Verification of the GCC-generated binary of the seL4 microkernel

Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹

¹ Data61 CSIRO & UNSW, Australia
² Chalmers, Sweden

the clever PhD student who did the hard part of the work
today’s speaker (borrowing some slides from Sewell)
L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly
> 500,000 lines of Isabelle/HOL proofs
Assumptions in L4.verified

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
- Cambridge ARM model

The aim of this work is to remove the first assumption.
And also to validate L4.verified’s C semantics.
Aim: extend downwards

- high-level design
- low-level design
- detailed model of C code
- real C code
- Haskell prototype

existing L4.verified work

Aim: remove need to trust C compiler and C semantics
Connection to CompCert

Incompatible:
- different view on what valid C is
- CompCert C is more conservative
- pointers & memory more abstract in CompCert C sem.
- different provers (Coq and Isabelle)
Using Cambridge ARM model

- Cambridge ARM model
- Detailed model of C code
- Low-level design
- High-level design
- Haskell prototype
- Real C code
- Machine code as graph
- Decompilation
- Refinement proof
- sel4 machine code
- Cambridge ARM model
- gcc (not trusted)

Existing L4 verified work

New extension
Translation validation

Translation Validation efforts:

- Many others for many languages and levels of connection to compilers.
- …
Talk outline

Part 1: proof-producing decompilation
- generating functions / graphs
- stack vs heap

Part 2: pseudo compilation and SMT refinement proof
- C semantics
- SMT proof search and proof checking
- examples
- complicated cases
Cambridge ARM model

- detailed model of the ARM instruction set architecture formalised in HOL4 theorem prover
- originates in a project on hardware verification (ARM6 verification)
- extensively tested against different hardware implementations

Web: http://www.cl.cam.ac.uk/~acjf3/arm/
Part 1: decompilation

- Cambridge ARM model
- seL4 machine code
- machine code as functions

↑ decompilation

first version produced functions; latest version produces graphs
Sample C code:

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

machine code:

```
gcc (not trusted)
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx   lr
```

HOL4 certificate theorem:

```plaintext
{ R0 i * R1 j * LR lr * PC p }  
p : e0810000 e1a000a0 e12fff1e  
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```

Resulting function:

```plaintext
avg (r0, r1) = let r0 = r1 + r0 in
    let r0 = r0 >> 1 in
    r0
```

Decompilation

- word arithmetic
- word right-shift
- return instruction
- separation logic:
Decompilation

```
{ R0 i * R1 j * PC p }
p+0 :
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 :
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC lr }
p+8 :
{ LR lr * PC lr }

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 ((i+j)>>1) * R1 j * LR lr * PC lr }
```

How to decompile:
1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)

```
e0810000 add r0, r1, r0
e1a000a0 lsr r0, r0, #1
e12fff1e bx lr
```

```
e0810000 e1a000a0 e12fff1e
```
Decompiling seL4: **Challenges**

- seL4 is ~12,000 ARM instructions (lines of assembly)
  - ✓ decompilation is compositional
- compiled using gcc -O1 and gcc -O2
  - ✓ gcc implements ARM+C calling convention
- must be compatible with L4.verified proof
  - ➡ stack requires special treatment
Some arguments are passed on the stack, and cause memory ops in machine code that are not present in C semantics.

C code:
```
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Stack is visible in machine code:
```
add r1, r1, r0
add r1, r1, r2
ldr r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr r3, [sp, #12]
add r0, r0, r3
lsr r0, r0, #3
bx lr
```
Solution (early version)

Use separation-logic inspired approach

stack pointer: sp

stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

3 slots of unused but required stack space

rest of stack

disjoint due to *

separation logic:*
**Solution**

**Method:**

1. static analysis to find stack operations,
2. derive stack-specific Hoare triples,
3. then run decompiler as before.

The new triples make it seem as if stack accesses are separate from the rest of memory.
Result (early version)

Stack load/stores become straightforward assignments.

Disadvantage: the automation is trying to prove stack safety with sometimes too little information.

What about arrays on the stack?

The new triples make it seem as if stack accesses are separate from the rest of memory.
Later version

Stack load/stores become accesses to “stack memory”.

In certificate theorems:

stack

heap

results in proof obligations higher up
Correct memory after compilation

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:

$$\forall (in, in\_heap) \in \text{domain}(\mathcal{C}). \mathcal{C}(in, in\_heap) = \mathcal{B}(in, in\_heap)$$

This would be difficult with higher level optimisations.
Other tricky cases

- **struct as return value**
  - case of passing *pointer of stack location*
  - stack approach is strong enough
- **switch statements**
  - *position dependent*
  - must decompile linked elf-files, not object files
- **infinite loops in C**
  - make *gcc produce strange output*
  - must be pruned from control-flow graph
Latest decompiler

- produces a graph instead of a function
  - functions are good for interactive proofs
  - graphs seem better for automation here

\[
\text{avg8}(r0, r1, r2, r3, sp, stack) = \\
\quad \text{let } r1 = r1 + r0 \text{ in} \\
\quad \text{let } r1 = r1 + r2 \text{ in} \\
\quad \text{let } r2 = \text{stack}(sp) \text{ in} \\
\quad \text{let } r1 = r1 + r3 \text{ in} \\
\quad \text{let } r0 = r1 + r3 \text{ in} \\
\quad \text{let } (r2, r3) = (\text{stack}(sp+4), \text{stack}(sp+8)) \text{ in} \\
\quad \text{let } r0 = r0 + r2 \text{ in} \\
\quad \text{let } r0 = r0 + r3 \text{ in} \\
\quad \text{let } r3 = \text{stack}(sp+12) \text{ in}
\]

```
Assign r1 := r1 + r0
Assign r1 := r1 + r2
Assign r2 := stack(sp)
Assign r1 := r1 + r3
```
Moving to Part 2

detailed model of C code
refinement proof
machine code as graph
automatic translation
seL4 machine code

new extension
Moving to Part 2

Questions about Part 1?

… before we continue to Part 2

Sydney Harbour Bridge during construction
Part 2

detailed model of C code

machine code as graph

seL4 machine code

refinement proof

automatic translation
Approach for refinement proof

detailed model of C code

\[\downarrow\]

semantics preserving rewriting

C code as graph

\[\downarrow\]

SMT proof

machine code as graph
the C semantics is produced on import into Isabelle/HOL

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

partial semantics to account for undefined behaviour
Why not just trust the C 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?

\[
(pt_t*)(pt[x] & 0xFFFFFFFF000)
\]

There are multiple interpretations of the C language.

- **NICTA sel4**: Liberal, portable assembler, soundy.
  - Strict aliasing rule but not object rule.
- **CompCert**: Conservative.
Translating C into graphs

```c
struct node *
find (struct tree *t, int k) {
    struct node *p = t->trunk;
    while (p) {
        if (p->key == k)
            return p;
        else if (p->key < k)
            p = p->right;
        else
            p = p->left;
    }
    return NULL;
}
```

1: `p := Mem[t + 4]`;
2: `p == 0 ?`
3: `Mem[p] == k ?`
4: `ret := p;`
5: `Mem[p] < k ?`
6: `p := Mem[p + 4];`
7: `p := Mem[p + 8];`
Bridging the gap

detailed model of C code → semantics preserving rewriting

C code as graph

machine code as graph

seL4 machine code

Cambridge ARM model

latest toolchain designed to have all of its heuristics in this step only

decompilation

SMT proof
The SMT proof step

Following Pnuelli’s original translation validation, we split the proof step:

Part 1: proof search (proof script construction)

Part 2: proof checking (checking the proof script)

The proof scripts consist of a state space description and a tree of proof rules:

Leaf, CaseSplit, Restrict, FunCall and Split

The heavy lifting is done by calls to SMT solvers for both the proof search and checking.
Generated proof scripts

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.
Translating graphs into SMT exps

**Figure 5.** Example Conversion to SMT

Here: ‘pc’ is the accumulated **path condition** and variables (x, y etc.) are **values w.r.t. inputs** (x_i, y_i, etc.)

(The actual translation avoids a blow up in size...)

\[ p_{c, i} = \text{True} \]
\[ x = x_i \]
\[ y = x_i + 1 \]

1. \( y := x + 1 \)

2. \( y < 3 \) ?

3. \( x := y - 12 \)

4. \( z = f(x, y) \)

\[ pc = (x_i + 1) < 3 \]
\[ x = x_i \]
\[ y = x_i + 1 \]

\[ pc = (x_i + 1) < 3 \lor (x_i + 1) \neq 3 \]
\[ x = \text{if} (x_i + 1) < 3 \text{ then } (x_i + 1) - 12 \text{ else } x_i \]
\[ y = x_i + 1 \]
Easy for SMT (1)

```c
int f1 (unsigned int x) {
    return ((x >> 4) & 15) == 3;
}

int f2 (unsigned int x) {
    return (x & (15 << 4)) == (3 << 4);
}

int f3 (unsigned int x) {
    return ((x << 24) >> 28) == 3;
}

int f4 (unsigned int x) {
    return ((x & (15 << 4)) | (3 << 4)) == 0;
}
```

Word games: solved problem.
- “Bit Vector” SMT theory.
Easy for SMT (2)

```c
void
f (struct foo *x, int y) {
    struct foo f = *x;
    f.a += y;
    f.b -= y;
    f = do_the_thing (f);
    *x = f;
}
```

Memory optimisation: mostly solved problem.
- “Array” SMT theory.
- QF_ABV SMT logic.
SMT use summary

SMT problems generated contain:

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

- Validity assertions and needed inequalities:
  \[ \texttt{ptr1\_valid \& ptr2\_valid} \Rightarrow \texttt{ptr1 > ptr2 + 7 \lor ptr2 > ptr1 + 15}. \]

Strong compatibility with \texttt{SMTLIB2 QF\_ABV}.
Examples
Example 1

int
g (int i) {
    return i * 8 + (i & 15);
}

void
f (int *p, int x) {
    int i;

    for (i = x; i < 100; i ++) {
        p[i] = g (i);
    }
}
Example 1 (cont)

```c
void f (int *p, int x) {
    int i;
    for (i = x; i < 100; i++) {
        p[i] = g (i);
    }
}
```

The C code as a graph:

```
i := x
---
i < 100 ?
---
return
---
m := m[p + (i * 4) := rv]
---
i := i + 1
---
rv := CALL g (i)
```
Example 1 (cont)

The machine code as a graph:

0x48: \[ v := \text{False}, z := (r4 = 0), n := \text{msb } r4, \ldots \]

0x44: \[ \text{...} \]

0x48: \[ z? \]
Example 1 (cont)

We are to prove that these compute the same:
Example 1 (cont)

We are to prove that these compute the same:

(simplified view of graphs)
Example 1 (cont)

What is going on?

for (i = x; i < 100; i ++) {

The loop has been unrolled.

The branches all encode i < 100.

Proof of correctness:

- relate the sequences of loop body visits.
Example 1 (cont)

Proof of correctness:

1. **Case split** on execution of 04c:
   - Consider even case

2. Relate visits to 0x68 to visits 3, 5, 7, ... to body by **induction**.

3. **Case split** on related sequences:
   - Infinite case.
   - Init case: < 4 visits to body. Expand.
   - Loop case: $2n$ visits to body for some $n > 1$. Expand.

The proof search script discovers this proof automatically.
Example 1 (cont)

Proof search:
- Unroll the first few loop iterations.
- Produce SMT model.
- Look for coincidences.
- Check for counterexamples.
Example 2: string compare

```c
int
strncmp(const char* s1, const char* s2, int n)
{
    word_t i;
    int diff;
    for (i = 0; i < n; i++) {
        diff = ((unsigned char*)s1)[i] - ((unsigned char*)s2)[i];
        if (diff != 0 || s1[i] == '\0') {
            return diff;
        }
    }
    return 0;
}
```
Example 2: string compare (cont)
**Example 2: string compare**

```c
int strncmp(const char* s1, const char* s2, int n)
{
    word_t i;
    int diff;
    for (i = 0; i < n; i++) {
        diff = ((unsigned char*)s1)[i] - ((unsigned char*)s2)[i];
        if (diff != 0 || s1[i] == '\0') {
            return diff;
        }
    }
    return 0;
}
```

**Complications:**

1. structure is different (complex induction required, case split on parity)
2. usual strategy of looking for coincidences doesn’t work (because values of i, s1 and s2 might not be there)
3. compiler optimises linear variables and might track a combination of them (e.g. s1+i+4)
4. ignoring linear variables doesn’t work because memory stays the same

---

"can waste hours of CPU time…"
Big picture (again)

- high-level design
- low-level design
- detailed model of C code
- refinement proof
- machine code as graph
- decompilation
- seL4 machine code
- Cambridge ARM model
- Haskell prototype
- real C code
- gcc (not trusted)

existing L4.verified work

new extension
Summary

Translation validation can be used to formally check the output of GCC -O1 and (very nearly) -O2.

Validates the C semantics as used for the seL4 proofs.

Questions?