Maksym Planeta   Björn Döbel

Operating Systems Meet Fault Tolerance

Microkernel-Based Operating Systems // Dresden (online), 16.01.2024
‘If there is more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.’

Edward Murphy jr.
Goal of the Lecture

OS in critical environments

• Safety
• Security
• Performance
Goal of the Lecture

OS in critical environments

- Safety
- Security
- Performance
Dependability

• Availability
  Average fraction of time that a component has been up and running
• Reliability
  Probability that a component has been up and running continuously
• Maintainability
  Time required to repair a faulty component

---

Textbook terminology

Dependability threats:
• Failure
• Error
• Fault

Dependability means
• Prevention
• Removal
• Forecasting
• Tolerance
Resilience

Persistence of dependability when facing changes
# Dependability vs. Resilience

<table>
<thead>
<tr>
<th>Dependability</th>
<th>Technologies for Resilience</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Evolvability</td>
</tr>
<tr>
<td>Fault Prevention</td>
<td>✓</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>✓</td>
</tr>
<tr>
<td>Fault Removal</td>
<td>✓</td>
</tr>
<tr>
<td>Fault Forecasting</td>
<td>✓</td>
</tr>
</tbody>
</table>
Faults:
- Software (bugs)
- Hardware

Measures:
- Software Engineering
- Software Architectures
A Classic Study

- A. Chou et al.: *An empirical study of operating system errors*, SOSP 2001
- Automated software error detection (today: https://www.coverity.com)
- Target: Linux (1.0 - 2.4)
  - Where are the errors?
  - What error types do exist?
  - How long do they survive?
  - Do bugs cluster in certain locations?
Revalidation of Chou’s Results

• N. Palix et al.: *Faults in Linux: Ten years later*, ASPLOS 2011

• 10 years of work on tools to decrease error counts - has it worked?

• Repeated Chou’s analysis until Linux 2.6.34
To give an overview of the software we are studying, we first consider the evolution in code size of the Linux kernel between version 1.0, released in March 1994, and the latest version, version 2.6.33, released in February 2010, as shown in Figure 1. We give the size of the development versions, which includes drivers/staging and other directories: drivers, sound, and include only the ANSI C code. The code sizes are broken down by directory, highlighting the largest directories.

For most directories, the code growth has been roughly linear since Linux 1.0. Some exceptions are the directory for new drivers that are not yet mature enough to be included in standard Linux distributions. When available, as it is in these versions that new code is added, and this added code is then maintained in the subsequent stable versions. Code sizes are computed using David A. Wheeler's 'SLOCCount' (v2.26) [27].

The largest directory is drivers/staging and includes new drivers that are not yet mature enough to be included in standard Linux distributions. Code in drivers/staging is added in Linux 2.6.28 as an incubator for new drivers. When available, this code is maintained in the subsequent stable versions. Code sizes are computed using David A. Wheeler's 'SLOCCount' (v2.26) [27].

The directory for new drivers that are not yet mature enough to be included in standard Linux distributions is drivers/staging. This directory includes new drivers that are not yet mature enough to be included in standard Linux distributions. Code in drivers/staging is added in Linux 2.6.28 as an incubator for new drivers. When available, this code is maintained in the subsequent stable versions. Code sizes are computed using David A. Wheeler's 'SLOCCount' (v2.26) [27].

Finally, staging grew substantially in 2.6.29. All in all, these changes amounted to around 180,000 lines of C code to arch. Finally, staging grew substantially in 2.6.29. All in all, these changes amounted to around 180,000 lines of C code to arch.
Analysis. It is unknown whether this set of bugs is representative of all errors. We attempt to compensate for this by examining bugs across time, or supplement static results with dynamic traces.

Bad programmers will be consistently bad. They are not presenting distributions, and aggregating samples. One likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering of the previous section, we want to answer the following questions: Where are the errors? Do drivers actually find a variety of different types of errors and (2) drivers.

This paper shows patterns in all bugs. An interesting improvement would be to find patterns only in important bugs. However, this effect is even more pronounced when we correct for code size. Figure 4 does so by plotting distributions, and aggregating samples. One likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering of the previous section, we want to answer the following questions: Where are the errors? Do drivers actually find a variety of different types of errors and (2) drivers.

These graphs show that driver code is the most buggy, both in terms of absolute number of bugs (as additional information, consider several thousand lines of code structured so that it only performs two potentially failing allocations along with how often it was done correctly (as the notes do). The result of this low-level focus is that good code tends to be more familiar with the device.

Figure 3 gives a breakdown of the absolute count of errors per directory in Linux [4]. Figure 4: This graph shows drivers have an error rate up to 7 times higher than the rest of the kernel. The clustering of the previous section, we want to answer the following questions: Where are the errors? Do drivers actually find a variety of different types of errors and (2) drivers.

The only checker that has a disproportionate number of bugs is the Range checker. We found three identical errors in the rang directory, which is unsurprising since it is the most code. Currently we only compile the rest of the kernel using the formula: err

Number of Errors per Directory in Linux

Figure: Number of errors per directory in Linux [4]
Fault Rate per Subdirectory (2001)

Figure: Rate of errors compared to other directories [4]
Fault Rate per Subdirectory (2011)

Figure: Rate of errors compared to other directories [19]
Bug Lifetimes (2011) [19]

Figure: Per directory

Figure: Per finding and fixing difficulty, and impact likelihood
Software Engineering Measures

- QA
  Examples: Manual testing, automated testing, fuzzing
- Continuous Integration
- Static analysis
- Using safer languages
- Guidelines, best practices, etc.
  Examples: MISRA C++, C++ Guideline Support Library
Rule 0-1-7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
**Example: MISRA C++ 2008**

<table>
<thead>
<tr>
<th>Rule 0-1-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 3-9-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>The underlying bit representations of floating-point values shall not be used.</td>
</tr>
</tbody>
</table>
**Example: MISRA C++ 2008**

<table>
<thead>
<tr>
<th>Rule 0-1-7</th>
<th>The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 3-9-3</td>
<td>The underlying bit representations of floating-point values shall not be used.</td>
</tr>
<tr>
<td>Rule 6-4-6</td>
<td>The final clause of a switch statement shall be the default-clause.</td>
</tr>
</tbody>
</table>
(Required) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

**Rationale**

Defining variables in the minimum block scope possible reduces the visibility of those variables and therefore reduces the possibility that these identifiers will be used accidentally. A corollary of this is that global objects (including singleton function objects) shall be used in more than one function.
Rule 3-4-1: Example

```c
void f(int32_t k)
{
    int32_t j = k * k; // Non-compliant
    {
        int32_t i = j; // Compliant
        std::cout << i << j << std::endl;
    }
}
```

In the above example, the definition of j could be moved into the same block as i, reducing the possibility that j will be incorrectly used later in f.
Safer languages

- Garbage collection (Go)
- Memory safety (Rust)
- No unused variables (Go, Rust)
- Check error return codes (Go, Rust)
- No uninitialised memory (Go, Rust)
- etc.
Writing a kernel in a high-level language\(^2\)

- Biscuit: a monolithic kernel implemented in Go

Writing a kernel in a high-level language

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection

---

Writing a kernel in a high-level language\textsuperscript{2}

• Biscuit: a monolithic kernel implemented in Go
• High-level features: closures, channels, garbage collection
• Development effort: 28k lines in Go and 1.5k lines in Assembly

Writing a kernel in a high-level language

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly
- Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers

---

Writing a kernel in a high-level language

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly
- Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers
- Out of 64 CVE-listed Linux kernel bugs, \( \approx 40 \) would be fully or partially alleviated by Go

Writing a kernel in a high-level language²

• Biscuit: a monolithic kernel implemented in Go
• High-level features: closures, channels, garbage collection
• Development effort: 28k lines in Go and 1.5k lines in Assembly
• Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers
• Out of 64 CVE-listed Linux kernel bugs, ≈ 40 would be fully or partially alleviated by Go
• 5% to 15% slower, up to 600µs latencies for GC

Writing a kernel in a safe language

- Tock: an embedded OS implemented in Rust

---

Writing a kernel in a safe language

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
  - Several immutable references or one mutable one
  - No null pointers
  - No reading undefined memory
  - etc.
- Unsafe code is annotated
- Memory or synchronization problems are impossible in safe code
- Performance like in C or C++ code
- Some software patterns don’t work with (safe) Rust well

---

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one

---

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one
  – No null pointers

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one
  – No null pointers
  – No reading undefined memory

---

Writing a kernel in a safe language

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
  - Several immutable references or one mutable one
  - No null pointers
  - No reading undefined memory
  - etc.

---

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one
  – No null pointers
  – No reading undefined memory
  – etc.
• Unsafe code is annotated

---

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one
  – No null pointers
  – No reading undefined memory
  – etc.
• Unsafe code is annotated
• Memory or synchronization problems are impossible in safe code

---

Writing a kernel in a safe language

• Tock: an embedded OS implemented in Rust
• Compiler enforced rules:
  – Several immutable references or one mutable one
  – No null pointers
  – No reading undefined memory
  – etc.
• Unsafe code is annotated
• Memory or synchronization problems are impossible in safe code
• Performance like in C or C++ code

---

Writing a kernel in a safe language

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
  - Several immutable references or one mutable one
  - No null pointers
  - No reading undefined memory
  - etc.
- Unsafe code is annotated
- Memory or synchronization problems are impossible in safe code
- Performance like in C or C++ code
- Some software patterns don’t work with (safe) Rust well

---

Safe Monoculture Operating Systems

- Safe language for the safe OS
- Maintaining safety guarantees requires using the same language for the subcomponents
- Examples: Theseus⁴ (Rust), RedLeaf⁵ (Rust), Singularity⁶ (C#)

---


Software architectures addressing faults

• Means:
  – Compartmentalisation
  – Redundancy
  – Hardening

Figure: Ship building
Software architectures addressing faults

• Means:
  – Compartmentalisation
  – Redundancy
  – Hardening

• Address hardware faults

Figure: Ship building
Software architectures addressing faults

• **Means:**
  - Compartmentalisation
  - Redundancy
  - Hardening

• **Address hardware faults**

• **Recovery**
  - Rollback: return to a previous state
    - Transactions
    - Checkpoint/Restart
  - Roll-forward: everything else
    - Error correcting codes
    - Triple modular redundancy + majority voting

Figure: Ship building
Minix\textsuperscript{3}: A Fault-tolerant OS

User processes
- User Processes
- Server Processes
- Device Processes

Kernel
- Kernel
- Clock Task
- System Task
Minix3: Fault Tolerance

- **Address Space Isolation**
  - Applications only access private memory
  - Faults do not spread to other components

- **User-level OS services**
  - Principle of Least Privilege
  - Fine-grain control over resource access
    - e.g., DMA only for specific drivers

- **Small components**
  - Easy to replace (micro-reboot)

---

Minix3: Fault Detection

- Fault model: transient errors caused by software bugs
- Fix: Component restart
- *Reincarnation server* monitors components
  - Program termination (crash)
  - CPU exception (div by 0)
  - Heartbeat messages
- Users may also indicate that something is wrong
Repair

• Restarting a component is insufficient:
  – Applications may *depend* on restarted component
  – After restart, *component state* is lost

• Minix3: explicit mechanisms
  – Reincarnation server signals applications about restart
  – Applications store state at data store server
  – In any case: program interaction needed
    – Restarted app: store/recover state
    – User apps: recover server connection
L4ReAnimator: Restart on L4Re

• L4Re Applications
  – Loader component: ned
  – Detects application termination: parent signal
  – Restart: re-execute Lua init script (or parts of it)
  – Problem after restart: capabilities
    – No single component knows everyone owning a capability to an object
    – Minix3 signals won’t work

L4ReAnimator: Lazy recovery

- Only the application itself can detect that a capability vanished
- Kernel raises Capability fault
- Application needs to re-obtain the capability: execute capability fault handler
- Capfault handler: application-specific
  - Create new communication channel
  - Restore session state
- Programming model:
  - Capfault handler provided by server implementor
  - Handling transparent for application developer
  - Semi-transparency
Distributed snapshots

- Localized checkpoints
- Problem: Unlimited rollbacks
- Solution: Create global snapshot
- No synchronized clock
- No shared memory
- Only point-to-point messages

---

Break

• Minix3 fault tolerance
  – Architectural Isolation
  – Explicit monitoring and notifications
• L4ReAnimator
  – semi-transparent restart in a capability-based system
• Next: CuriOS
  – smart session state handling
CuriOS: Servers and Sessions

- State recovery is tricky
  - Minix3: Data Store for application data
  - But: applications interact
    - Servers store *session-specific* state
    - Server restart requires potential rollback for every participant

---

Francis M David et al. ‘CuriOS: Improving Reliability through Operating System Structure..’
CuriOS: Server State Regions

- CuiK kernel manages dedicated session memory: *Server State Regions*
- SSRs are managed by the kernel and attached to a client-server connection
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server.
- Solves another problem: failure while handling A’s request will never corrupt B’s session state.

Diagram:
- Server
  - Server State
  - call()
- Client A
  - Client State A
- Client B
  - Client State B
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Protecting Sessions

• SSR gets mapped only when a client actually invokes the server
• Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A’s request will never corrupt B’s session state
CuriOS: Transparent Restart

- CuriOS is a *Single-Address-Space OS*:
  - Every application runs on the same page table (with modified access rights)
Transparent Restart

• Single Address Space
  – Each object has unique address
  – Identical in all programs
  – Server := C++ object

• Restart
  – Replace old C++ object with new one
  – Reuse previous memory location
  – References in other applications remain valid
  – OS blocks access during restart
Transient Hardware Faults

• Radiation-induced soft errors
  – Mainly an issue in avionics+space?

• DRAM errors in large data centers
  – Google study: >2% failing DRAM DIMMs per year [20]
  – ECC insufficient [12]

• Decreasing transistor sizes → higher rate of errors in CPU functional units [7]
Transparent Replication as OS Service [9, 8]

Application

L4 Runtime Environment

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [9, 8]

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [9, 8]

- Unreplicated Application
- Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service \([9, 8]\)
Transparent Replication as OS Service [9, 8]

Replicated Driver

Unreplicated Application

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel

Reliable Computing Base
Romain: Structure

Replica

Replica

Replica

Master
Romain: Structure

Replica

Replica

Replica

Master

= OS Resilience
Maksym Planeta, Björn Döbel
Dresden (online), 16.01.2024
Romain: Structure

- Replica
- Replica
- Replica

Resource Manager

System Call Proxy

Master

Replica

Maksym Planeta, Björn Döbel
Dresden (online), 16.01.2024

OS Resilience
Dresden concept

Slide 40 of 60
Replica Memory Management

Replica 1
- rw
- ro
- ro

Replica 2
- rw
- ro
- ro

Master
Replica Memory Management

Replica 1
- rw
- ro
- ro

Replica 2
- rw
- ro
- ro

Master
Replica Memory Management

Replica 1
rw  ro  ro

Replica 2
rw  ro  ro

Master
Replicating SPEC CPU 2006 [10]

Normalized Runtime

perl     bzip2     gamess     mcf     milc     gromacs     leslie3d     namd     gobmk     calculix

Normalized Runtime

hmmer     sjeng     libquant     h264ref     tonto     lbm     omnet++     astar     sphinx3     GEOM

Single Replica  Two Replicas  Three Replicas
Replicating SPEC CPU 2006 [10]

Sources of overhead:
- System call interception
  - Frequent memory allocation
- Cache effects

Single Replica  Two Replicas  Three Replicas

Normalized Runtime

perl  bzip2  gobmk  calculix

Normalized Runtime

hmmer  sjeng  libquant  h264ref  tonto  lbm  omnet++  astar  sphinx3  GEOM

OS Resilience
Maksym Planeta, Björn Döbel
Dresden (online), 16.01.2024
Error Coverage [10]

- **Bitcount**
  - No Effect: 60%
  - Crash: 20%
  - SDC: 20%
  - Timeout: 0%
  - Recovered (Compare): 0%
  - Recovered (Timeout): 0%

- **IPC**
  - No Effect: 70%
  - Crash: 10%
  - SDC: 10%
  - Timeout: 0%
  - Recovered (Compare): 0%
  - Recovered (Timeout): 0%

- **Dijkstra**
  - No Effect: 50%
  - Crash: 40%
  - SDC: 10%
  - Timeout: 0%
  - Recovered (Compare): 0%
  - Recovered (Timeout): 0%

- **CRC32**
  - No Effect: 80%
  - Crash: 20%
  - SDC: 0%
  - Timeout: 0%
  - Recovered (Compare): 0%
  - Recovered (Timeout): 0%
Error Coverage [10]
Romain: Summary

- Faults: CPU and memory bit-flips
- Best-effort resilience
- Tripple modular redundancy with small increase in makespan
- Multithreading support with deterministic multithreading\textsuperscript{11}

\textsuperscript{11}Björn Döbel and Hermann Härtig. ‘Can we put concurrency back into redundant multithreading?’ In: EMSOFT. 2014, pp. 1–10.
HAFT: Hardware-Assisted Fault Tolerance

- CPU single-event upsets (SEU)
- Instruction-level redundancy for fault detection
- Hardware transaction memory for fault recovery
- *Best-effort* fault tolerance
- Improve efficiency through instruction-level parallelism (ILP) and compiler optimisations

---

Instruction-level redundancy

(a) Native

1
2  z = add x, y
3
4
5
6
7  ret z
Instruction-level redundancy

(a) Native
1
2 \[ z = \text{add} \ x, y \]
3
4
5
6
7 \text{ret } z

(b) ILR
z = \text{add} \ x, y
z2 = \text{add} \ x2, y2
d = \text{cmp neq} \ z, z2
\text{br } d, \text{crash}
ret z

DMR

(c) HAFT
xbegin
z = \text{add} \ x, y
z2 = \text{add} \ x2, y2
d = \text{cmp neq} \ z, z2
\text{br } d, \text{xabort}
xend
ret z
Instruction-level redundancy

(a) Native

1  z = add x, y
2  
3  
4  
5  
6  
7  ret z

(b) ILR

\[
\begin{align*}
1. & \quad z = add x, y \\
2. & \quad z2 = add x2, y2 \\
3. & \quad d = cmp neq z, z2 \\
4. & \quad br d, crash \\
\end{align*}
\]

(b) ILR

\[
\begin{align*}
1. & \quad z = add x, y \\
2. & \quad z2 = add x2, y2 \\
3. & \quad d = cmp neq z, z2 \\
4. & \quad br d, crash \\
\end{align*}
\]

DMR

(c) HAFT

\[
\begin{align*}
1. & \quad z = add x, y \\
2. & \quad z2 = add x2, y2 \\
3. & \quad d = cmp neq z, z2 \\
4. & \quad br d, xabort \\
\end{align*}
\]

ret z

TMR [15]

\[
\begin{align*}
1. & \quad r1 = add r1, r2 \\
2. & \quad r1' = add r1', r2' \\
3. & \quad r1'' = add r1'', r2'' \\
4. & \quad majority(r1, r1', r1'') \\
5. & \quad majority(r3, r3', r3'') \\
6. & \quad cmp r1, r3 \\
7. & \quad jne loop
\end{align*}
\]
Instruction-level redundancy

(a) Native
1
2 \( z = \text{add} \ x, y \)
3
4
5
6
7 \( \text{ret} \ z \)

(b) ILR
loop:
1 \( r1 = \text{add} \ r1, r2 \)
2 \( r1' = \text{add} \ r1', r2' \)
3 \( r1'' = \text{add} \ r1'', r2'' \)
4 \( \text{majority} (r1, r1', r1'') \)
5 \( \text{majority} (r3, r3', r3'') \)
6 \( \text{cmp} \ r1, r3 \)
7 \( \text{jne} \ \text{loop} \)
8 \( \text{ret} \ z \)

(c) HAFT
xbegin
1 \( z = \text{add} \ x, y \)
2 \( z2 = \text{add} \ x2, y2 \)
3 \( d = \text{cmp neq} \ z, z2 \)
4 \( \text{br} \ d, \text{xabort} \)
xend
5 \( \text{ret} \ z \)

DMR
TMR [15]
Figure 6: Performance overhead over native execution with the increasing number of threads (on a machine with 14 cores).
## Romain vs. HAFT

<table>
<thead>
<tr>
<th></th>
<th>Romain</th>
<th>HAFT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Granularity</strong></td>
<td>Syscall</td>
<td>Instruction</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td>Thread-level</td>
<td>Instruction-level</td>
</tr>
<tr>
<td><strong>Runtime overhead</strong></td>
<td>$\approx 10%$</td>
<td>$\approx 100%$</td>
</tr>
<tr>
<td><strong>Resource overhead</strong></td>
<td>$\approx 210%$</td>
<td>$\approx 100%$</td>
</tr>
<tr>
<td><strong>Faults</strong></td>
<td>CPU &amp; (some) Memory</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>OS</td>
<td>Compiler &amp; CPU features</td>
</tr>
</tbody>
</table>
Software verification

• Combines software engineering and software architectures
• Define good and bad states
• Define axioms (i.e. initial state is good)
• Prove bad states (i.e. null pointer dereference) are unreachable
• Special theorem prover languages
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
seL4: Formal verification of an OS kernel

- seL4: https://sel4.systems/
- Formally verify that system adheres to specification
- Microkernel design allows to separate components easier
- Hence verification process is easier

---

Verification of a microkernel

Figure: The seL4 design process [13]
SeL4: Conclusion

• Assumes correctness of compiler, assembly code, and hardware
• DMA over IOMMU
• Architectures: arm, x86
• Virtualization
• Future: Verification on multicores
Hardening the RCB

• **We need:** Dedicated mechanisms to protect the RCB (HW or SW)
• **We have:** Full control over software
• Use FT-encoding compiler?
  – Has not been done for kernel code yet
• RAD-hardened hardware?
  – Too expensive

Why not split cores into resilient and non-resilient ones?
Summary

• Dependability is robust development practices + reliability techniques
• Do not let failures propagate
• Silent data corruptions are the worst
Summary

• Dependability is robust development practices + reliability techniques
• Do not let failures propagate
• Silent data corruptions are the worst
• Fail fast!

Further reading: D. Bernstein: Some thoughts on security after ten years of qmail 1.0

Next week (in previous life): Practical exercise starts at 14:50
Summary

• Dependability is robust development practices + reliability techniques
• Do not let failures propagate
• Silent data corruptions are the worst
• Fail fast!
• Further reading: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*
Summary

• Dependability is robust development practices + reliability techniques
• Do not let failures propagate
• Silent data corruptions are the worst
• Fail fast!
• Further reading: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*
Summary

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst
- Fail fast!
- Further reading: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*

Next week (in previous life): Practical exercise starts at 14:50
Bibliography I


Bibliography III


Bibliography IV


Bibliography V


