Present Systems are NOT Trustworthy!

Yet they are expensive:
- $1,000 per line of code for “high-assurance” software!

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
- Legacy Apps
- System-specific, simple!
- Mechanisms for enforcing isolation
- Linux Server
- Trusted Service
- Defines access rights
- Sensitive App
- Processors
- Trusted
- Critical, trusted
- General-purpose
- Trustworthy Microkernel – seL4

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
     o despite 99% of code untrusted!

Requirements for Trustworthy Systems

Safety
- Availability
- Timeliness
- Termination

Security
- Functional Correctness
- Integrity
- Confidential / Info Flow

Isolation!

Confidentiality
- Proof
- Abstract Model
- Functional correctness [ITP'11, S&P'13]

Integrity
- Proof
- Binary code
- Translation correctness [PLDI'13]

Availability
- Proof
- Isolation properties
- Worst-case execution time [RTSS'11, RTAS'16]

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

Provable Security and Safety
Proving Functional Correctness

Abstract Model

Executable Model

C Implementation

Refinement: All possible implementation behaviours are captured by model

117,000 lop

50,000 lop

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!
Isolation Goes Deep

High
TCBs
Caps
PTs
Low
TCBs
Caps
PTs

Kernel data partitioned like user data

Multicore

Microkernel vs Linux Execution

App
10s of ms
10s of ms
Linux
Kernel
10s of ms

App
10s of ms
10s of ms
Microkernel
Server
10s of ms
Kernel
0.1µs

Cache Line Migration Latencies

Core
HW context
L1 cache
10–20 cycles
Core
HW context
L1 cache
1,000–10,000 cycles
Core
HW context
L1 cache
Core
HW context
L1 cache

Main memory

Data transfer takes much longer than code execution!
Cost of Locking

Locks have a cost – significant in a fast microkernel!

Big-Lock Scalability

Cycles between system calls

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

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

How about timing channels?

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

Timing Side-Channel Attack in Public Cloud

Analysing Memory Access Latency

Mitigation: Partition Cache (Colouring)
Colouring the System is Easy

- System permanently coloured
- Partitions restricted to coloured memory

Global Resource Manager

Time slots
Cache sets

Analysing Memory Access Latency
Coloured System

Trace reveals No information

Cache misses

Partitions restricted to coloured memory

System permanently coloured

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);
}

Cache Covert Channel Through Kernel
Spy observations

Misses on sets used by kernel for trojan syscalls

Cache sets probed
Cache misses

Ticks
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 its own kernel image

**Global Resource Manager**

---

**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!

---

**Tackling Verification Cost**
### Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Task</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>

### Why So Hard for 9,000 LOC?

- seL4 call graph
- seL4 call graph

### Cost of Assurance

- **Confidentiality**
  - 4.5 py
  - By construction

- **Availability**
  - 2 py, 1.5 years
    - Mostly for tools
  - 21 py
    - 4.5 years
    - By construction

- **Integrity**
  - 0 py
  - 1 py
    - 4 months

- **C Implementation**
  - $400 per line of code!
  - $400 per line of code!
  - Revolution!

- **Binary code**
  - Estimate repeat cost: $200/LOC

### Microkernel Life-Cycle Cost in Context

- seL4 $400
  - Fast!
  - Slow!
  - Green Hills Integrity $1000
  - L4 Pistachio $100

- Revolution!

- ?

- Assurance vs. Cost ($/SLOC)
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: Verifying High-Assurance File System Implementations | Sidney Amani
Cogent workflow

- Cogent’s certifying compiler generates an C implementation

In-kernel file system, no language run-time and no garbage collector

Cogent workflow

- Cogent generates a specification and a proof that links it to the C code

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

• 20% CPU load for Cogent 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

IOZone random 4k writes

![Graph showing IOZone random 4k writes throughput vs file size](image)

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
Abstract Spec

8 py

Proof

Executable Spec

Cogent spec higher level than sel4 exec spec

3 py

Proof

C Implementation

Fully automated in Cogent