2023 T3 Week 8 Part 2

Formal Verification and seL4

@GernotHeiser
Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) 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/4.0/legalcode
Today’s Lecture

• Assurance and verification
  • Common Criteria
  • Formal verification

• seL4
  • Functional correctness
  • Translation correctness
  • Security enforcement
  • Verification limitations
  • WCET analysis
  • Cost of verification

• Security impact of OS design
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: 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
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><strong>Formal</strong></td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL7</td>
<td><strong>Formal</strong></td>
<td>Formal</td>
<td><strong>Formal</strong></td>
<td>Informal</td>
</tr>
</tbody>
</table>

The table shows the evaluation levels (EAL1 to EAL7) and their associated evaluation criteria across different phases of software development: Requirements, Specification, Design, and Implementation. The level of formality for each phase is indicated, with more formal approaches typically requiring higher levels of assurance or evaluation. The table also reflects the cost and thoroughness associated with these levels, with higher EALs typically involving more thorough and formalized evaluation processes.
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
    - TEE OS for Arm TrustZone

Get regularly hacked!
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 [Green Hills]
    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

Automatic ("push-button") techniques
• Model checking / abstract interpretation / SMT
• Systematic exploration of system state space
  ❏ Cannot generally prove code correct
    • Proves specific properties
    • Functional correctness in simple cases
  ❏ Generally have to
    • over-approximate (false positives), or
    • under-approximate (false negatives, unsound)
  ❏ Suffers state-space explosion
  ✓ Can scale to large code bases

Interactive techniques:
• Theorem proving
• Proofs about state spaces
  ✓ 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]
Theorem Proving

“Forward simulation”: Prove state correspondence of abstract and concrete levels.

\[ \sigma \xrightarrow{\text{operation}} \sigma' \]

\[ \text{state relation} \]

\[ \sigma \xrightarrow{\text{state relation}} \text{abstract} \]

\[ s \xrightarrow{\text{operation}} s' \]

\[ \text{concrete} \]

prove!
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?

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 Linux kernel today faces an unprecedented safety crisis. Much like when BlueBorne hit the news in 2016.
August 2009

A NICTA bejelentette a világ első, formális módszerekkel igazolt,

New Scientist
Saturday 29/8/2009
Page: 21
Section: General News
Region: National
Type: Magazines Science / Technology
Size: 196.31 sq.cms.
Published: -----S-

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 a new generation of operating systems.

Just mathematics, and you can reason about them mathematically,” says Klein.

His team formulated a model with more than 200,000 logical steps which allowed them to prove that the program would always behave as its developer intended.
Crash-Proof Code

Making critical software safer

7 comments

WILLIAM BULKELEY
May/June 2011
Proving Security and Safety (Armv6/7)

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

**Abstract Model**
- 2019 ACM SIGOPS Hall-of-Fame Award

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

**Binary code**
- Still most comprehensive verification
- Still only verified capability-based OS
Other Architectures

Confidentiality  Integrity  Availability

Security Enforcement

Armv7
RISC-V

Abstract Model

Functional Correctness

Armv7
x86
RISC-V

Armv8: Q1'24

C Implementation

Translation Correctness

Armv8: Q1’24

Binary code

Armv7
RISC-V

All code and proofs are open source
Security Is No Excuse For Bad Performance!

<table>
<thead>
<tr>
<th>Cost</th>
<th>seL4</th>
<th>Fiasco.OC</th>
<th>Zircon</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC RT latency (cycles)</td>
<td>986</td>
<td>2717</td>
<td>8157</td>
</tr>
<tr>
<td>Mand. HW cost (cycles)</td>
<td>790</td>
<td>790</td>
<td>790</td>
</tr>
<tr>
<td>Abs. overhead (cycles)</td>
<td>196</td>
<td>1972</td>
<td>7367</td>
</tr>
<tr>
<td>Rel. overhead (%)</td>
<td>25</td>
<td>240</td>
<td>930</td>
</tr>
</tbody>
</table>

Source: Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen: “SkyBridge: Fast and Secure Inter-Process Communication for Microkernels”, EuroSys, April 2019

Round-trip, cross-address-space IPC on x64 (Intel Skylake)

<table>
<thead>
<tr>
<th>Operation</th>
<th>1-way</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSCALL</td>
<td>82</td>
<td>164</td>
</tr>
<tr>
<td>SWAPGS</td>
<td>2×26</td>
<td>104</td>
</tr>
<tr>
<td>Switch PT</td>
<td>186</td>
<td>372</td>
</tr>
<tr>
<td>SYSRET</td>
<td>75</td>
<td>150</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>395</strong></td>
<td><strong>790</strong></td>
</tr>
</tbody>
</table>

Hardware cost dominates

SW overheads dominate
Functional Correctness
Proving Functional Correctness

- Abstract Model
  - 4.9 kLOC Isabelle

- Executable Model
  - 13 kLOC Isabelle

- Implementation
  - 5.7 kLOC Haskell

- Formalised C
  - (Isabelle)
  - 8.7 kLOC C
  - Cparser

- Refinement: all possible implementation behaviours are captured by the model

- Proof
  - 117,000 lop
  - 50,000 lop
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

Bugs found:

• 16 in (shallow) testing
• 460 in verification
  • 150 in spec,
  • 150 in design,
  • 160 in C
Translation Correctness
Binary Verification: Translation Validation

- **C Source** ➔ **Formalised C**
  - Formal C Semantics
  - Rewrite Rules

- **Compiler** ➔ **Symbol Tables** ➔ **Formalised Binary** ➔ **De-compiler**
  - Proof

- **Graph Language** ➔ **SMT Solver**

Target of functional correctness proof
Security Enforcement
Isolation Goes Deep

Kernel 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 be-half of well-defined user:
• Prove kernel only modifies data if presented write cap
Availability: Ensuring Resource Access

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

Nothing to do, implied by other properties!
To prove:
Low has no read capabilities to High objects
\(\Rightarrow\) no action will reveal High state to Low

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

Spec
bool a();

Implementation
bool a() {
  return !secret;
}

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

Non-determinism breaks confidentiality under refinement!

Infoflow is very strong property, requiring restrictions rarely met in real world
Limitations
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

- 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 (yet)

MMU model finished by recent PhD

Present research!
<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?**
WCET Analysis
WCET Analysis

Challenges:
- Minimise pessimism – establish tight bounds/models

Accurate & sound model of pipeline, caches

Analysis tool

Integer linear equations

ILP solver

WCET

Scalability!

Program binary

Control-flow graph

Micro-architecture model

Loop bounds

Infeasible path info

Pessimism!
Loop Bounds & Infeasible Paths

Tight loop bounds and infeasible path refutations infeasible to obtain from binary – lack of semantic information, especially pointer aliasing analysis.

Idea:
• prove on C level
• transfer to binary using translation-validation toolchain

Program binary → Control-flow graph → Micro-architecture model

Analysis tool → Integer linear equations

ILP solver → WCET

C Implementation

Loop bounds

Proof

Annotate

Extract

Binary code

Result: High-assurance & tight bounds!
WCET Analysis on ARM11

WCET presently limited by verification practicalities
• without regard to verification achieved 50 µs
• 10 µs seem achievable
• BCET ~ 1µs
• [Blackham’11, ‘12] [Sewell’16]

Problem: Latency information no longer published by Arm!

Pessimism mostly due to under-specified hardware

Observed
Computed

Likely worst case

99.5 µs
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!**

[Diagram showing the breakdown of verification costs with arrows indicating proof, executable spec, and C implementation stages.]
Why So Hard for 9,000 LOC?

seL4 call graph
Verification Cost

Confidentiality
- 3.4 py
- Mostly for tools

Integrity
- 2 py, 1.5 years
- Mostly for tools
- 2 py, 1 year
- Mostly for tools
- Design + implementation + verification = $400/LOC

Availability
- 0 py, by construction
- 0.6 py, 4 months
- 11.5 py, 4.5 years

Binary code
- C Implementation
- Proof

Abstract Model
- Proof

Design + implementation + verification = $400/LOC

© Gernot Heiser 2019 – CC BY 4.0
Microkernel Life-Cycle Cost in Context

- L4 Pistachio ($100–150)
- Green Hills INTEGRITY ($1000)

Cost ($/SLOC)

Assurance

Slow!

Fast!

Revolution!
Security Impact of OS Design
Quantifying OS-Design Security Impact

Approach:

• Examine all *critical* Linux CVEs (vulnerabilities & exploits database)
  - easy to exploit
  - high impact
  - no defence available
  - confirmed

• For each establish how microkernel-based design would change impact

115 critical Linux CVEs to Nov’17
Hypothetical seL4-based OS

OS structured in *isolated* components, minimal inter-component dependencies, *least privilege*

Functionality comparable to Linux
App requires:
• IP networking
• File storage
• Display output
All Critical Linux CVEs to 2017

- **Not in TCB:** Attack defeated
  - 41% eliminated
  - 58% low severity
  - 96% not critical

- **In microkernel:** Attack defeated by verification

- **Still full system compromise:** No effect

- **No full compromise, but violates integrity or confidentiality:** Weakly mitigated

- **Only crash essential service (availability):** Strongly mitigated

- 38% of CVEs
- 30% of CVEs
- 17% of CVEs
- 11% of CVEs
- 4% of CVEs
Conclusion: OS Structure Matters

- Microkernels definitely improve security
- Microkernel verification improves further
- Monolithic OS design is fundamentally flawed from security point of view

Use of a monolithic OS in security- or safety-critical scenarios is professional malpractice!

[Biggs et al., APSys’18]
John Lions Honours Scholarship for thesis in OS!

https://www.scholarships.unsw.edu.au/scholarships/id/1757/6077