Range Analysis of Microcontroller Code Using Bit-Level Congruences

Jörg Brauer (RWTH Aachen University)
Andy King (Portcullis Computer Security)
Stefan Kowalewski (RWTH Aachen University)

20.09.2010 @ FMICS
Motivating Example (1/2)

0x50: LDI r17 0
0x51: LDI r26 0
0x52: LDI r27 0
0x53: LDI r30 66
0x54: LDI r31 0
0x55: RJUMP 2
0x56: LPMPI r0 Z
0x57: STPI X r0
0x58: CPI r26 99
0x59: CPC r27 r17
0x5A: BRNE -5
0x5B: RET

• Loop copies three bytes from program memory into SRAM
Motivating Example (2/2)

0x50: LDI r17 0
0x51: LDI r26 0
0x52: LDI r27 0
0x53: LDI r30 66
0x54: LDI r31 0
0x55: RJUMP 2
0x56: LPMPI r0 Z
0x57: STPI X r0
0x58: CPI r26 99
0x59: CPC r27 r17
0x5A: BRNE -5
0x5B: RET

• Interval analysis derives $X \in [96, 98] \land Z \in T$
• Insufficient for proving correctness
• Key idea: Combine with relational invariants (bit-level congruences) to prove $X \in [96, 98] \land Z \in [66, 68]$
Bit-Level Congruences

- Linear equations of the form \( \sum_{i=0}^{n-1} \lambda_i \cdot v_i \equiv_m d \)
  - \( v_i \) are bits
  - \( \lambda_i \in \mathbb{Z} \) are coefficients
  - \( m \in \mathbb{N} \) is a modulus
  - \( d \in \mathbb{Z} \) is a constant
- Good to choose \( m = 2^8 \) for 8-bit microcontroller
- Can represent overflows that occur frequently
Example: Exclusive-Or

- Consider instruction EOR r0 r1
- Represented in Boolean logic:
  \[ \varphi = \land_{i=0}^{7} r0'[i] \leftrightarrow r0[i] \oplus r1[i] \]
- Congruent abstraction of \( \varphi \) gives equations
  \[ \alpha_{\text{cong}}(\varphi) = \land_{i=0}^{7} (128 \cdot r0'[i] \equiv_{256} 128 \cdot r0[i] + 128 \cdot r1[i]) \]
- Computed using algorithm of [KS10]
Computing Congruent Abstractions of Entire Instruction Set

- Bit-blast concrete semantics of each instruction as defined in the specification
- Compute congruent abstraction using SAT solving
- Gives a set of transfer functions for each instruction in a program
- Need to be computed once, and can be reused afterwards
Deriving Congruent Invariants (1/3)

- Consider \texttt{EOR r0 r1; EOR r1 r0; EOR r0 r1;}
- Well-known idiom for register-swapping
- Goal: Relate inputs \(r_0, r_1\) to outputs \(r_0', r_1'\) to derive a suitable program invariant
For the first two instructions, we have:

\[
\bigwedge_{i=0}^{7} (128 \cdot r0'[i] \equiv_{256} 128 \cdot r0[i] + 128 \cdot r1[i]) \land \bigwedge_{i=0}^{7} (r1'[i] \equiv_{256} r1[i]) \tag{1}
\]

\[
\bigwedge_{i=0}^{7} (128 \cdot r1'[i] \equiv_{256} 128 \cdot r0[i] + 128 \cdot r1[i]) \land \bigwedge_{i=0}^{7} (r0'[i] \equiv_{256} r0[i]) \tag{2}
\]

Connect outputs of (1) to inputs of (2) and eliminate intermediate variables

Elimination amounts to triangularization
Deriving Congruent Invariants (3/3)

• Invariant after second instruction:

\[ \bigwedge_{i=0}^{7} \left( r1'[i] \equiv_{256} r0[i] \right) \land \bigwedge_{i=0}^{7} \left( 128 \cdot r0'[i] \equiv_{256} 128 \cdot r0[i] + 128 \cdot r1[i] \right) \]

• Invariant after third instruction:

\[ \bigwedge_{i=0}^{7} \left( r1'[i] \equiv_{256} r0[i] \right) \land \bigwedge_{i=0}^{7} \left( r0'[i] \equiv_{256} r1[i] \right) \]

• That is, the implementation does what it is supposed to do
Initial Loop Revisited

• For the initial loop, this gives the invariant:
  \[ r_{26}' - r_{30}' \equiv_{256} 30 \land \]
  \[ \land_{i=0}^{7} (r_{17}'[i] \equiv_{256} 0 \land r_{27}'[i] \equiv_{256} 0 \land r_{31}'[i] \equiv_{256} 0) \]
  
• Difference between \( r_{26}' \) and \( r_{30}' \) as expected
• But no explicit information about range of \( r_{30}' \)
• Combine with intervals \( x \in [96, 98] \land z \in T \) to prove that \( x \in [96, 98] \land z \in [66, 68] \)
Reduction

• To do so, construct a map

\[
\text{reduce} : \text{Int} \times \text{Cong} \rightarrow \text{Int} \times \text{Cong}
\]

with \( \text{reduce}(i, c) = (i', c') \) and

\[
i' \subseteq i \quad c' \subseteq c
\]

• Intuitively, let information from one domain flow into the other computer narrower over-approximation
Stronger Intervals

- Convert constraint from intervals into logic:
  \[ i = 96 \leq r_{26'} \leq 98 \land 0 \leq 30' \leq 255 \]

- Convert congruent invariant into logic, say, \( \psi \)

- Put \( i \land \psi \land r_{30'}[7] \) and test satisfiability

- Unsatisfiable, hence \( r_{30'} \leq 127 \)

- Put \( i \land \psi \land \neg r_{30'}[7] \land r_{30'}[6] \) and test satisfiability

- Satisfiable, hence \( r_{30'} \geq 64 \)

- Proceed with all bits to get \( 66 \leq r_{30'} \leq 68 \)
Stronger Congruences

• Apply similar idea to congruence equations
• Encode additional constraints from intervals into equation system
• And then project these additional constraints
• Gives stronger congruence equations, e.g.
  \[ r_{26}^{'}[7] \equiv_{256} 0 \land r_{30}^{'}[7] \equiv_{256} 0 \]
• Technical details in the paper
Experiments: Transfer Function Synthesis

• Derived congruent abstractions for entire instruction set of ATMEL ATmega16 microcontroller
• Less than 1s for each instruction
• Some are exact (EOR, INC, ADD, etc.) others are not (AND, OR)
• By combining intervals and congruences, this loss of precision can be eased
Experiments: Computing Loop Invariants

• Invariant stabilized after 2 iterations
• Requiring 0.3s
• Computing join and eliminating variables is cubic in the number of bits
• Thus, it is a good idea to „slice“ variables not affected
• And use congruences where the interval analyzer loses precision
Experiments: Reducing Abstract Descriptions

- Reducing the intervals using SAT solving required 16 SAT instances
- Overall runtime using SAT4J amounts to 0.25s
- That is, two instances for each bit
- Reducing congruences requires computing upper-triangular form, approx. 0.1s
Related Work

- P. Cousot and R. Cousot: Systematic Design of Program Analysis Frameworks (POPL 1979)
- M. Codish, V. Lagoon and P. J. Stuckey: Logic Programming with Satisfiability (TPLP 2008)
Discussion

• Deriving congruent invariants for binary/assembly code
• Combination with interval analysis
• A novel reduction operator that combines congruences and intervals
• Approach relies heavily on SAT-solving, which appears natural when reasoning about bits
• Allows to prove memory safety for many examples of binary code
• Integrated into [mc]square program verification tool
Thank you very much!