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
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
- Critical, trusted
- Defines access rights
- General-purpose
- Trustworthy Microkernel – seL4
- Processor
- Policy Layer

Legacy Apps
- Linux Server
- Sensitive App
- Trusted Service

© 2016 Gernot Heiser. Distributed under CC Attribution License
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
  - Confident. / Info Flow

Isolation!
Provable Security and Safety

Confidentiality

Integrity

Availability

Isolation properties [ITP’11, S&P’13]

Functional correctness [SOSP’09]

Abstract Model

Proof

Proof

Proof

C Implementation

Translation correctness [PLDI’13]

Translation correctness [PLDI’13]

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

Binary code

Exclusions (at present):
- Initialisation
- Privileged state & caches
- Multicore
- Covert timing channels

Functional correctness [SOSP’09]
Proving Functional Correctness

Abstract Model

Proof 117,000 lop

Executable Model

Proof 50,000 lop

C Implementation

Refinement: All possible implementation behaviours are captured by model
constdefs
    schedule :: "unit s_monad"
    "schedule = do
        threads ← allActiveTCBs;
        thread ← select threads;
        do_machine_op flushCaches OR return ();
        modify (λq. q.run thread := thread 0)
    \"

    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] = tcbSchedDequeue(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!
Kernel data partitioned like user data
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!
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
- Work in progress (Yanyan’s PhD)
Side Channels
Side Channel Attacks

Information leakage through shared hardware, e.g. caches

E.g. encryption keys
Types of Side Channels

Storage Channels
- Use some shared state
- Could be inside the OS/hypervisor
  - Eg existence of a file
  - Eg accessibility of an object

Timing Channels
- Observe timing of events
- Eg memory access latency
  - Senses victim’s cache footprint

seL4: The world’s only OS proved free of storage channels!

How about timing channels?
Timing Side-Channel Attack in Public Cloud

High (Victim)

Low (Attacker)

Core

L1 Cache

L2 Cache

L3 Cache

Side Channel
Analysing Memory Access Latency

Trace reveals encryption key

![Graph showing cache misses and time slots](image)
Mitigation: Partition Cache (Colouring)

High (Victim)

Low (Attacker)

Core

L1 Cache

L2 Cache

Side Channel
Colouring the System is Easy

System permanently coloured

Partitions restricted to coloured memory

Resource Manager
- RM I+D

Resource Manager
- RM I+D

Global Resource Manager

RAM
- I+D

© 2016 Gernot Heiser. Distributed under CC Attribution License
Analysing Memory Access Latency
Coloured System

Trace reveals No information
Timing Channel Through Kernel

High (Trojan)

```c
int count = 0;
for( ; ;) {
    wait_for_new_system_tick();
    if ((count % 13) < 5)
        syscall(...);
    count++;
}
```

Low (Spy)

```c
for( t = 0; t < 100 ; t++) {
    wait_for_new_system_tick();
    for (i = 0; i < prob_sets; i++)
        result[t][i] = cache_probe(i);
}
```
Cache Covert Channel Through Kernel Spy observations

Misses on sets used by kernel for trojan syscalls
**Coleouring the Kernel**

**Only shared kernel data:**
- Scheduler queue array & bitmap
- Pointers to current: thread, kernel, page table, cap space, FPU state

Each partition has own kernel image

Kernel clone!

Global Resource Manager

Resource Manager

<table>
<thead>
<tr>
<th>RM</th>
<th>sel4</th>
<th>I+D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sel4</td>
<td>I+D</td>
</tr>
</tbody>
</table>

Resource Manager

<table>
<thead>
<tr>
<th>RM</th>
<th>sel4</th>
<th>I+D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sel4</td>
<td>I+D</td>
</tr>
</tbody>
</table>

RAM

<table>
<thead>
<tr>
<th>sel4</th>
<th>I+D</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>RM</td>
</tr>
<tr>
<td>M</td>
<td>I+D</td>
</tr>
</tbody>
</table>
Timing Channel Through Kernel

High (Trojan)

```c
int count = 0;
for( ; ;) {
    wait_for_new_system_tick( );
    if ((count % 13) < 5)
        syscall(...);
    count++;
}
```

Low (Spy)

```c
for( t = 0; t < 100 ; t++) {
    wait_for_new_system_tick( );
    for (i = 0; i < prob_sets; i++)
        result[t][i] = cache_probe(i)
}
```
Cache Covert Channel Through Kernel
Spy observations with coloured kernel

Only self-conflict misses, no time signal!
But – Your Processor Leaks!

**Intel Sandy Bridge I-cache channel**

- **No mitigation**
- **With cache flush**
- **With all mitigations**

**Intel Skylake**

- **All mitigations**

Causes: Un-flushable state in
- instruction prefetcher
- branch predictor

**ARM is no better!**
Tackling Verification Cost
## Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Activity</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><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>

### Reusable!
- Abstract Spec
- Executable Spec
- C Implementation
- Proof

---

The table above outlines the verification cost breakdown for the Haskell design, C implementation, debugging/testing, abstract spec refinement, executable spec refinement, fastpath verification, and formal frameworks. The total cost is estimated to be 24 person-years (py). Reuse is indicated for specific activities, and traditional engineering is estimated to range from 4 to 6 person-years.
Why So Hard for 9,000 LOC?
Cost of Assurance

Confidentiality
- 4.5 py

Availability
- Proof

Integrity
- 1 py
- 4 months

Abstract Model
- Proof
- 21 py
- 4.5 years

C Implementation
- Proof
- 2 py, 1.5 years
- Mostly for tools

Binary code
- Proof
- 2 py, 1 year
- Mostly for tools

By construction
- 0 py

$400 per line of code!

Estimate repeat cost: $200/LOC
Microkernel Life-Cycle Cost in Context

Cost ($/SLOC)

Assurance

L4 Pistachio
$100

seL4
$400

Green Hills Integrity
$1000

Slow!

Fast!

Revolution!
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 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
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)
Our Approach

- Cogent: code and proof co-generation
  - Implement FS in high-level functional language (and reason about it)
  - Generate efficient low-level code in C
  - Automatically prove correspondence between the two
Cogent Workflow

- Cogent: purely functional memory-safe language
Cogent Workflow

- Cogent's certifying compiler generates a C implementation
- Cogent generates a specification and a proof that links it to the C code
Cogent Workflow

- Prove high-level properties about Cogent-generated specifications using a proof assistant
Cogent File Systems

• We implemented two Linux FSs:
  - Ext2: functionally complete original spec
    - No ACLs, symlinks
  - BilbyFs: custom flash file system

• Invoked from VFS via a small C wrapper, which:
  - Uses a global lock to prevent concurrent execution of FS operations
  - Handles VFS caches
  - Calls Cogent FS entry points

• FSs interface with the storage device via external ADT functions
Evaluation

• Compare ext2 with Linux's native implementation
  - Hardware:
    • 4 core i7-6700 running at 3.1 GHz,
    • Samsung HD501JL 7200RPM 500G SATA disk

• Compare BilbyFs with handwritten C implementation
  - Hardware:
    • Mirabox development board
    • Marvell Armada 370 single-core 1.2 GHz ARMv7 processor
    • 1 GiB of NAND flash
IOZone Random 4KiB Writes

- 20% CPU load for Cogent BilbyFs vs 15% for C
- Both ext2 implementations have the same CPU load
## Postmark on RAM Disk

<table>
<thead>
<tr>
<th>System</th>
<th>Total time sec</th>
<th>Creation files/sec</th>
<th>Read rate kB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>C ext2</td>
<td>10</td>
<td>5025</td>
<td>248</td>
</tr>
<tr>
<td>COGENT ext2</td>
<td>21</td>
<td>2393</td>
<td>118</td>
</tr>
<tr>
<td>C BilbyFs</td>
<td>6</td>
<td>33375</td>
<td>431</td>
</tr>
<tr>
<td>COGENT BilbyFs</td>
<td>10</td>
<td>20025</td>
<td>259</td>
</tr>
</tbody>
</table>

- Degradation of a factor 2 for Cogent FSs
Postmark on RAM Disk

<table>
<thead>
<tr>
<th>System</th>
<th>Total time</th>
<th>creation</th>
<th>read rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec</td>
<td>files/sec</td>
<td>kB/sec</td>
</tr>
<tr>
<td>C ext2</td>
<td>10</td>
<td>5025</td>
<td>248</td>
</tr>
<tr>
<td>COGENT ext2</td>
<td>21</td>
<td>2393</td>
<td>118</td>
</tr>
<tr>
<td>C BilbyFs</td>
<td>6</td>
<td>33375</td>
<td>431</td>
</tr>
<tr>
<td>COGENT BilbyFs</td>
<td>10</td>
<td>20025</td>
<td>259</td>
</tr>
</tbody>
</table>

- Degradation of a factor 2 for Cogent FSs

- Overhead is due to two reasons:
  - extra copying involved when converting in-buffer directory entries into Cogent's internal data type
  - Cogent compiler is overly reliant on C compiler's optimiser to convert automatically C structs passed by copy to pointers
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
Real-World Use
DARPA HACMS Program

Retrofit existing system!

Boeing Unmanned Little Bird

US Army Autonomous Trucks

SMACCMcopter Research Vehicle

TARDEC GVR-Bot

Develop technology
Issue: Capabilities are Low-Level

>50 capabilities for trivial program!
Component Middleware: CAmkES

Higher-level abstractions of low-level seL4 constructs

Component

Connector

Interface

RPC

SharedData

CompA:A

CompB:B

CompC:C

AsynchEvent
Example: Simplified HACMS UAV

Security enforcement: Linux only sees encrypted data

Radio Driver

Data Link

Crypto

CAN Driver

Uncritical/untrusted, contained

Wifi

Camera

Linux
Enforcing the Architecture

Architecture specification language

Low-level access rights

Compiler/Linker

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

Eclipse-based IDE

AADL

Design

Component Description

CAmkES

Generate

.h, .c

Glue Code

Binary

Compile

Analysis Tools

Safety ✔

Architecture Analysis & Description Language
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 crypto communication device undergoing formal security evaluation in UK
Real-World Use
Courtesy Boeing, DARPA
Thank you!