Present Systems are NOT Trustworthy!

Windows

An exception 06 has occurred at 0020:0C860DCE in \W\Disk\TSD(03)+00000000. This was called from 0020:0C86048C in \W\voltrack(04)+00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CIR\W\ALT\RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue

Yet they are expensive:
- $1,000 per line of code for “high-assurance” software!
What's Next?

Fundamental issue: large stacks, need isolation

E.g. medical implant

- 1 kLOC critical code
- 20–100 kLOC trusted computing base (TCB)
- 100s of bugs
- dozens of exploits!

Dealing with Complexity: Physical Isolation

High Assurance Bad Practice

Does not scale!

Correctness of bus protocols?

Separate processors for critical functionality

Does not scale!

Isolation?

Hacker's delight!

Uncritical/untrusted

Sensitive/critical/trusted

Xen/VMMware/KVM hypervisor

Huge TCB

• TCB of millions of LOC
• Expect 1000s of bugs
• Expect 100s of vulnerabilities

Processor
High Assurance Best Practice

- Isolate
- Minimise the TCB
- Assure TCB by
  - testing
  - code inspection
  - bug-finding tools

Always incomplete!

Uncritical/ untrusted
Sensitive/ critical/ trusted
Minimal “trusted computing base” (TCB)
Separation kernel
Processor

Claim: A system must be considered untrustworthy unless proved otherwise!

Corollary [with apologies to Dijkstra]:
Testing, code inspection, etc. can only show lack of trustworthiness!

So, why don’t we prove trustworthiness?

State of the Art: NICTA’s seL4 Microkernel

- Provable isolation!
- Provable assurance!

No place for bugs to hide!

Uncritical/ untrusted
Sensitive/ critical/ trusted
seL4 microkernel
Processor

Truly dependable TCB

Our Vision: Trustworthy Systems

We will change the practice of designing and implementing critical systems, using rigorous approaches to achieve true trustworthiness

Suitable for real-world systems

Hard guarantees on safety/security/reliability
Isolation is Key!

**Identify, minimise and isolate critical components!**

- Complex, untrusted
- Legacy Apps
- System-specific, simple!
- Sensitive App
- Trusted Service
- General-purpose
- Policy Layer
- Linux Server
- Processor

**Critical, trusted**

**Trusted Microkernel – seL4**

**Core of trusted computing base: System can only be as dependable as the microkernel!**

**Identify, minimise and...**

**Core of trusted computing base: System can only be...**

- Critical, trusted
- Sensitive App
- Trusted Service
- General-purpose
- Policy Layer
- Linux Server
- Processor

**Trusted Microkernel – seL4**

**Isolation is Key!**

**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

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
     - despite 99% of code untrusted!

**Requirements for Trustworthy Systems**

- Safety
- Availability
- Security
- Functional Correctness
- Timeliness
- Confidentiality / Info Flow
- Integriti
- Termination

**Isolation!**
seL4: Designing and Formalising

Two Mentalities

Standard Kernel Design

Formal Design
Iterative Design and Formalisation

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

Inspired by existing code

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

High-Performance C Implementation

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
    - IO/MMU 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 decisions beneficial for other reasons too
  - Single kernel stack for memory footprint
  - Interrupt handling by polling has performance advantages
NICTA's seL4: Mathematical Proof of Isolation

Abstract

Model

C Implementation

Exclusions (at present):

- Initialisation
- Assembler, TLB & caches
- Multicore
- Covert timing channels

Binary code

Functional correctness [SOSP'09]

Translation correctness [PLDI'13]

Availabilty

Integrity

Confidentiality

Isolation properties [ITP'11, S&P'13]

Proving Functional Correctness

Proving Functional Correctness

Abstract Model

Executable Model

C Implementation

Refinement: All possible implementation behaviours are captured by model

117,000 lop

50,000 lop

const std::

schedule :: "unit s_mcad"

schedule :: do
thread <- allActiveTCBs;
do_machine_op flushCachesOP return;
modify (A.s (cur_thread := thread))
do*

schedule :: Kernel ()
schedule = do
action <- getSchedulerAction

Refinement: All possible implementation behaviours are captured by model

50,000 lop

117,000 lop

const std::

schedule :: "unit s_mcad"

schedule :: do
thread <- allActiveTCBs;
do_machine_op flushCachesOP return;
modify (A.s (cur_thread := thread))
do*

schedule :: Kernel ()
schedule = do
action <- getSchedulerAction

VC Technology

IO BREAKTHROUGH

Technologies

Compact and efficient

Crash-Proof Code

Making critical software safer

7 comments

WILLIAM BULKELEY

May/June 2011

Share

2011
seL4 Formal Verification Summary

**Kinds of properties proved**
- Behaviour of C code is fully captured by abstract model
- 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

Can prove further properties on abstract level!

How About Performance?

Let’s face it, seL4 is basically slow!
- C code (semi-blindly) translated from Haskell
- Many small functions, little regard for performance

IPC: one-way, zero-length
- Standard C code: 1455 cycles
- C fast path: 188 cycles

Fastest-ever IPC on ARM11!

Bare “pass” in Advanced Operating Systems course!

But can speed up critical operations by short-circuit “fast paths”
- … without resorting to assembler!

seL4 as Basis for Trustworthy Systems

Integrity: Limiting Write Access

To prove:
- Domain-1 doesn’t have write capabilities to Domain-2 objects
  ⇒ no action of Domain-1 agents will modify Domain-2 state
- Specifically, kernel does not modify on Domain-1’s behalf!
  – Event-based kernel operates on behalf of well-defined user thread
  – Prove kernel only allows write upon capability presentation

Kernel data partitioned like user data
Availability: Ensuring Resource Access

- Strict separation of kernel resources
  ⇒ agent cannot deny access to another domain’s resources
- Nothing to do: implied by other properties

Confidentiality: Limiting Read Accesses

To prove:
- Domain-1 doesn’t have read capabilities to Domain-2 objects
  ⇒ no action of any agents will reveal Domain-2 state to Domain-1

Non-interference proof:
- Evolution of Domain 1 does not depend on Domain-2 state
- Also shows absence of covert storage channels

Timeliness

Need worst-case execution time (WCET) analysis of kernel

WCET Analysis Approach

Automatic, from separate tool
Manually determined, proved by tool
Main source of pessimism
Result

Pessimism due to under-specified hardware

Observed Computed

0 100 200 300 μs

99.5 378

WCET presently limited by verification practicalities
• 10 μs seem achievable

Future: Whole-System Schedulability

Guarantee schedulability

Hardware

Not Critical
Moderately Critical
Highly Critical

Arbitrary behaviour

Requires model for managing time resource

Future: Whole-System Schedulability

Guarantee schedulability

Hardware

Not Critical
Moderately Critical
Highly Critical

Arbitrary behaviour

Requires model for managing time resource

seL4 as Basis for Trustworthy Systems

Safety

Availability

Timeliness

Termination

Security

Functional Correctness

Confident. / Info Flow

Memory Safety

Integrity

Binary Verification

IPC: one-way, zero-length

Compiler gcc CompCert

Standard C code: 1455 cycles 3749 cycles

C fast path: 188 cycles 730 cycles

Uncompetitive performance!

Bigger problem:
• Our proofs are in Isabel/HOL, CompCert uses Coq
• We cannot prove that they use the same C semantics!
**Binary Code Verification**

- C source
- Formal C
- Formal semantics
- Rewrite rules
- Function code
- SAT solver etc
- Formal binary
- Formal ISA spec
- Binary code
decompiler
- Symbol tables etc

**seL4 – the Next 24 Months**

- Confidentiality
- Availability
- Integrity
- Abstract Model
- Executable Model
- Multicore
- Initialization
- Proof
- C Implementation
- Binary code
- WCET Analysis
- Timing-Channel Mitigation

**seL4 Multicore Design: Clustered Multikernel**

- SMP Linux
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Virtual CPU
- Kernel
- Core
- HW context
- HW context
- L1 cache
- L2 cache
- Memory
- Kernel
- Core
- HW context
- HW context
- L1 cache
- L2 cache
- Memory
- Kernel
- Core
- HW context
- HW context
- L1 cache
- L2 cache
- Memory
- Kernel
- Core
- HW context
- HW context
- L1 cache
- L2 cache
- Memory

- Still no concurrency in the kernel!

**Multikernel Verification**

- By definition, multikernel images execute independently
- except for explicit messaging

- To prove:
  - isolated images are initialised correctly
  - images maintain isolation at run time

- Essentially non-interference
**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**

**Logical Function**

- Security Property:
  - No data leakage between red and blue networks
Logical Function

Security Property:
• No data leakage between red and blue networks

Minimal TCB

Implementation

Access Rights
Next Step: Full System Assurance

DARPA HACMS Program:
- Provable vehicle safety
- "Red Team" must not be able to divert vehicle

Boeing Unmanned Little Bird (AH-6) Deployment Vehicle

SMACCMcopter Research Vehicle

SMACCMcopter System Structure

Hardware
- Sensors
  - gyro,
  - accel,
  - ...

C&C
- Radio control
- Microcontroller

Verified RTOS

C&C
- CAN bus controller

Verified OS Kernel (seL4)

Mission Board
- Monitor
- Control
- CAN bus controller

Control Board
- C&C
- File system
- Device drivers
- Untrusted Linux kernel, image processing

Network camera

Architecting Security/Safety

Architecture Specification
- Requirements (specific set of security/safety properties)
- Component Model
  - Untr
  - Trusted

Automatic Analysis (Requirements fulfilled)

Automatic Generation of Glue code

Component Implementations
- Untr
- Trusted
- Untr

Formal proof
- Security

Functional correctness
- Glue Code Proof

Did you find bugs???
- During (very shallow) testing: 16
- During verification: 460
  - 160 in C, ~150 in design, ~150 in spec

seL4 Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell design</td>
<td>2 py</td>
</tr>
<tr>
<td>C implementation</td>
<td>2 months</td>
</tr>
<tr>
<td>Debugging/Testing</td>
<td>2 months</td>
</tr>
<tr>
<td>Kernel verification</td>
<td>12 py</td>
</tr>
<tr>
<td>Fastpath verification</td>
<td>5 months</td>
</tr>
<tr>
<td>Formal frameworks</td>
<td>9 py</td>
</tr>
<tr>
<td>Total</td>
<td>24 py</td>
</tr>
<tr>
<td>Repeat (estimated)</td>
<td>6 py</td>
</tr>
<tr>
<td>Traditional engineering</td>
<td>4–6 py</td>
</tr>
</tbody>
</table>

Key:
- Trusted, NICTA
- Untrusted
Why So Hard for 9,000 LOC?

seL4 call graph

Cost of Assurance

Industry Best Practice:
- "High assurance": $1,000/LOC, no guarantees, unoptimised
- Low assurance: $100–200/LOC, 1–5 faults/kLOC, optimised

State of the Art – seL4:
- $400/LOC, 0 faults/kLOC, optimised
- Estimate repeat would cost half
  - that’s about the development cost of the predecessor Pistachio!
- Aggressive optimisation [APSys’12]
  - much faster than traditional high-assurance kernels
  - as fast as best-performing low-assurance kernels

seL4: Cost of Assurance

Confidentiality
Availability
Integrity

Abstract Model

C Implementation

Binary code

4.5 py

2 py, 1.5 years Mostly for tools

2 py, 1 year Mostly for tools

0 py 4 months

21 py 4.5 years

$400 per line of code!

Estimate repeat cost: $200/LOC

What Have We Learnt?

Formal verification probably didn’t produce a more secure kernel
- In reality, traditional separation kernels are probably secure
But:
- We now have certainty
- We did it probably at less cost

Real achievement:
- Cost-competitive at a scale where traditional approaches still work
- Foundation for scaling beyond: 2 × cheaper, 10 × bigger!

How?
- Combine theorem proving with
  - synthesis
  - domain–specific languages (DSLs)
Device Drivers

- Complex, untrusted
- How make trustworthy?
- Some devices are critical!
- Drivers at user level – can encapsulate

Policy Layer
- Trustworthy Microkernel – seL4
- Processor

Device Driver
- Linux Server
- Trusted Service
- Legacy Apps

Sensitive App

How make trustworthy?

Synthesis: Device Drivers [SOSP’09]

Formal OS Interface Spec
Formal Device Spec
Formalise specs!

driver.c

In progress:
- Extract device spec from device design work-flow
- Manual optimisations
- Verified synthesis

Actually works! (On Linux & seL4)

- IDE disk controller
- UART controller
- USB-to-Eth adapter
- Intel PRO/1000 Ethernet

Working on proving correctness

- W5100 Eth shield
- Asix AX88772 USB-to-Eth adapter
- SD host controller
Hardware Design Workflow

Informal specification → High-level model → Register-transfer-level description → netlist

- Manual transformation
- Too detailed

- Low-level description: registers, gates, wires.
- Cycle-accurate
- Precisely models internal device architecture and interfaces
- "Gold reference"

Generation: File System

File-system properties:
- Multiple, pre-defined abstraction levels
- Naturally modular
- Lots of "boring" code
- (de-)serialisation
- error handling

Manual Proof

Component Spec (Isabelle)

Component Spec (Isabelle)

Component Implementation (C)

Generated Proof

Generated

Code and Proof Co-Generation from DSLs

Pilot project: Flash file system
- Linux-compatible
- Fits between VFS and flash abstraction (UBI)

Manual, FS-specific

Manual, FS-independent

Generated

DDSL code

Data layout spec

CDLS code

Marshalling code spec

Control Code spec

Isabelle specs & proofs

Functional spec

Control Code Spec

Proof

Control Code (C)

Verified code
Future: Full-Scale Trustworthy System

- Verified microkernel
- Verified Device Drivers
- Verified File systems
- Verified Network Stacks
- Verified High-level runtime
- Verified Resource Management

Untrusted VM

Untrusted Apps

Verified critical application

Research system

Target system

Trustworthy Systems–Be Part of It!

An anonymous reader writes:

"Operating systems usually have flaws, and so forth. Hence, they are usually verified. The idea is to prove that a particular OS kernel is verified, and to use such a verified kernel to build a more secure system."