Analysing Switch-Case Code with Abstract Execution

Niklas Holsti         Jan Gustafsson, Linus Källberg, Björn Lisper

Tidorum Ltd         School of Innovation Design and Engineering
Finland              Mälardalen University
                     Västerås, Sweden
Context and contribution

• Recovering the Control Flow Graph from machine code

• Applications:
  – WCET analysis in the traditional way
  – other analyses starting from machine code

• Particular problem: Dynamic Transfer of Control (DTC)
  – call or jump to dynamically computed code address
  – in particular from switch-case statements
    • switch-case DTC is amenable to *local* analysis
    • but hard to handle by annotations

• Contributions:
  – apply *Abstract Execution* analysis to DTC
  – evaluate *Circular Linear Progression* (CLP) domain
  – combine two analysis tools: Bound-T and SWEET
  – compare DTC analysis methods: *patterns vs value analysis*
From binary file to control-flow graph

Problem: *dynamic* transfer of control, DTC

for example *jump via register*
The "trivial" problem

- DTC makes it hard:
  - to get the control flow graph (CFG) from machine code
  - or even to find all the code to be analysed
- Switch-case code often compiles into DTC
  - *trivial*: the compiler usually knows the DTC targets
  - *non-trivial*: the compiler seldom tells us what it knows
  - *complex*: wide variety of machine-code forms
- Many library functions use switch-case DTC
  - sometimes hand-written, tricky assembler

⇒ Switch-case DTC must be solved by analysis
Overview

- Switch-case **analysis methods**, their good and their bad
  - code-pattern matching
  - value analysis
- Our **study questions** and methods
- Our analysis tools
  - Bound-T
  - SWEET
  - and their **combination**
- Our example programs = test cases
- Discussion: what worked, **what didn't**, and why
- Conclusions
Pattern-matching vs. value-analysis

- **Pattern-matching** approach:
  - compiled switch-case DTC code uses small set of code idioms
    - directly indexed address table (dense case numbering)
    - look-up table, value $\Rightarrow$ address (sparse numbering)
  - tool detects target- & compiler-specific code patterns
    - pattern-specific rules or analysis finds DTC targets

- **Value-analysis** approach:
  - get DTC target addresses from a general value analysis
  - set of values usually small, but irregular; often from a table
  - need highly accurate value analysis
    - small over-estimation $\Rightarrow$ big change in CFG
  - circular problem $\Rightarrow$ iterative analysis
    - value analysis depends on CFG
    - CFG depends on the analysis of DTC
Example: "dense address table" as pattern

**Source (C)**

```c
switch (i) {
    case 0 : foo(i+2);
    case 1 : ...
    ... // cases 2 .. 12
    case 13 : ...
    default : log_err(x);
}
```

**Code (pseudo)**

```
134  r1 := <computed>
135  cmp r1,13 (unsigned)
136  if r1>13 jump 204
137  r2 := mem[347+r1]
138  jump via r2
```

Results of pattern match:
- this is DTC from switch-case
- dense numbering of cases 0 .. 13
- case-code addresses in 14-entry table at data address 347
- (default case at code address 204)

1. Aha, a DTC!
2. Aha, target address table starts at 347!
3. Aha, there are 14 target addresses!
4. (Aha, default case is at address 204!)
Example: "dense address table" value-analysis

Source (C)

```c
switch (i) {
    case 0 : foo(i+2);
    case 1 : ...
    ... // cases 2 .. 12
    case 13 : ...
    default : log_err(x);
}
```

Code (pseudo)

```c
134 r1 := <computed>
135 cmp r1,13 (unsigned)
136 if r1>13 jump 204
137 r2 := mem[347+r1]
138 jump via r2
```

Results of value analysis:

- DTC target addresses = values in memory locations 347 .. 360
- possibly further restricted by value constraints on the <computed> value of r1 at code address 134
- possibly distorted by over-estimations in value analysis

At 137, (unsigned) r1 in 0 .. 13
Mem address in 347 .. 360
At 138, r2 in (mem content values)
DTC targets = values of r2
Value-analysis complications...

- Code address usually 16 or 32 bits
- Memory addressing unit is usually 8 bits
  - therefore, case-index is multiplied by 2 or 4 in table access

**Code (pseudo)**

```
134 r1 := <computed>
135 cmp r1,13 (unsigned)
136 if r1>13 jump 204
137 r1 := 2 * r1
138 r2 := mem[347+r1]
139 jump via r2
```

Assume code address is 16 bits, memory unit is 8 bits
Value-analysis complications...

- Code address usually 16 or 32 bits
- Memory addressing unit is usually 8 bits
  - therefore, case-index is multiplied by 2 or 4 in table access

**Code (pseudo)**

```plaintext
134 r1 := <computed>
135 cmp r1,13 (unsigned)
136 if r1>13 jump 204
137 r1 := 2 * r1
138 r2 := mem[347+r1]
139 jump via r2
```

- At 137, (unsigned) r1 in 0 .. 13
- At 138, (unsigned) r1 in 0 .. 26
- Mem address in 347 .. 373
- At 139, r2 in (mem content values)
- DTC targets = values of r2
### Value-analysis complications: strides

**Address Table (assume little-endian, \( A_i = \text{address of case } i \))**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Low Octet</th>
<th>High Octet</th>
</tr>
</thead>
<tbody>
<tr>
<td>347</td>
<td>A0</td>
<td>147</td>
</tr>
<tr>
<td>348</td>
<td>A0 high</td>
<td>0</td>
</tr>
<tr>
<td>349</td>
<td>A1 low</td>
<td>202</td>
</tr>
<tr>
<td>350</td>
<td>A1 high</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>372</td>
<td>A12 high</td>
<td>1</td>
</tr>
<tr>
<td>373</td>
<td>A13 low</td>
<td>158</td>
</tr>
<tr>
<td>374</td>
<td>A13 high</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Odd table offset** = spurious address
  - low octet = 0
  - high octet = 202
  - value = 51712

- **Even table offset** = valid address

- The interval 0 .. 26 overestimates the product \( 2 \times r_1 \)
  - if congruence (stride) is not included in abstract domain

- Precise value set for \( 2 \times r_1 \) is "0 .. 26 with stride 2"; a CLP
Iterative CFG construction - 1

CFG vs 1

Diagram with nodes and arrows.
Iterative CFG construction - 2

CFG vs 1

CFG vs 2

Analyse DTC
Iterative CFG construction - 3

CFG vs 1

CFG vs 2

Analyse DTC
Iterative CFG construction - 4

CFG vs 1

Analyse DTC

CFG vs 2

?
Iterative CFG construction - 5

CFG vs 1

Analyse DTC

CFG vs 2
Iterative CFG construction - 6

CFG vs 1

CFG vs 2

CFG vs 3 (final)

Analyse DTC
Study questions and methods

• Questions: given a very accurate value-analysis,
  – here: SWEET Abstract Execution with CLP and no merge,
  – can it handle tricky switch-case DTC code?
  – can it work as well as the pattern-based method?
  – or, are the two methods complementary?

• Study methods:
  – apply both analyses to set of examples/benchmarks
  – compare success/failure rate
  – understand success/failure reasons in detail

• Modular combination of two analysis tools
  – machine code $\Rightarrow$ Bound-T $\Rightarrow$ (ALF code) SWEET
  – SWEET results $\Rightarrow$ Bound-T $\Rightarrow$ (more ALF) SWEET ...

• Target: Atmel AVR, 8/16-bit microcontroller
Our tools: Bound-T and SWEET

• Bound-T: a traditional static WCET tool from Tidorum Ltd
  – input machine code, construct CFG, compute WCET bound

• Bound-T DTC analysis: code patterns guide other analyses
  – dense table indices: Presburger Arithmetic value analysis
  – sparse look-up tables: partial evaluation [7]

• SWEET: multi-purpose tool from Mälardalen University
  – input program in ALF language
  – value analysis with Abstract Interpretation (AI)
  – value and control-flow analysis with Abstract Execution (AE)
  – some kinds of WCET analysis (not used in this work)

• SWEET value-analysis domains:
  – Intervals with Wrap-Around, finite-width integers (IWA)
  – Circular Linear Progressions (CLP)
  – polyhedra (not used here)
Abstract Execution in SWEET

- Abstract Execution (J. Gustafsson et al.) is a form of AI, but
  - does not use widening to force convergence
    - thus risks non-termination of analysis
  - records control flow, including loop iterations
    - produces loop bounds, feasible/infeasible paths, ...
  - allows control of value merging at control-flow join points
    - "no merge" converts domain to \textit{powerset} of domain
      - interval $\Rightarrow$ set of intervals
      - CLP $\Rightarrow$ set of CLPs
  - "no merge" is feasible for \textit{local} analyses
    - switch-case DTC-analysis is local
    - general function-pointer analysis is global
- AE can use any domain supported by SWEET
  - here: either IWA or CLP
Tool combination and iteration
Examples/benchmarks

- Selected from Bound-T test suite
  - most forms of switch-case DTC Tidorum has seen
  - focus on machine-code form, not source-code form
  - small, because analysis is local

- P1: dense cases: indexed jump into table of jumps
- P2: dense cases: get address from table, jump to it
- P3: sparse cases: look-up table from ref. [7]
- P4: sparse cases: look-up table from IAR compiler
- P5: dense cases: look-up table from IAR compiler
- P6: two-index switch-case, 4 x 4, 16-entry address table
- P7: variant of P6 with different code
<table>
<thead>
<tr>
<th>Program</th>
<th>Bound-T</th>
<th>SWEET IWA</th>
<th>SWEET CLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Exact result, but needs annotation</td>
<td>Exact result</td>
<td>Exact result</td>
</tr>
<tr>
<td>P2</td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td>P3</td>
<td>Fail</td>
<td>Exact result</td>
<td>Exact result</td>
</tr>
<tr>
<td>P4</td>
<td>Exact result</td>
<td>Exact CFG, WCET overest.</td>
<td>Exact CFG, WCET overest.</td>
</tr>
<tr>
<td>P5</td>
<td>Exact result</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td>P6</td>
<td>Fail</td>
<td>Fail</td>
<td>Exact result</td>
</tr>
<tr>
<td>P7</td>
<td>Exact result</td>
<td>Exact result</td>
<td>Exact result</td>
</tr>
</tbody>
</table>
Statistics

- Bound-T (patterns): 4 successes, 3 failures
- SWEET (value-analysis): 5 successes, 2 failures
- Combined: 6 successes, 1 failure (P2)

⇒ Value-analysis with AE and CLP works quite well
⇒ Some complementarity with pattern-based analysis

- CLP wins over IWA in only one case (P6) - why?
Reasons for failures

• Code pattern not known (Bound-T):
  – P2, P3

• Imprecise instruction modelling (carry-out):
  – Bound-T on P6

• Lack of congruence in domain (SWEET IWA):
  – P2, P6

• Loss of congruence information (SWEET CLP):
  – P5

• Merging of values from array (SWEET CLP):
  – P2

• Non-relational domain (SWEET IWA and CLP):
  – P5
Loss of congruence - when is $x+x = 2x$?

- Multiplication by 2 is often compiled into addition

**Instead of ...**

\[
137 \ r1 := 2 \times r1
\]

**... we get this code:**

\[
137 \ r1 := r1 + r1
\]

<table>
<thead>
<tr>
<th>Domain</th>
<th>$x+x = 2x$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete values</td>
<td>yes</td>
</tr>
<tr>
<td>Intervals (IWA)</td>
<td>yes</td>
</tr>
<tr>
<td>Circular Linear Progressions (CLP)</td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td>Polyhedra</td>
<td>yes</td>
</tr>
<tr>
<td>Presburger Arithmetic</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Bound-T's ALF generator was modified to emit $x+x$ as $2x$
  - but (so far) only locally, per instruction
  - still problems for e.g. $y := x; y := y+x;$ using two instructions
Problems from non-relational domain

- The "x+x \neq 2x" problem can be seen as one example
- Other example: failure of SWEET CLP on P5:

\begin{verbatim}
103 r2 := 347 + r1
104 cmp r1,13 (unsigned)
105 if r1>13 jump 204
106 r2 := r2 + r1
107 r3 := mem[ r2 ]
108 jump via r3
\end{verbatim}

The computation of 347 + 2*r1 is split
Non-relational domain problems

- The "x+x ≠ 2x" problem can be seen as one example
- Other example: failure of SWEET CLP on P5:

```
103  r2 := 347 + r1
104  cmp r1,13 (unsigned)
105  if r1>13 jump 204
106  r2 := r2 + r1
107  r3 := mem[r2]
108  jump via r3
```

Constraints from later instructions do not constrain earlier uses of the same variable!

In relational domains, constraints from later instructions can be applied to relations derived from earlier instructions.
Conclusions

• Value-analysis (AE+CLP) is **promising** for switch-case DTC
  – to some extent complementary with pattern methods
  – also produces flow facts, as a bonus (not used here)

• Circular Linear Progressions is a "**fragile**" domain
  – more powerful than intervals and IWA
  – but only if "well fed" with program in good form
    • e.g. with 2x instead of x+x
  – and still non-relational
    • fragile with respect to instruction ordering

• Tentative solution = possible future work
  – restructure computation **before** submitting to AE+CLP:
    • collect instruction sequences into **affine expressions**
    • propagate **constraints** to all subject variable uses
Atmel AVR8

- 8/16 bit microcontroller, a little RISCy (load-store)
  - few true 16-bit operations, not orthogonal
  - hard case for analysing indexing and address arithmetic

- Separate code and data memories (Harvard arch)

- Switch-case tables are usually in code space (flash)
  - special instruction LPM: Load from Program Memory
  - LPM reads 8 bits: need two LPMs to read 16-bit address
    - increment table address/index in between
    - combine two octets to 16-bit DTC address
  - DTC usually with IJMP: Indirect Jump
  - switch-case DTC code is rather long...

- Compilers used: gcc, IAR, assembler
  - compiler chosen to get desired form of machine code