Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  – to share—to copy, distribute and transmit the work
  – to remix—to adapt the work
• under the following conditions:
  – Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Trustworthy Systems Vision

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
- System-specific, simple!
- Mechanisms for enforcing isolation

- Processor
- Policy Layer
- Trustworthy Microkernel – seL4
- Linux Server
- Legacy Apps
- Sensitive App
- Trusted Service

Critical, trusted

Defines access rights

General-purpose
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
- Timeliness
- Termination

Security
- Availability
- Functional Correctness
- Integrity
- Confidentiality / Info Flow

Isolation!
Provable Security and Safety

Confidentiality

Integrity

Availability

Abstract Model

Proof

Proof

Proof

Isolation properties

Functional correctness

[ITP’11, S&P’13]

[SOSP’09]

Exclusions (at present):

• Initialisation
• Privileged state & caches
• Multicore
• Covert timing channels

Translation correctness [PLDI’13]

C Implementation

Binary code

Worst-case execution time [RTSS’11, RTAS’16]
Proving Functional Correctness

Abstract Model

Executable Model

C Implementation

Proof

Refinement: All possible implementation behaviours are captured by model

117,000 lop

50,000 lop
Proof of Functional Correctness

Abstract

constdefs
schedule :: "unit s_monad"
"schedule = do
  threads ← allActiveTCBs;
  thread ← select threads;
  do_machine_op flushCaches OR return ();
  modify (λ o. o.runThread := thread)
" schedule :: Kernel ()
schedule = do
  action ← getSchedulerAction
  case action of
    void
      setPriority(tcb_t *tptr, prio_t prio) {
        prio_t oldprio;
        if(thread_state_get_tcbQueued(tptr->tcbState)) {
          oldprio = tptr->tcbPriority;
          ksReadyQueues[oldprio] = tcbSchedEnqueue(tptr, ksReadyQueues[);
          if(isRunnable(tptr)) {
            ksReadyQueues[prio] = tcbSchedEnqueue(tptr, ksReadyQueues
          } else {
            thread_state_ptr_set_tcbQueued(&tptr->tcbState, false);
          }
        }
        tptr->tcbPriority = prio;
      }
    void
      yieldTo(tcb_t *target) {
        target->tcbTimeSlice += ksCurThread->tcbTimeSlice;
      }
Crash-Proof Code

Making critical software safer

7 comments
WILLIAM BULKELEY
May/June 2011
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

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

Can prove further properties on abstract level!
Verification Assumptions

1. Hardware behaves as expected
   - Formalised hardware-software contract (ISA)
   - Hardware implementation free of bugs, Trojans, ...

2. Spec matches expectations
   - Can only prove “security” if specify what “security” means
   - Spec may not be what we think it is

3. Proof checker is correct
   - Isabel/HOL checking core that validates proofs against logic

With binary verification do not need to trust C compiler!
Present Verification Limitations

• Not verified boot code
  – **Assume** it leaves kernel in safe state

• Caches/MMU presently modeled at high level / axiomised
  – This is in progress of being fixed

• Not proved any temporal properties
  – Presently not proved scheduler observes priorities, properties needed for RT
  – Worst-case execution-time analysis applies only to dated ARM11/A8 cores
  – No proofs about timing channels
Isolation Goes Deep

High

TCBs
Caps
PTs

Low

TCBs
Caps
PTs

Kernel data partitioned like user data
Hardware Faults
How About Hardware Faults?

- Single-event upset: Random (transient) bit-flips due to cosmic rays, natural radioactivity
- May break “proved” isolation
Redundant Execution

Idea: fault-tolerance through redundancy
- Compare & vote at kernel entry/exit
- Transparent Application replication
Timing Channels
Timing Channels

Information leakage through timing of events

- Typically by observing response latencies or own execution speed

**Covert channel:** Information flow that bypasses the security policy

![Diagram showing Trojan encoding info and Spy observing]

**Side channel:** Covert channel exploitable without insider help

![Diagram showing Victim executing normally and Attacker observing]
Cause: Temporal Interference

- Inter-process interference
- Competing access to micro-architectural features
  - not exposed by the ISA
  - hidden by the HW-SW contract!

Affect execution speed
Sharing 1: Stateless Interconnect

H/W is *bandwidth-limited*

- Interference during concurrent access
- Generally reveals no data or addresses
- Must encode info into access patterns
- Only usable as covert channel, not side channel
Sharing 2: Stateful Hardware

HW is capacity-limited
- Interference during
  - concurrent access
  - time-shared access
- Collisions reveal data or addresses
  - Usable as side channel

Any state-holding microarchitectural feature:
- cache
- branch predictor
- pre-fetcher state machine
Time Protection
OS Must Enforce *Time Protection*

Preventing interference is core duty of the OS!

- *Memory protection* is well established
- *Time protection* is completely absent
Time Protection: No Sharing of State

Cannot partition on-core caches (L1, TLB, branch predictor, prefetchers)
- virtually-indexed
- OS cannot control access

Partition, e.g. page colouring

Flush useless for concurrent access
- between HW threads
- between cores
- for stateless channels

Need both!
Partitioning User Memory is Easy

System permanently coloured

Global Resource Manager

Partitions restricted to coloured memory

Resource Manager

Resource Manager

RM I+D

RM I+D

Resource Manager

Resource Manager

Global Resource Manager

Colouring user data automatically colours kernel data

RAM

I+D

Still share kernel image!
Colouring the Kernel

Remaining shared kernel data:
- Scheduler queue array & bitmap
- Few pointers to current thread state

Each partition has own kernel image
Flushing on Domain Switch

1. \( T_0 = \text{current\_time}() \)
2. Switch context
3. Flush caches
4. Touch all code/data needed for return
5. Reprogram timer
6. \( \text{while } (T_0 + \text{WCET} < \text{current\_time}()) ; \)
7. return

Latency depends on prior execution!

Ensure deterministic execution

Remove dependency

seL4 proof-of-concept works
- Needs proper integration with seL4 model
- Aim: prove absence of timing channels!
Tackling Verification Cost
## Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Task</th>
<th>Time/Units</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>Abstract spec refinement</td>
<td>8 py</td>
</tr>
<tr>
<td>Executable spec refinement</td>
<td>3 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><strong>Total</strong></td>
<td><strong>24 py</strong></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>
seL4 Why So Hard for 9,000 LOC?

seL4 call graph
Cost of Assurance

Confidentiality

Availability

Integrity

Abstract Model

C Implementation

Binary code

Proof

Proof

Proof

1 py 4 months

21 py 4.5 years

$400 per line of code!

2 py, 1.5 years Mostly for tools

2 py, 1 year Mostly for tools

By construction

4.5 py

4 months

0 py

Mostly for tools

Mostly for tools

$400 per line of code!
Microkernel Life-Cycle Cost in Context

Assurance

Cost ($/SLOC)

L4 Pistachio
$100

seL4
$400

Green Hills Integrity
$1000

Slow!

Fast!

Revolution!
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 twice 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
Beyond the Kernel

Critical control

Device driver

NW stack

File system

Uncritical/untrusted

Aim: Verified TCB at affordable cost!

1 kLOC?

5 kLOC?

10 kLOC?

100 kLOC?

10 kLOC?

100 kLOC?

Linux

Apps

10 kLOC

11 py

Apps

Apps

10 kLOC

11 py

File system

NW stack

Device driver

Critical control
Cogent: Code & Proof Co-Generation

- Reduces the cost of formally verifying systems code
- Restricted, purely functional language
- Type- and memory safe, not managed
- Case-studies: BilbyFs, ext2, F2FS, VFAT

[O’Connor et al, ICFP’16; Amani et al, ASPLOS’16]
### Manual Proof Effort

<table>
<thead>
<tr>
<th>BilbyFS functions</th>
<th>Effort</th>
<th>Isabelle LoP</th>
<th>Cogent SLoC</th>
<th>Cost $/SLoC</th>
<th>LoP/ SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>isync()</td>
<td>9.25 pm</td>
<td>13,000</td>
<td>1,350</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>iget()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>library</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sync() - specific</td>
<td>3.75 pm</td>
<td>5,700</td>
<td>300</td>
<td>260</td>
<td>19</td>
</tr>
<tr>
<td>iget() - specific</td>
<td>1 pm</td>
<td>1,800</td>
<td>200</td>
<td>100</td>
<td>9</td>
</tr>
<tr>
<td>seL4</td>
<td>12 py</td>
<td>180,000</td>
<td>8,700 C</td>
<td>350</td>
<td>20</td>
</tr>
</tbody>
</table>

BilbyFS: 4,200 LoC Cogent
Remember: Verification Cost Breakdown

Abstract Spec

8 py

Proof

Executable Spec

3 py

Proof

C Implementation

Cogent spec higher level than seL4 exec spec

Fully automated in Cogent
Cogent: Present Work

• Relax type-system restrictions
  • purely linear types lead to excessive copying, 100% overhead
• Extend base language with domain-specific syntactic sugar
  • increased expressiveness → less code
  • automate boilerplate code
• Apply to other systems code
  • device drivers
  • network protocols stacks
Real-World Use
DARPA HACMS Program

- Boeing Unmanned Little Bird
- SMACCMcopter Research Vehicle
- TARDEC GVR-Bot

Retrofit existing system!

Develop technology

US Army Autonomous Trucks
Issue: Capabilities are Low-Level

>50 capabilities for trivial program!
Component Middleware: CAmkES

Higher-level abstractions of low-level seL4 constructs

Component

Connector

Interface

SharedData

RPC
Example: Simplified HACMS UAV

Security enforcement: Linux only sees encrypted data

Uncritical/untrusted, contained

- Radio Driver
- Data Link
- Crypto
- CAN Driver
- Wifi
- Camera
- Linux
Enforcing the Architecture

Architecture specification language

Low-level access rights

VSpace

Radio Driver

Data Link

Crypto

CAN Driver

Uncritical/untrusted, contained

Linux

 Compiler/Linker

binary

driver.c

VMM.c

glue.c

init.c

Thread Object

CNode

CSpace

Thread Object

CNode

CSpace

VSpace

A

B

Send

Receive

Radio

Driver

Data

Link

Crypto

CAN

Driver

Context

CONTEXT

CONTEXT

Radio

Driver

Crypto

CAN

Driver

Low-level access rights
Open-source AADL tools from Rockwell-Collins / U Minnesota

Eclipse-based IDE

AADL

Analysis Tools

Safety ✓

Architecture Analysis & Description Language

Component Description

CAmkES

Generate

.h, .c

Glue Code

Binary

Generate

Compile
Military-Grade Security

Cross-Domain Desktop Compositor

- Multi-level secure terminal
  - Successful defence trial in AU
  - Evaluated in US, UK, CA
  - Formal security evaluation soon

Pen10.com.au secure communication device
- approved for military use
- deployed in defence
Real-World Use
Courtesy Boeing, DARPA