Trustworthy Systems Vision

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

Hard guarantees on safety/security/reliability

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

Isolation is Key!

- Identify, minimise and isolate critical components!

- Critical, trusted
- Defines access rights
- General-purpose
- Complex, untrusted
- System-specific, simple!
- Mechanisms for enforcing isolation
- Policy Layer
- Trustworthy Microkernel – seL4
- Processor
- Linux Server
- Sensitive App
- Trusted Service
- Legacy Apps
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
     o despite 99% of code untrusted!

Requirements for Trustworthy Systems

Provable Security and Safety

Proving Functional Correctness
### Proving Functional Correctness

```c
const defu

schedule :: "unit s_monad"
"Schedule = do
threads ← allActiveTSCs;
thread ← selectThreads;
do_machine_op flushCaches OR return ()
end
end

schedule :: Kernel ()
schedule = do
    action ← getSchedulerAction
    case action of
```

### 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

---

**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
- **Low**
  - TCBs
  - Caps
  - PTs

Kernel data partitioned like user data

**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

Information leakage through timing of events
- Typically by observing response latencies or own execution speed

Covert channel: Information flow that bypasses the security policy

Side channel: Covert channel exploitable without insider help

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

H/W 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

- **High** Cache
- **Low** Cache

**Context Switch**
- **Flush**
- **Partitions, e.g. page colouring**
- **Need both!**
- **Cannot partition on-core caches (L1, TLB, branch predictor, prefetchers)**
  - virtually-indexed
  - OS cannot control access

**Flushing useless for concurrent access**
- between HW threads
- between cores
- for stateless channels

**Partitioning User Memory is Easy**

- **Resource Manager**
  - **Resource Manager**

- **Global Resource Manager**
  - **Colouring user data automatically colours kernel data**

**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. while \((T_0 + WCET < \text{current\_time}())\):
7. return

- **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>Component</th>
<th>Time</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>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>

Why So Hard for 9,000 LOC?

Cost of Assurance

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>4.5 py</td>
</tr>
<tr>
<td>Availability</td>
<td>Proof</td>
</tr>
<tr>
<td>Integrity</td>
<td>1 py</td>
</tr>
<tr>
<td>Abstract Model</td>
<td>21 py</td>
</tr>
<tr>
<td>C Implementation</td>
<td>2 py, 1.5 years</td>
</tr>
<tr>
<td>Binary code</td>
<td>2 py, 1 year</td>
</tr>
</tbody>
</table>

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

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

Cost of Assurance

Aim: Verified TCB at affordable cost!

Beyond the Kernel

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() library</td>
<td>3.75 pm</td>
<td>5,700</td>
<td>300</td>
<td>260</td>
<td>19</td>
</tr>
<tr>
<td>sync() - specific</td>
<td>1 pm</td>
<td>1,800</td>
<td>200</td>
<td>100</td>
<td>9</td>
</tr>
<tr>
<td>iget() - specific</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>seL4</strong></td>
<td><strong>12 py</strong></td>
<td><strong>180,000</strong></td>
<td><strong>8,700 C</strong></td>
<td><strong>350</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>

BilbyFS: 4,200 LoC 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
- US Army Autonomous Trucks
- SMACCMcopter Research Vehicle
- TARDEC GVR-Bot

**Issue: Capabilities are Low-Level**

- >50 capabilities for trivial program!

**Component Middleware: CAmkES**

- Higher-level abstractions of low-level sel4 constructs

**Example: Simplified HACMS UAV**

- Security enforcement: Linux only sees encrypted data
Enforcing the Architecture

Architecture specification language

Low-level access rights

Component code + proof

Thread Object

Thread Object

Compiler/Linker

glue.c
driver.c
VMM.c

Binary

initialised system + proof

Radio Driver

Crypto CAN Driver

Data Link

Uncritical/untrusted, contained

Linux Camera

Wifi

COMP9242 S2/2018 W13

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

COMP9242 S2/2018 W13

Architecture Analysis

Open-source AADL tools from Rockwell-Collins / U Minnesota

Eclipse-based IDE

Analysis Tools

Safety

AADL

Architecture Analysis & Description Language

Generate

Component Description

CAmKES

Generate

.h, .c

Glue Code

Compiler/Linker

binary

Driver.c

VMM.c

init.c

Radio Driver

Crypto CAN Driver

Data Link

Uncritical/untrusted, contained

Linux Camera

Wifi

COMP9242 S2/2018 W13

Real-World Use

Courtesy Boeing, DARPA