Translation Validation for a Verified OS Kernel

Thomas Sewell[1] Magnus Myreen[2] Gerwin Klein[1]

[1]UNSW & NICTA

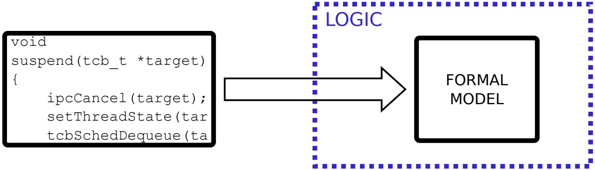
[2]University of Cambridge

12 Feb 2013

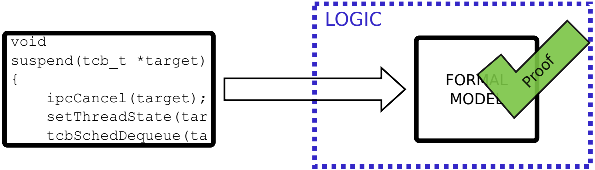
Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

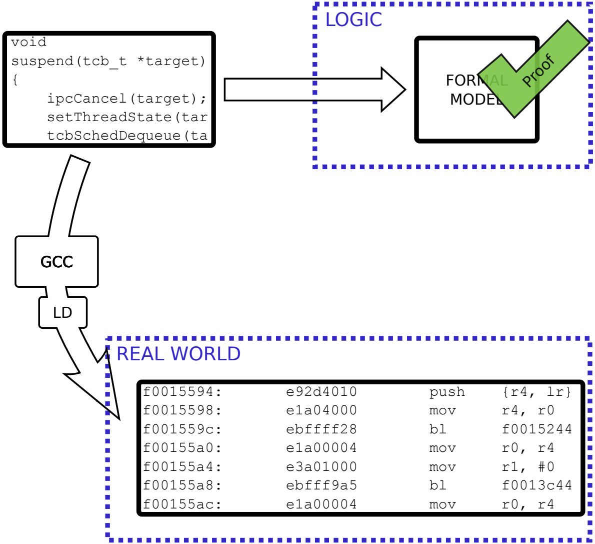
Overview Picture



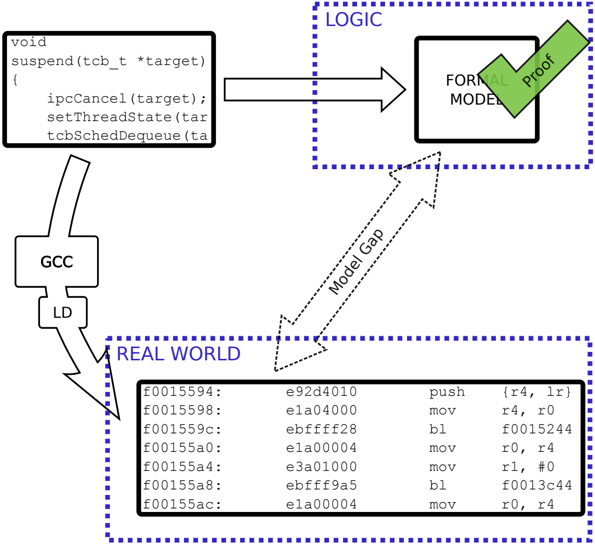
Overview Picture



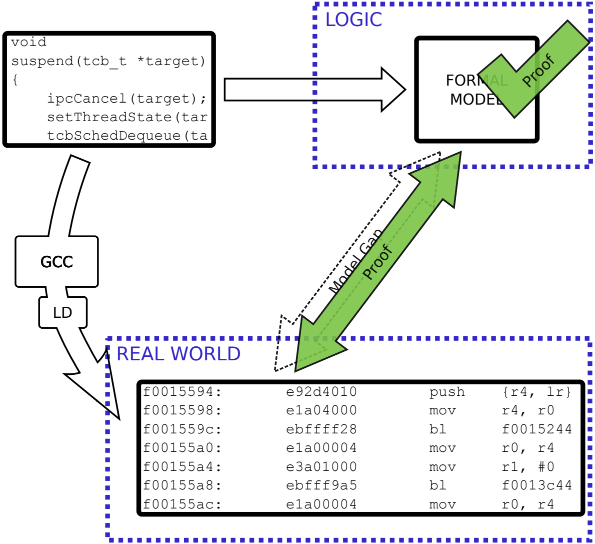
Overview Picture



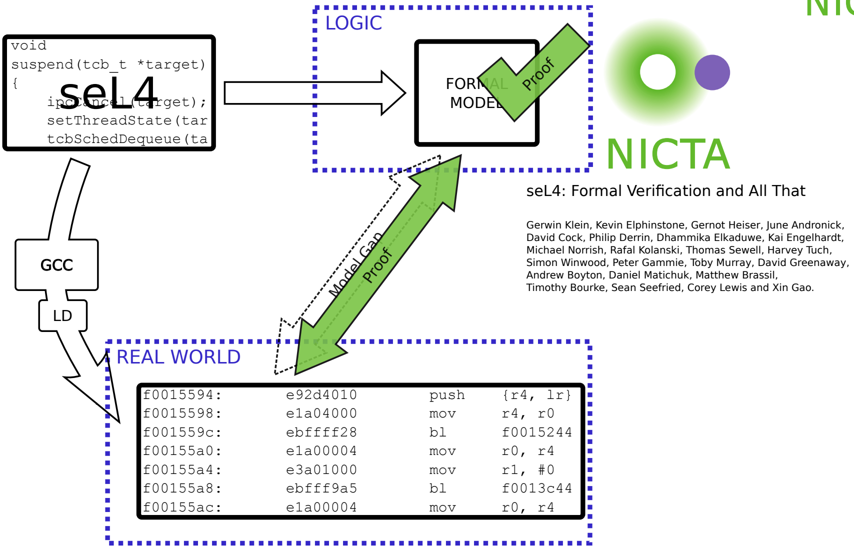
Overview Picture



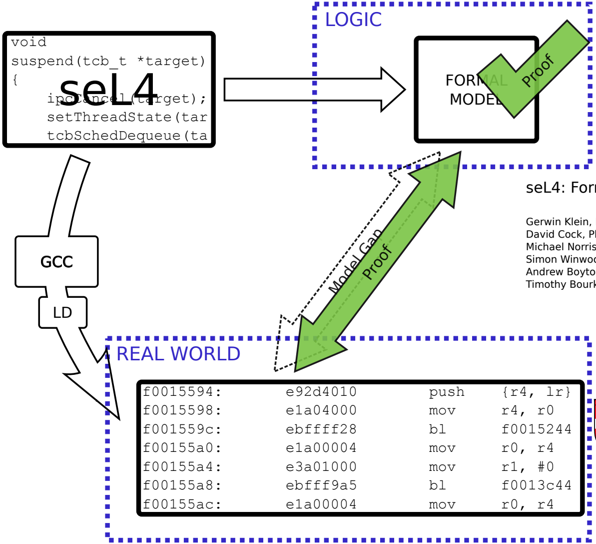
Overview Picture



Overview Picture



Overview Picture



NICTA

NICTA

seL4: Formal Verification and All That

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, Simon Winwood, Peter Gammie, Toby Murray, David Greenaway, Andrew Boyton, Daniel Matichuk, Matthew Brassil, Timothy Bourke, Sean Seefried, Corey Lewis and Xin Gao.

UNIVERSITY OF CAMBRIDGE

ARM ISA and Decompiler

Anthony Fox and Magnus Myreen

Overview

This talk in one bullet point:

- We can prove the binary refines the formal model, for seL4's verified components and `gcc-4.5.1 -O1`.

Overview

NICTA

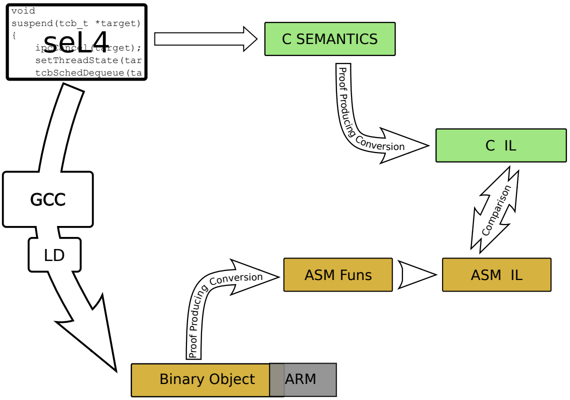
This talk in one bullet point:

- We can prove the binary refines the formal model, for seL4's verified components and `gcc-4.5.1 -O1`.

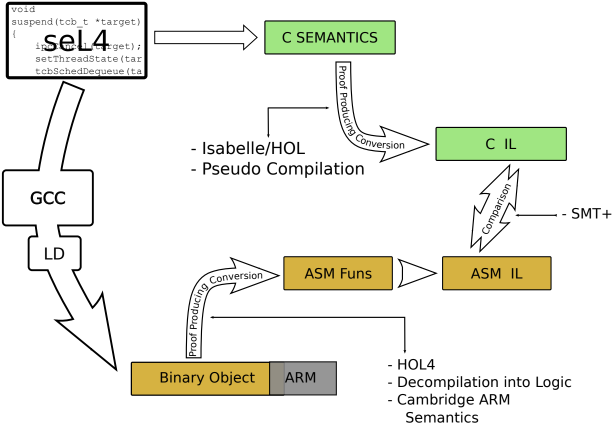
Talk contents:

- Structure of proof and approach.
  - Alternative approaches.
  - C Semantics & C Standard.
  - Challenges.
  - Restrictions & workarounds.

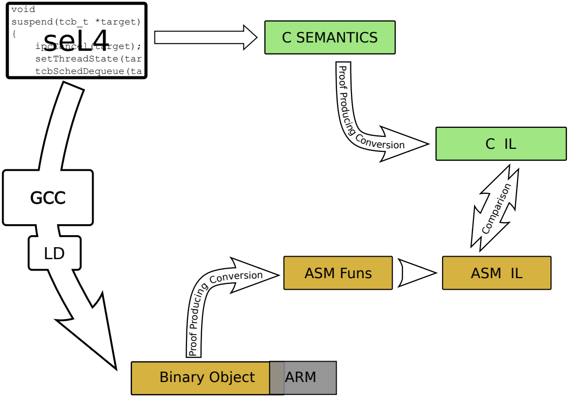
Approach


void
suspend(tcb_t *target)
{
ipcCancel(target);
setThreadState(tar
tcbSchedDequeue(ta
seL4
C SEMANTICS
Proof Producing Conversion
C IL
Comparison
GCC
LD
ASM Funs
ASM IL
Proof Producing Conversion
Binary Object
ARM

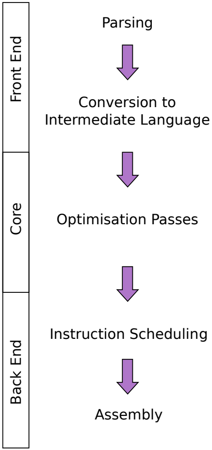
Approach


void
suspend(tcb_t *target)
{
ipcCancel(target);
setThreadState(tar
tcbSchedDequeue(ta
seL4
C SEMANTICS
Proof Producing Conversion
C IL
- Isabelle/HOL
- Pseudo Compilation
GCC
LD
Comparison
- SMT+
ASM Funs
ASM IL
Proof Producing Conversion
Binary Object
ARM
- HOL4
- Decompilation into Logic
- Cambridge ARM
Semantics

Approach

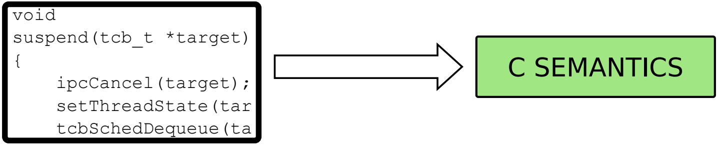


void
suspend(tcb_t *target)
{
ipcCancel(target);
setThreadState(tar
tcbSchedDequeue(ta
seL4
C SEMANTICS
Proof Producing Conversion
C IL
Comparison
GCC
LD
ASM Funs
ASM IL
Proof Producing Conversion
Binary Object
ARM



Front End
Core
Back End
Parsing
Conversion to Intermediate Language
Optimisation Passes
Instruction Scheduling
Assembly

C Program Semantics



C Program Semantics

Maps syntax of C to a deeply embedded language in Isabelle/HOL with an operational semantics.

```
if (...)  {...}  ⇒  IF (...)  THEN ...  FI
f (1, 2);        ⇒  CALL f_'proc (1, 2);;
x ++;            ⇒
                        (´x :== ´x + 1);;

*p = *q;         ⇒

                        mem :== h_upd ´p
                          (h_val ´q ´mem) ´mem
```

C Program Semantics

Maps syntax of C to a deeply embedded language in Isabelle/HOL with an operational semantics.

Partial semantics to explain **undefined behaviour.**

```
if (...)  {...}  ⇒  IF (...)  THEN ...  FI
f (1, 2);        ⇒  CALL f_'proc (1, 2);;
x ++;            ⇒  Guard {´x <=s ´x + 1}
                        (´x :== ´x + 1);;
*p = *q;         ⇒  Guard {ptr_valid ´p}
                      Guard {ptr_valid ´q}
                        mem :== h_upd ´p
                          (h_val ´q ´mem) ´mem
```

C Standard Semantics

Aside: Why not just trust the compiler?

The `ptr_valid` assertion used in `Guard` is subtle.

The **object rule** says that a pointers may come from arithmetic within an object, `&` and `malloc`.

C Standard Semantics

NICTA

Aside: Why not just trust the compiler?

The `ptr_valid` assertion used in `Guard` is subtle.

The **object rule** says that a pointers may come from arithmetic within an object, `&` and `malloc`.

What about casts from numbers?

```
(pt_t *)(pt[x] & 0xFFFFF000)
```

C Standard Semantics

Aside: Why not just trust the compiler?

The `ptr_valid` assertion used in `Guard` is subtle.

The **object rule** says that a pointers may come from arithmetic within an object, `&` and `malloc`.

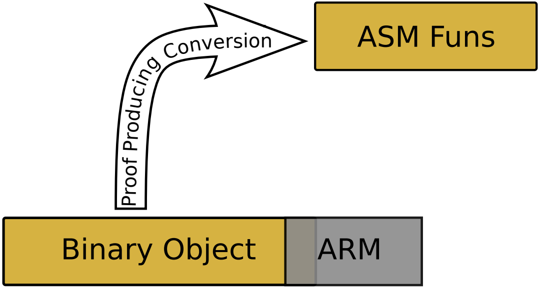
What about casts from numbers?

```
(pt_t *)(pt[x] & 0xFFFFF000)
```

There are multiple interpretations of the C language.

- **NICTA seL4:** Liberal, portable assembler, soundy.
  - Strict aliasing rule but not object rule.
- **CompCert:** Conservative.

Decompilation



Example Decompilation

NICTA

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

```
<avg>:
avg+0    e0810000    add r0, r1, r0     // add r1 to r0
avg+4    e1a000a0    lsr r0, r0, #1     // shift r0 right
avg+8    e12fff1e    bx  lr             // return
```

Example Decompilation

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

```
<avg>:
avg+0    e0810000    add r0, r1, r0    // add r1 to r0
avg+4    e1a000a0    lsr r0, r0, #1    // shift r0 right
avg+8    e12fff1e    bx  lr            // return
```

$$
\begin{aligned}
\mathrm{avg}\,(r_0, r_1) \quad = \quad & \mathrm{let}\ r_0 = r_1 + r_0\ \mathrm{in} \\
& \mathrm{let}\ r_0 = r_0 >\!>\!> 1\ \mathrm{in} \\
& \quad r_0
\end{aligned}
$$

Example Decompilation

NICTA

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

```
<avg>:
avg+0    e0810000     add r0, r1, r0     // add r1 to r0
avg+4    e1a000a0     lsr r0, r0, #1     // shift r0 right
avg+8    e12fff1e     bx  lr             // return
```

$$\begin{aligned}\text{avg}\,(r_0, r_1) \quad = \quad & \text{let } r_0 = r_1 + r_0 \text{ in} \\ & \text{let } r_0 = r_0 \ggg 1 \text{ in} \\ & \quad r_0\end{aligned}$$

$$\{\, \text{R0}\ r_0 * \text{R1}\ r_1 * \text{R14}\ lr * \text{PC}\ p \,\}$$
$$p : \texttt{e0810000 e1a000a0 e12fff1e}$$
$$\{\, \text{R0}\ (\text{avg}\ (r_0, r_1)) * \text{R1}\ \_ * \text{R14}\ \_ * \text{PC}\ lr \,\}$$

Challenges for Decompilation

```
uint avg8 (uint i1, i2, i3, i4, i5, i6, i7, i8) {
  return (i1+i2+i3+i4+i5+i6+i7+i8) / 8;
}
```

Challenges for Decompilation

```
uint avg8 (uint i1, i2, i3, i4, i5, i6, i7, i8) {
  return (i1+i2+i3+i4+i5+i6+i7+i8) / 8;
}
```

```
<avg8>:
e0811000     add r1, r1, r0
e0811002     add r1, r1, r2
e59d2000     ldr r2, [sp]        // load
e0811003     add r1, r1, r3
e0810002     add r0, r1, r2
e99d000c     ldmib sp, {r2, r3}  // load
e0800002     add r0, r0, r2
e0800003     add r0, r0, r3
e59d300c     ldr r3, [sp, #12]   // load
e0800003     add r0, r0, r3
e1a001a0     lsr r0, r0, #3
e12fff1e     bx   lr
```

Stack and Heap

Aside: Hiding stack accesses mean they must not be aliased.

Our C semantics forbids pointers to the stack.

We also eliminate padding, clearly separating:

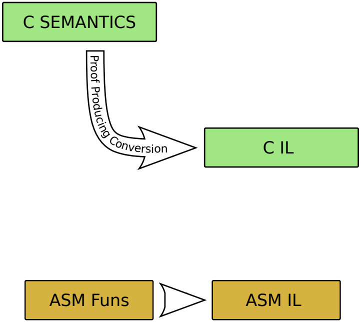
- the heap, under user control.
- the stack, under compiler control.

Enables a simple notion of correct compilation:

$$\forall(\mathit{in}, \mathit{in\_heap}) \in \mathrm{domain}(\mathfrak{C}).\ \mathfrak{C}(\mathit{in}, \mathit{in\_heap}) = \mathfrak{B}(\mathit{in}, \mathit{in\_heap})$$

This would be difficult with higher level optimisations.

Conversion to Graph



Not going to discuss this in detail.

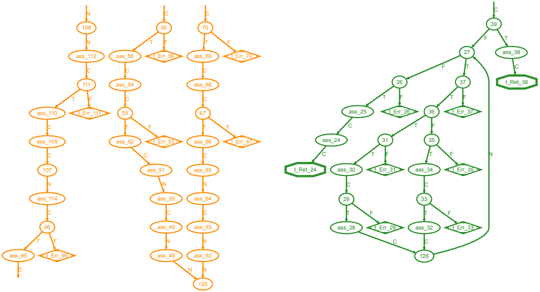
Graph Refinement

The proof of refinement between graphs involves two processes:

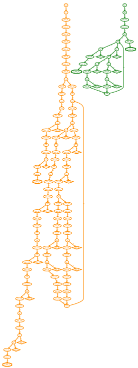
- A search process, which heuristically discovers a proof object.
- A check process, which checks the proof is sound.

This follows Pnueli's translation validation design.

Both processes use SMT solving extensively.



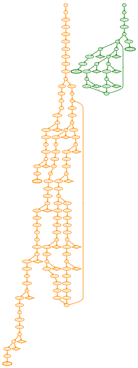
Proof Objects



Proof objects contain:

- An **inlining** of all needed function bodies into one space.

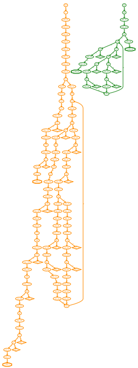
Proof Objects



Proof objects contain:

- An **inlining** of all needed function bodies into one space.
- **Restrict** rules, which observe that a given point in a loop may be reached only *n* times.

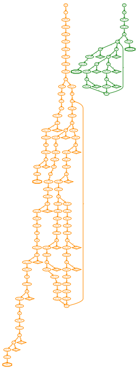
Proof Objects



Proof objects contain:

- An **inlining** of all needed function bodies into one space.
- **Restrict** rules, which observe that a given point in a loop may be reached only *n* times.
- **Split** rules, which observe that a C loop point is reached as often as a loop point in the binary.

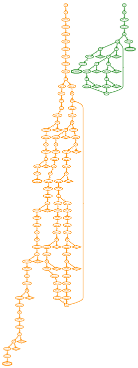
Proof Objects



Proof objects contain:

- An **inlining** of all needed function bodies into one space.
- **Restrict** rules, which observe that a given point in a loop may be reached only *n* times.
- **Split** rules, which observe that a C loop point is reached as often as a loop point in the binary.
    - Checked by *k*-induction.

Proof Objects



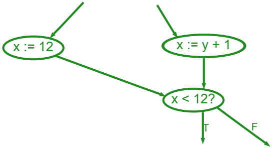
Proof objects contain:

- An **inlining** of all needed function bodies into one space.
- **Restrict** rules, which observe that a given point in a loop may be reached only *n* times.
- **Split** rules, which observe that a C loop point is reached as often as a loop point in the binary.
  - Checked by *k*-induction.
  - Parameter *eqs* must relate enough of binary state to C state to relate events after the loop.

SMT

SMT problems generated contain:

- Fixed-length values and arithmetic: `word32, +, -, <=` etc.
- Arrays to model the heap: `heap :: word30 => word32`.
- If-then-else operators to handle multiple paths.

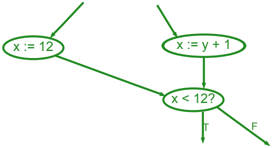


- Validity assertions and needed inequalities:
  `ptr1_valid` & `ptr2_valid` $\Rightarrow ptr1 > ptr2 + 7 \vee ptr2 > ptr1 + 15$.

SMT

SMT problems generated contain:

- Fixed-length values and arithmetic: `word32`, `+`, `-`, `<=` etc.
- Arrays to model the heap: `heap :: word30 => word32`.
- If-then-else operators to handle multiple paths.



- Validity assertions and needed inequalities:
  `ptr1_valid` & `ptr2_valid` $\Rightarrow ptr1 > ptr2 + 7 \vee ptr2 > ptr1 + 15$.

Strong compatibility with **SMTLIB2 QF_ABV**.

Strong similarity to **QF_ABV** category of the SMT competition.

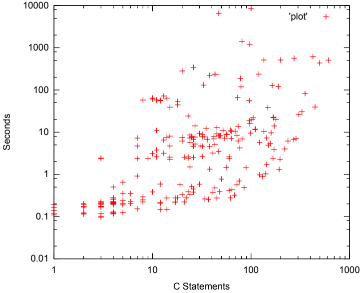
We ran this experiment with Z3 (version 4.0) and SONOLAR (version 2012-06-14).

The solvers are efficient at producing both `sat` and `unsat` results, which is important in discovering and checking a proof.

Results

The proof rules and inlining heuristic mentioned are sufficient for seL4's verified code with `gcc-4.5.1 -O1`.

Nested loops and some `-O2` loop optimisations are not yet handled.



Conclusions

Translation validation can scale up to substantial problem size, using naive approaches, for a carefully managed problem.

Supporting factors:

- Simple looping structure.
- C Semantics already at the level of bits and bytes.
- Clear separation of compiler and user control.
- Strong compatibility with SMT QF_ABV.

Software is available at
http://www.ssrg.nicta.com.au/software/TS/graph-refine