Translation Validation for a Verified OS Kernel

Thomas Sewell\textsuperscript{1}  Magnus Myreen\textsuperscript{2}  Gerwin Klein\textsuperscript{1}

\textsuperscript{1}UNSW & NICTA
\textsuperscript{2}University of Cambridge

12 Feb 2013
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
    tcbSchedDequeue(ta

void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
tcbSchedDequeue(ta

Translation Validation for a Verified OS Kernel
Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}

Translation Validation for a Verified OS Kernel
Thomas Sewell\textsuperscript{1}, Magnus Myreen\textsuperscript{2}, Gerwin Klein\textsuperscript{1}

\textsuperscript{1}School of Computer Science and Software Engineering, University of New South Wales, Sydney
\textsuperscript{2}Department of Information Technology, Linköping University, Sweden
Copyright NICTA 2013
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
    tcbSchedDequeue(ta
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}

sel4: Formal Verification and All That

ARM ISA and Decompiler
Anthony Fox and Magnus Myreen

Translation Validation for a Verified OS Kernel
Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹
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 -01.
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.
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
tcbSchedDequeue(ta

Translation Validation for a Verified OS Kernel
Copyright NICTA 2013
Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}

- Isabelle/HOL
- Pseudo Compilation
- HOL4
- Decompilation into Logic
- Cambridge ARM Semantics
- SMT+
Approach

void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
tcbSchedDequeue(ta

sel4

C SEMANTICS

Proof Producing Conversion

C IL

Comparison

Parsing

Conversion to Intermediate Language

Optimisation Passes

Instruction Scheduling

Assembly

GCC

LD

Binary Object

ARM

ASL Funs

ASM IL

Translation Validation for a Verified OS Kernel
Thomas Sewell\textsuperscript{1}, Magnus Myreen\textsuperscript{2}, Gerwin Klein\textsuperscript{1}

Copyright NICTA 2013
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
    tcbSchedDequeue(ta

C SEMANTICS
C Program Semantics

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

\[
\begin{align*}
\text{if} \ (\ldots) \ \{ \ldots \} & \implies \text{IF} \ (\ldots) \ \text{THEN} \ \ldots \ \text{FI} \\
f \ (1, \ 2); & \implies \text{CALL} \ f\_\text{\textquote{proc}} \ (1, \ 2) ;; \\
x \ \text{++}; & \implies \ (\text{\textquote{x} := \text{\textquote{x} + 1}}) ;; \\
*p = *q; & \implies \text{mem := h\_upd} \ \text{\textquote{p}} \\
& \quad (\text{h\_val} \ \text{\textquote{q} \ \text{\textquote{mem}}} \ \text{\textquote{mem}})
\end{align*}
\]
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.**

\[
\begin{align*}
\text{if} \ (\ldots) \ \{\ldots\} & \Rightarrow \text{IF} \ (\ldots) \ \text{THEN} \ \ldots \ \text{FI} \\
f \ (1, \ 2); & \Rightarrow \text{CALL} \ f\_\text{’proc} \ (1, \ 2);; \\
x \ +++; & \Rightarrow \text{Guard} \ \{\ ‘x <=s \ ‘x + 1} \\
& \quad \left(‘x ::= \ ‘x + 1);; \right. \\
*p = *q; & \Rightarrow \text{Guard} \ \{\text{ptr\_valid \ ‘p}\} \\
& \quad \text{Guard} \ \{\text{ptr\_valid \ ‘q}\} \\
& \quad \text{mem} ::= \ h\_\text{upd \ ‘p} \\
& \quad \quad \left(\text{h\_val \ ‘q \ ‘mem}\right) \ ‘mem
\end{align*}
\]
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`. 
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?

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

Translation Validation for a Verified OS Kernel
Thomas Sewell¹, Magnus Myreen², Gerwin Klein¹

Copyright NICTA 2013
Example Decompilation

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

```assembly
<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

```c
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
```

\[
\text{avg}(r_0, r_1) = \text{let } r_0 = r_1 + r_0 \text{ in }
\begin{align*}
\text{let } r_0 &= r_0 >>> 1 \text{ in } \\
& r_0
\end{align*}
\]
Example Decompilation

```c
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
```

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

{ R0 r0 * R1 r1 * R14 lr * PC p }
```

```plaintext
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg (r0, r1)) * R1 _ * R14 _ * PC lr }
```
Challenges for Decompilation

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

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

```asm
<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:

\[ \forall (in, \text{in_heap}) \in \text{domain}(\mathcal{C}). \mathcal{C}(in, \text{in_heap}) = \mathcal{B}(in, \text{in_heap}) \]

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 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.

```
x := 12
x < 12?
x := y + 1
T F
```

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

Strong compatibility with \texttt{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.
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