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

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
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**
- **Functional Correctness**
- **Timeliness**
- **Termination**

**Provable Security and Safety**

- **Confidentiality**
  - Proof
- **Integrity**
  - Proof
- **Availability**
  - Proof
- **Isolation properties**
  - [ITP'11, S&P'13]
- **Functional correctness**
  - [SOSP'09]
- **Translation correctness**
  - [PLDI'13]
- **Worst-case execution time**
  - [RTSS'11, RTAS'16]
- **C Implementation**
- **Binary code**

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

**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
Proving Functional Correctness

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

Formal Verification Summary

Isolation Goes Deep
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!

How About Hardware Faults?

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

Hardware Faults

Redundant Execution

Idea: fault-tolerance through redundancy
- Compare & vote at kernel entry/exit
- Work in progress (Yanyan’s PhD)
Side Channels

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?

Side Channel Attacks

Information leakage through shared hardware, e.g. caches

E.g. encryption keys

Timing Side-Channel Attack in Public Cloud

High (Victim)  Low (Attacker)

Core
L1 Cache
L2 Cache
L3 Cache

Side Channel

Core
L1 Cache
L2 Cache
Analysing Memory Access Latency

Mitigation: Partition Cache (Colouring)

Colouring the System is Easy

Analysing Memory Access Latency

Coloured System
**Timing Channel Through Kernel**

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

**Low (Spy)**
```
for (t = 0; t < 100 ; t++) {
    wait_for_new_system_tick();
    for (i = 0; i < prob_sets; i++)
        result[t][i] = cache_probe(i);
}
```

**Covert Channel**

**Cache Covert Channel Through Kernel**

Spy observations

Misses on sets used by kernel for trojan syscalls

**Timeline**

**Ticks (ms)**

**Cache sets probed**

**Cache misses**

**Colouring 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

**Resource Manager**

**Global Resource Manager**

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

**Timing Channel Through Kernel**

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

**Low (Spy)**
```
for (t = 0; t < 100 ; t++) {
    wait_for_new_system_tick();
    for (i = 0; i < prob_sets; i++)
        result[t][i] = cache_probe(i);
}
```
Tackling Verification Cost

Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Category</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>

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!
seL4 Why So Hard for 9,000 LOC?

seL4 call graph

seL4 Cost of Assurance

Abstract Model

Availability

Confidentiality

Integrity

Proof

Proof

Proof

By construction

0 py

1 py

4.5 years

4 months

21 py

4.5 years

2 py, 1.5 years

Mostly for tools

2 py, 1 year

Mostly for tools

Binary code

C Implementation

$400 per line of code!

Estimate repeat cost: $200/LOC

seL4 Microkernel Life-Cycle Cost in Context

L4

Pistachio

$100

Green Hills

Integrity

$1000

seL4

$400

Fast!

Slow!

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
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)

Cogent Workflow

- Cogent: purely functional memory-safe language

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’s certifying compiler generates a C implementation

In-kernel file system, no language run-time and no garbage collector
Cogent generates a specification and a proof that links it to the C code

Cogent compiler

ADT library

generated C code

import

Cogent: Verifying High-Assurance File System Implementations | Sidney Amani

• 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

Cogent Workflow

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

Cogent:

C code semantics

Isabelle/HOL

Cogent: Verifying High-Assurance File System Implementations | Sidney Amani

VFS

C wrapper

Cogent FS

Storage Device

COMP9242 S2/2017 W12 © 2016 Gernot Heiser. Distributed under CC Attribution License
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

- 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

---

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

- 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

---

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

- 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

<table>
<thead>
<tr>
<th>Spec</th>
<th>Abstract Spec</th>
<th>Executable Spec</th>
<th>Cogent spec higher level than sel4 exec spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>Fully automated in Cogent</td>
<td>Cogent spec higher level than sel4 exec spec</td>
</tr>
</tbody>
</table>

Abstract Spec

Executable Spec

Cogent spec higher level than sel4 exec spec

8 py

3 py

Fully automated in Cogent

C Implementation

---
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

Higher-level abstractions of low-level sel4 constructs

Component Middleware: CAmkES

>50 capabilities for trivial program!
Example: Simplified HACMS UAV

Security enforcement: Linux only sees encrypted data

Radio Driver
Data Link
Uncritical/untrusted, contained
Crypto
CAN Driver

Linux

Enforcing the Architecture

Architecture specification language

Low-level access rights

Compiler/Linker

Glue Code

Binary

Analysis Tools

Safety

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

Eclipse-based IDE

AADL

Architecture Analysis & Description Language

Component Description

CAmkES

Generate

.h .c

Glue Code

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
Thank you!