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

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 DOT "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>

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**
  - 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
- **Conclusion (NSA, March 2010)**
  - SKPP validation for commodity hardware platforms infeasible due to their complexity
  - SKPP has limited relevance for these platforms

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]
Disappointing rate of improvement for bugs that are automatically detectable!
Proving Functional Correctness

Abstract Model

Executable Model

C Implementation

Refinement: all possible implementation behaviours are captured by the model

117,000 lop

50,000 lop

From Haskell

Proving Functional Correctness

Abstract Model

Executable Model

C Implementation

Can prove further properties on abstract level!

Refinement: all possible implementation behaviours are captured by the model

117,000 lop

50,000 lop

From Haskell

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 dereference 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

Binary Code Verification

C source

Formal C semantics

Rewrite rules

Formalised C

Formal code

SAT solver

Functional code

Symbol tables etc.

Formalised binary

Formal ISA spec

Binary code

Isolation Goes Deep

Kemel data partitioned like user data

Integrity: Control Write Access

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!

Event-based kernel always operates on behalf of well-defined user:
- Prove kernel only modifies data if presented write cap
### Confidentiality: Control Information Flow

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

- **TCB**
- **Caps**
- **PTs**

**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

### 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

### Common Criteria?

<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>not eval</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL3</td>
<td>not evaluated</td>
<td>informal</td>
<td>not eval</td>
<td>not evaluated</td>
</tr>
<tr>
<td>EAL4</td>
<td>not evaluated</td>
<td>informal</td>
<td>not eval</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>EAL8</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
</tbody>
</table>

### 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

### Confidentiality Proof Challenge

**Spec**

- **reveal x();
  - x = secrets;
  - return x;

**Implementation**

- **reveal x();
  - return x;

**Solution**:

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

**Infoprot is very strong property, requiring restrictions rarely met in real world

- Non-determinism breaks confidentiality under refinement!

### Availibility: Ensuring Resource Access

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

- **TCB**
- **Caps**
- **PTs**

**Strict separation of kernel resources**

- Low cannot deny High access to resources

**Nothing to do, implied by other properties!**
Cost of Verification

Verification Cost Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Duration/Resource</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>8 months</td>
</tr>
<tr>
<td>Formal frameworks</td>
<td>9 py</td>
</tr>
<tr>
<td>Total</td>
<td>24 py</td>
</tr>
<tr>
<td>Non-reusable verification</td>
<td>11.5 py</td>
</tr>
<tr>
<td>Traditional engineering</td>
<td>4-6 py</td>
</tr>
</tbody>
</table>

Why So Hard for 9,000 LOC?

Verification Cost

<table>
<thead>
<tr>
<th>Assurance Type</th>
<th>Cost ($/SLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4</td>
<td>$400</td>
</tr>
<tr>
<td>Pistachio</td>
<td>$100-150</td>
</tr>
<tr>
<td>Green Hills INTEGRITY</td>
<td>$1000</td>
</tr>
</tbody>
</table>

Microkernel Life-Cycle Cost in Context

Revolution! Fast! Slow!