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

Chair in Distributed Systems
and Operating Systems

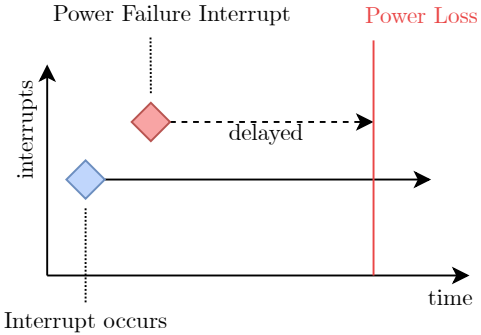
Friedrich-Alexander-Universität
Technische Fakultät

Motivation

Many systems rely on timely processing of interrupts

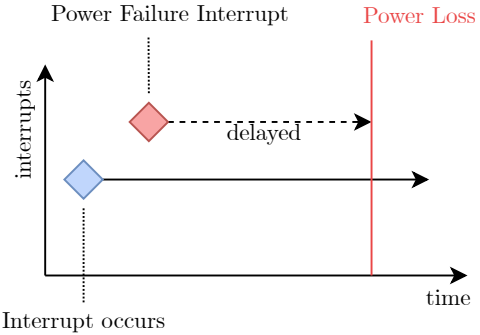
Motivation

Many systems rely on timely processing of interrupts



Motivation

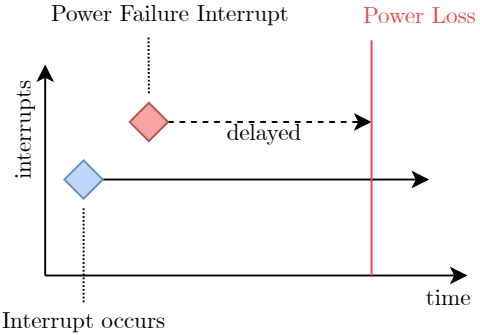
Many systems rely on timely processing of interrupts



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

Motivation

Many systems rely on timely processing of interrupts



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

$\Rightarrow$ Blocking time analysis on assembly level

Table of Contents

Interrupt Analysis

Kernel Disassembly

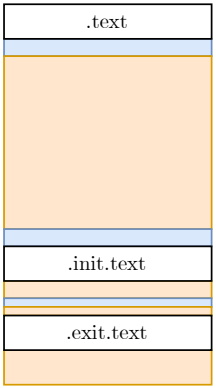
- Operating systems support numerous architectures

Kernel Disassembly

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

Kernel Disassembly

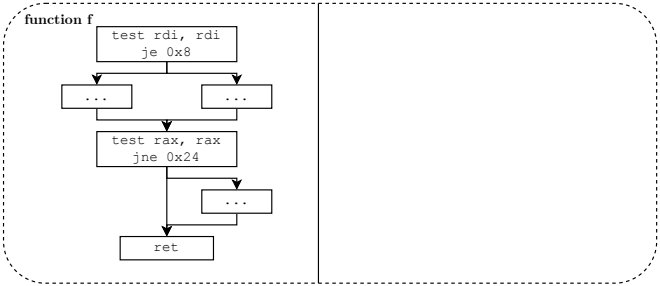
- Operating systems support numerous architectures
- One binary can be composed of multiple kinds




```
# 16/32 Bit
0x1000000: <startup>:
    mov ax, bx
    ...
# 64 Bit
0x1001000: <kernel_main>:
    sub rsp, 64
    ...
```

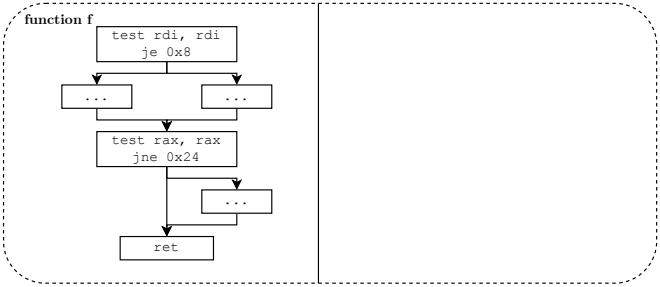
Control Flow

- Control Flow Graphs



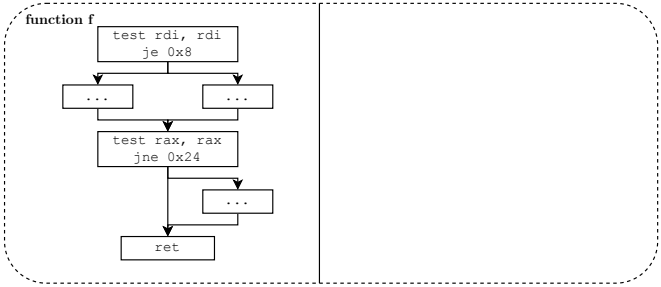
Control Flow

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



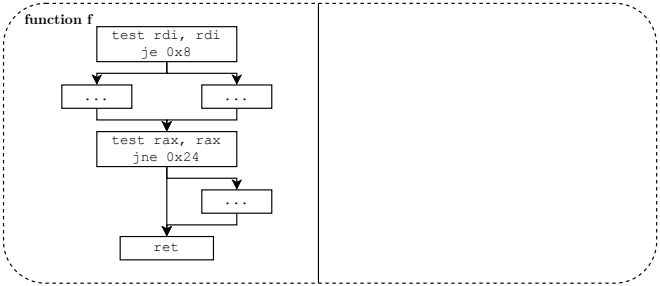
Control Flow

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



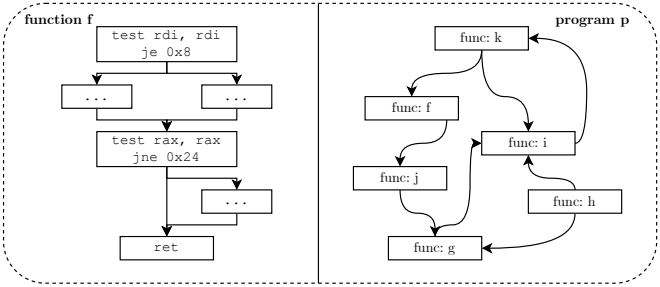
Control Flow

- 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



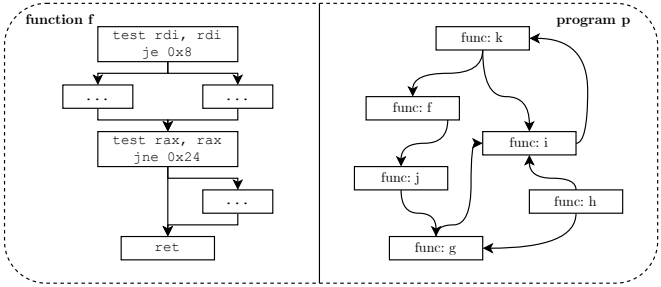
Control Flow

- 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



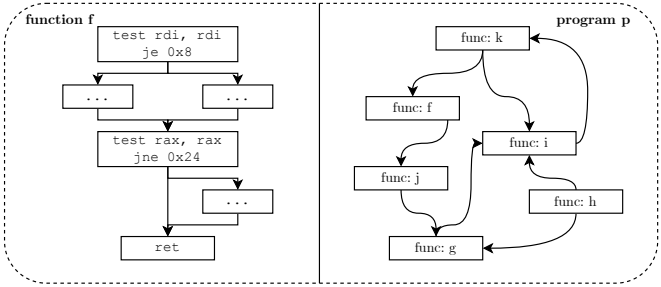
Control Flow

- 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

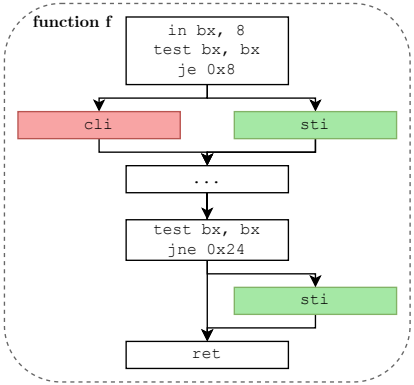


Interrupt Knowledge

Constant interrupt knowledge is caused by {`sti`: on, `cli`: off, `popf`: ?}.

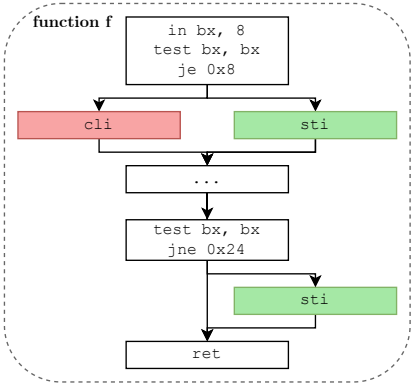
Interrupt Knowledge

Constant interrupt knowledge is caused by {`sti`: on, `cli`: off, `popf`: ?}.



Interrupt Knowledge

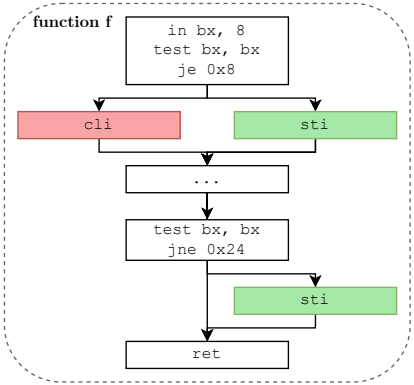
Constant interrupt knowledge is caused by {`sti`: on, `cli`: off, `popf`: ?}.



- Unaffected by surrounding blocks

Interrupt Knowledge

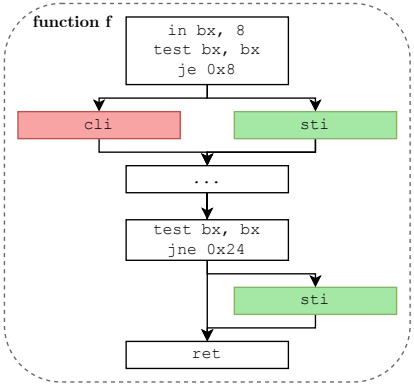
Constant interrupt knowledge is caused by {`sti`: on, `cli`: off, `popf`: ?}.



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

Interrupt Knowledge

Constant 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

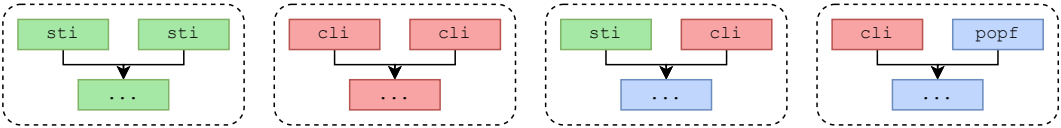
- Requires knowledge present in a function's entry block
- Each block is dependent on its predecessors
- Contradicting interrupt states lead to worst-case (unknown)

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



Determining Interrupt Latency

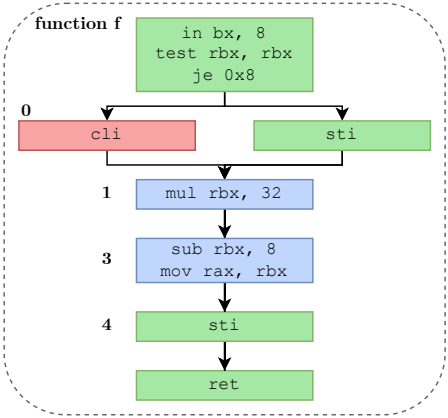
Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

Determining Interrupt Latency

Interrupt Latency

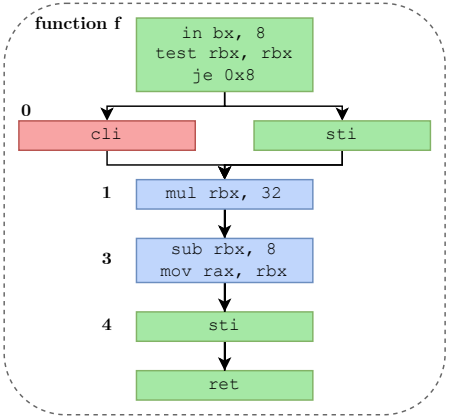
Longest possible instruction count until interrupts are enabled again.



Determining Interrupt Latency

Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

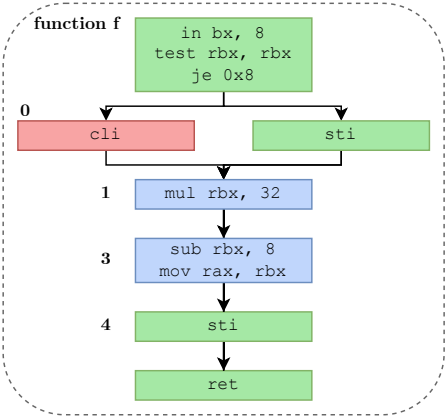


- DFS to find longest path

Determining Interrupt Latency

Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

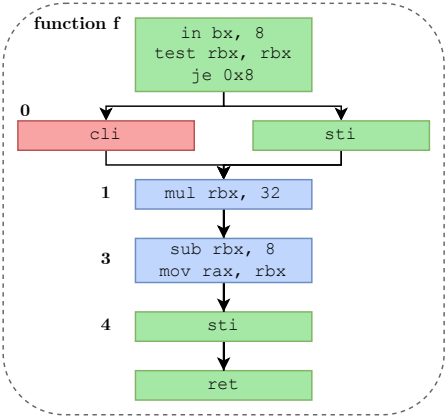


- DFS to find longest path
- Unknown and disabled states

Determining Interrupt Latency

Interrupt Latency

Longest possible instruction count until interrupts are enabled again.

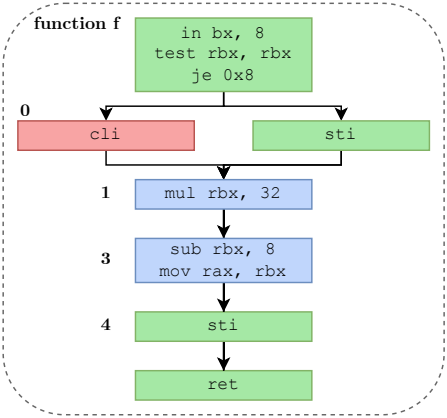


- DFS to find longest path
- Unknown and disabled states
- Weigh by instruction kind

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

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

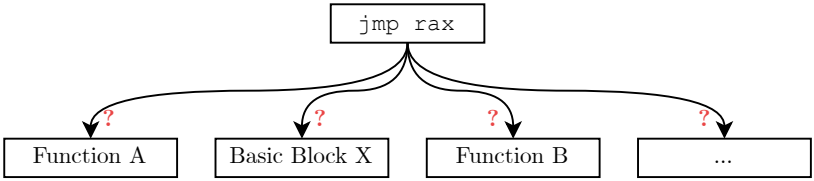
- Not all control flow instructions target a static location
- `call rax`/`jmp rax` depend on run-time values
- Execution can continue nearly anywhere

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, …

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, ...



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++)

```
dispatch:
  mov rbx, rdi
  mov rax, rbx
  call rax
```

```
dispatch:
  mov rbx, rdi
  mov rax, rbx
  call 0x21610
```

```
dispatch:
  mov rbx, rdi
  mov rax, rbx
  call 0xABA80
```

Inter-Function Analysis

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

Inter-Function Analysis

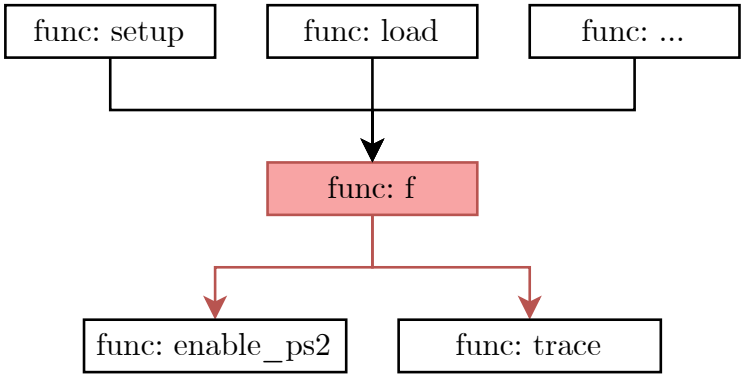
What happens when interrupts are still disabled in the exit block?

$\Rightarrow$ Follow the call tree upwards

Inter-Function Analysis

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

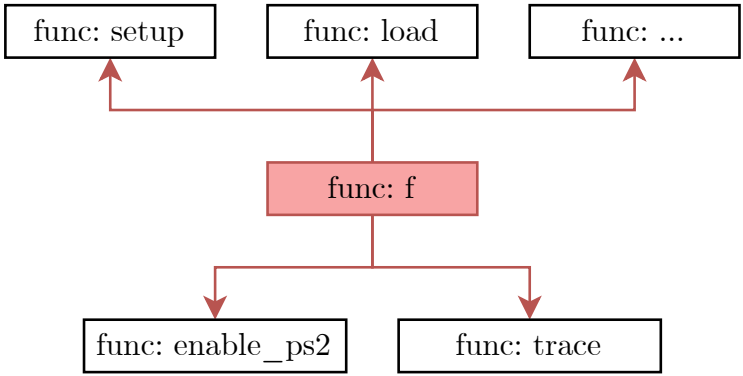
⇒ Follow the call tree upwards



Inter-Function Analysis

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

⇒ Follow the call tree upwards



Loops

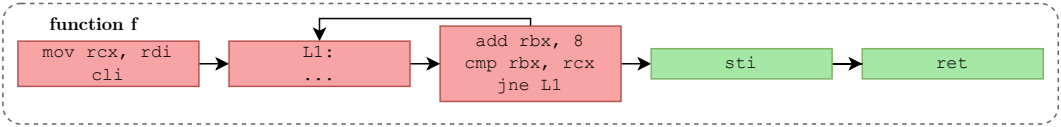
- Terminate early when encountering loops with disabled interrupts

Loops

- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches

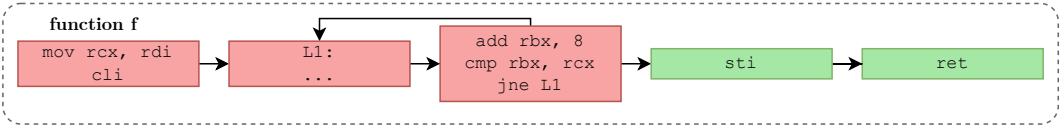
Loops

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



Loops

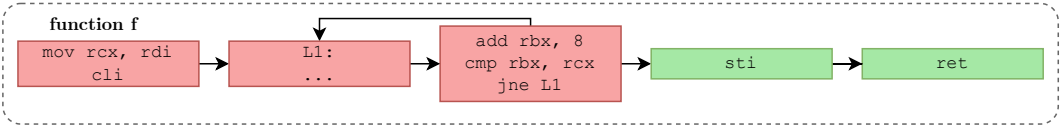
- Terminate early when encountering loops with disabled interrupts
- Inherent limitation present in static approaches
- Dataflow analysis could provide loop bound approximations



→ 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

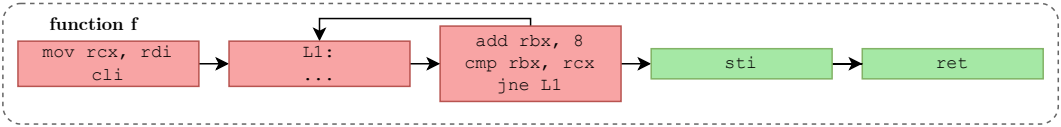


→ Repetitions limited by `rdi` (function parameter)

→ 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:

	Version	Flags
clang	17.0.6	{-Os, -O2}, fcf-protection=none
gcc	13.2.1	{-Os, -O2}, fcf-protection=none

Analysis Environment

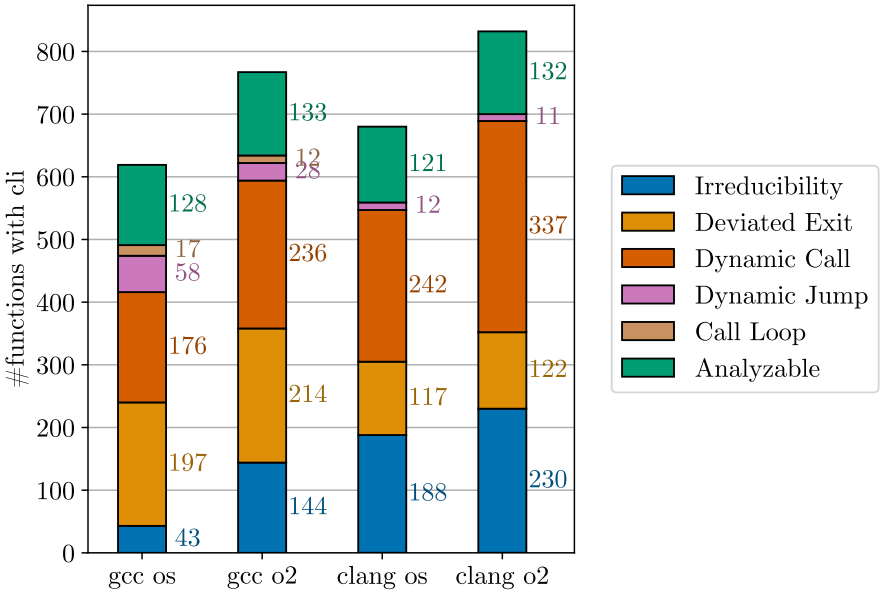
Compilers:

	Version	Flags
clang	17.0.6	{-Os, -O2}, fcf-protection=none
gcc	13.2.1	{-Os, -O2}, fcf-protection=none

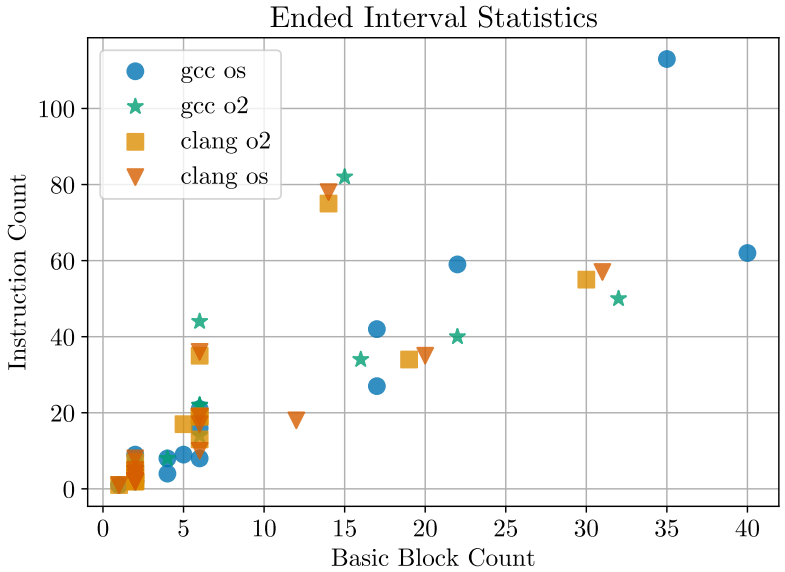
Operating Systems:

	Version	Notable Options
Linux	6.5.7	tinyconfig, readable asm, ...
BSD	13.1	GENERIC*
StuBs	-	-
RuStuBs	-	-

Error Kinds


#functions with cli
0
100
200
300
400
500
600
700
800
gcc os
gcc o2
clang os
clang o2
128
17
58
176
197
43
133
12
28
236
214
144
121
12
242
117
188
132
11
337
122
230
Irreducibility
Deviated Exit
Dynamic Call
Dynamic Jump
Call Loop
Analyzable

Intervals


Ended Interval Statistics
gcc os
gcc o2
clang o2
clang os
Instruction Count
Basic Block Count
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40

Validation

- Not every `cli` is made equal

Validation

- Not every `cli` is made equal
  - Startup code

Validation

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

Validation

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

Validation

- 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

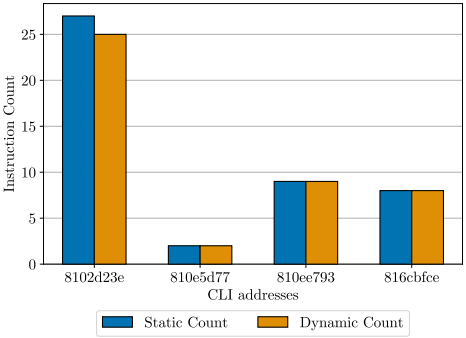
→ QEMU Plugin to count interrupts

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

Discussion

- 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 to overcome

Discussion

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

Discussion

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

Discussion

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

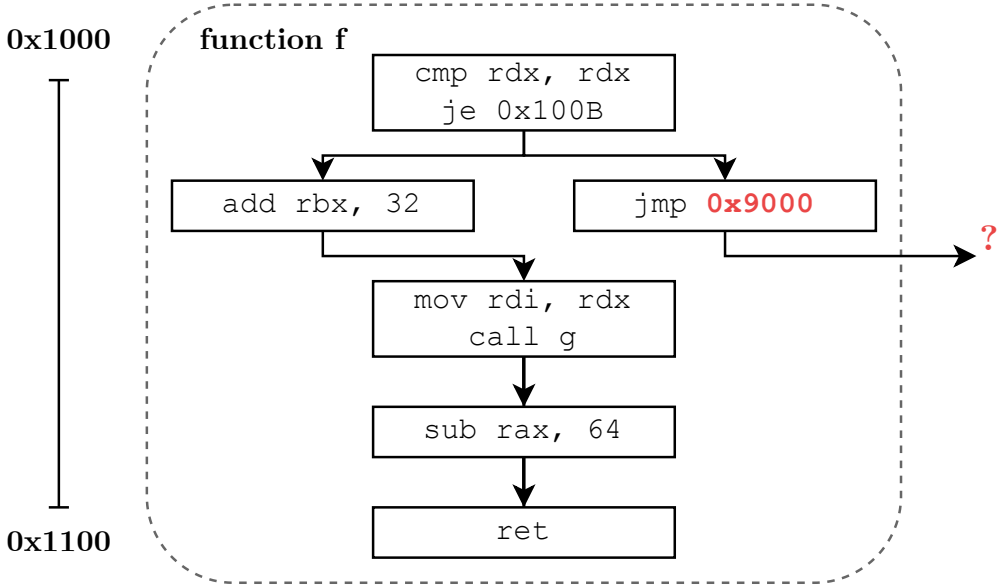
Can we provide safe guarantees on how long interrupts can be delayed?

Discussion

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


0x1000
function f
cmp rdx, rdx
je 0x100B
add rbx, 32
jmp 0x9000
?
mov rdi, rdx
call g
sub rax, 64
ret
0x1100

```
1  int measure(Sensor* s) {
2    int total = 0, limit;
3    if (s->kind == 0x0) {
4      limit = 16;
5    } else {
6      limit = 32;
7    }
8    for(int i = 0; i < limit; ++i) {
9      total += sense(s);
10   }
11   return total;
12 }
```

statically obtainable
run-time dependency