Precise Control Flow Reconstruction Using Boolean Logic

Thomas Reinbacher[1] and Jörg Brauer[2]

[1] Embedded Computing Systems Group
Vienna University of Technology, Austria

[2] Embedded Software Laboratory
RWTH Aachen University, Germany

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Binary code verification

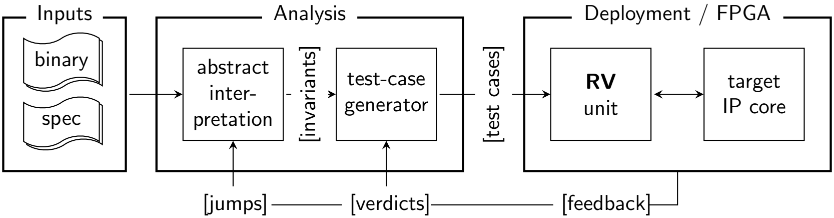
1. Embedded software mostly not in plain ANSI C
   - Side effects, embedded assembler, direct hardware access
2. No source code required (closed source libraries)
3. Who verified your compiler?
   - GCC 4.3.5 has 8M loc (2.5M C, 1.5M C++, 1.5M Java, 60k ASM ...)
   - Proving correctness of the compiler is very hard
   - Even translation validation is hard (and not really widespread in industry)
4. As close as possible to the actual execution

Big picture:

- Formal methods to derive a set of test cases (guess)
- Runtime Verification to check validity of test cases during execution (check)

CevTes Approach

1. Use Abstract Interpretation to derive an over-approximation of the reachable states
2. Find program locations where the specification is violated
3. Backward analysis derives counterexamples (test cases)
4. Interface a hardware unit attached to the SUT to replay a test case and automatically identify spurious warnings



Control Flow Graph Recovery

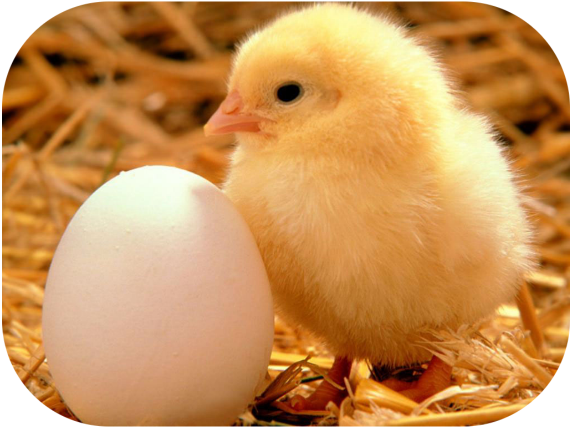
Problem Statement

Given a binary, return its (overapproximated) Control Flow Graph

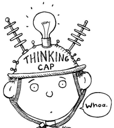
Control Flow Graph Recovery

Problem Statement

Given a binary, return its (overapproximated) Control Flow Graph



- Recovery requires invariants over registers
- A CFG is required to generate these invariants



Bit-twiddling

Some C-code bit-twiddling:

```
#define SWPC(a,b) (a^=b,b^=a,a^=b,a&=0xf,b&=0xf)
```

(swap the value of two variables without an auxiliary variable)

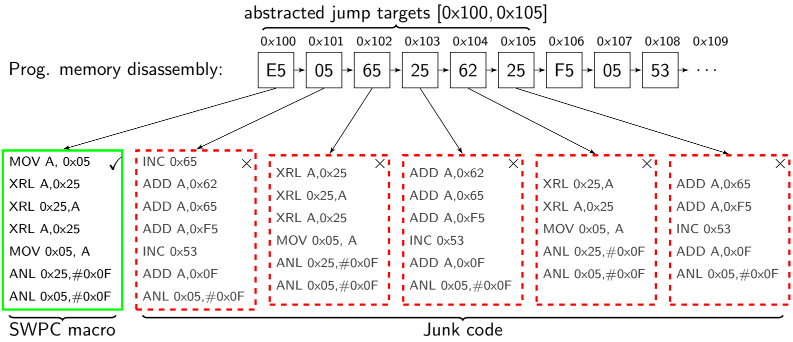
used within a `switch-case` statement:

```
switch (p) {
 case 10: SWPC(x,y); break;
 case 20: foo(x,y);  break;
  ...
 default: bar(x,y);
}
```

Motivating Example

Noise propagation

Compilation transforms the `switch-case` into a jump table:



⇝ Falsely recovered jump targets generate significant noise for subsequent analyses ☹

Basic Block Abstraction

- Encode instruction set as propositional logic
- Lift encoding to basic block level
- Use existential quantification to reason about input/output values

Bit blasting & Basic blocks

1. Microcontroller instr. can be (precisely) encoded in prop. logic

$$[\![XRL\ A, B]\!] := \bigwedge_{i=0}^{n-1} \left((a'[i] \leftrightarrow a[i] \oplus b[i]) \wedge (b'[i] \leftrightarrow b[i]) \right)$$

$$[\![INC\ \ C]\!] := \bigwedge_{i=0}^{n-1} \left(c'[i] \leftrightarrow c[i] \oplus \bigwedge_{j=0}^{i-1} c[j] \right)$$

($a[i]$ is the i-th bit of vector a; primed vectors are outputs)

Bit blasting & Basic blocks

1. Microcontroller instr. can be (precisely) encoded in prop. logic

$$[\![XRL\ A, B]\!] := \bigwedge_{i=0}^{n-1} ((a'[i] \leftrightarrow a[i] \oplus b[i]) \wedge (b'[i] \leftrightarrow b[i]))$$

$$[\![INC\ \ C]\!] := \bigwedge_{i=0}^{n-1} \left(c'[i] \leftrightarrow c[i] \oplus \bigwedge_{j=0}^{i-1} c[j] \right)$$

($a[i]$ is the i-th bit of vector a; primed vectors are outputs)

2. A basic block is a straight sequence of instructions with a single entry point and a single exit point. Propositionally encode bb's:

```
(0x114) MOV A, 0x05
(0x116) XRL A, 0x25
(0x118) XRL 0x25, A
(0x11a) XRL A, 0x25
(0x11c) MOV 0x05, A
```

$$\rightsquigarrow \qquad \Phi_{bb} := \bigwedge_{i=\text{fst}(bb)}^{lst(bb)} \text{bitblast}(i)$$

3. Translate Φ_{bb} into CNF by Tseitin Encoding [Tseitin'70]

Block-Wise interpreters

pre: 0x05 : {10, 20, 30}, 0x25 : {50, 60, 70}

Φ_{bb}

```
(0x114) MOV A, 0x05
(0x116) XRL A, 0x25
(0x118) XRL 0x25, A
(0x11a) XRL A, 0x25
(0x11c) MOV 0x05, A
```

Φ_{bb}

post: 0x05 : {??, ??, ??}, 0x25 : {??, ??, ??}

$\overrightarrow{\mathcal{F}}$ $c \leftrightarrow Pre$

$\forall m \in M_{\text{out}}$:

$\phi_m = \text{proj}(\Phi_{bb} \wedge c, \text{lit}(m))$

$\overleftarrow{\mathcal{B}}$ $c \leftrightarrow Post$

$\forall m \in M_{\text{inp}}$:

$\phi_m = \text{proj}(\Phi_{bb} \wedge c, \text{lit}(m))$

$\rightsquigarrow$ (Existential) quantifier elimination needed to compute $\text{proj}(\Phi_{bb})$

SAT-based projection scheme [Brauer et. al; CAV'11]

- enumerate prime implicants of a quantified formula
- translate cubes into clauses to get CNF of the projection

Range abstraction [Barrett,King; ENCS 267,'10]

- Infer the range of values of a n-wide bit vector $x = \langle x_0, \ldots, x_{n-1} \rangle$ where x is constrained by a boolean formula f
- Alternate runs of over- and underapproximations
- Solving $f \wedge c$ (blocking clause $c \bigvee_{i=0}^{n-1} y_i$; put $y_i = x_i$ if $b_i = 0$ and $y_i = \neg x_i$ othw.) excludes previously found solutions
- Adoptions yield a SAT-based Value-Set abstract domain

```
while (|k| < n){
  if(SAT(f /\ !x[n-|k|-1])){
    f <- f /\ !x[n-|k|-1]
    k <- <0> :: k
  }else{
    f <- f /\ x[n-|k|-1]
    k <- <1> :: k
  }
}
```

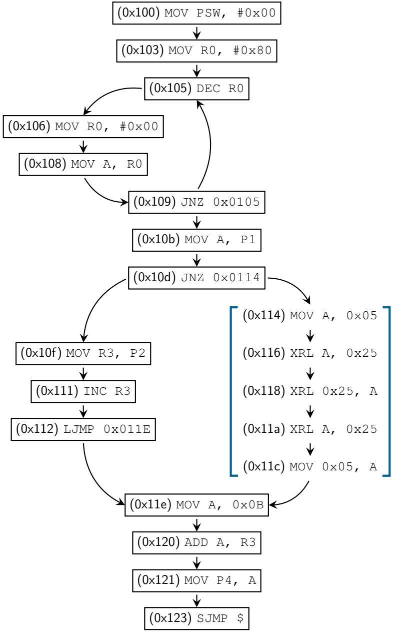
1. k is the minimum value of vector x in f
2. similar for maximum use $x[n - |k| - 1]$; invert truth values prepended to k

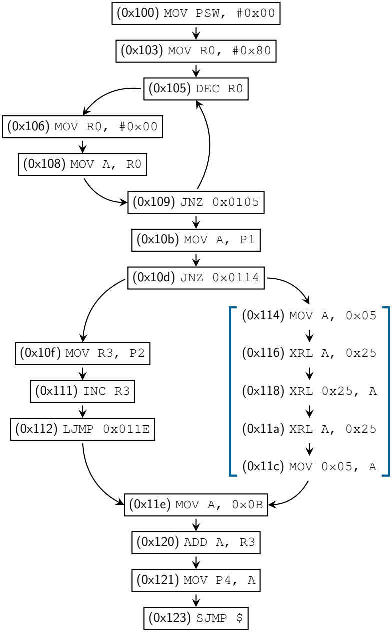
Putting it together

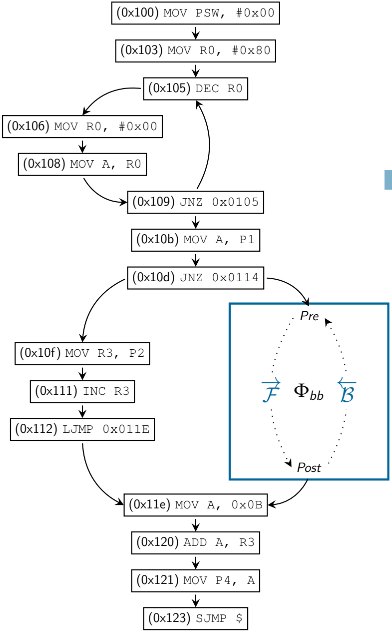
C macro: #define SWP(a,b) (a^=b,b^=a,a^=b)

Prog Cnt	Mnemonic & Instruction		
C:0x0100	E505	MOV	A,0x05
C:0x0102	6525	XRL	A,0x25
C:0x0104	6225	XRL	0x25,A
C:0x0106	6525	XRL	A,0x25
C:0x0108	F505	MOV	0x05,A

1. Bitblast $\{\phi_{0x0100}, \phi_{0x0102}, \phi_{0x0104}, \phi_{0x0106}, \phi_{0x0108}\}$
2. Tseitin $\{\phi'_{0x0100}, \phi'_{0x0102}, \phi'_{0x0104}, \phi'_{0x0106}, \phi'_{0x0108}\}$
3. Basic Block encoding $\Phi_{bb} := \bigwedge_{i=\mathrm{fst}(bb)}^{lst(bb)} \mathrm{bitblast}(i)$
4. Determine output registers/bits $\overrightarrow{\mathcal{F}}$
5. Determine input registers/bits $\overleftarrow{\mathcal{B}}$


(0x100) MOV PSW, #0x00
(0x103) MOV R0, #0x80
(0x105) DEC R0
(0x106) MOV R0, #0x00
(0x108) MOV A, R0
(0x109) JNZ 0x0105
(0x10b) MOV A, P1
(0x10d) JNZ 0x0114
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x114) MOV A, 0x05
(0x116) XRL A, 0x25
(0x118) XRL 0x25, A
(0x11a) XRL A, 0x25
(0x11c) MOV 0x05, A
(0x11e) MOV A, 0x0B
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $


(0x100) MOV PSW, #0x00
(0x103) MOV R0, #0x80
(0x105) DEC R0
(0x106) MOV R0, #0x00
(0x108) MOV A, R0
(0x109) JNZ 0x0105
(0x10b) MOV A, P1
(0x10d) JNZ 0x0114
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x114) MOV A, 0x05
(0x116) XRL A, 0x25
(0x118) XRL 0x25, A
(0x11a) XRL A, 0x25
(0x11c) MOV 0x05, A
(0x11e) MOV A, 0x0B
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $


(0x100) MOV PSW, #0x00
(0x103) MOV R0, #0x80
(0x105) DEC R0
(0x106) MOV R0, #0x00
(0x108) MOV A, R0
(0x109) JNZ 0x0105
(0x10b) MOV A, P1
(0x10d) JNZ 0x0114
Pre
$\overrightarrow{\mathcal{F}}$ Φ_{bb} $\overleftarrow{\mathcal{B}}$
Post
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x11e) MOV A, 0x0B
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $

Program Level Abstraction

Program Level Abstraction

Preprocessing

- Sweep linear disassembly of the hex file
- Stop when iJmp instruction is detected
- Analyse program fragment, abstract targets and restart

Alternating $\overrightarrow{\mathcal{F}}$ and $\overleftarrow{\mathcal{B}}$ abstract interpretation

Result for the analysis are value-sets (or intervals) for all memory locations

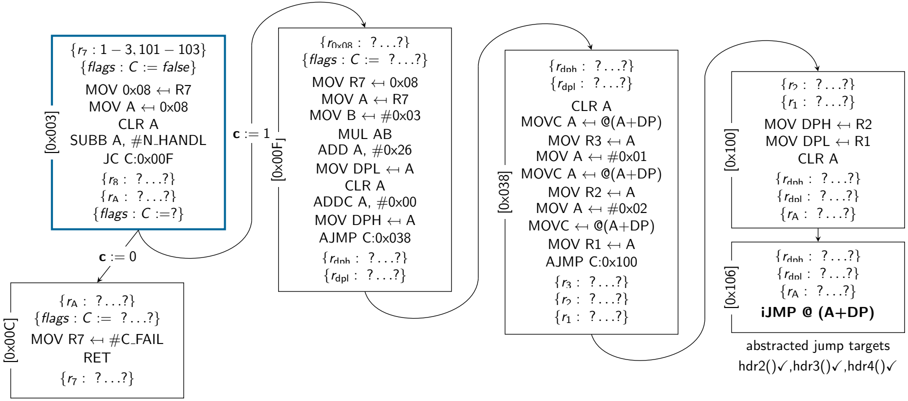
- e.g., for the MCS-51 architecture: DPL and DPH

Depth bounded backtracking on conditional branches

- Invariant refinement

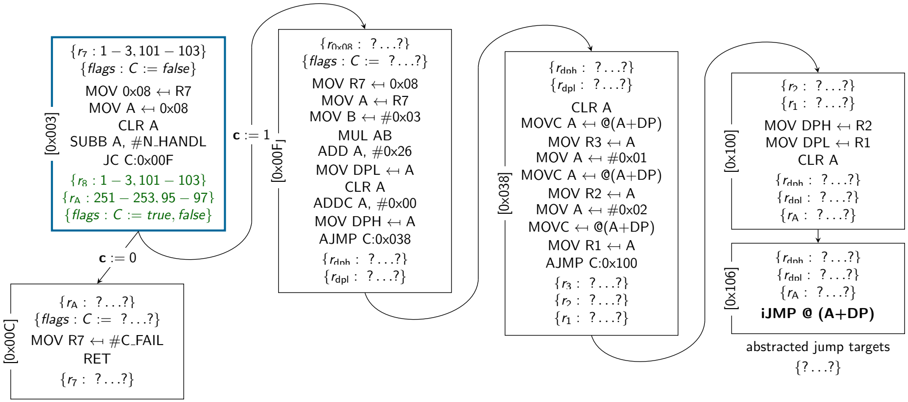
Algorithm

Worked example

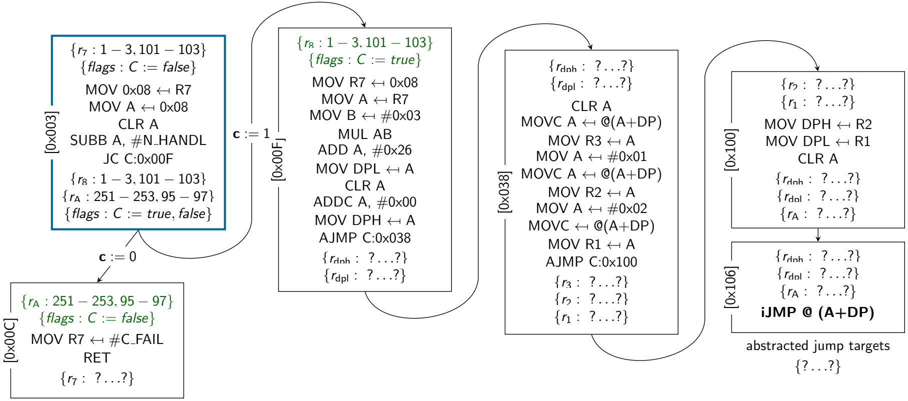


1. Use $\overrightarrow{\mathcal{F}}$ to derive a postcondition $\psi_{\text{post}}(V_{\text{out}})$ for initial block b

Algorithm $\overrightarrow{\mathcal{F}}$ interpretation

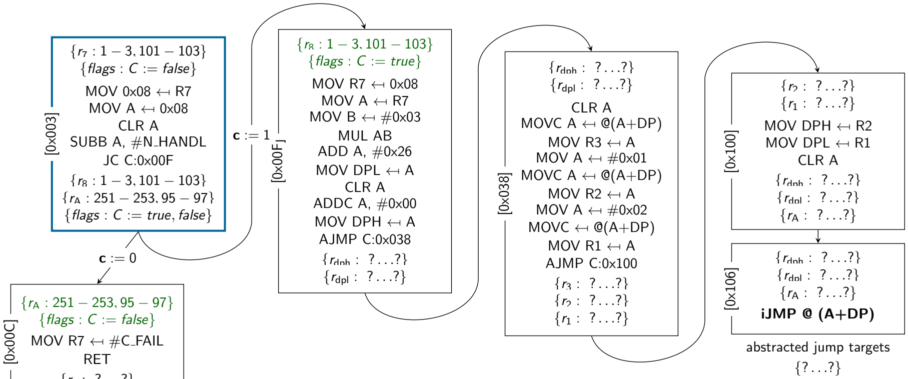


Naive propagation



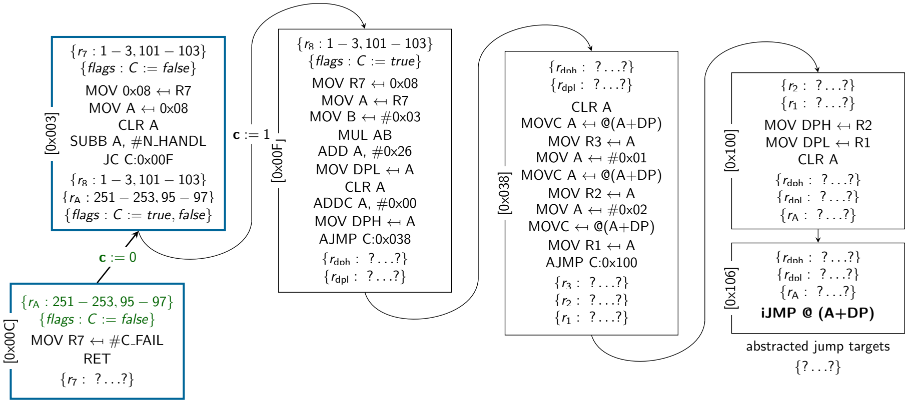
1. Use $\overrightarrow{\mathcal{F}}$ to derive a postcondition $\psi_{\text{post}}(V_{\text{out}})$ for initial block b
2. $\forall b_{\text{succ}} \in \text{succ}(b)$
   - If the edge does not impose any constraints then join the precondition $\psi_{\text{pre}}^{\text{succ}}(V_{\text{in}})$ of b_{succ} with the postcondition $\psi_{\text{post}}(V_{\text{out}})$ of b; repeat step 2 with the next successor
   - Else continue with backtracking at step 3 and set $i = k$

Naive propagation

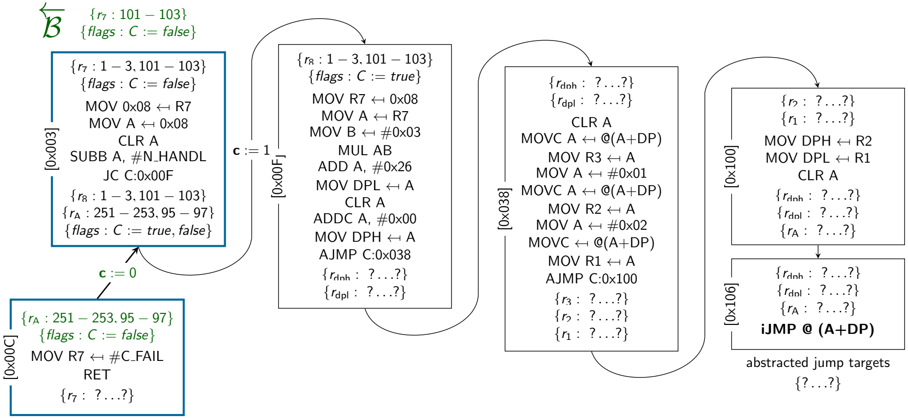


1. Use $\overrightarrow{\mathcal{F}}$ to derive a postcondition $\psi_{\text{post}}(V_{\text{out}})$ for initial block b
2. $\forall b_{\text{succ}} \in \text{succ}(b)$
   - If the edge does not impose any constraints then join the precondition $\psi^{\text{succ}}_{\text{pre}}(V_{\text{in}})$ of b_{succ} with the postcondition $\psi_{\text{post}}(V_{\text{out}})$ of b; repeat step 2 with the next successor
   - Else continue with backtracking at step 3 and set $i = k$
3. Backtracking
   - Rename the edge constraint to range over variables in V_{out}; denote the resulting constraint by σ and put $\xi = \phi_b(T) \wedge \sigma$
   - Apply $\overleftarrow{\mathcal{B}}$ to b, derive $\eta(V_{\text{in}}) = \overleftarrow{\mathcal{B}}(\xi, \psi^{b}_{\text{post}}(V_{\text{out}}))$
   - Refine the precondition of b by computing the intersection of value set $\psi'^{b}_{\text{pre}}(V_{\text{in}}) = \psi^{b}_{\text{pre}}(V_{\text{in}}) \sqcap \eta(V_{\text{in}})$
   - Dec. i; If i is pos and $|\text{pred}(b)| = 1$ do step 3 for $\text{pred}(b)$; else 4

Algorithm $\overleftarrow{\mathcal{B}}$ refinement

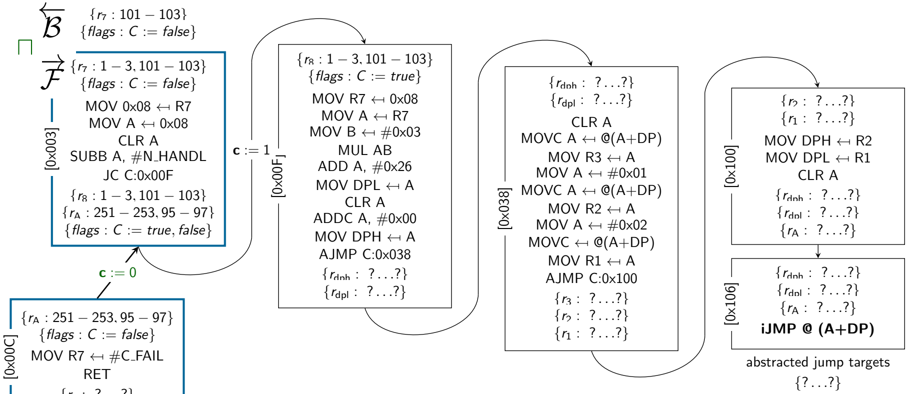


Algorithm $\overleftarrow{\mathcal{B}}$ refinement

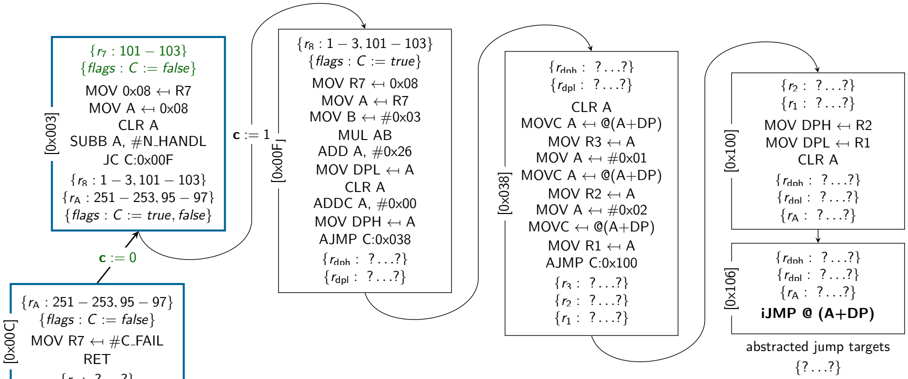


1. Use $\overrightarrow{\mathcal{F}}$ to derive a postcondition $\psi_{\text{post}}(V_{\text{out}})$ for initial block b
2. $\forall b_{\text{succ}} \in \text{succ}(b)$
   - If the edge does not impose any constraints then join the precondition $\psi_{\text{pre}}^{\text{succ}}(V_{\text{in}})$ of b_{succ} with the postcondition $\psi_{\text{post}}(V_{\text{out}})$ of b; repeat step 2 with the next successor
   - Else continue with backtracking at step 3 and set $i = k$
3. Backtracking
   - Rename the edge constraint to range over variables in V_{out}; denote the resulting constraint by σ and put $\xi = \phi_b(T) \wedge \sigma$
   - Apply $\overleftarrow{\mathcal{B}}$ to b, derive $\eta(V_{\text{in}}) = \overleftarrow{\mathcal{B}}(\xi, \psi_{\text{post}}^{b}(V_{\text{out}}))$
   - Refine the precondition of b by computing the intersection of value set $\psi_{\text{pre}}^{\prime b}(V_{\text{in}}) = \psi_{\text{pre}}^{b}(V_{\text{in}}) \sqcap \eta(V_{\text{in}})$
   - Dec. i; If i is pos and $|\text{pred}(b)| = 1$ do step 3 for $\text{pred}(b)$; else 4

$\overleftarrow{\mathcal{B}}$ refinement (intersection)

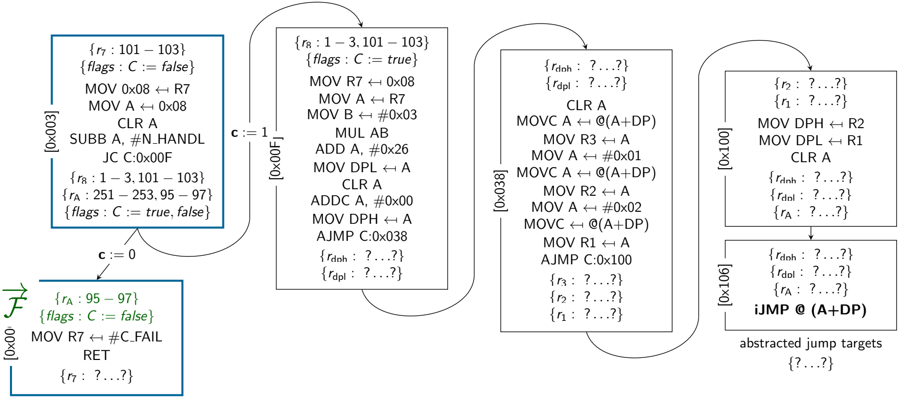


Algorithm $\overleftarrow{\mathcal{B}}$ refinement (intersection result)

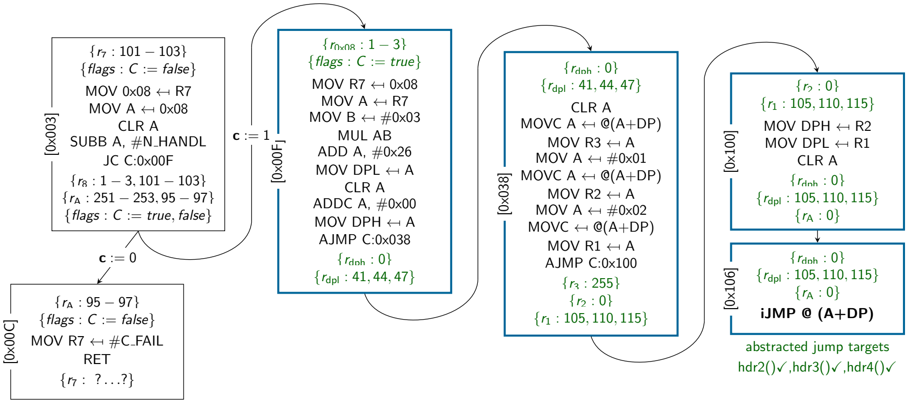


1. Use $\overrightarrow{\mathcal{F}}$ to derive a postcondition $\psi_{post}(V_{out})$ for initial block b
2. $\forall b_{succ} \in \mathrm{succ}(b)$
   - If the edge does not impose any constraints then join the precondition $\psi_{pre}^{succ}(V_{in})$ of b_{succ} with the postcondition $\psi_{post}(V_{out})$ of b; repeat step 2 with the next successor
   - Else continue with backtracking at step 3 and set $i = k$
3. Backtracking
   - Rename the edge constraint to range over variables in V_{out}; denote the resulting constraint by σ and put $\xi = \phi_b(T) \wedge \sigma$
   - Apply $\overleftarrow{\mathcal{B}}$ to b, derive $\eta(V_{in}) = \overleftarrow{\mathcal{B}}(\xi, \psi_{post}^{b}(V_{out}))$
   - Refine the precondition of b by computing the intersection of value set $\psi_{pre}^{\prime b}(V_{in}) = \psi_{pre}^{b}(V_{in}) \sqcap \eta(V_{in})$
   - Dec. i; If i is pos and $|\mathrm{pred}(b)| = 1$ do step 3 for $\mathrm{pred}(b)$; else 4
4. Forward Refinement
   - Derive $\psi_{post}^{\prime b}(V_{out}) = \overrightarrow{\mathcal{F}}(\phi_b(T), \psi_{pre}^{\prime b}(V_{in}))$
   - Increment i. If $i < k$ then set b to $\mathrm{succ}(b)$ and repeat step 4, otherwise continue at step 5
5. Join refined precondition
   - Rename $\psi_{post}^{\prime b}(V_{out})$ to range over inputs V_{in}; denote by σ'
   - Set $\psi_{pre}^{succ}(V_{in}) = \sigma' \sqcup \psi_{pre}^{succ}(V_{in})$; Continue with next succ in 2

Algorithm $\overrightarrow{\mathcal{F}}$ refinement



Algorithm $\overrightarrow{\mathcal{F}}$ refinement



Experiments

Implementation

- Integrated into the [MC]SQUARE binary code verification framework
- Using the SAT4J SAT solver
- Running on an Intel Core i5 CPU with 4 GB of RAM

Benchmark sets

1. Based on *Arrays of Pointers to Functions* article [N. Jones, Embedded Systems Programming Magazine'99]
    - Single Row Input, Keypad, Communication Link, Task Scheduler
2. Non trivial `switch-case` statements
    - Single switch-case, Emergency stop [PLC Open Spec]

Two different embedded compilers

- KEIL μVISION 3 v3.23
- SDCC v3.0.0

Binary Program						$\overrightarrow{\mathcal{F}}$ interpreter			$\overrightarrow{\mathcal{F}}+\overleftarrow{\mathcal{B}}$ interpreter				
Name	C	loc_C	$instr_B$	JT	RT	FT	Time	RS	k	RT	FT	Time
Single Row	K	80	67	6	2401	2395	2.6	2	2	6	–	3.32
	S		52		460	454	2.4	2	2	6	–	2.0
Keypad	K	113	113	9	3844	3835	3.49	4	2	9	–	4.33
	S		80		1508	1499	3.08	4	2	9	–	2.57
Comm Link	K	111	164	8	6889	6881	4.56	2	2	8	–	4.37
	S		118		84	76	3.38	2	2	8	–	4.29
Scheduler	K	81	105	5	>1000	>995	>5m	17	2	5	–	14.03
	S		97					23	2	5	–	10.23
Switch Case	K	82	166	19	>5000	>4981	>5m	94	2	19	–	17.49
	S		180		3304	3285	2.31	6	2	38	19	2.6
Emergency	K	138	150	9	768	759	2.8	2	2	9	–	2.6
	S		141		256	247	2.9	2	2	9	–	3.1

loc_C ... Lines of C code
$instr_B$ ... Num of assembly instructions
JT ... Num of jump targets
RT ... Num of recovered targets
FT ... Num of recovered false targets
RS ... Num of refinement steps
k ... Backtracking depth
Time ... Analysis time in seconds

- Pure $\overrightarrow{\mathcal{F}}$ analysis insufficient for recovering iJump targets
- Interleaved $\overrightarrow{\mathcal{F}}/\overleftarrow{\mathcal{B}}$ analysis greatly eliminates spurious targets

Conclusion & Future work

Precise control-flow graph recovery for microcontroller binary code

- Expressing the concrete semantics of a program as propositional boolean formulae
- Alternating runs of forward and backward analysis are useful to soundly recover control flow in the Value-Set abstract domain
- Same encoding for forward and backward interpretation
- No difficult to design backward interpreters
- Benefits from the progress of cutting-edge SAT solvers

Conclusion & Future work

Precise control-flow graph recovery for microcontroller binary code

- Expressing the concrete semantics of a program as propositional boolean formulae
- Alternating runs of forward and backward analysis are useful to soundly recover control flow in the Value-Set abstract domain
- Same encoding for forward and backward interpretation
- No difficult to design backward interpreters
- Benefits from the progress of cutting-edge SAT solvers

More recent and future work

- Generic assembly model (ARM, AVR, C167, . . .)
- Automatic test-case generation, combine with runtime verification

CevTes ⇔ Framework for Testing / RV of Embedded Software
[http://ti.tuwien.ac.at/ecs/research/projects/cevtes]

Thank you. . .


义务指路点
If you get lost or need getting directions, please ask me, the volunteer.