Australian Government
Department of Defence
Defence Science and Technology Group

Lift-off: Trustworthy ARMv8 semantics from formal specifications

FMCAD 2023, October 2023

Kait Lam and **Nicholas Coughlin**
DST Group / The University of Queensland

Motivation

- Compilers can't be trusted.
  - In particular, not for security-critical code.
  - Fine details such as running time and memory clearing are essential.

Motivation

- Compilers can't be trusted.
  - In particular, not for security-critical code.
  - Fine details such as running time and memory clearing are essential.
- Off-the-shelf software is in use and needs to be analysed.
  - We cannot influence their development but must verify their compatibility and suitability in secure environments.

Motivation

- Compilers can't be trusted.
    - In particular, not for security-critical code.
    - Fine details such as running time and memory clearing are essential.
- Off-the-shelf software is in use and needs to be analysed.
    - We cannot influence their development but must verify their compatibility and suitability in secure environments.
- Binary analysis becomes necessary when you require absolute assurance.

Binary Analysis

- Requires detailed architecural-level semantics.
- Existing decompilers use hand-written semantics with limited correctness arguments.
  - Often, written in custom languages with unclear semantics (e.g. Ghidra's P-Code, Valgrind's VEX IR).
- Formal architecture models have shown promise (e.g. Sail architecture definition language, Dasgupta et al. for x86-64 in $\mathbb{K}$).
- There is a gap between the formal models and decompilation tools, due to differences in expressivity and abstractions.

Background

- Arm publishes machine-readable architecture (MRA) files with ISA semantics (Reid 2016).
- Expressed in Arm's Architecture Specification Language (ASL).
- Previous work has developed ASLi, an open-source parser and interpreter for ASL.
- We extend this into *ASLp*, a partial evaluator for ASL based on ASLi.

Partial evaluation

- Given some known initial state, reduces the program by propagating this static information.
- For unknown values or side-effecting operations, it emits a *residual program* to compute these.

Partial evaluation

- Given some known initial state, reduces the program by propagating this static information.
- For unknown values or side-effecting operations, it emits a *residual program* to compute these.
- Correctness: Executing residual program behaves identically to the original program with the specified initial state.

Example

Consider `add x1, x2, x3, LSL 4` = 41 10 03 8B.

```
0x    8    b    0    3    1    0    4    1
   1000 1010 0000 0011 0001 0000 0100 0001
   xxx0 1011 xx0x xxxx xxxx xxxx xxxx xxxx
   │││       └┤ └────┤ └─────┤└────┤└────┤
 sf┘│└S    shift      Rm    imm6    Rn    Rd
   op
```

Example

Consider `add x1, x2, x3, LSL 4` = 41 10 03 8B.

```
0x    8    b    0    3    1    0    4    1
   1000 1010 0000 0011 0001 0000 0100 0001
   xxx0 1011 xx0x xxxx xxxx xxxx xxxx xxxx
    ││ │        └┤  └────┤ └──────┤└────┤└────┤
  sf┘│ └S   shift     Rm    imm6    Rn    Rd
     op
```

```
__encoding aarch64_integer_arith_add_sub_shiftedreg
__opcode 'xxx01011 xx0xxxxx xxxxxxxx xxxxxxxx'
__instruction_set A64
__field sf    31 +: 1    __field Rm   16 +: 5
__field op    30 +: 1    __field imm6 10 +: 6
__field S     29 +: 1    __field Rn    5 +: 5
__field shift 22 +: 2    __field Rd    0 +: 5
```

Example

```
__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 = 4;
```

Example

Known: { datasize, n, m, d, shift_type, sub_op, setflags, ... }

```
__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;
```

Example

Known: `{ datasize, n, m, d, shift_type, sub_op, setflags, ... }`

```
__execute
  bits(64) 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;
```

Example

Known: `{ operand1, datasize, n, m, d, shift_type, ... }`

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  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;
```

Example

Known: { operand2, operand1, datasize, n, ... }

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  bits( 64 ) operand2 =
    X[3][0 +: 60] : '0000' ;
  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;
```

Example

Known: { operand2, operand1, datasize, n, ... }

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  bits(64) operand2 =
    X[3][0 +: 60] : '0000';
  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;
```

Example

Known: `{ carry_in, operand2, operand1, datasize, n, ... }`

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  bits(64) operand2 =
    X[3][0 +: 60] : '0000';
  bits(4) nzcv;
  bit carry_in;
  if false 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;
```

Example

Known: { `result, carry_in, operand2, operand1, datasize, n, ...` }

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  bits(64) operand2 =
    X[3][0 +: 60] : '0000';
  bits(4) nzcv;
  bit carry_in;
  if false then
    operand2 = NOT(operand2);
    carry_in = '1';
  else
    carry_in = '0';
  result = X[2] + (X[3][0 +: 60] : '0000');
  if setflags then
    PSTATE.[N,Z,C,V] = nzcv;
  X[d] = result;
```

Example

Known: { `result, carry_in, operand2, operand1, datasize, n, ... }`

```
__execute
  bits(64) result;
  bits(64) operand1 = X[2];
  bits(64) operand2 =
    X[3][0 +: 60] : '0000';
  bits(4) nzcv;
  bit carry_in;
  if false then
    operand2 = NOT(operand2);
    carry_in = '1';
  else
    carry_in = '0';
  result = X[2] + (X[3][0 +: 60] : '0000');
  if false then
    PSTATE.[N,Z,C,V] = nzcv;
  X[1] = X[2] + (X[3][0 +: 60] : '0000');
```

Example

```
X[1] = X[2] + (X[3][0 +: 60] : '0000');
```

Example

```
X[1] = X[2] + (X[3][0 +: 60] : '0000');
```

```
ASLp> :sem A64 0x8b031041
Decoding instruction A64 8b031041
__array _R [ 1 ] = add_bits.0 {{ 64 }} (
  __array _R [ 2 ] [ 0 +: 64 ],
  append_bits.0 {{ 60,4 }} (
    __array _R [ 3 ] [ 0 +: 60 ],
    '0000'
  )
) [ 0 +: 64 ] ;
```

Program transformations

- Conventional transformations:
  - Constant propagation.
  - Dead code / unused variable elimination.
  - Common subexpression elimination.
  - Function inlining.
  - Loop unrolling.
- ASL syntax desugaring (e.g. simultaneous assignments).
- Expression simplification (e.g. for arithmetic and bit slices).
- Integer & real to bitvector conversions.

Simplification rules

- Integer to bitvector conversions.
  - ASL has an arbitrary-precision `integer` type.
  - However, we can calculate the number of bits needed to maintain precision.
  - Allows integration with any tool supporting bitvectors.
- Real to floating-point conversion.
  - ASL's `real` type approximates $\mathbb{R}$.
  - Represented as a rational in the interpreter.
  - Since this is only used in floating-point contexts, we emit these as float primitives for downstream handling.

Implementation

Duplicate and modify ASLi's interpreter to extract semantics.

- Interpreter:

$$\text{eval} : \textit{stmt list} \rightarrow \textit{state} \rightarrow \textit{state}$$

  where $\textit{state} = \textit{var} \rightarrow \textit{val}$.

- Disassembler:

$$\text{aslp} : \textit{stmt list} \rightarrow \textit{symstate} \rightarrow \textit{stmt list}$$

  where

$$\begin{aligned} \textit{symstate} &= \textit{var} \rightarrow \textit{symval}, \text{ and} \\ \textit{symval} &= \mathrm{Val}\ \textit{val} \mid \mathrm{Sym}\ \textit{expr} \mid \mathrm{Unknown}. \end{aligned}$$

Correctness

$$\begin{tikzcd}
prog_{orig} \arrow[r, "\textbf{aslp}"'] \arrow[d, "\text{eval}"] & prog_{resid} \arrow[d, "\text{eval}"'] \\
state_1 \arrow[r, dashed, dash, bend right, "=?"'] & state_2
\end{tikzcd}$$

Testing

- We need to test this property to validate ASLp.
- Test cases compute matching concrete and symbolic states, run aslp and eval, then compare the resulting states for equality.
- Automatically generate these by enumerating opcodes, varying register operands, mode flags, and other parameters.

```
add x1, x1, x1
     ⋮
add x30, x30, x30
```

Application: BAP

- BAP (the CMU Binary Analysis Platform) supports aarch64 by encoding semantics in its Primus Lisp DSL.
  - Encoding is done manually and incomplete.
- We build a BAP/ASLp plugin which connects ASLp into BAP's knowledge base.
- Substantially more complete and reliable than BAP's existing Primus Lisp semantics.

Application: Evalution of lifters

- Compare ASLp to two lifters which target LLVM IR:
  - RetDec, by Avast.
  - Remill, by Trail of Bits, used in McSema and Anvill.
- Use LLVM as a common language for semantics.
- Alive2 translation validation (Lopes et al. 2021) to check equivalence of lifter outputs.
- If they do not match, we manually compare their outputs against the Arm specification.
- Note: Alive2 is very precise with its LLVM semantics (particularly undef and poison).

LLVM lifters evaluation

$$\begin{tikzcd}
 & \text{LLVM IR (ASL)} \arrow[r, "\textit{translator}"] & \text{LLVM IR} \arrow[d, dashed, leftrightarrow, "\textit{Alive2}"'] \arrow[dd, dashed, leftrightarrow, bend left, "\textit{Alive2}"] \\
\text{ARMv8 bytecode} \arrow[ur, bend left, "\textit{ASLp + llvm-translator}"] \arrow[r, "\textit{RetDec}"] \arrow[dr, bend right, "\textit{Remill}"'] & \text{LLVM IR (RetDec)} \arrow[r, "\textit{translator}"] & \text{LLVM IR} \\
 & \text{LLVM IR (Remill)} \arrow[r, "\textit{translator}"] & \text{LLVM IR}
\end{tikzcd}$$

Evaluation: RetDec

		RetDec			
	Count	Equiv.	Mismatch	Timeout	Unsupp.
Branch	162	74	9	10	69
Integer	14442	10147	608	415	3272
Memory	6414	3864	618	0	1932
Vec. binary	19170	264	426	0	18480
Vec. unary	1098	9	153	0	936
Total	41286	14358	1814	425	24689
	100.0%	34.8%	4.4%	1.0%	59.8%

Evaluation: RetDec

	Count	RetDec Equiv.	Mismatch	Timeout	Unsupp.
Branch	162	74	9	10	69
Integer	14442	10147	608	415	3272
Memory	6414	3864	618	0	1932
Vec. binary	19170	264	426	0	18480
Vec. unary	1098	9	153	0	936
Total	41286	14358	1814	425	24689
	100.0%	34.8%	4.4%	1.0%	59.8%

- Mismatches occur in variants of: `adcs`, `add`, `and`, `ldr`, `str`, `sdiv`, `udiv`.

RetDec inaccuracies

- In RetDec, inaccuracies in flag computation, shifting, handling LLVM poison, and sign extension.
- SIMD is only implemented as scalar operations, not vectorised.

RetDec `adcs`

```
adcs xzr, x0, xzr
```

RetDec adcs

`adcs xzr, x0, xzr`

Bug in signed overflow flag when $\texttt{X0} = \texttt{INT\_MAX} - 1$ and $\texttt{CF} = 1$.

RetDec: $\texttt{VF} = 1$.   ARM spec / ASLp: $\texttt{VF} = 0$.

```
Mismatch in pointer(non-local, block_id=3, offset=0)
Source value: #x01
Target value: #x00
```

RetDec adcs

```
adcs xzr, x0, xzr
```

Bug in signed overflow flag when `X0` $=$ `INT_MAX` $- 1$ and `CF` $= 1$.

RetDec: `VF` $= 1$. ARM spec / ASLp: `VF` $= 0$.

```
Mismatch in pointer(non-local, block_id=3, offset=0)
Source value: #x01
Target value: #x00
```

```
RetDec:
i1 %_CF = #x1 (1)
i1 %_VF = #x0 (0)
i64 %_X0 = #x7ffffffffffffffe
           (9223372036854775806)
  >> Jump to %entry
[...]
i1 %7 = #x1 (1)
```

RetDec adcs

```
adcs xzr, x0, xzr
```

Bug in signed overflow flag when $\texttt{X0} = \texttt{INT\_MAX} - 1$ and $\texttt{CF} = 1$.

RetDec: $\texttt{VF} = 1$. ARM spec / ASLp: $\texttt{VF} = 0$.

```
Mismatch in pointer(non-local, block_id=3, offset=0)
Source value: #x01
Target value: #x00
```

```
RetDec:
i1 %_CF = #x1 (1)
i1 %_VF = #x0 (0)
i64 %_X0 = #x7ffffffffffffffe
           (9223372036854775806)
  >> Jump to %entry
[...]
i1 %7 = #x1 (1)
```

```
ASLp:
i1 %_CF = #x1 (1)
i1 %_VF = #x0 (0)
i64 %_X0 = #x7ffffffffffffffe
           (9223372036854775806)
  >> Jump to %stmts_root
[...]
i1 %6 = #x0 (0)
```

RetDec adcs

```
define void @retdec_ll() {
  ; ...
entry:
  %cf_load = zext i1 %CF to i64
  %1 = add i64 %X0, %cf_load
  %4 = add i64 %1, %cf_load  ; <- (!)

  %5 = xor i64 %X0, -1
  %6 = and i64 %4, %5
  %7 = icmp slt i64 %6, 0
  store i1 %7, ptr @VF, align 1
  ret void
}
```

RetDec adcs

```
define void @retdec_ll() {
  ; ...
entry:
  %cf_load = zext i1 %CF to i64
  %1 = add i64 %X0, %cf_load
  %4 = add i64 %1, %cf_load  ; <- (!)

  %5 = xor i64 %X0, -1
  %6 = and i64 %4, %5
  %7 = icmp slt i64 %6, 0
  store i1 %7, ptr @VF, align 1
  ret void
}
```

```
define void @aslp_ll() {
  ; ...
stmts_root:
  %cf_64 = zext i1 %CF to i64
  %cf_65 = zext i1 %CF to i65

  %2 = add i64 %X0, %cf_64
  %add_64 = sext i64 %2 to i65

  %4 = sext i64 %X0 to i65
  %add_65 = add nsw i65 %4, %cf_65

  %6 = icmp ne i65 %add_65, %add_64
  store i1 %6, ptr @VF, align 1
  ret void
}
```

Evaluation: Remill

	Count	Remill Equiv.	Mismatch	Timeout	Unsupp.
Branch	162	120	1	37	4
Integer	14442	12058	15	1238	1131
Memory	6414	5057	88	0	1269
Vector binary	19170	2997	0	0	16173
Vector unary	1098	333	0	0	765
Total	41286	20565	104	1275	19342
	100.0%	49.8%	0.3%	3.1%	46.8%

Evaluation: Remill

	Count	Remill Equiv.	Mismatch	Timeout	Unsupp.
Branch	162	120	1	37	4
Integer	14442	12058	15	1238	1131
Memory	6414	5057	88	0	1269
Vector binary	19170	2997	0	0	16173
Vector unary	1098	333	0	0	765
Total	41286	20565	104	1275	19342
	100.0%	49.8%	0.3%	3.1%	46.8%

Remill inaccuracies

- Minor mismatches occur in variants of: `smaddl`, `sdiv`, and writeback-conflict `ldp`, `ldpsw` and `strb`, `strh` (e.g. `strb w1, [x1, #1]!`).
- Very few inaccuracies in Remill's semantics and none that were major.
- One case of undefined behaviour (overflow) within `sdiv` in the case of $(-2^{63})/(-1)$.
- Differs from ASLp in some minor constrained-unpredictable behaviours.

Bugs found

- While testing, we found some bugs in the tools used.
- In ASLi, small but impactful bugs relating to register field assignment, reference parameters, and parsing.
- In Alive2, a soundness bug due to overflow, and type-punning was not defined as poison.
- Issues have been reported to upstream maintainers.

Conclusion

- We have developed ASLp, a tool for extracting ARMv8 semantics from ARM's machine-readable architecture.
- To demonstrate its usefulness, we integrate it with BAP and validate the semantics of existing lifters against its output.
- Work continues to integrate this into a full toolchain for binary information flow analysis and rely/guarantee reasoning.
- `https://github.com/UQ-PAC/aslp`