What's Next?

- Windows

An exception 06 has occurred at 002b:21830060 in \Windows\DiskTSD(03) + 00000000. This was called from 002b:219800c8 in \Windows\voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press Ctl+Alt+Del to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
Trust Without Trustworthiness

Core Issue: Complexity

- Massive functionality → huge software stacks
  - Expensive recalls of CE devices
- Increasing usability requirements
  - Wearable or implanted medical devices
  - Patient-operated
  - GUIs next to life-critical functionality
- On-going integration of critical and entertainment functions
  - Automotive infotainment and engine control

Our Vision: Trustworthy Systems

We will change industry’s approach to the design and implementation of critical systems, resulting in true trustworthiness.

Trustworthy means highly dependable, with hard guarantees on security, safety or reliability.

Dealing With Complexity

- Complexity of critical devices will continue to grow
  - Critical systems with millions of lines of code (LOC)
- We need to learn to ensure dependability despite complexity
  - Need to guarantee dependability
- Correctness guarantees for MLOCs unfeasible

- Key to solution: isolation
  - … with controlled communication
Isolation: Physical

Dedicated CPUs for critical tasks

Cost: Space, costly interconnects, poor use of hardware

Isolation: Logical

- Protect critical components by sandboxing complex components
- Provide tightly-controlled communication channels
- **Trustworthy microkernel** provides general mechanisms to enforce isolation
- Policy layer defines access rights
- Microkernel becomes core of trusted computing base
  - System trustworthiness only as good as microkernel!

Isolation Requirements

To guarantee dependability, following must be guaranteed:
- Isolation infrastructure impact must be specified
  - To allow reason about operation of isolated critical instances
- Isolation infrastructure must behave as specified
  - Functional correctness
  - Bounded and know worst-case latencies
- Isolation infrastructure must provide actual isolation
  - Integrity guarantees
  - Temporal isolation

Dependability Requirements
NICTA Trustworthy Systems Agenda

1. Dependable microkernel (seL4) as a rock-solid base
   - Formal specification of functionality
   - Proof of functional correctness of implementation
   - Proof of safety/security properties
   - Timeliness guarantees

2. Lift microkernel guarantees to whole system
   - Use kernel correctness and integrity to guarantee critical functionality
   - Ensure correctness of balance of trusted computing base
   - Prove dependability properties of complete system

Two Mentalities

Formal Methods Practitioners vs Kernel Developers

seL4: Designing and Formalising

Design & Specify
Formal Model
Proof

High-Performance C implementation
Safety Theorem

Standard Kernel Design

Kernel Hacker View

Design & Specify
Formal Model
Proof

Prototype on Real Hardware
Safety Theorem

Step 2
White-board
Formal Design

**Formal Methods View**

- Design & Specify
- Formal Model
- Proof
- High-Performance C implementation
- Safety Theorem

**Step 2**

Iterative Design and Formalisation

- Design & Specify
- Haskell Prototype
- Formal Model
- Proof
- Safety Theorem

- Inspired by existing code
- Prototype kernel executes native binaries on simulator
- Exposes usability issues early
- Tight formal design integration

Kernel Design for Verification

- Main objective: minimise complexity
  - global invariants must be proven for each state change
  - must prove pre- and post-conditions for statements/blocks
  - effort determined by complexity of conditions and state change

- … without sacrificing performance

- Affects design in many ways
  - global variables, side effects
  - kernel memory management
  - concurrency and non-determinism
  - I/O

Global Variables

- Not a difficulty per se, but potential source of complexity
- Eg: scheduler queue as doubly-linked list
- Show that
  - all pointers are to valid nodes
  - front- and back-pointers are consistent
  - nodes point to TCBs

- Requires proof that any pointer operation maintains invariants
- Challenge is temporary violation
  - eg adding a node
  - Requires ensuring atomicity
Kernel Memory

- seL4 kernel memory management model pushes policy to userland
  - aids verification
  - need to ensure strict hierarchy
  - capability derivation tree
- Challenge is re-use
  - most difficult part of verification!
  - use derivation tree to detect all references
  - global data structure that requires invariants in all parts of the system

Concurrency

- Proofs about concurrent programs are inherently hard!
- seL4 strictly limits concurrency to the bare minimum
  - Single processor
    - multicore via big kernel lock or multikernel approach
  - User-level device drivers
    - Non-preemptible, event-based
    - single kernel stack
  - Interrupt points to limit real-time latencies
    - poll interrupt status
    - insert new kernel event (ahead of user)
    - return to user boundary and re-enter kernel
    - allows maintaining all invariants

Preempting object destruction:

- Keep one cap as zombie during object cleanup
  - only retained to reference partially cleaned-up object
  - stores state of cleanup, maintaining invariants
  - attempt by preemptor to remove zombie can just execute

Exceptions in kernel:

- Prevent memory exceptions
  - ensure kernel page tables are complete
  - map into every address space
- Disallow other exceptions
  - verification is its own friend 😊

I/O

- Mostly a non-issue
  - user-level drivers
    - IOMMU support for DMA security
  - non-preemptible kernel
- Exception is timer tick
  - essentially a source of interrupts
  - handled in-kernel as separate event
  - no real complication
Lessons for Kernel Design

- Need to reduce complexity forced simple and clean design
  - beneficial even with traditional validation
  - does not necessarily impact performance
- Some design decision beneficial for other reasons too
  - single kernel stack for memory footprint
  - interrupt handling by polling has performance advantages

Kernel Functional Verification

<table>
<thead>
<tr>
<th>Initial protection state</th>
<th>Confidentiality (???)</th>
</tr>
</thead>
<tbody>
<tr>
<td>capDL Model (4,800)</td>
<td>22,000 lop</td>
</tr>
<tr>
<td>Manual Spec (Isabelle/HOL)</td>
<td>Integrity (1,000)</td>
</tr>
<tr>
<td>Abstract Model (4,900)</td>
<td>10,000 lop</td>
</tr>
<tr>
<td>Executable Model (13,000)</td>
<td>117,000 lop</td>
</tr>
<tr>
<td>High Performance Implementation</td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td></td>
</tr>
<tr>
<td>Asm Code (320)</td>
<td></td>
</tr>
<tr>
<td>C Code (8,700)</td>
<td>Sane initial state</td>
</tr>
</tbody>
</table>

Binary Code Verification (In Progress)

C source → Formal C semantics → Abstracted C code

Abstract compiler → Abstract machine code

Equivalence checker → Abstract executable of real machine

Formal ISA spec → Executable binary
Formal Verification Summary

Kinds of properties proved

• Behaviour of C code is fully captured by abstract model
  – Can prove many interesting properties on higher-level models
• Behaviour of C code is fully captured by executable model
  – Kernel never fails, behaviour is always well-defined
    – Assertions never fail
    – Will never de-reference null pointer
    – Cannot be subverted by misformed input
• All syscalls terminate, reclaiming memory is safe, ...
• Well typed references, aligned objects, kernel always mapped...
• Access control is decidable

Effort:

• Average 6 people over 5.5 years
• Only 50–100% more than comparable (low-assurance) projects

Verification vs Certification

Common Criteria: Military-Strength Security

<table>
<thead>
<tr>
<th>Evaluation Level</th>
<th>Requirements</th>
<th>Functional Specification</th>
<th>Top Down Design</th>
<th>Implementation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL1</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>EAL2</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>EAL3</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>EAL4</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>EAL5</td>
<td>Semi-formal</td>
<td>Semi-formal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>EAL6</td>
<td>Formal</td>
<td>Semi-formal</td>
<td>Semi-formal</td>
<td>Informal</td>
<td>1K/LoC</td>
</tr>
<tr>
<td>EAL7</td>
<td>Formal</td>
<td>Formal</td>
<td>Informal</td>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>sel4</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>0.6K/LoC</td>
</tr>
</tbody>
</table>

Kernel Worst-Case Execution Time

Issues for WCET analysis of sel4:

• Need knowledge of worst-case interrupt-latency
  – Longest non-preemptible path + IRQ delivery cost
  – sel4 runs with interrupts disabled
    • System calls in well-designed microkernel are short!
    • Strategic preemption points in long-running operations
    • Optimal average-case performance with reasonable worst-case
• Applications also need to know cost of system calls
  – Need WCET analysis of all possible code paths

Challenges for WCET analysis of OS kernels in general:

• Kernel code notoriously unstructured
• Low-level system-specific instructions
• Context-switching
• Assembly code

sel4-specific advantages:

• (Relatively) structured design (evolved from Haskell prototype)
• Event-based kernel (single kernel stack)
• Small (as far as operating systems go!)
• No function pointers in C
• Preemption points are explicit and preserve code structure
• Memory allocation performed in userspace
**WCET analysis process**

1. **seL4 binary**
2. **CFG extractor**
3. **Loop bounds**
4. **ILP equations**
5. **Hardware platform**
6. **WCET Upper bound**
7. **Observed execution time**

**Evaluation platform**

- OMAP3-based BeagleBoard-xM
  - ARM Cortex-A8 @ 800 MHz
  - 128 MB memory
  - 32KB 4-way set-associative L1 instruction cache
    - Random replacement ← pessimistic model
  - Disabled L2 cache
    - Cache analysis does not (yet) scale
- Fairly accurate (but sound) model
  - Dual-issue pipeline (simplified)
  - No branch prediction

**Early Days...**

- **Open system - untrusted code**
  - Observed: 305.2 μs, 1635 μs
  - Computed

- **Closed system**
  - Observed: 46.4 μs, 387.4 μs
  - Computed

**Improve WCET**

- Analysis helps placing preemption points
  - Can optimise further
- Knowledge about seL4 can eliminate many paths
  - Invariants proved during verification
  - E.g. loop iteration counts, non-interference
  - Can easily prove new invariants
- Cache pinning
  - Big reduction in WCET
  - Eliminate cache pessimism
- Improved pipeline modelling
  - May have practical approach for complex pipelines
- Aim: IRQ WCET < 10 μs
Phase Two: Full-System Guarantees

• Achieved: Verification of microkernel (8,700 LOC)

• Next step: Guarantees for real-world systems (1,000,000 LOC)

Overview of Approach

• Build system with minimal TCB
• Formalize and prove security properties about architecture
• Prove correctness of trusted components
• Prove correctness of setup
• Prove temporal properties (isolation, WCET, …)
• Maintain performance

Proof of Concept: Secure Access Controller

SAC Aim

Providers A & B should not be able to leak info between each other even if they actively cooperate
Solution Overview

SAC Significance

- Prototype of seL4-based security architecture
  - Demonstrates feasibility of seL4-based secure systems
    - Minimal TCB
  - Demonstrates feasibility of proving relevant properties
  - Mostly hand-knitted
- Future:
  - High-level specification of architecture and properties
  - Automation of system generation
  - Automation of verification

Specifying Security Architecture
**Needed: Component Architecture (CAmkES)**

- **Example System: Modular Access Router**

  **NAT to multiplex a single global IP address**
  - Includes drivers, network stack,...

  **Performance:**
  - ARM11 (Freescale iMX31)
  - RPC
    - 1 int arg, returning int
  - CAmkES round trip:
    - 778 cycles
  - raw seL4 round trip:
    - 698 cycles
    - not yet optimal!
  - CAmkES overhead:
    - 80 cycles

**Other Bits**

- Dependable device drivers
  - Event-driven drivers
  - Driver synthesis
    - exploring widening this to synthesising other system components
- Energy management
- Security
  - Information flow
  - Side channels and covert channels

**Summary:**

- Trustworthy systems are possible
  - … and we’re the leaders in the field
- You can be part of it!
The ultimate way to keep your computer safe from harm

FLAWS in the code, or "kernel", that sits at the heart of modern computers leave them prone to occasional malfunction and vulnerable to attack by worms and viruses, so the development of a secure general-purpose microkernel could prove the just mathematics, and you can reason about them mathematically," says Klein.

Last week he formulated a model with more than 200,000 logical steps which allowed them to prove that the program would always behave as expected.