Sequoll: a framework for model checking binaries

Bernard Blackham, Gernot Heiser

School of Computer Science & Engineering, UNSW
Software Systems Research Group, NICTA
Sydney, Australia
Motivation

• The desire for a *trustworthy* kernel to build reliable *mixed-criticality real-time* systems
Motivation

• The desire for a trustworthy kernel to build reliable multi-criticality real-time systems

• Using seL4 to guarantee:
  – functional correctness through formal proof  
    (Klein et al., SOSP 2009)
  – timing constraints through sound WCET analysis  
    (Blackham et al., RTSS 2011)
Motivation

• Current analysis uses annotations to specify:
  – loop counts
  – infeasible paths

• We want to reduce scope for errors in WCET analysis.
Results

Estimated worst-case execution time of seL4

- Without infeasible path information: 213
- With infeasible path information: 657

000’s of cycles

© NICTA 2013
seL4 is large

- Small by microkernel standards
- Large by WCET standards

<table>
<thead>
<tr>
<th>C source</th>
<th>Binary (ARM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~8,700 lines</td>
<td>~10,000 instructions</td>
</tr>
<tr>
<td>316 functions</td>
<td>228 functions</td>
</tr>
<tr>
<td>76 loops</td>
<td>56 loops</td>
</tr>
<tr>
<td></td>
<td>2,384 basic blocks</td>
</tr>
<tr>
<td></td>
<td>~400,000 basic blocks when inlined</td>
</tr>
</tbody>
</table>
WCET computation process

1. Compute WCET
2. Reconstruct path
3. Path valid?
   - Y: Success!
   - N: Add infeasible path criteria

© NICTA 2013
Infeasible paths
Infeasible paths

\[ a \leq 3 \quad \text{or} \quad a > 3 \]
Infeasible paths

\[ a \leq 3 \quad \text{or} \quad a > 3 \]
Infeasible paths
Infeasible paths

\[
a = 0 \\
a \neq 0
\]

\[
b = 0 \\
b \neq 0
\]
Infeasible paths

\[ b \neq 0 \]

\[ a = 0 \]

\[ a \neq 0 \]

\[ b = 0 \]

\[ b \neq 0 \]
Pruning infeasible paths

Express constraints as one of:

- \( a \) conflicts with \( b \) when called under \( f \)
- or
- \( a \) is consistent with \( b \) when called under \( f \)
Infeasible paths

conflicts with/
is consistent with

© NICTA 2013

Wednesday, 10 April 13
Verifying annotations

• How can we verify these manual annotations?

• For two basic blocks \( a, b \):
  – show all paths execute \( a \) and \( b \) the same number of times
  – show all paths execute at most one of \( a \) or \( b \)

• For some loop \( \ell \) in the binary:
  – How many times can \( \ell \) iterate?
Why model checking?

- Many infeasible paths and even some loop bounds cannot be determined without program invariants.

- Invariants are known to the formal proof.

- Formal proof statements and invariants are a natural fit for model checkers.
Why model checking?

```c
int count_bits(uint32_t x) {
    int c = 0;
    while (x != 0) {
        x &= x - 1; /* clear lowest bit */
        c++;
    }
    return c;
}
```

What loop bound?
From binary → model checker

ARM formalization, CFG extraction

Semantic representation

Symbolic execution, SSA

SSA representation

Simplification, slicing

Reduced CFG

Property of interest

Symbolic Model

Model checker

True/False!
Extracting instruction semantics

- Reused existing formalization of ARM ISA
  (Fox & Myreen, ITP 2010)

machine code: \texttt{E2813002}
add r3, r1, #2
r3 ← r1 + 2
r15 ← r15 + 4

machine code: \texttt{E8AD0028}
stmia sp!, \{r3, r5\}
mem r13 ← r3<7:0>
mem (r13 + 1) ← r3<15:8>
mem (r13 + 2) ← r3<23:16>
mem (r13 + 3) ← r3<31:24>
mem (r13 + 4) ← r5<7:0>
mem (r13 + 5) ← r5<15:8>
mem (r13 + 6) ← r5<23:16>
mem (r13 + 7) ← r5<31:24>
r13 ← r13 + 8
r15 ← r15 + 4
Identifying properties of interest

How many times can this node execute?
Computing a program slice

Find all nodes which can impact upon execution of that block through:

- data dependencies
- control dependencies
Reducing the CFG further
Model checking the CFG

\[
\begin{align*}
\text{init}(n) & := 0 \\
\text{next}(n) := \begin{cases} 
 0 & n=0 \\
 1 & n=1 \land \text{cond}_1 \\
 2 & n=2
\end{cases} \\
\text{init}(c) & := 0 \\
\text{next}(c) := \begin{cases} 
 0 & n=0 \\
 c + 1 & n=1 \\
 c & \text{else}
\end{cases}
\end{align*}
\]

\text{VAR} \\
n : -1..2; \\
memRead : unsigned word[8]; \\
\text{COND}_3 : boolean; \\
r3_2 : unsigned word [32]; \\
c : unsigned word[32];

\text{DEFINE} \\
\text{psrZ}_3 := (((1) >= 32) ? 0ud32_0 : (r3_2) >> (1)) = (0ud32_0); \\
\text{cond}_1 := \neg\text{psrZ}_1; \\
\text{r3}_1 := r0_0; \\
\text{r3}_3 := (((1) >= 32) ? 0ud32_0 : (r3_2) >> (1)); \\
\text{psrZ}_1 := r0_0 = (0ud32_0); \\
\text{ASSIGN} \\
\text{next}(%20) := \begin{cases} 
 0 & n=1 \\
 \text{cond}_5 & n=0 \\
 \text{cond}_2 & \text{TRUE}
\end{cases} \\
\text{next}(r3_2) := \begin{cases} 
 r3_3 & n=1 \\
 r3_1 & n=0 \\
 r3_2 & \text{TRUE}
\end{cases}
Model checking the CFG

For loop counts:

Ask a model checker: is $c < k$?
(and binary search for $k$)
Model checking the CFG

For conflict constraints:

Add “visited” flag for nodes a and b

Ask a model checker to ensure that a and b are never both true

In CTL: $\text{AG } ! (\text{visitedA } \& \text{ visitedB})$
Model checking the CFG

For consistency constraints:

Add “count” variables for nodes $a$ and $b$

Ask a model checker to ensure counts for $a$ and $b$ are equal (eventually)

In CTL: $\text{AF} (\text{countA} = \text{countB})$
Loop bound verification

Can compute 18/32 loop bounds in seL4:

- 1 loop depends upon invariants in the proof

- 1 loop cannot be bounded due to complex exit conditions
  • model checker attempts to find the smallest loop bound
  • complex state space must be explored to deduce bound

- 12 loops, all identical in structure, cannot be bounded due to poor memory alias analysis
Loop bound verification

```c
void f(uint32_t n)
{
    uint32_t i = 1 << n;

    if (i > 256)
        i = 256;

    while (i > 0) {
        ...
        i -= 4;
    }
}
```

If $n \geq 32$, $i$ is undefined
If $n \leq 1$, loop is infinite
Infeasible paths

Of 35 infeasible path constraints

–4 validated

–1 shown untrue (oops!)

–11 cannot be validated without better alias analysis

–19 depend on kernel invariants
Results

Estimated worst-case execution time of seL4

- Without infeasible path information: 213
- With verified infeasible path information: 481
- With all infeasible path information: 657

000’s of cycles
Research directions

• Integrate with proof invariants
• Automate infeasible path detection (WIP)
• Use a faster ISA formalization
Summary

Sequoll is able to:

• apply model checkers to reason about compiled ARM binaries
• validate manual infeasible path annotations
• compute “interesting” loop bounds
• (eventually) integrate formal proof with infeasible path information

⇒ reduce scope for errors in WCET analysis!

Download it!

Bernard.Blackham@nicta.com.au