Translation Validation for a Verified OS Kernel

Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹

¹ NICTA, Australia
² University of Cambridge, UK
L4.verified

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 in L4.verified

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
Assumptions in L4.verified

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 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.
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

existing L4.verified work

---

high-level design

low-level design

detailed model of C code

Haskell prototype

real C code

trusted
Aim: extend downwards

existing L4.verified work

\[\cdots\] high-level design
\[\cdots\] low-level design
\[\cdots\] detailed model of C code

\[\cdots\] Haskell prototype

\[\text{trusted}\] real C code

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

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

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

new extension
existing L4.verified work
Connection to CompCert

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

new extension   existing L4.verified work

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

existing L4.verified work

new extension
Connection to CompCert

- Haskell prototype
- real C code
- manual tweaks (by Matthew Fernandez)
- incompatible
- new extension
- existing L4.verified work

Diagram:
- high-level design
- low-level design
- detailed model of C code
- seL4 as CompCert C code
- CompCert ARM assembly
- CompCert compiler
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)
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- 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
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

gcc (not trusted)
Using Cambridge ARM model

Cambridge ARM model

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

existing L4.verified work

new extension

Haskell prototype

real C code

machine code as functions

seL4 machine code

Cambridge ARM model

gcc (not trusted)
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 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
  - refinement proof
  - machine code as functions
    - decompilation
      - seL4 machine code
      - Cambridge ARM model
  - real C code
    - Haskell prototype
Translation validation

Translation Validation efforts:

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

Part 1: automatic translation / decompilation

Part 2: pseudo compilation and refinement proof (SMT)
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

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

Decompile
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:

```
egcc
add  r0, r1, r0
lsl  r0, r0, #1
bx   lr
```
Decompilation

Sample C code:

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

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
```

decompilation via ARM model
Decompilation

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
```

decompilation 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 }  
p : e0810000  e1a000a0  e12fff1e
{ 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:

```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
```

machine code:

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

Decompilation via ARM model

return instruction
Decompilation

Sample C code:

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

machine code:

```c
e0810000 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:

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

bit-string arithmetic

decompilation

return instruction
Decompilation

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:

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

HOL4 certificate theorem:

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

- bit-string arithmetic
- bit-string right-shift
- return instruction
- decompilation via ARM model
- gcc (not trusted)
Decompilation

Sample C code:

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

machine code:

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

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
```

- bit-string arithmetic
- bit-string right-shift
- return instruction
- decompilation
- 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  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx   lr

e1a000a0

e12fff1e
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
Decompile

How to decompile:

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

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

```
{ 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 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 }
```
Decompilation

How to decompile:
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 -O1 and 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 -O1 and 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 -O1 and 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 -O1 and gcc -O2
  - ✔ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
  - ➡ stack requires special treatment
Stack visible in machine code

C code:

```
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 machine code

Some arguments are passed on the stack,
Stack visible in machine 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,

```assembly
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...
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack
Solution

Use separation-logic inspired approach

3 slots of unused but required stack space

rest of stack

stack pointer: sp
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 pointer: sp

3 slots of unused but required stack space

rest of stack

(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

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

disjoint due to *

separation logic:*
Solution (cont.)

Method:

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

```assembly
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]
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.
avg8(r0,r1,r2,r3,s0,s1,s2,s3) =

\[
\begin{align*}
&\text{let } r1 = r1 + r0 \text{ in} \\
&\text{let } r1 = r1 + r2 \text{ in} \\
&\text{let } r2 = s0 \text{ in} \\
&\text{let } r1 = r1 + r3 \text{ in} \\
&\text{let } r0 = r1 + r3 \text{ in} \\
&\text{let } (r2,r3) = (s1,s2) \text{ in} \\
&\text{let } r0 = r0 + r2 \text{ in} \\
&\text{let } r0 = r0 + r3 \text{ in} \\
&\text{let } r3 = s3 \text{ in} \\
&\text{let } r0 = r0 + r3 \text{ in} \\
&\text{let } r0 = r0 >> 3 \text{ in} \\
&\text{r0}
\end{align*}
\]

Result

Stack load/stores become straightforward assignments.
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
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

- detailed model of C code
- machine code as functions
- seL4 machine code

New extension

refinement proof

automatic translation
Approach for refinement proof

detailed model of C code

machine code as functions
Approach for refinement proof

- detailed model of C code
  - C code as graph
    - mc functions as graph
      - machine code as functions
Approach for refinement proof

- Detailed model of C code
- C code as graph
- mc functions as graph
- Machine code as functions
- SMT proof
Translating C into graphs

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;
}
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;
}
```

Figure 3. One simple restriction is that all structures of interest are packed (see Section 4.3).
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];
8: ret := 0
```
Translating mc functions into graphs

\[ f(x, y) = \]
\[
\text{let } (a, b) = \text{if } x < y \text{ then } (1, 2) \text{ else } (3, x) \text{ in let } c = a + b - y \text{ in } (c, 0) \]

1: \( \text{x < y?} \)
2: \( a := 1 \)
3: \( b := 1 \)
4: \( a := 3 \)
5: \( b := x \)
6: \( c := a + b - y \)
7: \( r1 := c \)
8: \( r2 := 0 \)
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 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: **Restrict**, **Split** and **Leaf**.
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: Restrict, Split and Leaf.

The heavy lifting is done by calls to SMT solvers for both the proof search and checking.
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...)
Figure 2. The three dotted boxes in the diagram represent the three artefacts in the correctness proof: the C program and binary ELF file on the top of the diagram, the two inputs of the decompiler, and the output of the decompiler. The decompiler proves that the extracted function is indeed accurate with respect to the C program and binary. The main tool is called a decompiler, and the decompiler is automatic and proof producing. For each run, the decompiler produces a proof script that can be imported into a proof tool for verification. The proof tool then verifies the correctness of the decompiled function against the C program and binary. The proof tool is called a SMT-based proof tool, and it uses Z3 and SONOLAR as SMT solvers.

The main tool is called a decompiler, and the decompiler is automatic and proof producing. For each run, the decompiler produces a proof script that can be imported into a proof tool for verification. The proof tool then verifies the correctness of the decompiled function against the C program and binary. The proof tool is called a SMT-based proof tool, and it uses Z3 and SONOLAR as SMT solvers.

The idea of the graph conversion is to replace the language of the C program with a graph language. This also serves as an introduction to the graph language. The graph conversion is done by Isabelle/HOL, and the graph language is then exported to HOL4 for proof verification. The graph conversion is done by Isabelle/HOL, and the graph language is then exported to HOL4 for proof verification.

The graph conversion is done by Isabelle/HOL, and the graph language is then exported to HOL4 for proof verification. The graph conversion is done by Isabelle/HOL, and the graph language is then exported to HOL4 for proof verification.
Results and Summary

We have (almost) proved a full connection between the verified C and seL4 binary.

<table>
<thead>
<tr>
<th>Table 1. Decompilation and Proof Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions in Binary</td>
</tr>
<tr>
<td>Decomplied Functions</td>
</tr>
<tr>
<td>Placeholders</td>
</tr>
<tr>
<td>Function Pairings</td>
</tr>
<tr>
<td>Successes</td>
</tr>
<tr>
<td>Failures</td>
</tr>
<tr>
<td>Aborted</td>
</tr>
<tr>
<td>Nested Loops</td>
</tr>
<tr>
<td>Machine Operations Inlined</td>
</tr>
<tr>
<td>Time Taken in Proof</td>
</tr>
</tbody>
</table>