lang. 3(POPL) (jan 2019), https://doi.org/10.1145/3290384
3. Avast Software: avast/retdec: RetDec is a retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec (2022)
4. Balakrishnan, G., Reps, T.W.: WYSINWYX: What You See Is Not What You eXecute. ACM Trans. Program. Lang. Syst. 32(6), 23:1–23:84 (2010), https://doi.org/10.1145/1749608.1749612
5. Barthe, G., Blazy, S., Hutin, R., Pichardie, D.: Secure compilation of constant-resource programs. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–12. IEEE (2021), https://doi.org/10.1109/CSF51468.2021.00020
6. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA. pp. 41–46. USENIX (2005), http://www.usenix.org/events/usenix05/tech/freenix/bellard.html
7. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986), https://doi.org/10.1109/TC.1986.1676819
9. Christensen, N.H., Glück, R.: Offline partial evaluation can be as accurate as online partial evaluation. ACM Trans. Program. Lang. Syst. 26(1), 191–220 (2004), https://doi.org/10.1145/963778.963784
10. Coughlin, N., Michael, A., Lam, K.: Artifact for "Lift-offline: Instruction Lifter Generators" (Aug 2024), https://doi.org/10.5281/zenodo.13219113
11. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1133–1148. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3314221.3314601
12. Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. 213, 102725 (2022), https://doi.org/10.1016/j.scico.2021.102725
13. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015. pp. 73–87. IEEE Computer Society (2015), https://doi.org/10.1109/SPW.2015.33
14. Flores-Montoya, A., Schulte, E.M.: Datalog disassembly. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 1075–1092. USENIX Association (2020), https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya
15. Fox, A.C.J.: Formal specification and verification of ARM6. In: Basin, D.A., Wolff, B. (eds.) Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings. Lecture Notes in

Computer Science, vol. 2758, pp. 25–40. Springer (2003), https://doi.org/10.1007/10930755_2
16. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods Syst. Des. 10(2/3), 149–169 (1997), https://doi.org/10.1023/A:1008647823331
17. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86-64 instruction set. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 237–250. ACM (2016), https://doi.org/10.1145/2908080.2908121
18. Intel Corporation: Intel A64 and IA-32 Architectures Software Developer's manual (2023)
19. Jancar, J., Fourné, M., Braga, D.D.A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P., Acar, Y.: They're not that hard to mitigate: What cryptographic library developers think about timing attacks. In: Rabiser, R., Wimmer, M., Groher, I., Wortmann, A., Wiesmayr, B. (eds.) Software Engineering 2024, Fachtagung des GI-Fachbereichs Softwaretechnik, Linz, Austria, February 26 - March 1, 2024. LNI, vol. P-343, pp. 143–144. Gesellschaft für Informatik e.V. (2024), https://doi.org/10.18420/sw2024_47
20. Jeannet, B.: Bddapron (2012)
21. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall international series in computer science, Prentice Hall (1993)
22. Kim, S., Faerevaag, M., Jung, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. p. 353–364. ASE 2017, IEEE Press (2017)
23. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976), https://doi.org/10.1145/360248.360252
24. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 179–192. ACM (2014), https://doi.org/10.1145/2535838.2535841
25. Lam, K., Coughlin, N.: Lift-off: Trustworthy ARMv8 semantics from formal specifications. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023. pp. 274–283. IEEE (2023), https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_36
26. Leino, K.R.M.: This is Boogie 2, https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
27. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009), https://doi.org/10.1007/s10817-009-9155-4
28. Lhotak, O.: Program Analysis Using Binary Decision Diagrams. Ph.D. thesis, School of Computer Science, McGill University, Montreal (2006)
29. Lindner, A., Guanciale, R., Metere, R.: TrABin: Trustworthy analyses of binaries. Sci. Comput. Program. 174, 72–89 (2019), https://doi.org/10.1016/j.scico.2019.01.001
30. Lopes, N.P., Lee, J., Hur, C., Liu, Z., Regehr, J.: Alive2: bounded translation validation for LLVM. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM

SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 65–79. ACM (2021), https://doi.org/10.1145/3453483.3454030
31. Mattsen, S.: BDD-based value analysis for X86 executables. Ph.D. thesis, Technical University of Hamburg, Germany (2017), http://tubdok.tub.tuhh.de/handle/11420/1510
32. Meng, X., Miller, B.P.: Binary code is not easy. In: Zeller, A., Roychoudhury, A. (eds.) Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016. pp. 24–35. ACM (2016), https://doi.org/10.1145/2931037.2931047
33. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - Improved. In: Cabodi, G., Singh, S. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012. pp. 78–81. IEEE (2012), https://ieeexplore.ieee.org/document/6462558/
34. National Security Agency: Sleigh. https://github.com/NationalSecurityAgency/ghidra (2022)
35. Naus, N., Verbeek, F., Walker, D., Ravindran, B.: A formal semantics for P-Code. In: Lal, A., Tonetta, S. (eds.) Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13800, pp. 111–128. Springer (2022), https://doi.org/10.1007/978-3-031-25803-9_7
36. Palsberg, J., Schwartzbach, M.I.: Binding-time analysis: Abstract interpretation versus type inference. In: Bal, H.E. (ed.) Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France. pp. 277–288. IEEE Computer Society (1994), https://doi.org/10.1109/ICCL.1994.288372
37. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1384, pp. 151–166. Springer (1998), https://doi.org/10.1007/BFb0054170
38. Reid, A.: Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In: Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design. p. 161–168. FMCAD '16, FMCAD Inc, Austin, Texas (2016)
39. Reid, A.: Who guards the guards? Formal validation of the Arm v8-M architecture specification. Proc. ACM Program. Lang. 1(OOPSLA), 88:1–88:24 (2017), https://doi.org/10.1145/3133912
40. Reid, A.: Using ASLi with Arm's V8.6-A ISA specification (Jan 2020), https://alastairreid.github.io/using-asli/
41. Sammler, M., Hammond, A., Lepigre, R., Campbell, B., Pichon-Pharabod, J., Dreyer, D., Garg, D., Sewell, P.: Islaris: Verification of machine code against authoritative ISA semantics. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 825–840. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3519939.3523434
42. Sumii, E., Kobayashi, N.: Online-and-offline partial evaluation: A mixed approach (extended abstract). In: Lawall, J.L. (ed.) Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000. pp. 12–21. ACM (2000), https://doi.org/10.1145/328690.328694

43. Trail of Bits: lifting-bits/remill: Library for lifting machine code to LLVM bitcode. https://github.com/lifting-bits/remill (2022)
44. UQ-PAC: UQ-PAC/BASIL. https://github.com/UQ-PAC/BASIL (2024)
45. Verbeek, F., Olivier, P., Ravindran, B.: Sound C code decompilation for a subset of x86-64 binaries. In: de Boer, F.S., Cerone, A. (eds.) Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12310, pp. 247–264. Springer (2020), https://doi.org/10.1007/978-3-030-58768-0_14
46. Wang, F., Shoshitaishvili, Y.: Angr - the next generation of binary analysis. In: IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017. pp. 8–9. IEEE Computer Society (2017), https://doi.org/10.1109/SecDev.2017.14
47. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–16. IEEE (2021)