Using the Cambridge ARM model to verify the concrete machine code of seL4

Magnus Myreen\textsuperscript{12}, Thomas Sewell\textsuperscript{1}, Michael Norrish\textsuperscript{1} and Gerwin Klein\textsuperscript{1}

\textsuperscript{1}NICTA, Australia
\textsuperscript{2}University of Cambridge, UK
seL4 = a formally verified general-purpose microkernel
L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly
L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly
200,000 lines of Isabelle/HOL proofs
Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
Assumptions

L4.verified project assumes correctness of:

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

The aim of this work is to remove the first assumption.
Assumptions

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.
Assumptions

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.
Aim: extend downwards

existing L4.verified work

high-level design

low-level design

detailed model of C code

Haskell prototype

real C code

trusted
Aim: extend downwards

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

Aim: remove need to trust C compiler and C semantics
Aim: extend downwards

existing L4.verified work

high-level design

low-level design

detailed model of C code

Haskell prototype

real C code

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

existing L4.verified work

new extension

detailed model of C code

high-level design

low-level design

Haskell prototype

real C code
Connection to CompCert

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- seL4 as CompCert C code

existing L4.verified work

new extension
Connection to CompCert

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- manual tweaks (by Matthew Fernandez)

new extension
existing L4.verified work

detailed model of C code
Haskell prototype
real C code
manual tweaks (by Matthew Fernandez)
Connection to CompCert

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- manual tweaks (by Matthew Fernandez)

- seL4 as CompCert C code
- CompCert compiler

existing L4.verified work
new extension
Connection to CompCert

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- manual tweaks (by Matthew Fernandez)
- seL4 as CompCert C code
- CompCert ARM assembly
- CompCert compiler
Connection to CompCert

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- manual tweaks (by Matthew Fernandez)
- incompatible

- sel4 as CompCert C code
- CompCert compiler
- CompCert ARM assembly

existing L4.verified work
new extension
Connection to CompCert

Incompatible:
• different view on what valid C is
• pointers treated differently
• memory more abstract in CompCert C sem.
• different provers (Coq and Isabelle)
Connection to CompCert

Incompatible:
- different view on what valid C is
- pointers treated differently
- memory more abstract in CompCert C sem.
- different provers (Coq and Isabelle)

A separate project at NICTA aims to resolve these differences.
Using Cambridge ARM model

Cambridge ARM model

new extension
existing L4.verified work

detailed model of C code

low-level design

high-level design

Haskell prototype

real C code
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- gcc (not trusted)

Cambridge ARM model

existing L4.verified work

new extension
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- seL4 machine code
- Cambridge ARM model

existing L4.verified work

new extension

gcc (not trusted)
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- seL4 machine code
- Haskell prototype
- real C code
- gcc (not trusted)

Cambridge ARM model
existing L4.verified work
new extension
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- machine code as functions
- decompilation
- seL4 machine code
- Cambridge ARM model

existing L4.verified work

new extension

gcc (not trusted)
Using Cambridge ARM model

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

Existing L4.verified work with new extensions.
Talk outline

- automatic translation / decompilation
- progress and lessons learnt
Cambridge ARM model

- high-fidelity 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/
Stage 1: decompilation

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

Decompilation
Decompilation

Sample C code:

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

Sample C code:

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

Machine code:

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

Sample C code:

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

机器码:

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

解码结果:

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

Sample C code:

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

Machine code:

```
add r0, r1, r0
lsr r0, r0, #1
bx lr
```

decomposition via ARM model

Resulting function:

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

HOL4 certificate theorem:

```
\{ R0 i * R1 j * LR lr * PC p \}
\{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr \}
```
Decompileation

Sample C code:

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

machine code:

```
ge0810000  add   r0, r1, r0
e1a000a0  lsr   r0, r0, #1
e12fff1e  bx    lr
```

Resulting function:

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

HOL4 certificate theorem:

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

Sample C code:

```c
#include <stdint.h>

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

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

Resulting function:
```
avg (r0, r1) = let r0 = r1 + r0 in
              let r0 = r0 >> 1 in
              r0
```

HOL4 certificate theorem:
```
{ R0 i * R1 j * LR lr * PC p }
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```
Decompilation

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

HOL4 certificate theorem:
```hol
{ R0 i * R1 j * LR lr * PC p } 
P : e0810000 e1a000a0 e12fff1e 
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```

Resulting function:
```c
avg (r0, r1) = let r0 = r1 + r0 in
    let r0 = r0 >> 1 in
    r0
```
machine code:
```c
gcc (not trusted)
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr
```

decompilation via ARM model

bit-string arithmetic

return instruction
Decompilation

Sample C code:

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

HOL4 certificate theorem:

```
{ R0 i * R1 j * LR lr * PC p }

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

Resulting function:

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

machine code:

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

bit-string arithmetic

bit-string right-shift

decompilation via ARM model

return instruction

separation logic: *

machine code:

```
decomposition
```

return instruction

separation logic: *
Decompilation

How to decompile:

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

How to decompile:

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

e0810000

e1a000a0

e12fff1e
Decompilation

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

How to decompile:
e0810000  add r0, r1, r0
e1a000a0  lsr r0, r0, #1
e12fff1e  bx lr

1. derive Hoare triple theorems using Cambridge ARM model
Decompilation

How to decompile:

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

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
Decompile

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }
{ LR lr * PC lr }

p+8 : e12fff1e
{ 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:
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr

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

(Loops result in recursive functions.)

avg (i,j) = (i+j)>>1
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
- compiled using gcc -O2
- must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  ✓ decompilation is compositional
- compiled using gcc -O2
- must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  ✓ decompilation is compositional
- compiled using gcc -O2
  ✓ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - ✔ decompilation is compositional
- compiled using gcc -O2
  - ✔ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
  - ➡ stack requires special treatment
Stack visible in m. code

C code:

```c
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```
Some arguments are passed on the stack,

C code:

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

Stack visible in m. code
Stack visible in m. code

C code:

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

Some arguments are passed on the stack,

```
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
```
Some arguments are passed on the stack, and cause memory ops in machine code that are not present in C semantics.

C code:

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

Some arguments are passed on the stack, and cause memory ops in machine code...

...that are not present in C semantics.
Solution

Use separation-logic inspired approach

3 slots of unused but required stack space

rest of stack
Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

m
Solution

Use separation-logic inspired approach

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

3 slots of unused but required stack space

rest of stack
Solution

Use separation-logic inspired approach

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

Use separation-logic inspired approach

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

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

disjoint due to *

separation logic: *

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

Method:

1. static analysis to find stack operations,

2. derive stack-specific Hoare triples,

3. then run decompiler as before.
Solution (cont.)

Method:

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

Stack load/stores become straightforward assignments.

avg8(r0, r1, r2, r3, s0, s1, s2, s3) =

let r1 = r1 + r0 in
let r1 = r1 + r2 in
let r2 = s0 in
let r1 = r1 + r3 in
let r0 = r1 + r3 in
let (r2, r3) = (s1, s2) in
let r0 = r0 + r2 in
let r0 = r0 + r3 in
let r3 = s3 in
let r0 = r0 + r3 in
let r0 = r0 >> 3 in
r0

Result

Stack load/stores become straightforward assignments.

avg8(r0, r1, r2, r3, s0, s1, s2, s3) =

let r1 = r1 + r0 in
let r1 = r1 + r2 in
let (r2, r3) = (s0, s1) in
let r0 = r0 + r2 in
let r0 = r0 + r3 in
let r3 = s2 in
let r0 = r0 + r3 in
let r0 = r0 >> 3 in
r0
Result

Stack load/stores become straightforward assignments.

Additional benefit:
automatically proved certificate theorem
states explicitly stack shape/usage:

\[
\{ \text{stack } sp \ n \ (s0::s1::s2::s3::s) * \ldots * \text{PC } p \} \\
p : \text{code} \\
\{ \text{stack } sp \ n \ (s0::s1::s2::s3::s) * \ldots * \text{PC } lr \} \\
\]
Result

Stack load/stores become straightforward assignments.

Additional benefit:
automatically proved certificate theorem
states explicitly stack shape/usage:

\{ stack \text{sp} \ n \ (s0::s1::s2::s3::s) * ... * PC \ p \} \\
p : code \\
\{ stack \text{sp} \ n \ (s0::s1::s2::s3::s) * ... * PC lr \} \\

bx lr \\
r0
Result

Stack load/stores become straightforward assignments.

Additional benefit:

- Automatically states explicitly stack
  - four arguments passed on stack
  - does not require temp space, works for “any n”

```prolog
{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }  
p : code  
{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }  

bx lr  
r0
```
Result

Stack load/stores become straightforward assignments.

Additional benefit:
- automatically proves certificate
- states explicitly stack shape/usage

\[
\begin{align*}
\{ & \text{stack } sp \ n \ (s0::s1::s2::s3::s) \ \ast \ \ast \ \ast \ \text{PC} \ p \} \\
\text{p : code} \\
\{ & \text{stack } sp \ n \ (s0::s1::s2::s3::s) \ \ast \ \ast \ \ast \ \text{PC} \ lr \} \\
\end{align*}
\]
- promises to leave stack unchanged
- does not require temp space, works for “any n”
- four arguments passed on stack
Other C-specifics

- **struct as return value**
  - case of passing *pointer of stack location*
  - stack assertion strong enough

- **switch statements**
  - position dependent
  - must decompile elf-files, not object files

- **infinite loops in C**
  - make **gcc go weird**
  - must be pruned from control-flow graph
Progress

A 6-week visit to NICTA resulted in:

75% of seL4 decompiled
Progress

A 6-week visit to NICTA resulted in:

75 % of seL4 decompiled

Next visit scheduled for end of this year:

- complete decompilation
  (make stack heuristic stronger)
- concentrate on stage 2... (next slide)
Moving on to stage 2

detailed model of C code

machine code as functions

seL4 machine code

refinement proof

automatic translation

new extension
Moving on to stage 2

- New extension
  - Detailed model of C code
    - Refinement proof
  - Machine code as functions
    - Automatic translation
  - Sel4 machine code
Proving C refinement

Approach 1:

- use a verification condition generator (VCG) to prove C Hoare-triple theorems, approximately:

  \[
  \{ \text{true} \} \text{code} \{ \text{state\_after} = \text{code\_fun}(\text{state\_before}) \}
  \]

Aim:

- make solution as automatic as possible
- must deal with reordering of load/store instructions
Proving C refinement

Approach 1:

- use a verification condition generator (VCG) to prove C Hoare-triple theorems, approximately:

\[
\{ \text{true} \} \text{code} \{ \text{state_after} = \text{code_fun(state_before)} \}
\]

Aim:

- make solution as automatic as possible
- must deal with reordering of load/store instructions
Proving C refinement

Approach 2:

- compose C code inside existing correctness C Hoare triple, e.g.

\[
\{ \text{pre} \} \ (\text{Assign } f; \ \text{Assign } g) \ \{ \text{post} \} = \\
\{ \text{pre} \} \ (\text{Assign } (g \circ f)) \ \{ \text{post} \}
\]

- then prove, for almost any \text{pre}, \text{post}:

\[
\{ \text{pre} \} \ \text{code} \ \{ \text{post} \} \Rightarrow \\
\{ \text{pre} \} \ (\text{Assign } \text{code}_\text{fun}) \ \{ \text{post} \}
\]
Proving C refinement

Approach 2:

Solution to inlined assembly:

naturally compatible with decompilations of inlined assembly, e.g.

\[
\{ \text{pre} \} \ (\text{Assign inline_asm_fun}) \ \{ \ \text{post} \}
\]

Gets around the problem of C’s \texttt{__asm__}.

\[
\{ \text{pre} \} \ (\text{Assign code_fun}) \ \{ \ \text{post} \}
\]
Final part:

Lessons learnt
gcc: weird and wonderful

Wonderful:

• gcc -O2 produces good/clever code
• decompilation can be made to work on its output
• gcc -O0 produces simple “reference” machine code

Weird:

• fails to spot a few ‘obvious’ optimisations
• gcc -O2 sometimes invents new subroutines
Hardest part?
Hardest part?

So far: connection with C semantics.
Hardest part?

So far: connection with C semantics.

C semantics best avoided?
Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:

• use verification-friendly domain-specific language
Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:
  • use verification-friendly domain-specific language
Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:
- use verification-friendly domain-specific language

... but C is the reality of OS code
- a simple "hacker’s semantics of C"?
“a hacker’s C semantics”
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.

✓ approach reflects the observation: “OS hackers use C as convenient way to write assembly”
Possibility: use decompilation from gcc -O0 as semantics of C code.

✓ approach reflects the observation:
  “OS hackers use C as convenient way to write assembly”

✓ potentially simpler than current C semantics
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.

✓ approach reflects the observation:
   “OS hackers use C as convenient way to write assembly”

✓ potentially simpler than current C semantics

✓ does not require trusting gcc
  ▸ proof relates only to the generated machine code
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.

✓ approach reflects the observation:
  “OS hackers use C as convenient way to write assembly”
✓ potentially simpler than current C semantics
✓ does not require trusting gcc
  ▸ proof relates only to the generated machine code
✓ separately prove transition from gcc -O0 to gcc -O2
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.

✓ approach reflects the observation: “OS hackers use C as convenient way to write assembly”
✓ potentially simpler than current C semantics
✓ does not require trusting gcc
  ▸ proof relates only to the generated machine code
✓ separately prove transition from gcc -O0 to gcc -O2

⇒ impossible: current L4.verified proofs tied to C sem.
Connecting provers

In general, hard.
Easy in this case.
Connecting provers

In general, hard. Easy in this case.

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- seL4 machine code

Connecting provers

- existing L4.verified work
- new extension in HOL4
- in Isabelle/HOL
Connecting provers

In general, hard.
Easy in this case.

high-level design

low-level design

detailed model of C code

machine code as functions

machine code as functions

seL4 machine code

existing L4.verified work

in Isabelle/HOL

new extension in HOL4

automatic translation of definitions from HOL4 to Isabelle/HOL
Summary

L4.verified is being extended downwards using the Cambridge ARM model
Summary

L4.verified is being extended downwards using the Cambridge ARM model

Aim:

- remove need to trust gcc and C
Summary

L4.verified is being extended downwards using the Cambridge ARM model

Aim:
• remove need to trust gcc and C

Lesson learnt:
• decompilation scales!
  (at least to 10,000 ARM instructions)
Summary

L4.verified is being extended downwards using the Cambridge ARM model

Aim:
• remove need to trust gcc and C

Lesson learnt:
• decompilation scales!
  (at least to 10,000 ARM instructions)

Questions?