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
Assurance and Verification
Refresher: Assurance and Formal Verification

• **Assurance:**
  • systematic evaluation and testing
  • essentially an intensive and onerous form of quality assurance

• **Formal verification:**
  • mathematical proof

• **Certification:** independent examination
  • confirming that the assurance or verification was done right

Assurance and formal verification aim to establish correctness of
• mechanism design
• mechanism implementation
Assurance: Substantiating Trust

- Specification
  - Unambiguous description of desired behaviour

- System design
  - Justification that it meets specification

- Implementation
  - Justification that it implements the design

- Maintenance
  - Justifies that system use meets assumptions

Informal (English) or formal (maths)

Compelling argument or formal proof

Code inspection, rigorous testing, proof
Common Criteria

- **Common Criteria for IT Security Evaluation** [ISO/IEC 15408, 99]
  - ISO standard, for general use
  - Evaluates QA used to ensure systems meet their requirements
  - Developed out of the famous US DOD “Orange Book”:
    *Trusted Computer System Evaluation Criteria* [1985]

- Terminology:
  - **Target of evaluation** (TOE): Evaluated system
  - **Security target** (ST): Defines requirements
  - **Protection profile** (PP): Standardised ST template
  - **Evaluation assurance level** (EAL): Defines thoroughness of evaluation
    - PPs have maximum EAL they can be used for
CC: Evaluation Assurance Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Requirements</th>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL1</td>
<td>not evaluated</td>
<td>Informal</td>
<td>not eval</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL2</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL3</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL4</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL5</td>
<td>not evaluated</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL6</td>
<td>Formal</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL7</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Informal</td>
</tr>
</tbody>
</table>

Thoroughness, cost
Common Criteria: Protection Profiles (PPs)

- **Controlled Access PP** (CAPP)
  - standard OS security, up to EAL3

- **Single Level Operating System PP**
  - superset of CAPP, up to EAL4+

- **Labelled Security PP** (LSPP)
  - MAC for COTS OSes

- **Multi-Level Operating System PP**
  - superset of CAPP, LSPP, up to EAL4+

- **Separation Kernel Protection Profile** (SKPP)
  - strict partitioning, for EAL6-7
COTS OS Certifications

• EAL3:
  • 2010 Mac OS X (10.6)

• EAL4:
  • 2003: Windows 2000
  • 2005: SuSE Enterprise Linux
  • 2006: Solaris 10 (EAL4+)
    • against CAPP (an EAL3 PP!)
  • 2007: Red Hat Linux (EAL4+)

• EAL6:
  • 2008: Green Hills INTEGRITY-178B (EAL6+)
    • against SKPP, relatively simple PPC-based hardware platform in TOE

• EAL7:
  • 2019: Prove & Run PROVENCORE
SKPP on Commodity Hardware

• SKPP: OS provides only separation

• One Box One Wire (OB1) Project
  • Use INTEGRITY-178B to isolate VMs on commodity desktop hardware
  • Leverage existing INTEGRITY certification
    • by “porting” it to commodity platform

Conclusion [NSA, March 2010]:
• SKPP validation for commodity hardware platforms infeasible due to their complexity
• SKPP has limited relevance for these platforms

NSA subsequently dis-endorsed SKPP, discontinued certifying ≥EAL5
Common Criteria Limitations

• Very expensive
  • rule of thumb: EAL6+ costs $1K/LOC
design-implementation-evaluation-certification

• Too much focus on development process
  • rather than the product that was delivered

• Lower EALs of little practical use for OSes
  • c.f. COTS OS EAL4 certifications

• Commercial Licensed Evaluation Facilities licenses rarely revoked
  • Leads to potential “race to the bottom” [Anderson & Fuloria, 2009]
Formal Verification

• Prove properties about a mathematical model of a system

Model checking / abstract interpretation:
- Cannot generally prove code correct
  • Proves specific properties
  • Has false positives or
  false negatives (unsoundness)
- Suffers state-space explosion
  ✓ May scale to large code bases

Theorem proving:
✓ Can deal with large (even infinite) state spaces
✓ Can prove functional correctness against a spec
- Very labour-intensive

Recent work automatically proved functional correctness of simple systems using SMT solvers [Hyperkernel, SOSP’17]
Model Checking and Linux: A Sad Story

- Static analysis of Linux source [Chou & al, 2001]
  - Found high density of bugs, especially in device drivers
- Re-analysis 10 years later [Palix & al, 2011]

Disappointing rate of improvement for bugs that are automatically detectable!
And the Result?

**arsTECHNICA**

**RISK ASSESSMENT** —

Unsafe at any clock speed: Linux kernel security needs a rethink

Ars reports from the Linux Security Summit—and finds much work that needs to be done.

J.M. PORUP (UK) - 9/27/2016, 10:57 PM
The ultimate way to keep your computer safe from harm

FLAWS in the code, or "kernel", that sits at the heart of modern computers leave them prone to occasional malfunction and vulnerable to attack by worms and viruses. So the development of a secure general-purpose microkernel could pave the way to safer computing.

"We're pleased to say that it does. Recently, we have a para-virtualized version of the kernel that can be used on any general-purpose computer."

"The ultimate way to keep your computer safe from harm"
Crash-Proof Code

Making critical software safer

7 comments
WILLIAM BULKLEY
May/June 2011
Proving Security and Safety

Confidentiality
- Isolation properties [ITP’11, S&P’13]

Integrity
- Functional correctness [SOSP’09]
- Translation correctness [PLDI’13]

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

Abstract Model
- Proof

C Implementation
- Proof

Binary code
- Proof

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

World’s fastest microkernel!

Exclusions (at present, Armv7):
- Kernel initialisation not yet verified
- MMU & caches modelled abstractly
- Multicore not yet verified
- Covert *timing* channels not precluded
Proving Functional Correctness

Abstract Model
4.9 kLOC Isabelle

Executable Model
13 kLOC Isabelle

Implementation
5.7 kLOC Haskell

Implementation
8.7 kLOC C

Proof
117,000 lop

Proof
50,000 lop

Refinement: all possible implementation behaviours are captured by the model
Proving Functional Correctness

constdefs
    schedule :: "unit_s Monad"
    "schedule = do
        threads ← allActiveTCBs;
        thread ← select threads;
        do machine_op flushCaches OR return ();
        modify (λs. s \{ cur_thread := thread \})
    od"

schedule :: Kernel ()
schedule = do
    action ← getSchedulerAction
    case action of
        Normally → do
            thread ← getCurThread
            lo ← isRunnable curThread
            threadGet tcbTimeSlice curThread
            if (not runnable || time -= 0) chooseThread

void
    getPriority(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;
Functional Correctness 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
  - Will never access array out of bounds
  - 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

Can prove further properties on abstract level!

Bugs found:

- 16 in (shallow) testing
- 460 in verification
  - 160 in C,
  - 150 in design,
  - 150 in spec
Binary Code Verification

- C Source
  - Formal C Semantics
  - Compiler
  - Proof?

- Formalised C
  - Rewrite Rules
  - Symbol Tables
  - Formal ISA Spec

- Graph Language
  - Proof
  - SMT Solver

- Formalised Binary
  - De-compiler

Target of functional correctness proof

Proof?
Isolation Goes Deep

Kernel data partitioned like user data
To prove:
Low has no *write* capabilities to High objects
⇒ no action of Low will modify High state
Specifically, *kernel does not modify on Low’s behalf!*
Availability: Ensuring Resource Access

Strict separation of kernel resources
⇒ Low cannot deny High access to resources

Nothing to do, implied by other properties!
Confidentiality: Control Information Flow

Non-interference proof:
- Evolution of Low does not depend on High state
- Also shows absence of covert storage channels

To prove:
Low has no read capabilities to High objects
⇒ no action will reveal High state to Low
Confidentiality Proof Challenge

Spec

```c
bool a();
bool b() {
    int secret;
}
```

Idiotic but valid refinement

Implementation

```c
bool a() {
    return !secret;
}
```

Non-determinism breaks confidentiality under refinement!

Solution:

- Remove non-determinism where it affects confidentiality
- Eg: scheduler strictly round-robin

Infoflow is very strong property, requiring restrictions rarely met in real world
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
  - MMU model just finished

- Not proved any temporal properties
  - Presently not proved scheduler observes priorities, properties needed for RT
  - WCET analysis applies only to dated ARM11/A8 cores
  - No proofs about timing channels
<table>
<thead>
<tr>
<th>Level</th>
<th>Requirements</th>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL1</td>
<td>not evaluated</td>
<td>Informal</td>
<td>not eval</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL2</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL3</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL4</td>
<td>not evaluated</td>
<td>Informal</td>
<td>Informal</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL5</td>
<td>not evaluated</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL6</td>
<td>Formal</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL7</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Informal</td>
</tr>
<tr>
<td></td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
</tbody>
</table>
Cost of Verification
## Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Verification</th>
<th>Time (py)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell design</td>
<td>2</td>
</tr>
<tr>
<td>C implementation</td>
<td>0.15</td>
</tr>
<tr>
<td>Debugging/Testing</td>
<td>0.15</td>
</tr>
<tr>
<td>Abstract spec refinement</td>
<td>8</td>
</tr>
<tr>
<td>Executable spec refinement</td>
<td>3</td>
</tr>
<tr>
<td>Fastpath verification</td>
<td>0.4</td>
</tr>
<tr>
<td>Formal frameworks</td>
<td>9</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>24</strong></td>
</tr>
<tr>
<td>Non-reusable verification</td>
<td>11.5</td>
</tr>
<tr>
<td><strong>Traditional engineering</strong></td>
<td><strong>4–6</strong></td>
</tr>
</tbody>
</table>

**Reusable!**
seL4 Why So Hard for 9,000 LOC?

seL4 call graph
Verification Cost

Confidentiality
- 3.4 py

Integrity
- 0.6 py, 4 months

Availability
- 0 py, by construction

C Implementation
- 2 py, 1.5 years
  Mostly for tools
- 2 py, 1 year
  Mostly for tools
- 11.5 py, 4.5 years

Binary code

Abstract Model

Design + implementation + verification = $400/LOC

Mostly for tools
Microkernel Life-Cycle Cost in Context

Revolution!

L4 Pistachio $100–150

$400

Fast!

Green Hills INTEGRITY $1000

Slow!

Assurance

Cost ($/SLOC)
Update:
RISC-V Verification was completed in April 2020
Update:

We now have the seL4 Foundation to raise funds to support on-going seL4 development and verification!