Present Systems are NOT Trustworthy!

An exception 06 has occurred at 0025:C18E50C in VD DiskTSD(03) + 00001060. This was called from 0025:C18A0C8 in VD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press C:\WALT\RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
What's Next?

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!

Core Issue: Complexity

- Massive functionality of CE devices ⇒ huge software stacks
  - How secure are your payments?
- Increasing usability requirements
  - Wearable or implanted
  - Patient-operated
  - GUIs next to life-critical functionality
- On-going integration of critical and entertainment functions
  - Automotive infotainment and engine control
  - Gigabytes of software on 100 CPUs...

Dealing with Complexity: Physical Isolation

Systems far too complex to prove their trustworthiness!

Does not scale!

Separate processors for critical functionality

Correctness of bus protocols?

OS

RTOS

RT App

RT App

RTOS

RT App

RT App

RT App

RT App

RT App
How About Logical Isolation?

Shared processor with software isolation

Remember: A system is not trustworthy unless proved otherwise!

Xen: 0.3 MLOC

Linux: 7.5 MLOC

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!

Critical, trusted

Complex, untrusted

System-specific, simple!

Mechanisms for enforcing isolation

Processor

Trustworthy Microkernel – seL4

General-purpose

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

System-specific, simple!

Mechanisms for enforcing isolation

Processor

Trustworthy Microkernel – seL4

General-purpose

Identify, minimise and

Critical, trusted

System-specific, simple!

Mechanisms for enforcing isolation

Processor

Trustworthy Microkernel – seL4

General-purpose
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
- Security
- Availability
- Timeliness
- Functional Correctness
- Confidence / Info Flow
- Integrity
- Termination
- Isolation!

seL4: Designing and Formalising

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

Formal Methods Practitioners

Systems Researchers

Standard Kernel Design

Kernel Hacker View

Design & Specify

Formal Model

High-Performance C Implementation

Prototype on Real Hardware

Safety Theorem

Step 2

Proof

Design in Theorem Prover

Formal Methods View

Formal Design

Iterative Design and Formalisation

Design & Specify

Formal Model

Haskell Prototype

High-Performance C Implementation

Step 2

Proof

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
Concurrent

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

Proving Functional Correctness

Abstract

Model

Executable

Imple-
Proving Functional Correctness

Abstract Model

$117,000$ lop

Proof: $30–35$ py

Executable Model

$50,000$ lop

Proof: $30–35$ py

C Implementation

$4.5$ years

Refinement: All possible implementation behaviours are captured by model

Why So Long for $9,000$ LOC?

seL4 call graph

Costs Breakdown

<table>
<thead>
<tr>
<th>Activity</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell design</td>
<td>$2$ py</td>
</tr>
<tr>
<td>C implementation</td>
<td>$2$ weeks</td>
</tr>
<tr>
<td>Debugging/Testing</td>
<td>$2$ months</td>
</tr>
<tr>
<td>Kernel verification</td>
<td>$12$ py</td>
</tr>
<tr>
<td>Formal frameworks</td>
<td>$10$ py</td>
</tr>
<tr>
<td>Total</td>
<td>$25$ 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>

Did you find bugs???

- During (very shallow) testing: $16$
- During verification: $460$
  - $160$ in C, $150$ in design, $150$ in spec

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: 185 cycles

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

Fastest-ever IPC on ARM11!

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
Availability: Ensuring Resource Access

- Strict separation of kernel resources
  ⇒ agent cannot deny access to another domain’s resources

Confidentiality: Limiting Read Accesses

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

Confidentiality: Limiting Read Accesses

Non-interference proof in progress:
- Evolution of Domain 1 does not depend on Domain-2 state
- Presently cover only overt information flow

seL4 as Basis for Trustworthy Systems

- Safety
- Security
- Availability
- Functional Correctness
- Memory Safety
- Integrity
- Timeliness
- Confident. / Info Flow
- Termination

seL4 as Basis for Trustworthy Systems

- Safety
- Security
- Availability
- Functional Correctness
- Memory Safety
- Integrity
- Timeliness
- Confident. / Info Flow
- Termination
Timeliness

- Microkernel
- Makes arbitrary system calls
- Delivery with bounded latency
- Non-preemptible

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

WCET Analysis Approach

- Automatic, from separate tool
- Manually determined, proved by tool
- Tune WCET by inserting interrupt checks
- Infeasible path information
- Accurate & sound model of ARM pipeline

Future: Whole-System Schedulability

- seL4
- Guarantee schedulability
- Requires model for managing time resource

Result

- Observed: 378 µs
- Computed: 378 µs

WCET presently limited by verification practicalities
- 10 µs seem achievable
seL4 as Basis for Trustworthy Systems

- Safety
  - Availability
  - Timeliness
  - Termination
- Security
  - Functional Correctness
  - Memory Safety
  - Integrity

Proving seL4 Trustworthiness

- Confidentiality
  - Abstract Model
  - Availability
  - Integrity
  - Non-Interference
  - Executable Model
  - C Implementation
  - Binary Code
  - WCET Analysis

≈ 2 py (estimate)
1 py
4 months
30–35 py
4.5 years
0 py
By construction
2 py, 1 year
Mostly for tools

seL4 – the Next 24 Months

- Confidentiality
- Availability
- Integrity
- Abstract Model
- Non-Interference
- Executable Model
- Multicore
- Initialization
- C Implementation
- Timing-Channel Mitigation?
- Binary Code
- WCET Analysis

seL4 Multicore Design: Clustered Multikernel

SMP Linux

Kernel

Core

Virtual CPU

Virtual CPU

Virtual CPU

Virtual CPU

Virtual CPU

Core

HW context

HW context

HW context

L1 cache

L1 cache

L2 cache

L3 cache / Main memory

Still no concurrency in the kernel!
**Multikernel Verification**

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

**Binary Verification**

- IPC: one-way, zero-length
  - Compiler | gcc | Compcert
  - Standard C code: 1455 cycles | 3749 cycles
  - C fast path: 185 cycles | 730 cycles

**Binary Code Verification**

- C source
- Formalised C
- Rewriting rules
- Function code
- Binary code
- Symbol tables etc

**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

Architecting Security/Safety

Requirements (specific set of security/safety properties)

Component Model

Untrusted

Automatic Generation of Glue code

Verified Glue Code

Component Implementations

Untrusted

Automatic Analysis (Requirements fulfilled)

Glue Code Proof

Functional correctness

Security

Automatic Generation

seL4 Proof

Correctness

Formal proof

Synthesis

seL4 Kernel

Communication

Init

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
Synthesis: The Next Big Thing?

- Correctness FS spec (Isabelle)
- Proof

FS components spec (Isabelle)

- Component Synthesiser
- FS components spec (Domain Specific Language)
- Proof
- FS components implementation (language C)

Building Secure Systems: Long-Term View

- Formal Verification
- Your choice! (… but managed is clearly better)

Hardware

- Trusted Userland
- seL4 Microkernel
- Native App
- Managed runtime
- Other Stuff
- GC
- Zephyr App

- Managed App

- DSL
- C + asm

Formal Verification

Trustworthy Systems–Be Part of It!

Full-System Assurance in 4 Years!

- Research system
- Target system

- New Scientist
- Saturday 29/8/2009
- Page: 21
- Section: General News
- Region: National
- Type: Magazine
- Size: 196.31 sq.cms.
- Published: —

- Slashdot
- "The ultimate way to keep your computer safe from harm"