Interrupt Latency in Operating-System Kernels

Challenges and Benefits of Static Analysis

March 14, 2024

Kevin Kollenda, Thomas Preisner, Dustin Nguyen, Phillip Raffeck

Friedrich-Alexander-Universität Erlangen-Nürnberg
Many systems rely on timely processing of interrupts
Motivation

Many systems rely on timely processing of interrupts.

Can we provide safe guarantees on how long interrupts can be delayed?

⇒ Blocking time analysis on assembly level
Motivation

Many systems rely on timely processing of interrupts

Can we provide safe guarantees on how long interrupts can be delayed?
Motivation

Many systems rely on timely processing of interrupts. Can we provide safe guarantees on how long interrupts can be delayed?

⇒ Blocking time analysis on assembly level
1. Interrupt Analysis
2. Pitfalls and Limitations
3. Evaluation
4. Discussion
Interrupt Analysis
Operating systems support numerous architectures
Kernel Disassembly

- Operating systems support numerous architectures
- One binary can be composed of multiple kinds
Kernel Disassembly

- Operating systems support numerous architectures
- One binary can be composed of multiple kinds

```plaintext
# 16/32 Bit
0x1000000: <startup>:
   mov ax, bx
   ...

# 64 Bit
0x1000100: <kernel_main>:
   sub rsp, 64
   ...
```
Control Flow

- Control Flow Graphs

```plaintext
function f
  test rdi, rdi
  je 0x8
  ...
  ...
  test rax, rax
  jne 0x24
  ...
  ret
```
Control Flow

- Control Flow Graphs
  - Composed of **basic blocks**, with single entry- and exit-point

```
function f
  test rdi, rdi
  je 0x8
  ...
  ...
  test rax, rax
  jne 0x24
  ...
  ret
```
Control Flow

Control Flow Graphs

- Composed of **basic blocks**, with single entry- and exit-point
- Edges indicate control flow deviations (e.g. (un-)conditional jumps)
Control Flow

- Control Flow Graphs
  - Composed of **basic blocks**, with single entry- and exit-point
  - Edges indicate control flow deviations (e.g. (un-)conditional jumps)
  - Valid for each function
Control Flow

- Control Flow Graphs
  - Composed of basic blocks, with single entry- and exit-point
  - Edges indicate control flow deviations (e.g. (un-)conditional jumps)
  - Valid for each function

- Call Graphs
Control Flow

- **Control Flow Graphs**
  - Composed of **basic blocks**, with single entry- and exit-point
  - Edges indicate control flow deviations (e.g. (un-)conditional jumps)
  - Valid for each function

- **Call Graphs**
  - Vertices are the binary’s functions, edges inferred by call targets
Control Flow

- Control Flow Graphs
  - Composed of **basic blocks**, with single entry- and exit-point
  - Edges indicate control flow deviations (e.g. (un-)conditional jumps)
  - Valid for each function

- Call Graphs
  - Vertices are the binary’s functions, edges inferred by call targets
  - Depicts **inter**-function relationships

![Diagram of control flow graphs and call graphs]

*Interrupt Latency in Operating-System Kernels*
Constant interrupt knowledge is caused by \{sti: on, cli: off, popf: ?\}.
**Constant** interrupt knowledge is caused by \{sti: on, cli: off, popf: ?}. 

```assembly
function f
  in bx, 8
  test bx, bx
  je 0x8
  cli
  sti
  ...
  test bx, bx
  jne 0x24
  sti
  ret
```
Constant interrupt knowledge is caused by \{sti: on, cli: off, popf: ?\}.

- Unaffected by surrounding blocks
**Constant** interrupt knowledge is caused by \{sti: on, cli: off, popf: \?\}.

- Unaffected by surrounding blocks
- Occurs in small subset of functions
StatusCode interrupt knowledge is caused by \{sti: on, cli: off, popf: ?}\.

- Unaffected by surrounding blocks
- Occurs in small subset of functions

⇒ Propagate knowledge dynamically throughout the control flow graph
Interrupt Knowledge

- Requires knowledge present in a function’s entry block
Interrupt Knowledge

- Requires knowledge present in a function’s entry block
- Each block is dependent on its predecessors
Interrupt Knowledge

- Requires knowledge present in a function’s entry block
- Each block is dependent on its predecessors
- Contradicting interrupt states lead to worst-case (*unknown*)
Interrupt Knowledge

- Requires knowledge present in a function’s entry block
- Each block is dependent on its predecessors
- Contradicting interrupt states lead to worst-case (unknown)
- Fixed-point iteration until results are stable
Interrupt Knowledge

- Requires knowledge present in a function’s entry block
- Each block is dependent on its predecessors
- Contradicting interrupt states lead to worst-case (unknown)
- Fixed-point iteration until results are stable
## Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

```assembly
in bx, 8
test rbx, rbx
je 0x8
cli
sti
mul rbx, 32
ret
sti
sub rbx, 8
mov rax, rbx
function f
0
1
3
4
```

DFS to find longest path

Unknown and disabled states
Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

```
in bx, 8
  test rbx, rbx
  je 0x8
cli
sti
mul rbx, 32
ret
sti
sub rbx, 8
mov rax, rbx
```
Determining Interrupt Latency

Interrupt Latency
Longest possible instruction count until interrupts are enabled again.

- DFS to find longest path
Determining Interrupt Latency

**Interrupt Latency**

Longest possible instruction count until interrupts are enabled again.

- DFS to find longest path
- Unknown and disabled states
Determining Interrupt Latency

Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

- DFS to find longest path
- Unknown and disabled states
- Weigh by instruction kind
Determining Interrupt Latency

Interrupt Latency
Longest possible instruction count until interrupts are enabled again.

DFS to find longest path
- Unknown and disabled states
- Weigh by instruction kind

```
in bx, 8
test rbx, rbx
je 0x8
cli
sti
mul rbx, 32
ret
sti
sub rbx, 8
mov rax, rbx
function f
0
1
3
4
```
Pitfalls and Limitations
Indirect Branches

- Not all control flow instructions target a static location
Indirect Branches

- Not all control flow instructions target a static location
- call rax/jmp rax depend on run-time values
Indirect Branches

- Not all control flow instructions target a static location
- `call rax/jmp rax` depend on run-time values
- Execution can continue nearly anywhere
Indirect Branches

- Not all control flow instructions target a static location
- `call rax/jmp rax` depend on run-time values
- Execution can continue nearly anywhere
- Caused by jump tables, `{interrupt, syscall}` dispatchers, ...
Not all control flow instructions target a static location

- `call rax/jmp rax` depend on run-time values
- Execution can continue nearly anywhere
- Caused by jump tables, {interrupt, syscall} dispatchers, ...
Injecting External Knowledge

- Specific compiler options like -fno-jump-tables
Injecting External Knowledge

- Specific compiler options like -fno-jump-tables
- Determine possible targets using source code level analysis
Injecting External Knowledge

- Specific compiler options like `-fno-jump-tables`
- Determine possible targets using source code level analysis
  - Interrupt service routines
Injecting External Knowledge

- Specific compiler options like -fno-jump-tables
- Determine possible targets using source code level analysis
  - Interrupt service routines
  - Virtual function tables (C++)
Injecting External Knowledge

- Specific compiler options like `-fno-jump-tables`
- Determine possible targets using source code level analysis
  - Interrupt service routines
  - Virtual function tables (C++)

```
1 dispatch:
2   mov rbx, rdi
3   mov rax, rbx
4   call rax
```
Injecting External Knowledge

- Specific compiler options like `-fno-jump-tables`
- Determine possible targets using source code level analysis
  - Interrupt service routines
  - Virtual function tables (C++)

```
1  dispatch:
2    mov rbx, rdi
3    mov rax, rbx
4    call rax
```

```
1  dispatch:
2    mov rbx, rdi
3    mov rax, rbx
4    call 0x21610
```

```
1  dispatch:
2    mov rbx, rdi
3    mov rax, rbx
4    call 0xABA80
```
Inter-Function Analysis

What happens when interrupts are still disabled in the exit block?
Inter-Function Analysis

What happens when interrupts are still disabled in the exit block?
⇒ Follow the call tree upwards
Inter-Function Analysis

**What happens when interrupts are still disabled in the exit block?**

⇒ Follow the call tree upwards

```
func: setup
func: load
func: ...

func: f

func: enable_ps2
func: trace
```
Inter-Function Analysis

What happens when interrupts are still disabled in the exit block?

⇒ Follow the call tree upwards
Loops

- Terminate early when encountering loops with disabled interrupts
Loops

- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches
Terminate early when encountering loops with disabled interrupts
Inherent limitation present in static approaches
Dataflow analysis could provide loop bound approximations
Loops

- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches
- Dataflow analysis could provide loop bound approximations

→ Repetitions limited by rdi (function parameter)
Loops

- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches
- Dataflow analysis could provide loop bound approximations

→ Repetitions limited by rdi (function parameter)
→ Caller with the highest value "wins"
Loops

- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches
- Dataflow analysis could provide loop bound approximations

→ Repetitions limited by rdi (function parameter)
→ Caller with the highest value "wins"
→ Only possible for trivial cases
Evaluation
## Analysis Environment

### Compilers:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td>17.0.6</td>
<td>{-0s, -02}, fcf-protection=none</td>
</tr>
<tr>
<td>gcc</td>
<td>13.2.1</td>
<td>{-0s, -02}, fcf-protection=none</td>
</tr>
</tbody>
</table>
## Analysis Environment

### Compilers:

<table>
<thead>
<tr>
<th>Version</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td><code>-Os</code>, <code>-O2</code>, fcf-protection=none</td>
</tr>
<tr>
<td>gcc</td>
<td><code>-Os</code>, <code>-O2</code>, fcf-protection=none</td>
</tr>
</tbody>
</table>

### Operating Systems:

<table>
<thead>
<tr>
<th>Version</th>
<th>Notable Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>tinyconfig, readable asm, ...</td>
</tr>
<tr>
<td>BSD</td>
<td>GENERIC*</td>
</tr>
<tr>
<td>StuBs</td>
<td>-</td>
</tr>
<tr>
<td>RuStuBs</td>
<td>-</td>
</tr>
</tbody>
</table>
Error Kinds

Interrupt Latency in Operating-System Kernels
Interrupt Latency in Operating-System Kernels
Validation

- Not every cli is made equal
Validation

- Not every cli is made equal
  - Startup code
- Not every CLI is made equal
  - Startup code
  - Scheduling
Not every CLI is made equal
- Startup code
- Scheduling

Various control flow paths per CLI
Not every cli is made equal
- Startup code
- Scheduling
- Various control flow paths per cli

→ Validate results using dynamic analysis
Validation

- Not every CLI is made equal
  - Startup code
  - Scheduling
- Various control flow paths per CLI
  → Validate results using dynamic analysis
  → QEMU Plugin to count interrupts
Validation

- Not every cli is made equal
  - Startup code
  - Scheduling
- Various control flow paths per cli
  - Validate results using dynamic analysis
  - QEMU Plugin to count interrupts
Discussion
Discussion

- Static analysis can provide reliable guarantees
Static analysis can provide reliable guarantees
High flexibility due to source-code independence
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
  - Support for additional architectures (ARM, RISC-V, ...)

Questions?
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
  - Support for additional architectures (ARM, RISC-V, ...)
  - Incorporate more external knowledge during analysis

Questions?
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
  - Support for additional architectures (ARM, RISC-V, ...)
  - Incorporate more external knowledge during analysis
  - Verify applicability of existing WCET algorithms
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
  - Support for additional architectures (ARM, RISC-V, ...)
  - Incorporate more external knowledge during analysis
  - Verify applicability of existing WCET algorithms

Can we provide safe guarantees on how long interrupts can be delayed?
Discussion

- Static analysis can provide reliable guarantees
- High flexibility due to source-code independence
- Many limitations still need to be overcome
  - Support for additional architectures (ARM, RISC-V, ...)
  - Incorporate more external knowledge during analysis
  - Verify applicability of existing WCET algorithms

Can we provide safe guarantees on how long interrupts can be delayed?

Questions?
Appendix

Control Flow Deviations

function f

```
cmp rdx, rdx
je 0x100B

add rbx, 32
jmp 0x9000

mov rdi, rdx
call g
sub rax, 64
ret
```
Appendix

Static Analysis

```c
int measure(Sensor* s) {
    int total = 0, limit;
    if (s->kind == 0x0) {
        limit = 16;
    } else {
        limit = 32;
    }
    for(int i = 0; i < limit; ++i) {
        total += sense(s);
    }
    return total;
}
```