Test-Case Generation for Embedded Binary Code Using Abstract Interpretation

Thomas Reinbacher¹, Jörg Brauer², Martin Horauer³, Andreas Steininger¹, and Stefan Kowalewski²

¹ Institute of Computer Engineering
Vienna University of Technology, Austria

² Embedded Software Laboratory
RWTH Aachen University, Germany

³ Department of Embedded Systems
University of Applied Sciences Technikum Wien, Austria
Verification of Embedded Binary Code - Motivation

1. Embedded software mostly not in **plain** ANSI C
   - expr. with side effects, inline assembler, direct hardware register access

2. Who verified your compiler?
   - assumption
     - Good SW has about 1 error in 500 loc
     - GCC 4.3.5 has 8M loc
       - GCC $\rightarrow \frac{8\text{M}}{500} = 16k$ flaws
     - Small devices C compiler (SDCC) has 3M loc
       - SDCC $\rightarrow \frac{3\text{M}}{500} = 6k$ flaws
   - you can bug-fix GCC & SDCC; you can’t COTS compilers

3. Interrupts and interrupt service routines

4. Exact semantics is unambiguously specified in high-level representations

Supporting $\mu$CU specifics is essential for embedded SW verification $\rightsquigarrow$ Analysis of binary code
Verification Strategies for Embedded Binary Code

1. Model checking binary code, e.g., [MC]SQUARE
   - can detect errors that others can’t (is complete) ✓
   - suffers from state-explosion problem ×

2. Theorem Proving
   - deriving formal proofs is most times expensive ×
   - no “push-button” approach ×

3. Abstract Interpretation
   - scalable approach to verification ✓
   - suffers from imprecision, spurious traces ×

Our goal: derive real counterexample traces for binary programs using Abstract Interpretation
Abstract Interpretation computes an over-approximation of the exact behavior.

Thus, found bugs may be spurious.

How to separate real bug reports from spurious ones?
Abstract Interpretation for Better!

Spurious warnings are a major issue of abstract interpretation in practice

- examining them relies on manual inspection of program invariants
- **but**: embedded code may be very complex, so are the derived invariants
  - manual inspection is difficult and time-intensive

↓↓ Our framework (CEVTES) ↓↓

1. applies abstract interpretation to generate assertion-directed test cases
2. provides a link to the actual execution environment of the target
3. automatically identifies spurious test traces
The CEVTES Framework

- read a binary file and a specification
- abstract interpretation computes invariants
- potential violations of the specification are detected
- backward analysis to derive test cases
- load test cases onto target hardware
- online monitoring rules out spurious warnings
- runtime feedback, indirect jump targets
Specification Language

... Assertions figure strongly in Microsoft code. A recent count discovered more than a quarter million of them in the code for its Office product suite; (C.A.R. Hoare 2003) ...

Our industrial case study showed that the full expressive power of temporal logics is often not understood/needed by test engineers.

\[
\begin{align*}
\Phi & ::= A(pc, \varphi) \mid I(\varphi) \\
\varphi & ::= \text{true} \mid \text{false} \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \text{AP}
\end{align*}
\]

- Local assertions \( A(pc, \varphi) \) hold on certain program locations
- Global invariants \( I(\varphi) \) hold on every program location

![Diagram of the CEVTES framework]

- Inputs: spec, binary
- Analysis: abstract interpretation, test-case generator
- Deployment / FPGA: test-case monitor, target IP core
- [jumps] [verdicts] [feedback]
Program Analysis

0000 020100 LJMP STARTUP
0100  STARTUP:
0100  7D000  MOV  PSW,#0x00
0103  7880  MOV  R0,#0x80
0105  RAM_CLR:
0105  18  DEC  R0
0106  7600  MOV  @R0,#0x00
0108  E8  MOV  A,R0
0109  70FA  JNZ  RAM_CLR
010B  E590  MOV  A, P1
010D  7006  JNZ  READ_P3
010F  ABA0  MOV  R3, P2
0111  0B  INC  R3
0112  02011E LJMP CONT
0115  READ_P3:
0115  75D008  MOV  PSW,#0x08
0118  ABB0  MOV  R3, P3
011A  0B  INC  R3
011B  7D000  MOV  PSW,#0x00
011E  CONT:
011E  E50B  MOV  A, 0x0B
0120  2B  ADD  A, R3
0121  F5B1  MOV  P4, A
0123  80FE SJMP $
Computing Invariants using Interval Analysis

Interval abstract domain

- determine invariants for program variables (registers)
- word-level interval \([a, b]\) with \(a, b \in \{0, \ldots, 255\}\) and \(a \leq b\).
  (complete lattice with \(\top = [0, 255], \bot = \emptyset, [a_1, b_1] \sqcup [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)]\))

Analysis result

- list of word-level intervals over memory locations attached to every \(pc\) location

\[\langle pc, (A[a_0, b_0],B[a_1, b_1], \ldots) \rangle\]
Computing Invariants II

(0x000) LJMP 0x0100
(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 0x0115
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x115) MOV PSW, #0x08
(0x118) MOV R3, P3
(0x11a) INC R3
(0x11b) MOV PSW, #0x00
(0x11e) MOV A, 0x0B
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $
Computing Invariants III

(0x000) LJMP 0x0100
(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 0x0115
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x115) MOV PSW, #0x08
(0x118) MOV R3, P3
(0x11a) INC R3
(0x11b) MOV PSW #0x00
(0x11e) MOV A, 0x0B
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $
Synthesized Transformers for Basic Blocks

Bit-Blasting

- semantics of a microcontroller instruction is encoded in propositional logic (bit-blasting)
  - example: \( \text{INC } A := \bigwedge_{i=0}^{7} (a'[i] \leftrightarrow a[i] \oplus \bigwedge_{j=0}^{i-1} a[j]) \)
- derive optimal transfer functions for sequences of bit-blasted instructions by use of a decision procedure

Brauer and King (SAS 2010)

- successive calls to a SAT solver
- derive a system of affine equalities \( \bigwedge_{i=0}^{m-1} (\sum_{j=0}^{n_i-1} \lambda_{i,j} \cdot v_j = d_i) \), where \( v_j \) are program variables and \( \lambda_{i,j}, d_i \in \mathbb{Z} \)
- use octagonal constraints (Miné 2006) to distinguish inputs that lead to overflows, e.g., resulting affine system for \( \text{INC } A \):
  - \((\llangle a \rrangle \leq 254) \Rightarrow (\llangle a' \rrangle = \llangle a \rrangle + 1)\)
  - \((\llangle a \rrangle = 255) \Rightarrow (\llangle a' \rrangle = 0)\)
Synthesized Transformers for Basic Blocks II

```asm
(0x00) LJMP 0x0100
(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 0x0115
(0x10f) MOV R3, P2
(0x111) INC R3
(0x112) LJMP 0x011E
(0x115) MOV PSW, #0x08
(0x118) MOV R3, P3
(0x11a) INC R3
(0x11b) MOV PSW, #0x00
(0x120) ADD A, R3
(0x121) MOV P4, A
(0x123) SJMP $
```
Test-trace Generation

1. find a set of program locations where the specification may be violated
2. systematically search for traces that lead to this violation
3. negate the specification assertion to $\neg \Psi$
4. for a given invariant $\Theta$, use $\neg \Psi \land \Theta$ as the desired postcondition
5. apply the affine transfer function in reverse using Integer Linear Programming
6. obtain a precondition
7. iteratively apply steps 5-6 until the entry of the program
Test-trace Deployment

Test trace $t$

- $\pi := \langle pc_0, \ldots, pc_n \rangle$ a path of PC locations
- $ln := \langle pc, i \rangle$ (e.g., $ln := \langle 0xc1c1, p1 \leftarrow 0xb2 \rangle$)

Hardware monitor unit

- attached to industrial $\mu$CU IP core
- monitor and track memory accesses
- compare current PC with the one given in $t$
- check global invariants on the fly
- check local assertions off-line at host
Worked Example

Reading a sensor value from an 8-bit input port and convert it to its absolute value. Will the assertion hold?

```
#define ABS(a) (((a)<0)?-(a):(a))
char getSensor(void){return P1;}
void sendPWM(void){...;}

int main(void){
    char val;
    while(1){
        val = ABS(getSensor());
        while(val > 0){
            sendPWM();
            val--;
        }
        ASSERT(val == 0);
    }
}
```

Analysis results

\[ (\text{getSensor()} \geq 0 \land \text{getSensor()} \leq 127) \quad \Rightarrow (\text{val}' = \text{getSensor()}), \]
\[ (\text{getSensor()} \geq -127 \land \text{getSensor()} \leq -1) \quad \Rightarrow (\text{val}' = -\text{getSensor()}), \]
\[ (\text{getSensor()} \geq -128 \land \text{getSensor()} \leq -128) \quad \Rightarrow (\text{val}' = -128) \]
Conclusion

We presented a framework (CEVTES)

- for test-case generation based on abstract interpretation of binary code
- guarantees
  - found errors are real ones ✓
  - reproducible on target hardware, not only on intermediate representation ✓
  - test data is exact ✓
  - sound due to over-approximations ✓
  - industry friendly ✓
- no guarantees
  - completeness (no full correctness proof)
  - dynamic jump targets calculated at run time
  - incomplete CFG (open issue)
Future Work

1. finish implementation and show feasibility when applied to industrial embedded code
2. time-bounded properties of the form \( \Theta(\varphi_1, \varphi_2, \delta) \). (If the predicate \( \varphi_1 \) holds then \( \varphi_2 \) must hold within \( \delta \in \mathbb{N} \) clock cycles)
3. control flow graph reconstruction
If you get lost or need getting directions, please ask me, the volunteer.