Automated Synthesis of Symbolic Instruction Encodings from I/O Samples

Patrice Godefroid
Microsoft Research

Ankur Taly
Stanford University
Need for Symbolic Instruction Encodings

Symbolic Execution is a key component of precise binary program analysis tools

- SAGE, BitBlaze, BAP, etc.
- Static analysis tools

11: mov eax, inp1
    mov cl, inp2
    shl eax, cl
    jnz 12
    jmp  13
12: div    ebx, eax
// Is this safe ?
// Is eax != 0 ?
13:  ...
Problem: Symbolic Instruction Encoding

Problem: Given a processor and an instruction name, symbolically describe the input-output function for the instruction

- Express the encoding as bit-vector constraints (ex: SMT-Lib format)
So far, only manual solutions...

- From the instruction architecture manual (X86, ARM, …) implemented by the processor

**SHLD—Double Precision Shift Left (Continued)**

**Operation**

COUNT ← COUNT MOD 32;
SIZE ← OperandSize

IF COUNT = 0
  THEN
  no operation
ELSE
  IF COUNT ≥ SIZE
    THEN (* Bad parameters *)
      DEST is undefined;
      CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
      CF ← BIT[DEST, SIZE – COUNT];
      (* Last bit shifted out on exit *)
      FOR i ← SIZE – 1 DOWNTO COUNT
        DO
          Bit(DEST, i) ← Bit(DEST, i – COUNT);
          OD;
      FOR i ← COUNT – 1 DOWNTO 0
        DO
          BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
          OD;
      FI;

X86 spec for SHLD

Limitations:

- **Tedious, expensive**
  - X86 has more than 300 unique instructions, each with ~10 OPCodes, 2000 pages

- **Error-prone**
  - Written in English, many corner cases

- **Imprecise**
  - Spec is often under-specified

- **Partial**
  - Not all instructions are covered

- **Can we trust the spec?**
Here: **Automated Synthesis Approach**

**Goals:**

- **As automated** as possible so that we can boot-strap a symbolic execution engine on an arbitrary instruction set
  - But **search spaces are enormous** (ex: $2^{2048}$ 8-bit to 8-bit functions!)

- **As precise** as possible: $f$ must capture behavior for inputs outside the partial truth table $S$ as well
  - But **exhaustive sampling is impossible** ($32 \times 32$ bits = $2^{64}$ inputs!)
Challenge: Enormous Search Space

How can we reduce the search space?

Solution: Templates (Program Sketching, Oracle-guided component synthesis)

- A template is a parametric function $T(c_1, \ldots, c_n, i, o)$ with certain unknown parameters(coeffs) $c_1, \ldots, c_n$
- A concretization of the template is obtained by substituting specific values for the coefficients
- Restrict the search space to all possible concretizations of $T(c_1, \ldots, c_n, i, o)$

Warning: this fails if template cannot express the actual function
Designing Templates

Design Principles

• Template $T(c_1, ..., c_n, i)$ must be expressible using bit-vector constraints (for compactness requirements)

• Must capture the common structure of a set of instructions
  - A template abstracts a set of concrete instructions

• Must not have too much freedom => enormous search space

• Must not have too little freedom => cover too few instructions

Architecture-specification is useful

• help in grouping instructions based on similar behaviors

• help in capturing the common structure
Intel X86 Instruction Set

- Complex Instruction-set Architecture (CISC)
  - 300+ unique instructions, each with ~10 OPCodes

Assumption: behavior is independent of where the operands come from
We want a symbolic function from \( i_1, i_2, i_3 \) to \( res_1, res_2 \) and the E-FLAGS
This Work: ALU Instructions from X86

• Why ALU? Current bit-vector solvers provide the necessary building blocks

• 46 relevant unique instructions (irrelevant instr: MOV, LOAD, ...)  
  - Each has approx. 6 to 21 instances (8/16/32 bits, 2Result + 5 EFLAGS)

• Based on the spec, we divide ALU instructions into 3 groups:  
  - Bit-shift instructions (BS): SHL, SHR, ROL, ...  
  - Bit-wise instructions (BW): AND, OR, NOT, ...  
  - Arithmetic instructions (ARI): ADD, MUL, IMUL ...

• We define 2 templates (Result + EFlags) for each group  
  - templates are parametric on the register size (8/16/32)  
  - In total 3*2 = 6 templates to cover 534 ALU instruction instances!
State-of-the-art: Distinguishing Input Synthesis

[Initial I/O Samples] -> SYN -> Function -> VERIFY

- Incorrect Template
- Failed Samples

If YES, sample input

PASS -> YES

DSample

DInput?

FAIL -> Random Testing

FAIL

[DSample ?] -> YES

[Jha-Gulwani-Seshia-Tiwari, ICSE’2010]
## Problem: too slow! (or OOM)

Intel XEON 3.07ghz processor, 8GB RAM

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$n_{syn}$</th>
<th>$n_{ver}$</th>
<th>S-Iters</th>
<th>D-Iters</th>
<th>Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHL32</td>
<td>10</td>
<td>100</td>
<td>31</td>
<td>4</td>
<td>24,168,853</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1000</td>
<td>31</td>
<td>3</td>
<td>20,107,259</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10000</td>
<td>31</td>
<td>1</td>
<td>11,754,805</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>100</td>
<td>21</td>
<td>3</td>
<td>16,877,223</td>
</tr>
<tr>
<td>SHL32</td>
<td>100</td>
<td>1000</td>
<td>22</td>
<td>3</td>
<td>17,577,444</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>10000</td>
<td>20</td>
<td>4</td>
<td>21,620,686</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>4,382,472</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>4,456,942</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>10000</td>
<td>1</td>
<td>1</td>
<td>4,707,855</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>10000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Z3 runs out of memory in the DInput phase
New Approach: Smart Sampling

- The distinguishing-input check is expensive, can we eliminate it?
- **Intuition:**
  - 2 points are enough to uniquely determine coefficients of a linear template,
  - 3 points are enough for a circle template
  - ...

**Smart Inputs:** A set of inputs \( I \) is said to be **smart** for a template \( T \) if for all samples obtained using the inputs, there exists a unique coefficient up to logical equivalence, for which the template respects the samples.

**Ex:** there are 16 bitwise operations (functions from 1x1 bits to 1 bit)

**What are the smart inputs?**

**Answer:** Inputs must have 4 bitwise pairs \((0,0),(0,1),(1,0),(1,1)\)

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
= 12
\]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
= 10
\]

Smart Inputs for **Bit-wise** template is the singleton \((12,10)\)!
Templates Summary

- $n$ is the size of the input and output bit-vectors (8, 16, 32)

<table>
<thead>
<tr>
<th>Template</th>
<th>Search space</th>
<th>Smart sample size</th>
<th>Circuit size [RESULT]</th>
<th>Circuit size [EFLAGS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-shift</td>
<td>$(2n+2)^{32n}$</td>
<td>32 $(\log(n)+2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bit-wise</td>
<td>16</td>
<td>1</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>$21 2^{2n}$</td>
<td>3</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Synthesis with Templates and Smart Sampling

Synthesis is much faster!

<table>
<thead>
<tr>
<th>Instr</th>
<th>Exhaust</th>
<th>DInput</th>
<th>Smart Sampl</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND8</td>
<td>26,478</td>
<td>48 (÷551)</td>
<td>3 (÷16)</td>
</tr>
<tr>
<td>AND16</td>
<td>-</td>
<td>55</td>
<td>4 (÷14)</td>
</tr>
<tr>
<td>AND32</td>
<td>-</td>
<td>71</td>
<td>4 (÷18)</td>
</tr>
<tr>
<td>MUL8</td>
<td>32,462</td>
<td>189 (÷172)</td>
<td>17 (÷11)</td>
</tr>
<tr>
<td>MUL16</td>
<td>-</td>
<td>609</td>
<td>20 (÷30)</td>
</tr>
<tr>
<td>MUL32</td>
<td>-</td>
<td>1,997</td>
<td>29 (÷68)</td>
</tr>
<tr>
<td>SHL8</td>
<td>181,857</td>
<td>21,501 (÷9)</td>
<td>867 (÷25)</td>
</tr>
<tr>
<td>SHL16</td>
<td>-</td>
<td>250,105</td>
<td>8,064 (÷31)</td>
</tr>
<tr>
<td>SHL32</td>
<td>-</td>
<td>4,382,472</td>
<td>303,970 (÷14)</td>
</tr>
</tbody>
</table>

new “smart sampling” synthesis algorithm takes <2 hours with Z3 to synthesis functions for 534 x86 instruction instances
Lessons Learned

• Uncovered behaviors for “undefined” cases

Ex: \texttt{ADD/SUB} \textbf{: Overflow Flag (OF)}

- X86 Spec: OF is set “according to the result”
- Intel XEON3.7: Is set only when XOR of MSB of the two inputs is negation of MSB of output!

• Discrepancies found compared to spec

Ex: \texttt{IMUL[8]} 65, 254

- X86 Spec: OF is set to 0
- Intel XEON3.7: OF is set to 1
Discrepancies Found Across Machines

- X86 Spec and Intel XEON3.7 and Core2 (left laptop): instructions ROL, SHL, SHR do not set OF if count argument is not 1

- Intel I7-2620M 2.8ghz (right laptop): OF is set to 1 even for certain cases where count argument > 1
Current Limitations

- Instructions like `CMPXCHG` set EFLAGS according to an intermediate value that is throw away at the end
  - difficult to construct a template for such instructions

- Instructions like `DIV, IDIV` crash on certain inputs (example: when quotient is > register range)
  - these pre-conditions are currently hard-wired in the system
  - in future we would like to synthesize them automatically

- Instruction like `SHL, SHR` leave ZF, PF, SF “unchanged” when count operand = 0
  - therefore ZF, PF, SF must also be inputs to the functions
  - currently we sample all instructions after clearing all flags
Conclusion

• Automated Synthesis of Symbolic Instruction Encodings for X86-ALU instructions
  - 6 abstract instruction templates
  - for 534 x86 ALU instructions (8/16/32bits, outputs, EFLAGS)
  - new “smart sampling” synthesis algorithm takes <2 hours with Z3
  - building blocks are bit-vector constraints (SMT-lib format)
  - synthesis against specific x86 processor as I/O oracle:

• Future work: x64, AMD64, ARM, SIMD instructions, floating point instructions,...
Related Work

- **Deriving Abstract Transfer Functions for Embedded CPUs**
  - Ex: [HOIST, Regehr et al.]
  - Like us, but small CPUs (8-bits), large encodings (BDDs), abstraction (simplifications -> imprecise)

- **Black-box analysis of processors/assemblers**
  - Ex: [DERIVE, Hsieh-Engler-Back], [Giano, Forin et al.]
  - Emphasis on testing all aspects (addressing modes, clock cycles, privilege levels) of a processor (no symbolic/circuit generation)

- **Connection with Machine Learning**
  - Close connection between smart inputs for a template and VC dimension of a concept class, to be explored in the future

- **Automatic Program Synthesis**
  - From I/O examples [Gulwani et al., …], “Program Sketching” (“templates”) [Bodik et al., Solar-Lezama et al.,…]
  - Here, new app. domain, smart sampling, verification oracle is a black box