Formal Methods in Computer-Aided Design 2023

Lift-off: Trustworthy ARMv8 semantics from formal specifications

Kait Lam and Nicholas Coughlin
Defence Science and Technology Group, Australia
School of EECS, University of Queensland, Brisbane, Australia
{kait.lam, n.coughlin}@uq.edu.au

***Abstract*—Disassembly and lifting tools are essential in the verification of software binaries. However, existing tools are difficult to validate and hence not suitable when high levels of assurance are needed. We address this by deriving a trustworthy lifter for ARMv8, named ASLp, based on ARM's official machine-readable architecture files. ASLp is capable of extracting usable semantics for a large subset of ARMv8, covering almost all integer, control flow, memory and vector instructions.**

We demonstrate the utility of ASLp by integrating it with the CMU Binary Analysis Platform. Furthermore, we translate the lifter's output into LLVM IR and compare the resulting semantics with those from existing lifters Remill and RetDec, leveraging the trustworthiness afforded by our lifter to find a number of major and minor bugs in their outputs.

I. Introduction

Binary analysis techniques enable program verification over executable machine code, in contrast to reasoning over the abstract programming languages from which it may be derived. These techniques are essential in high assurance domains where software development toolchains represent a liability [1] and may obfuscate hardware behaviours of critical concern [2], [3], [4]. Due to the complexity of binary executables, multiple tools have been developed to provide a foundation for further analysis [5], [6], [7]. These tools *disassemble* binaries, identifying contents such as its machine instructions and static data, and then *decompile* them, lifting the behaviours of machine instructions into machine-generic imperative code. This is advantageous as domain-specific analyses can then be applied to this generic, simplified representation.

To soundly perform this transformation, these foundational tools require trustworthy semantic models of each architecture they aim to support. For disassembly, the architecture's instruction encoding logic [8] is required to identify instructions, along with some limited understanding of the control flow implications of instructions [7]. To then decompile them, detailed knowledge of the architecture's state and the effects of instructions on this state is essential [9].

While the information required for disassembly is widely available in the form of reusable decoding libraries [7], the task of specifying detailed instruction semantics for decompilation presents considerable difficulty. Modern architectures support thousands of instructions, with frequent additions to address performance and security issues [10], [11], [12]. Manually encoding each instruction's behaviour is a time consuming and error-prone task [13]. There are limited alternatives to a manual approach however, as instruction semantics are generally only specified as informal prose in large instructions set architecture (ISA) manuals [14], [15]. Correctly interpreting all behaviours described in these documents is a difficult task [16], with some projects deferring to incomplete hardware testing to derive semantics instead [17].

An additional concern is the fidelity of these semantic models. Encodings are generally simplified and optimised for a particular application given assumptions over the architecture behaviour, such as ignoring privileged execution modes. While this may benefit the implementation, it limits model reuse between tools due to over-specialisation to an intended purpose [18]. Furthermore, these implicit assumptions may not be clearly documented, potentially invalidating the soundness of any subsequent analysis. Evidently, these issues may negate the high assurance benefits that motivate binary analysis.

Trustworthy architecture models are required in a variety of other domains, such as compiler verification [19], [20], hardware verification [21], [22], and emulation [23]. Given this common motivation, multiple efforts have been made to develop formal architecture models for use across verification projects [24], [25], [18], [26]. While these models have seen use in certain binary analysis applications [27], [28], they have not been broadly used as a semantic underpinning for decompilation tools. This can be attributed to the significant semantic gap between these formal architecture models and the semantic encodings expected by these tools. For instance, formal models may exploit language features such as dependent types, recursive functions and exceptions. Such specification styles are not supported by decompilation tools, which instead encode semantics in simple imperative languages [29], [30].

In this paper, we propose the application of *partial evaluation* to bridge this semantic gap, specialising and translating the formal architecture semantics for each instruction to an encoding suitable for decompilation tools. In Section II, we detail an implementation of partial evaluation for a formal semantics of ARMv8 [25]. Following this, we describe two distinct use cases of the partial evaluator. First, in Section III, we demonstrate the feasibility of its direct use by integrating it into the CMU Binary Analysis Platform [5] to obtain a full binary analysis toolchain. Second, in Section IV, we compare the instruction semantics from two decompilation tools [29], [31] with those derived from the partial evaluator, leveraging existing translation validation techniques to automate the com-

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_36

parison [32]. Finally, we explore related work and conclude in Sections V and VI respectively.

II. Approach

The foundation of this work is the machine-readable architecture (MRA) published by ARM [33]. This is the formal specification of the ARM architecture, used internally for verification and validation of their hardware. The MRA specification is a comprehensive description of the architecture's intricacies and behaviours, describing registers, memory behaviour (including faults and translation), interrupts, and the behaviour of exceptions. For our purposes, we are most interested in the opcode decoder and instruction semantics. These are expressed within the MRA using ARM's architecture specification language (ASL).

ASL [34] is domain-specific imperative language for specification of instruction behaviours and architectural details. Some notable features are its arbitrary-precision integer and real types, dependently-sized bitvector types, pattern matching, and exception handling.

We make use of ASLi, an open-source library for interacting with ASL [35] which provides a lexer, parser, interpreter, and AST transformer. We extend ASLi with a static transformation process to extract instruction semantics and simplify them using *partial evaluation*.

In doing so, we produce the semantics of individual instructions in a proper subset of ASL called *reduced ASL*. Reduced ASL represents instruction semantics with restricted control flow statements and a minimal set of primitive operations based on SMT-LIB's theory of bitvectors. This allows for easy integration with other tools and straightforward translation into other intermediate languages for binary analysis.

Our extension of ASLi with partial evaluation, which we call *ASLp*, is introduced in Section II-B.

A. Machine-readable architecture example

The MRA specification provides a comprehensive description of the hardware's behaviours. To detail these behaviours concisely, the specification groups instructions into broad classes based on their function and addressing mode. Specifically, a single `__encoding` in the specification handles several mnemonics, disambiguating them by fields extracted from the opcode. As an example, the encoding in Listing 1 describes the semantics for `add` and `sub` with shifted operands. The pseudocode contains considerable complexity with branches and subroutines to handle flags, different data sizes, and various bitshift options. Even more details are contained within the function calls and overloaded array operations.

In the encoding, `__field` defines slices of the opcode, the `__decode` section extracts information from fields, and the `__execute` block gives the operational semantics of the instruction. When lifting a single opcode, the `__decode` block can be evaluated ahead of time and combined with the `__execute` statements to produce a simplified summary of the instruction's behaviour. This is explained in the next section, Section II-B.

```
__encoding aarch64_integer_arith_add_sub_shiftedreg
  __field Rd 0 +: 5
  [...]
  __decode
    integer d = UInt(Rd); // destination operand
    integer n = UInt(Rn); // first operand
    integer m = UInt(Rm); // second operand
    integer datasize = if sf == '1' then 64 else 32;
    boolean sub_op = (op == '1');  // add or sub
    boolean setflags = (S == '1'); // set flags?

    if shift == '11' then UNDEFINED;
    if sf == '0' && imm6[5] == '1' then UNDEFINED;

    // logical/arithmetic, left/right shift
    ShiftType shift_type = DecodeShift(shift);
    integer shift_amount = UInt(imm6);

__execute
  bits(datasize) result;
  bits(datasize) operand1 = X[n];
  bits(datasize) operand2 =
      ShiftReg(m, shift_type, shift_amount);
  bits(4) nzcv;
  bit carry_in;

  if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
  else
    carry_in = '0';
  (result, nzcv) = AddWithCarry(operand1,
                                operand2, carry_in);
  if setflags then
    PSTATE.[N,Z,C,V] = nzcv;
  X[d] = result;
```

Listing 1: ASL encoding of integer add/subtract with a shifted register operand, from the `arch_instrs.asl` file.

To support a common specification across architecture versions and optional extensions, instruction behaviours are often guarded by *feature flags* denoting differences in their behaviours across possible hardware implementations. This is accomplished with stub functions, i.e. functions declared without implementations, that are later overridden to select the desired behaviour. Some examples of these are:

```
boolean HasArchVersion(ArchVersion version);
boolean HaveEL(bits(2) el); // exception levels
boolean HaveSVE();     // scalable vectors
boolean HaveBTIExt(); // branch target ident.
boolean HaveDITExt(); // data indep. timing
```

The feature flags offer substantial control over the specification, permitting it to be tailored to the underlying hardware as necessary. We leverage this functionality to make our assumptions over the hardware precise and explicit, overriding these feature flags as a form of configuration to our partial evaluation process.

B. Partial evaluation implementation

The key to extracting useful semantics from the MRA specification is *partial evaluation*, a program transformation applicable in contexts where a subset of the program's inputs are known in advance. These known inputs are propagated throughout the program body, permitting the early evaluation of computations and simplification of language structures

```
X[1] = ZeroExtend(
  (X[2][0 +: 32] + (X[3][0 +: 32] << 4)), 64);
```

Listing 2: Residual program for `add w1, w2, w3, LSL 4`

based on identified constraints. The *residual* program produced by this transform consumes any remaining unknown inputs to generate a result equivalent to that of the original program [36].

For example, Listing 2 is the residual program of `add w1, w2, w3, LSL 4`, from the partial evaluation of Listing 1. Since this specifies various inputs ahead of time (e.g. register usage, bitvector widths and operation mode), the program can be significantly simplified. This process extends to function calls, such as `AddWithCarry()`, which are inlined and simplified down to primitive operations (e.g. `ZeroExtend()`, `<<` and `+`). The final residual program succinctly represents the instruction's effects in terms of unknown inputs—here, values held in the register array `X`.

We implement our ASL partial evaluator (ASLp)[1] by augmenting the existing ASLi [35] to perform *online* partial evaluation [37]. This approach preserves the structure of the existing interpreter but extends it to consider a *symbolic* state, in which variables map to one of the following:

1) `Known` v: A known concrete value v
2) `Expr` e: The result of a simplified pure expression e
3) `Unknown`: An unknown value

`Known` encodes inputs and intermediary calculations that are known ahead of time, with `Unknown` encoding the inverse. `Expr` encodes intermediary calculations over `Unknown` variables and is used to identify simplifications.

Our partial evaluator $\texttt{aslp}(prog, sym)$ produces a residual program for the program $prog$ given the symbolic state sym. We define correctness of `aslp` in terms of the existing interpreter, `eval`, such that the residual program will produce an equivalent final state as the original program given agreement between sym and the initial concrete state, st:

$$\begin{aligned} &\forall prog, st, sym \cdot \\ &\quad (\forall x, e \cdot sym(x) = \texttt{Expr}\ e \implies st(x) = \texttt{eval}(e, st)) \land \\ &\quad\quad (\forall x, v \cdot sym(x) = \texttt{Known}\ v \implies st(x) = v) \implies \\ &\quad\quad\quad \texttt{eval}(\texttt{aslp}(prog, sym), st) = \texttt{eval}(prog, st) \end{aligned}$$

where $\texttt{eval}(prog, st)$ returns the final state for program $prog$ and $\texttt{eval}(e, st)$ returns the value of expression e.

The partial evaluator maintains the symbolic state through a forward traversal of the program, evaluating language structures where possible and building the residual program otherwise. We list some of the applied partial evaluation techniques:

1) Expression Simplification: Rewrite rules are applied during the construction of `Expr` e terms. These transforms are critical to matching the simplicity of existing lifter outputs as, without them, the abstract nature of ASL introduces redundant operations. For instance, bitvector calculations may include redundant slicing, concatenation and extension operations due to the use of shared code paths. These are aggressively rewritten, generally by distributing slicing operations over subexpressions to identify additional simplification opportunities.

2) Aggregate Values: ASL enables various aggregate values, such as tuples, records and arrays. We unpack these structures into their individual components and transform operations over them accordingly. As an example, consider ASL's syntax for a destructuring assignment to multiple fields of a record in `PSTATE.[N,Z,C,V] = nzcv`. In this operation, the 4 bits of the bitvector `nzcv` are extracted into the corresponding fields (`N`,`Z`,`C`,`V`) of the `PSTATE` record. The partial evaluation process lowers this operation into four assignments to the individual fields of the appropriate slice of `nzcv`.

3) Function Inlining: ASL supports functions to implement common functionality. We inline all calls to ensure we emit a single code sequence for an instruction, excluding those to a configurable set of primitive functions. Inlining is implemented by introducing a fresh variable to represent the function result and stitching the callee and caller bodies together appropriately. This stitching process is complicated by the limited control flow expressible in ASL, demanding transforms detailed later in Item II-B5.

The primitive function set enables the abstraction of complex processor features. For instance, memory accesses in the MRA specification are complex, including details such as virtual address translation. As these complexities are generally ignored during binary analysis, the inlining process is configured to treat various memory operations as primitives. These function calls are later translated into corresponding primitives in binary analysis tools, as detailed in Sections III and IV. A similar technique is applied to model floating-point operations.

4) Iteration: Loops are widely used in instruction specifications, notably to encode operations over vector elements. The bounds on these loops are generally known during partial evaluation, permitting their elimination via unrolling. While this increases program size, it significantly simplifies subsequent analysis. When loop bounds are unknown, all iterating language structures are lowered into `while` loops and emitted into the residual program.

5) Branching Control Flow: ASL supports various branching control flow structures, such as `if` statements, ternary operators and pattern matching. Often, branch conditions can be resolved during partial evaluation, eliminating the branch. If not, these structures are lowered into `if` statements and all possible branches are explored, merging their symbolic states when control flow eventually rejoins.

As the state merge process can cause loss of analysis precision, we may defer the control flow join by duplicating the statements appearing after an `if` statement into both branches. Moreover, this transform is necessary when inlining a function with a `return` statement within a branch, as a means to represent the execution of the inlined function's body after the conditional `return`.

Note that this transform may result in an exponential increase in code size, given a sequence of branches. Consequently, it is not applied to branching structures derived from an unrolled loop, as seen in various vector instructions

[1]Available at: https://github.com/UQ-PAC/aslp

$$\begin{tikzcd}
 & & & \text{assumptions} & & \\
\text{hex encoding} \arrow[rr, "\textit{decoder}", ""{name=D, below}] & & \text{ASL} \arrow[r, "\textit{partial eval}", ""{name=P}] \arrow[d, "\textit{eval}"] \arrow[from=1-4, to=P, dotted] & \text{reduced ASL} \arrow[r, "\textit{transforms}", ""{name=T}] \arrow[from=1-4, to=T, dotted] & \text{reduced ASL} \arrow[r] \arrow[d, "\textit{eval}"] & \cdots \\
 & \text{ARM MRA} \arrow[to=D, dotted] & \text{state} \arrow[rr, dashed, leftrightarrow, "\textit{state comparison}"] & & \text{state} &
\end{tikzcd}$$

Fig. 1. Overview of ASLp's partial evaluation pipeline. Dotted lines indicate some external input and dashed lines represent state comparison.

with conditional behaviours per vector element. It is applied in a majority of cases however, as residual programs for instructions generally contain at most a single branch.

C. Transforms

After partial evaluation, transformations are applied to the output to further simplify its representation. This includes dead code elimination and common sub-expression elimination to remove unnecessary and duplicated calculations introduced during partial evaluation.

Furthermore, the MRA specification makes extensive use of arbitrary-precision integers, a feature rarely considered in binary analysis frameworks. Consequently, we convert these operations into bounded equivalents to successfully integrate residual programs into these frameworks.

This is possible as the processor state is specified using types of a bounded size, e.g. a 64-bit register's value is known to fit within 64 bits. Therefore, while instructions may be specified with arbitrary-precision operations, only some bounded component of their effects will ultimately interact with the processor state. We determine these bounds using a simple interval analysis over the residual program. Given these intervals, we transform all integer variables and operations into corresponding bitvector equivalents of the required size.

D. Testing

To validate the partial evaluator, we implement differential testing [38] between the original ASL specification and its reduced form (denoted by the "state comparison" dashed arrow in Figure 1). Although this is not a formal proof of correctness, it increases our confidence in the validity of our implementation.

We perform this testing on a subset of the ARMv8 instructions, chosen to be a representative sample of application-level opcodes with predictable behaviours. Specifically, we include:

- integer and arithmetic instructions,
- floating-point instructions,
- branch instructions,
- vector instructions,
- memory load/store instructions, and
- atomic instructions (sequential semantics only).

To implement differential testing, we leverage the existing ASL interpreter to evaluate both the original specification and the partial evaluator's output for a particular instruction encoding. We then compare the final states for the two executions, with a mismatch indicating a potential bug in the partial evaluator. For each ARMv8 mnemonic, we test various combinations of register operands and flag values with a randomly initialised machine state for the interpreter.

The partial evaluator passes these tests for almost all instruction families, with errors limited to uncommon ASL features which we currently do not support. Notably, multiple bugs in the existing ASL interpreter were identified during this testing, which we detail in Appendix A.

III. Binary Analysis Toolchain

The CMU Binary Analysis Platform (BAP) is a toolkit and library supporting the analysis of binary programs. Given a binary file, it provides disassembly, control flow reconstruction, and instruction semantics in its BAP intermediate language (BIL). Moreover, it is built with a modular plugin system, enabling the development of additional analyses and lifters.

We develop a plugin to integrate our ARMv8 semantics into BAP by translating ASLp's reduced ASL into BIL and interfacing with BAP's knowledge base. With this, we are able to leverage the existing BAP machinery and combine it with our derived semantics, easily obtaining a full binary analysis toolchain.

To demonstrate the viability of the resulting lifter, we compare its output with that of an existing ARMv8 plugin[2] across a series of programs, summarised in Table I. The ASLp variant of BAP successfully lifts a superset of instructions relative to those supported by the existing plugin, capturing additional memory and vector operations. Notably, the ASLp variant successfully lifts all instructions in the example binaries except for 2508 instructions implementing floating-point operations, for which the conversion to BAP's semantics is unclear.

To evaluate the complexity of the produced programs, we compare the line counts of their outputs and average line length. The ASLp variant consistently produced a shorter representation with an average line length of 29.45 characters, comparable to the 29.47 characters of the existing implementation. A manual inspection of the results attributes these differences to alternative representations of flag calculations and branching conditions. Moreover, multiple semantic errors in the existing lifter were identified, such as incorrect operation

[2]We use the following, as it is the most comprehensive ARMv8 plugin to our knowledge: https://github.com/BinaryAnalysisPlatform/bap/pull/1546

TABLE I
BAP COMPARISON

Program	Lifted Instructions		Time (s)		Size (lines)	
	BAP	ASLp	BAP	ASLp	BAP	ASLp
bzip2	25254	25275	8.55	10.60	37658	35727
cntlm	15899	15908	7.59	8.64	34691	33064
gcc	152036	153673	63.88	77.43	391241	375623
gzip	17501	17554	6.82	8.00	35283	33682
oggenc	56227	56817	18.70	24.67	106448	101464

widths and memory address calculations. We do not consider a detailed semantic comparison of the two outputs, instead focusing such efforts on other lifters in Section IV.

The ASLp variant introduces additional overhead, increasing lifting time by roughly 20%. This is unsurprising, due to the additional analysis required to reduce and translate the ASL specification into BAP's representation. We believe this is an acceptable trade-off, as the ASL implementation provides greater coverage of the ARMv8 architecture with a stronger argument for correctness. Moreover, these benefits extend to any other architectures specified in ASL, avoiding the substantial effort needed to manually encode such semantics in BAP's existing infrastructure.

IV. SEMANTIC COMPARISON OF LIFTERS

Ours is not the first project to provide semantics for the ARMv8 architecture. Although we have trustworthy semantics from the architecture model, there are many existing lifters in active use with their own instruction semantics. Instead of replacing these, we can validate their semantics by comparing them with the ASL lifter. In this way, we can gain a level of confidence in their instruction semantics over and above fuzz testing or hardware testing.

The semantic comparison is done using the translation validation tool from Alive2 [32] which verifies that a given LLVM IR program refines a source program—that is, the target program's behaviours are a subset of the source program's behaviours. The tool was developed to verify compiler and optimisation passes, but here it is used to test for equivalence of the semantics from different lifters. Alive2 supports this by performing its refinement checks bidirectionally.

To test the output of ASLp, which emits reduced ASL, we developed a translator[3] from reduced ASL to LLVM IR and compare its result with other lifters that produce LLVM IR. We choose RetDec and Remill to demonstrate this process.

Although these lifters share a common output language, the representation of registers, memory, and other hardware-level state differs in each. To compare them, we translate the lifter outputs into a unified "dialect" of LLVM IR. This dialect needs to be simple to aid Alive2's reasoning while capturing enough of the machine state to faithfully represent the semantics we wish to compare.

We design this unified state representation as follows. Registers are modelled as global integer variables of various sizes:

- 31 64-bit general purpose registers, `x0` to `x30`,
- 32 128-bit vector registers, `v0` to `v31`,

[3]Available at: https://github.com/UQ-PAC/llvm-translator

```
declare noundef i8 @load_8(i64 noundef)
  inaccessiblememonly nounwind willreturn
  readonly

declare noundef void @store_8(i64 noundef, i8)
  inaccessiblememonly nounwind willreturn
```

Listing 3: Memory load/store functions. 8, 16, 32, and 64-bit versions are defined.

- 4 1-bit flag registers, `nf`, `zf`, `cf`, and `vf`,
- a 64-bit `pc` register, and a 64-bit `sp` register.

Representing memory requires more careful consideration, since many instructions can load/store memory at arbitrary addresses. In LLVM, this is conventionally done with a **`inttoptr`** ("integer to pointer") cast, but Alive2 cannot reason about these operations. Instead, we approximate these by modelling memory as uninterpreted functions, as seen in Listing 3. In order for two programs to verify as equivalent, the source and target must have identical calls in the same order. This is an overapproximation of memory behaviours but is sufficient for verifying a single instruction's semantics.

To reduce the overapproximation, LLVM attribute tags are used on each declarations to constrain the effects of these intrinsics on memory and global variables.

The **`inaccessiblememonly`** tag indicates the functions only read from or write to memory not visible to the caller. This is well-suited to representing the memory and its effects (virtual address translation, alignment, etc.) and indicates that the loads/stores are independent of register values but interacts with other loads and stores. Load functions are additionally tagged as **`readonly`**, indicating that the inaccessible memory is not modified.

The **`willreturn`** tag indicates that these functions will terminate (i.e. not loop forever). Combined with **`nounwind`**, it means the function will terminate without raising an exception (i.e. without jumping back up the call stack).

As a consequence of using Alive2 for validation, the semantic comparison in this work is done with respect to the formal semantics of LLVM IR [39]. However, LLVM IR is designed for use within optimising compilers as a compilation target of higher-level languages As such, it is intentionally underspecified; some details are left as undefined behaviour (UB) to allow for different implementations. Moreover, it introduces "undefined" and "poison" values into each type so compilers may exploit particular instances of undefined behaviour for optimisation. These features are useful in compilation but less suitable for our purposes.

Formal semantics of instruction-internal behaviours, including the ARMv8 model we consider, should be precise and not exhibit any undefined behaviour. Undefined and poison values do not occur naturally within the architecture specification language. To handle these, we annotate many LLVM operations as **`noundef`** and **`nonnull`** to assert values are never undefined/poison or null. These will invoke undefined behaviour when their assumption is violated, which is acceptable since two programs will verify as long as the UB occurs in the same way in both cases.

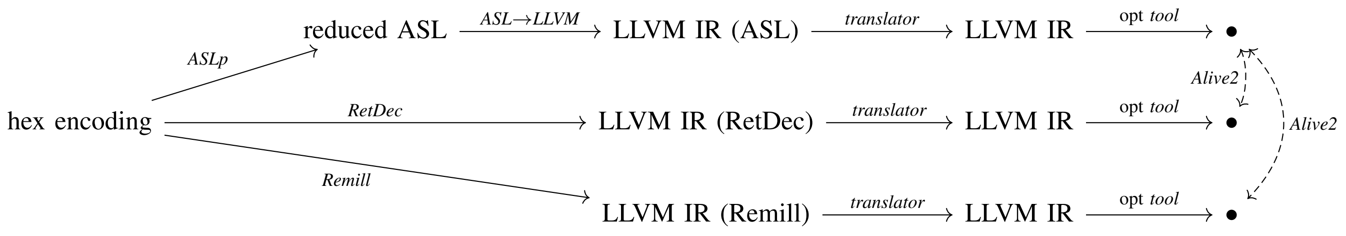

Fig. 2. Overview diagram of the lifter evaluation process. ASLp indicates the pipeline of Figure 1. Dashed lines indicate semantic comparison, and filled dots are the same type as their predecessor.

TABLE II
EVALUATION RESULTS

		RetDec				Remill			
	Count	Equivalent	Mismatch	Timeout	Unsupported	Equivalent	Mismatch	Timeout	Unsupported
Branch	162	74	9	10	69	120	1	37	4
Integer	14442	10147	608	415	3272	12058	15	1238	1131
Memory	6414	3864	618	0	1932	5057	88	0	1269
Vector binary	19170	264	426	0	18480	2997	0	0	16173
Vector unary	1098	9	153	0	936	333	0	0	765
Total	41286	14358	1814	425	24689	20565	104	1275	19342

With the comparison framework set up as above, evaluation of the lifters was conducted for each opcode, as outlined in Figure 2. For each opcode of interest, we perform the following:

1) Separately disassemble the instruction with each lifter. The ASL lifter produces a reduced ASL program which is translated to LLVM IR by a pattern-matching translator.
2) The LLVM IR from each lifter is transformed into the unified state and memory representation described above.
3) The LLVM optimiser `opt` is run on each output to simplify the resulting structures and allow for easier comparisons.
4) For each lifter under test, `alive-tv` checks for equivalence between its output and ASLp's LLVM IR.
5) If Alive2 can prove the equivalence, it reports the two outputs are equivalent. Otherwise it reports a mismatch or time out. A mismatch may be a difference in memory states or undefined behaviour. Where lifters do not match, Alive2 gives input values which cause the mismatch and we investigate the discrepancies by manually comparing the LLVM IR with the ARM ISA.

We used a subset of the integer, logical, branching and vector instructions tested in Section II-D to evaluate the three lifters. The results are summarised in Table II, organised by classes of instructions. Both RetDec and Remill were compared with the ASL lifter with a timeout of 20 seconds. The "Equivalent" column indicates both are semantically equivalent, and "Mismatch" indicates a difference in memory state or undefined behaviours. The "Unsupported" instructions are those supported by the ASL lifter but not the lifter under test.

A. *RetDec*

RetDec [31], developed by Avast Software, is a retargetable decompiler with plugins for IDA Pro and radare2. Here, we analyse its Capstone2LLVMIR component which provides its instruction semantics in LLVM IR. The produced IR is similar to our unified representation: registers are mutable global variables, memory operations used **`inttoptr`** instructions, and intrinsic functions handle program flow and branching. Since the goal of Capstone2LLVMIR is to lift to higher-level C/C++, it "does not aim to fully translate (give meaning/semantics to) all assembly instructions" [31].

Despite this, RetDec lifted a large fraction of the instructions tested. However, it had some inaccuracies in key implementation details and shortcomings with vector instructions. These are explained in more detail below.

Overflow flag computation: The overflow flag (`vf`) computation is incorrect for instructions which incorporate the carry flag (`cf`), e.g., `adcs`, `sbcs`. In these cases, `vf` is set when adding the carry *twice* would result in an overflow, when it should only be considered once. For example, this occurs with `adcs xzr, x0, xzr` (bytecode `1f001fba`) when `cf` is set and `x0` is $2^{63}-2$. This computation would not overflow, but RetDec's semantics indicate `vf` would be set. This error affects approximately 240 integer opcodes tested.

lshl/lshr/ashr poison: The LLVM instructions **`shl`**, **`lshr`** and **`ashr`** for bitvector shifts are defined to return a poison value when the shift amount is equal to or greater than the register size. However, ARMv8 shift instructions, such as `lsl`, `lsr`, `asr` and `ror`, are well defined in such scenarios, shifting by the desired amount modulo the size. RetDec ignores this difference, naively converting between the two.

For example, RetDec lifts `lsr x1, x1, x0` (`2124c09a`) to an LLVM snippet containing **`lshr i64`**`, %X1, %X0`. This operation will return poison when %X0 exceeds 63, where the ARMv8 specification would return 0. These inaccuracies make up 72 mismatches of integer opcodes.

Moreover, RetDec generates invalid shifts in various other cases, such as extr mnemonics and instructions with register operand rotations. These cases contain shifts that will always return poison, e.g., **shl i32** %3, 32. This affects 162 opcodes, such as and x0, x0, xzr, ror #0 (0000df8a).

clz poison: clz counts the number of zero bits before the first one bit in a bitvector, starting at the most significant bit. RetDec uses LLVM's @llvm.ctlz.* intrinsic functions to implement this behaviour. However, these calls are configured such that they will return poison when the bitvector is zero, instead of returning its width. For example, when x0 is zero clz x0, x0 (0010c0da) should set x0 to 64, but the RetDec result will produce poison.

uxtx/sxtx truncating to 32-bit: Various ARMv8 mnemonics accept register extension modes to specify how registers of different widths should be extended prior to applying an operation. For instance, uxtw specifies a zero-extension from 32-bit to 64-bit and sxtw specifies a signed-extension from 32-bit to 64-bit. Due to encoding quirks, it is possible to encode various modes that are effectively no-ops, such as uxtx for a zero-extension from 64-bit to 64-bit. However, RetDec lifts such cases incorrectly, truncating 64-bit registers down to 32-bit and then extending back to the original size. This error affects 220 opcodes, such as add x0, x1, x0, uxtx (2060208b).

Shifted uxtw/sxtw truncation: ARMv8 allows for the specification of shifted 32-bit offsets in memory address calculations. For instance, str w0, [x0, w0, uxtw #2] (005820b8), will perform a store to $\texttt{x0} + (\texttt{ZeroExtend}(\texttt{w0}, 64) << 2)$. where w0 is a 32-bit register. However, RetDec lifts these address calculations such that the shift is applied before the appropriate sign- or zero-extension. This results in the loss of w0's upper two bits and, subsequently, an incorrect address calculation. This affects 360 opcodes with sxtw operands, and 126 with uxtw.

sxtw extension: When specifying a memory access with a 32-bit offset, ARMv8 permits the application of either a zero- or a sign-extension to pad the value to 64-bit. RetDec always produces a zero-extension however, leading to incorrect addresses for sxtw. For example, str w0, [x1, w0, sxtw] (20c820b8) exhibits this behaviour. This error affects the same 360 sxtw opcodes as above.

udiv/sdiv by zero: LLVM's **udiv** and **sdiv** integer division instructions trigger undefined behaviour when the denominator is zero, instead of a zero result as defined in the ARMv8 specification. RetDec fails to account for this mismatch, for example, lifting udiv x0, x0, x0 (0008c09a) to an LLVM snippet that triggers undefined behaviour when x0 is zero.

Pre-increment address: Load/store pair instructions load or store two words at adjacent locations in memory, given the address of the first word and an increment. With pre-increment addressing, the address register should be incremented by the given offset prior to the memory access. However, the increment is added to the address of the second word instead of the first, leading to incorrect values in the updated address register. For example, stp xzr, xzr, [x0, #16]! (1f7c81a9) increments x0 by 24 instead of 16. This affects 102 variants of stp. Other paired memory operations appear to lift correctly.

SIMD instructions: For the majority of vector instructions, RetDec returns incorrect semantics; it treats the operands as ordinary registers and does not vectorise operations. For example, add v0.8b, v1.8b, v1.8 (2084210e) produces:

```
%0 = load i128, i128* @v1
%1 = load i128, i128* @v1
%2 = add i128 %0, %1
store i128 %2, i128* @v0
```

whereas the correct operation would consider v1 as 8 separate bytes, adding each byte elementwise. The same deficiency affects all vector instructions, leading to mismatches for all such opcodes. There are 579 such opcodes across the binary and unary vector instructions. The few 'equivalent' results for vector instructions occur when this discrepancy does not affect correctness, e.g. bitwise logical operations and subtraction of a register from itself.

B. Remill

Remill [29], developed by Trail of Bits, is a library providing LLVM IR instruction semantics for various architectures. As it "focuses on accurately lifting instructions" [29], it has seen wide adoption both individually and as part of the McSema binary decompiler [40]. Remill was found to have much fewer discrepancies than RetDec when compared with the ASL semantics.

sdiv overflow: When performing signed division over n-bit two's complement integers, the calculation $(-2^{n-1})/(-1)$ will overflow, as 2^{n-1} is not representable in n-bit two's complement. Under ARMv8, this is defined as returning -2^{n-1}, i.e. the truncation of 2^{n-1}. However, Remill lifts this operation to LLVM's **sdiv** instruction, which treats an overflow as undefined behaviour. For example, this error manifests in sdiv x1, x0, x1 (010cc19a).

ldp/strb with writeback: ARMv8's memory addressing modes support incrementing an address register before or after memory is accessed, in a process called *writeback*. When the data and address registers of these operations overlap, the implementation is permitted to select one of several acceptable behaviours. For instance, loads with overlapping registers may: skip the writeback operation, writeback an unknown value, treat the operation as a no-op or consider the instruction undefined. For stores with overlaps, it may: store the original value of the overlapping register (before writeback), store an unknown value, or act as a no-op or undefined opcode.

As an example, ldp x1, x0, [x1], #-8 (2180ffa8) accesses a pair of words at memory address x1 then decrements x1 by 8. However, this overlaps with the use of x1 as a data register, holding the first loaded value. The ASL lifter is explicitly configured to skip the writeback, i.e., the decrement, keeping the loaded value. Remill does the opposite, overwriting the loaded value with the decremented address. The correct outcome here is ultimately dependent on the hardware implementation, potentially leading to either interpretation being correct. However, the ASLp approach makes such configuration explicit, whereas the behaviour is silently assumed by Remill.

`strb w0, [x0], #-1` (`00f41f38`) is a more concerning example, featuring an overlapping post-indexed store. The ASL lifter stores the original value of `w0` to the memory address `x0`, and then decrements `x0` by 1. Remill, however, performs the store and skips the writeback operation. According to the specification, this is not an allowed behaviour, indicating an invalid lifting.

These inconsistencies affect overlapping `ldp`, `ldpsw`, `strb`, and `strh` opcodes. These make up the 88 mismatching memory instructions. Other variants, notably `ldr` and `str`, did not have these discrepancies.

smaddl: When lifting fused multiply-add instructions, Remill includes "no signed wrap" annotations on the corresponding LLVM operations, returning poison in the case of signed overflow. However, this is unnecessary, as these ARMv8 instructions are specified to implement standard truncation behaviour in such scenarios. This accounts for all 15 of Remill's mismatched integer instructions.

br xzr: The `e0031fd6` opcode should disassemble to `br xzr`, semantically a jump to address 0. However, the semantics produced by Remill jump to the value held in `sp` instead. This is the only mismatch on branch instructions.

C. Bugs found

We reported the above inconsistencies to the relevant projects. Additionally, we identified bugs in Alive2:

- There was a soundness issue caused by an incorrect peephole optimisation in an integer comparison operation.
- Type punning in LLVM (loading from a pointer that stores a different type) is defined to return poison, but this was not implemented by Alive's semantics due to an incorrect optimisation.

These Alive2 bugs were reported and fixed by its maintainers during the course of our work.

V. Related Work

The field of binary decompilation and analysis is vast and represents decades of ongoing research. Similar to this work, standalone hardware architecture models have been developed for use in decompilation tools. Notable examples of this include Remill and GHIDRA's sleigh library [30]. While these libraries are immediately applicable to many decompilation tasks and support multiple ISAs, they lack a formal foundation. For instance, the formal semantics of their output language is unclear [41], [42] and the derivation of their models is rarely documented.

Various approaches to decompilation have been proposed that build on trustworthy architecture models. For instance, *Decompilation into Logic* [27] leverages architecture encodings specified within the HOL4 theorem prover [43]. These specifications are simplified, using the theorem prover's rewriting engines, to derive concise semantics for individual instructions. Similar techniques have been applied in other theorem provers [44], [45], [28], potentially leveraging symbolic execution to further improve the rewriting process [46]. The produced representations are suitable for reasoning within theorem provers, but cannot be easily integrated into decompilation tools, due to the use of abstract logic constructs in their results. An exception to this is HolBA [47], which successfully coverts these logic constructs into an imperative language, at the expense of significantly slower lifting times.

Existing work has explored the application of partial evaluation to decompilation. For instance, Gómez-Zamalloa et al. [48] develop a lifter from Java bytecode to Prolog via the partial evaluation of an interpreter for the former written in the latter. While their approach features similar implementation details to ours, such as careful inlining configuration, they consider a wholistic perspective, partially evaluating whole programs with the intention to directly validate the residual Prolog representation.

In recent years, attention has turned to the validation of decompilers. The concept of differential testing for lifted IRs was first explored by Kim et al. [13] with the MeanDiff tool. In this work, three unproven x86 lifters (PyVEX [49], BAP [5], and BINSEC [50]) were compared to each other using an equivalent approach to Section IV. While similar to our work, we benefit from a trustworthy ISA semantics for ARMv8, providing a higher level of assurance and greater instruction coverage.

Dasgupta et al. [26] apply a similar technique to validate instruction semantics derived from Remill with respect to their own model of x86-64, written in K [51]. The validated Remill instruction semantics are subsequently concatenated to decompile the entire program. It is unclear whether instruction semantics could be directly derived from their trusted model, as done in Section III.

VI. Conclusion

This work documents the wide-reaching benefits and applications of a canonical, accurate, and comprehensive ISA specification. With architecture specifications provided by ARM, we can provide a solid, trustworthy foundation for binary disassembly and lifting. Partial evaluation allows us to summarise these semantics into a minimal IR, for easy integration with other analysis tools, with our BAP ARMv8 plugin as an example. Moreover, we demonstrate that the produced representation is no more complex than that of existing, manually encoded lifters. We also show the effectiveness of formal semantics for validating the results of established binary lifters using existing LLVM analysis and reasoning tools. The amount of errors we found, in some cases central to whole families of opcodes, demonstrates the importance of a reference semantics that can be instrumented for automated checkers. Altogether, this leads to more trustworthy binary lifting with promising future applications, providing trustworthiness in a field which demands high levels of assurance.

Acknowledgements: We would like to thank James Paterson and Andrew Brown, who both contributed to the implementation of this work.

REFERENCES

[1] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984. [Online]. Available: https://doi.org/10.1145/358198.358210

[2] V. D'Silva, M. Payer, and D. X. Song, "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*. IEEE Computer Society, 2015, pp. 73–87. [Online]. Available: https://doi.org/10.1109/SPW.2015.33

[3] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie, "Secure compilation of constant-resource programs," in *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–12. [Online]. Available: https://doi.org/10.1109/CSF51468.2021.00020

[4] G. Balakrishnan and T. W. Reps, "WYSINWYX: What You See Is Not What You eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[6] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. IEEE Computer Society, 2017, pp. 8–9. [Online]. Available: https://doi.org/10.1109/SecDev.2017.14

[7] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[8] N. Ramsey and M. F. Fernandez, "Specifying representations of machine instructions," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 492–524, 1997. [Online]. Available: https://doi.org/10.1145/256167.256225

[9] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, 1998, pp. 126–133. [Online]. Available: https://doi.org/10.1109/WPC.1998.693332

[10] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: attacking ARM pointer authentication with speculative execution," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 685–698. [Online]. Available: https://doi.org/10.1145/3470496.3527429

[11] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[12] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.35

[13] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 353–364.

[14] ARM, "ARM Architecture Reference Manual for A-profile architecture," 2023.

[15] Intel Corporation, "Intel A64 and IA-32 Architectures Software Developer's manual," 2023.

[16] A. Reid, "Who guards the guards? Formal validation of the Arm v8-M architecture specification," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 88:1–88:24, 2017. [Online]. Available: https://doi.org/10.1145/3133912

[17] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: automatically learning the x86-64 instruction set," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 237–250. [Online]. Available: https://doi.org/10.1145/2908080.2908121

[18] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290384

[19] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: https://doi.org/10.1145/2535838.2535841

[20] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, 2009. [Online]. Available: https://doi.org/10.1007/s10817-009-9155-4

[21] K. Nienhuis, A. Joannou, T. Bauereiss, A. C. J. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1003–1020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00055

[22] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, "Verified security for the Morello capability-enhanced prototype Arm architecture," in *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, ser. Lecture Notes in Computer Science, I. Sergey, Ed., vol. 13240. Springer, 2022, pp. 174–203. [Online]. Available: https://doi.org/10.1007/978-3-030-99336-8_7

[23] D. Lockhart, B. Ilbeyi, and C. Batten, "Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*. IEEE Computer Society, 2015, pp. 256–267. [Online]. Available: https://doi.org/10.1109/ISPASS.2015.7095811

[24] A. C. J. Fox, "Formal specification and verification of ARM6," in *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, ser. Lecture Notes in Computer Science, D. A. Basin and B. Wolff, Eds., vol. 2758. Springer, 2003, pp. 25–40. [Online]. Available: https://doi.org/10.1007/10930755_2

[25] A. Reid, "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*, ser. FMCAD '16. Austin, Texas: FMCAD Inc, 2016, p. 161–168.

[26] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "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*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1133–1148. [Online]. Available: https://doi.org/10.1145/3314221.3314601

[27] M. O. Myreen, M. J. C. Gordon, and K. Slind, "Decompilation into logic - Improved," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 78–81. [Online]. Available: https://ieeexplore.ieee.org/document/6462558/

[28] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell, "Islaris: Verification of machine code against authoritative ISA semantics," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming*

Language Design and Implementation, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 825–840. [Online]. Available: https://doi.org/10.1145/3519939.3523434

[29] Trail of Bits, “lifting-bits/remill: Library for lifting machine code to LLVM bitcode,” https://github.com/lifting-bits/remill, 2022.

[30] National Security Agency, “Sleigh,” https://github.com/NationalSecurityAgency/ghidra, 2022.

[31] Avast Software, “avast/retdec: RetDec is a retargetable machine-code decompiler based on LLVM.” https://github.com/avast/retdec, 2022.

[32] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr, “Alive2: bounded translation validation for LLVM,” in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 65–79. [Online]. Available: https://doi.org/10.1145/3453483.3454030

[33] ARM, “ARM developer exploration tools,” 2023.

[34] A. Reid, “ARM's architecture specification language,” Aug 2016. [Online]. Available: https://alastairreid.github.io/specification_languages/

[35] ——, “Using ASLi with Arm's V8.6-A ISA specification,” Jan 2020. [Online]. Available: https://alastairreid.github.io/using-asli/

[36] Y. Futamura, “Partial computation of programs,” in *RIMS Symposia on Software Science and Engineering*. Springer Berlin Heidelberg, 1983, pp. 1–35. [Online]. Available: https://doi.org/10.1007/3-540-11980-9_13

[37] E. Sumii and N. Kobayashi, “A hybrid approach to online and offline partial evaluation,” *High. Order Symb. Comput.*, vol. 14, no. 2-3, pp. 101–142, 2001. [Online]. Available: https://doi.org/10.1023/A:1012984529382

[38] W. M. McKeeman, “Differential testing for software,” *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[39] LLVM Project, “LLVM language reference manual,” https://llvm.org/docs/LangRef.html, 2022.

[40] Trail of Bits, “lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode,” https://github.com/lifting-bits/mcsema, 2022.

[41] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, “A formal semantics for P-Code,” in *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Lal and S. Tonetta, Eds., vol. 13800. Springer, 2022, pp. 111–128. [Online]. Available: https://doi.org/10.1007/978-3-031-25803-9_7

[42] L. Li and E. L. Gunter, “K-LLVM: A relatively complete semantics of LLVM IR,” in *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:29. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[43] K. Slind and M. Norrish, “A brief overview of HOL4,” in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, ser. Lecture Notes in Computer Science, O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds., vol. 5170. Springer, 2008, pp. 28–32. [Online]. Available: https://doi.org/10.1007/978-3-540-71067-7_6

[44] I. Roessle, F. Verbeek, and B. Ravindran, “Formally verified big step semantics out of x86-64 binaries,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, A. Mahboubi and M. O. Myreen, Eds. ACM, 2019, pp. 181–195. [Online]. Available: https://doi.org/10.1145/3293880.3294102

[45] F. Verbeek, P. Olivier, and B. Ravindran, “Sound C code decompilation for a subset of x86-64 binaries,” in *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*, ser. Lecture Notes in Computer Science, F. S. de Boer and A. Cerone, Eds., vol. 12310. Springer, 2020, pp. 247–264. [Online]. Available: https://doi.org/10.1007/978-3-030-58768-0_14

[46] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell, “Isla: Integrating full-scale ISA semantics and axiomatic concurrency models,” in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 303–316. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_14

[47] A. Lindner, R. Guanciale, and R. Metere, “TrABin: Trustworthy analyses of binaries,” *Sci. Comput. Program.*, vol. 174, pp. 72–89, 2019. [Online]. Available: https://doi.org/10.1016/j.scico.2019.01.001

[48] M. Gómez-Zamalloa, E. Albert, and G. Puebla, “Modular decompilation of low-level code by partial evaluation,” in *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*. IEEE Computer Society, 2008, pp. 239–248. [Online]. Available: https://doi.org/10.1109/SCAM.2008.35

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware,” 2015.

[50] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, “The BINCOA framework for binary code analysis,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 165–170. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_13

[51] G. Rosu and T. Serbanuta, “An overview of the K semantic framework,” *J. Log. Algebraic Methods Program.*, vol. 79, no. 6, pp. 397–434, 2010. [Online]. Available: https://doi.org/10.1016/j.jlap.2010.03.012

APPENDIX

A. ASL Interpreter Bugs

We identified multiple bugs in the existing ASL interpreter in the process of testing our partial evaluator. Note that the ASL interpreter, as open-sourced by ARM, offered no guarantee of correctness. We fixed these issues in our partial evaluation fork[1].

- Simultaneous assignments to multiple record fields were evaluated in the wrong order. For example, `PSTATE.[N,Z,C,V] = nzcv` should assign bit 0 of `nzcv` to the `V` field, bit 1 to `C` and so on. Instead, the reverse order was used, e.g. bit 0 was incorrectly assigned to `N`.
- ASL defines reference parameters which allow functions to modify their arguments directly. This was implemented in the parser but its semantics were not handled in the evaluation code, instead treating them as regular parameters.
- The interpreter's evaluation function would become stuck after breaking from a ‘while’ loop, due to a parsing ambiguity.