FFXE

Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries

Ryan Tsang, Asmita, Doreen Joseph, Soheil Salehi, Prasant Mohapatra, Houman Homayoun

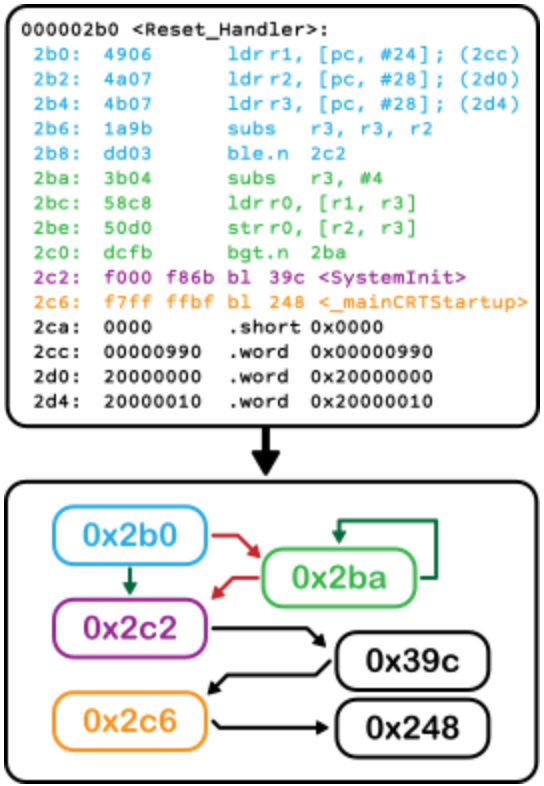
UC DAVIS
ELECTRICAL AND COMPUTER ENGINEERING

This work is funded in part by Robert N. Noyce Trust and by the National Science Foundation under grant Computing Research Association for CIFellows Project 2030859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Motivation

- Auditing Embedded Systems Firmware
  - Software/Firmware supply chain

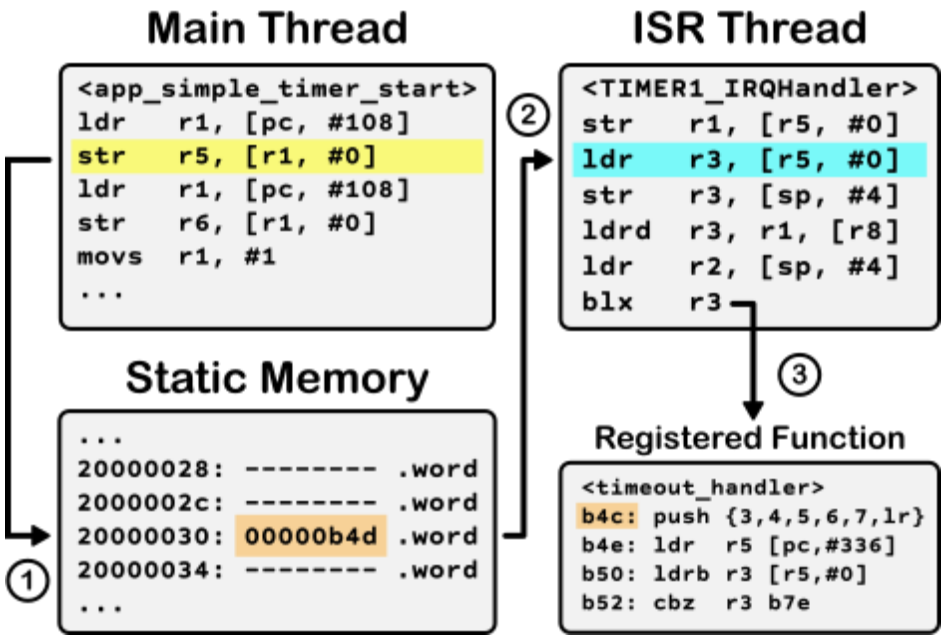
- Source code unavailable → Binary analysis
  - Human-in-the-loop process
  - Reliance on automated program analysis techniques
  - Quality depends on **Control Flow Graph** (CFG) **Recovery**



Motivation

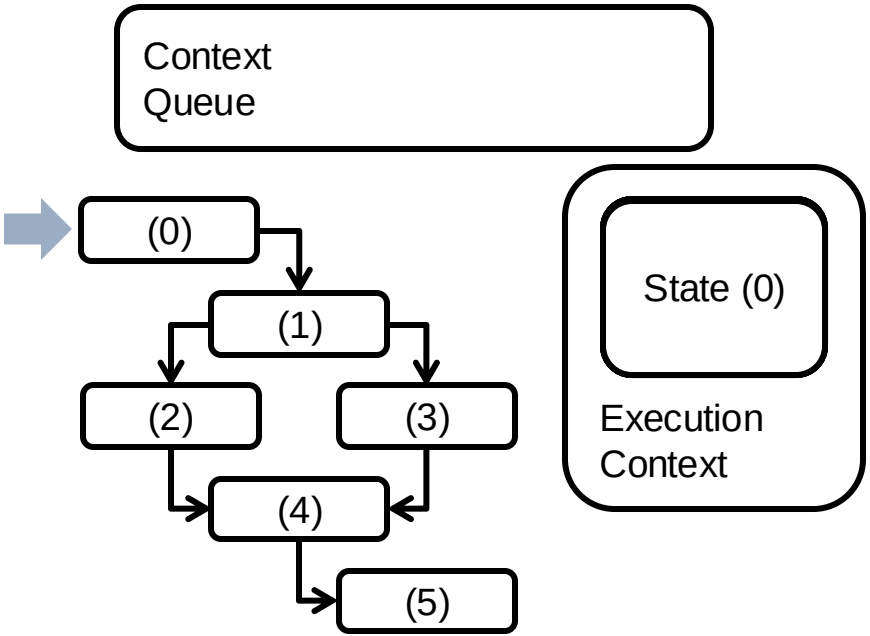
- Registered Interrupt Handlers
  - Common pattern in embedded systems firmware
  - Branch targets computed and registered in different thread

→ Asynchronous memory accesses
  - Difficult to resolve indirection
  - Runtime information useful



Background: Forced Execution

- Xu et al. 2009
- Concretely execute all conditional branch paths in DFS manner



L. Xu, F. Sun, and Z. Su, “Constructing Precise Control Flow Graphs from Binaries,” CSE-2009-27, 2009. [Online]. Available: https://citeseerx.ist.psu.edu/pdf/8a80f0d173ec7420478e4b96a8264e21e0dafac0

Problem

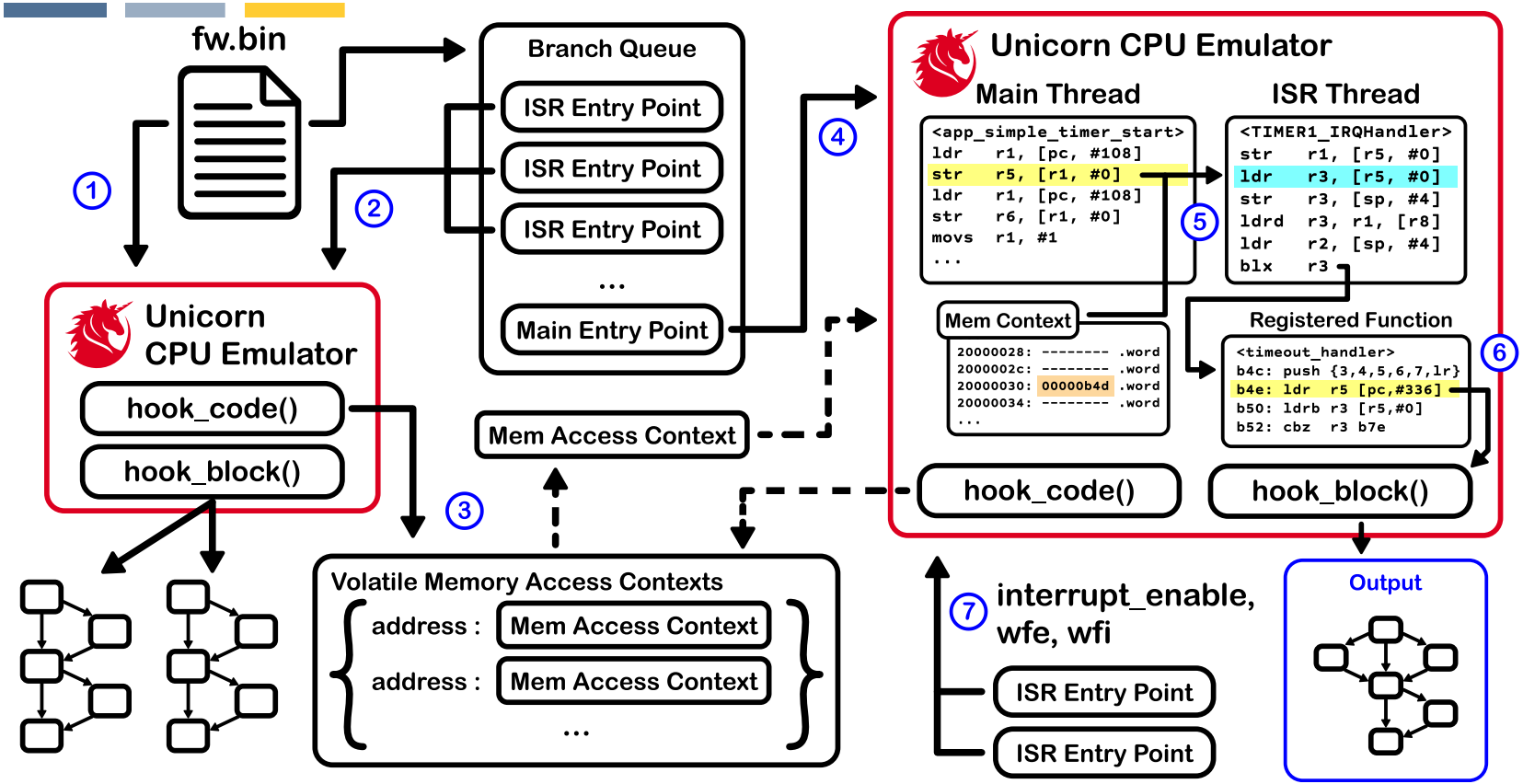
Forced Execution Assumption 2

"The target of an indirect branch is completely determined by a control flow path to this indirect branch and is independent of intermediate program states."

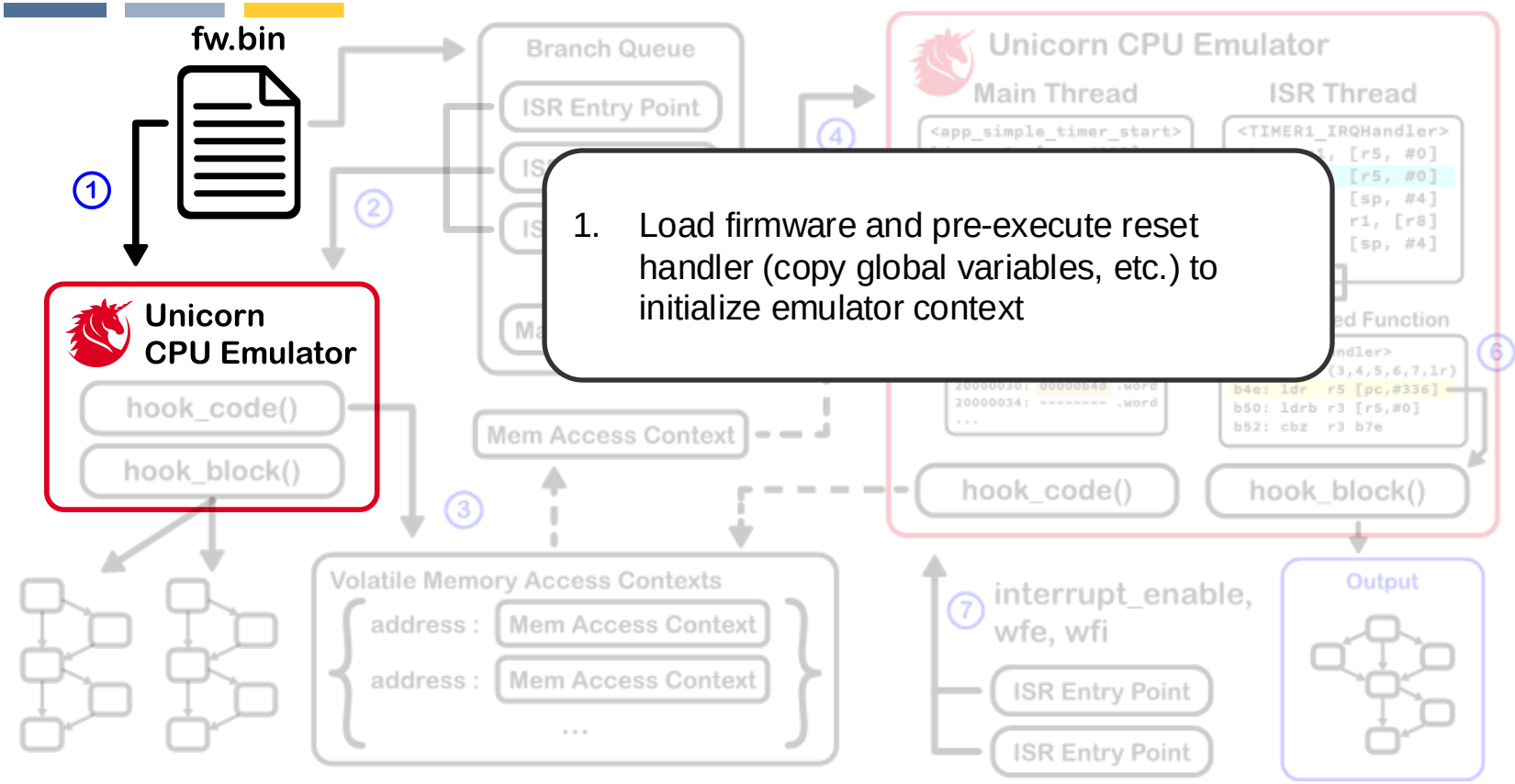
- Interrupt handler registration violates original assumptions
  - Branch target address is determined and written asynchronously
  - Forced execution will result in incorrect jumps

- **Forced Firmware Execution Engine (FFXE)**
  - Account for asynchronous memory accesses to resolve indirect branches
  - Resumes forced execution from volatile memory reads

FFXE

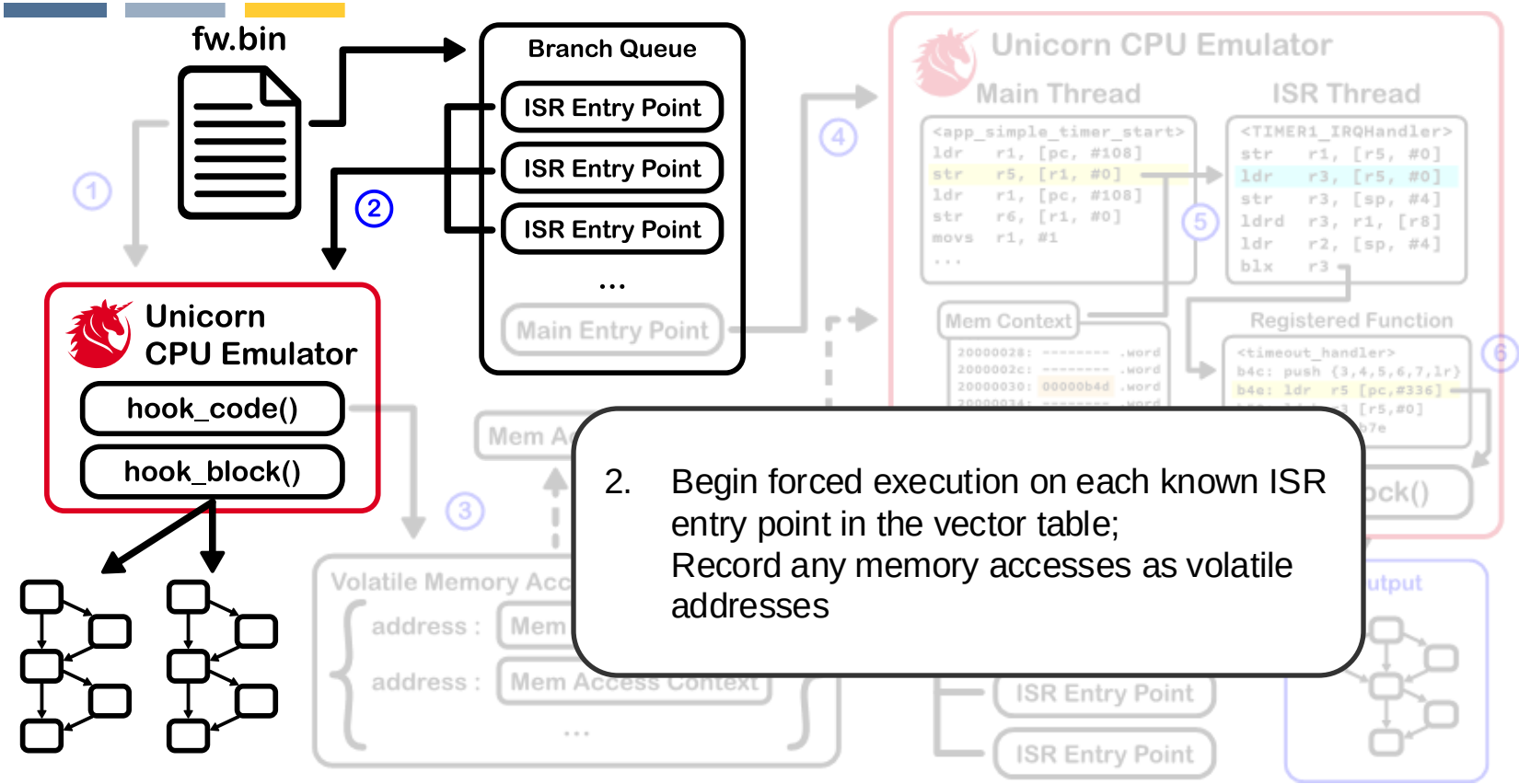

fw.bin
Branch Queue
ISR Entry Point
ISR Entry Point
ISR Entry Point
...
Main Entry Point
Unicorn CPU Emulator
hook_code()
hook_block()
Mem Access Context
Volatile Memory Access Contexts
address : Mem Access Context
address : Mem Access Context
...
Unicorn CPU Emulator
Main Thread
ISR Thread
<app_simple_timer_start>
ldr r1, [pc, #108]
str r5, [r1, #0]
ldr r1, [pc, #108]
str r6, [r1, #0]
movs r1, #1
...
<TIMER1_IRQHandler>
str r1, [r5, #0]
ldr r3, [r5, #0]
str r3, [sp, #4]
ldrd r3, r1, [r8]
ldr r2, [sp, #4]
blx r3
Mem Context
20000028: -------- .word
2000002c: -------- .word
20000030: 00000b4d .word
20000034: -------- .word
...
Registered Function
<timeout_handler>
b4c: push {3,4,5,6,7,lr}
b4e: ldr r5 [pc,#336]
b50: ldrb r3 [r5,#0]
b52: cbz r3 b7e
hook_code()
hook_block()
interrupt_enable, wfe, wfi
ISR Entry Point
ISR Entry Point
Output
1
2
3
4
5
6
7

Method

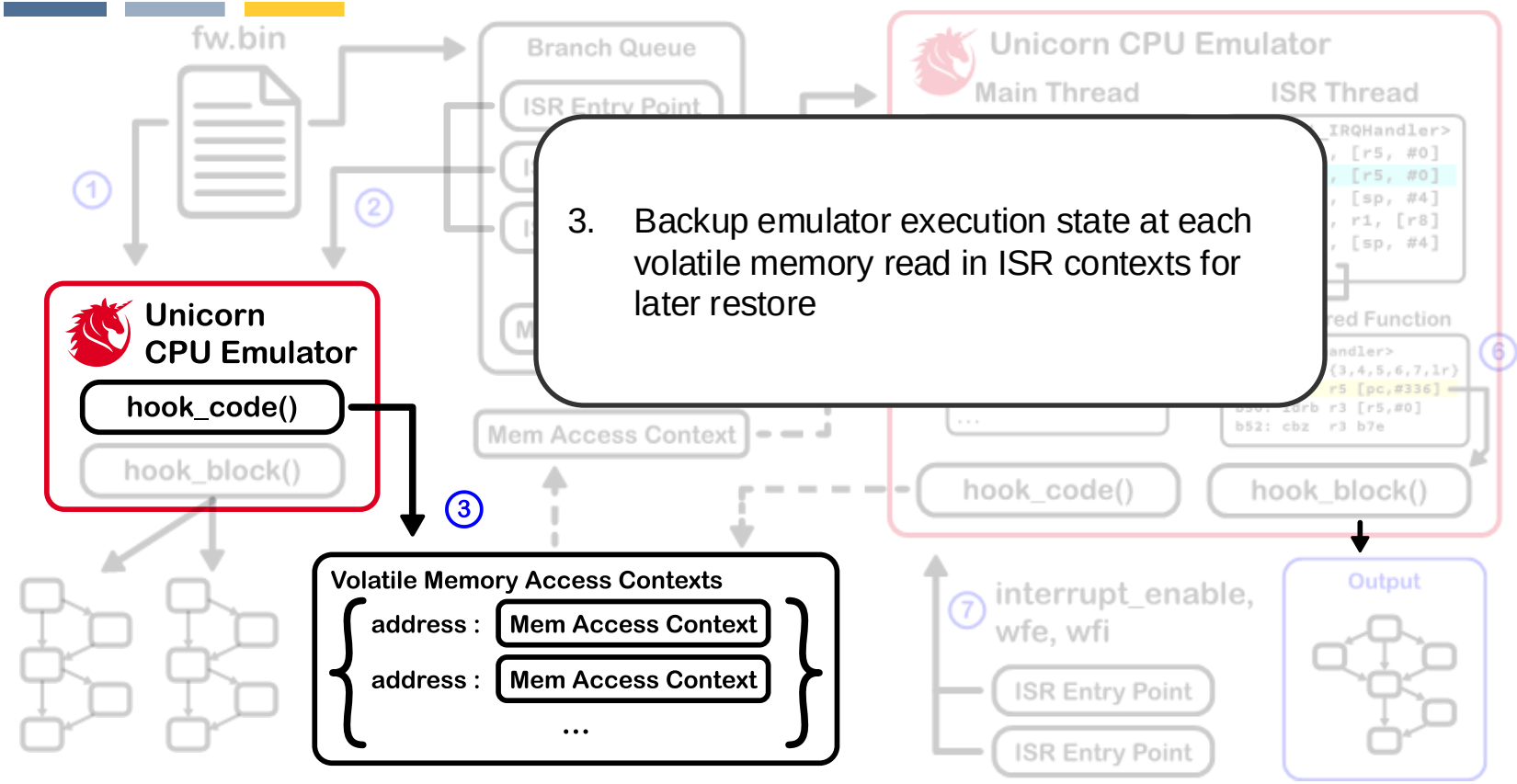


1. Load firmware and pre-execute reset handler (copy global variables, etc.) to initialize emulator context

Method

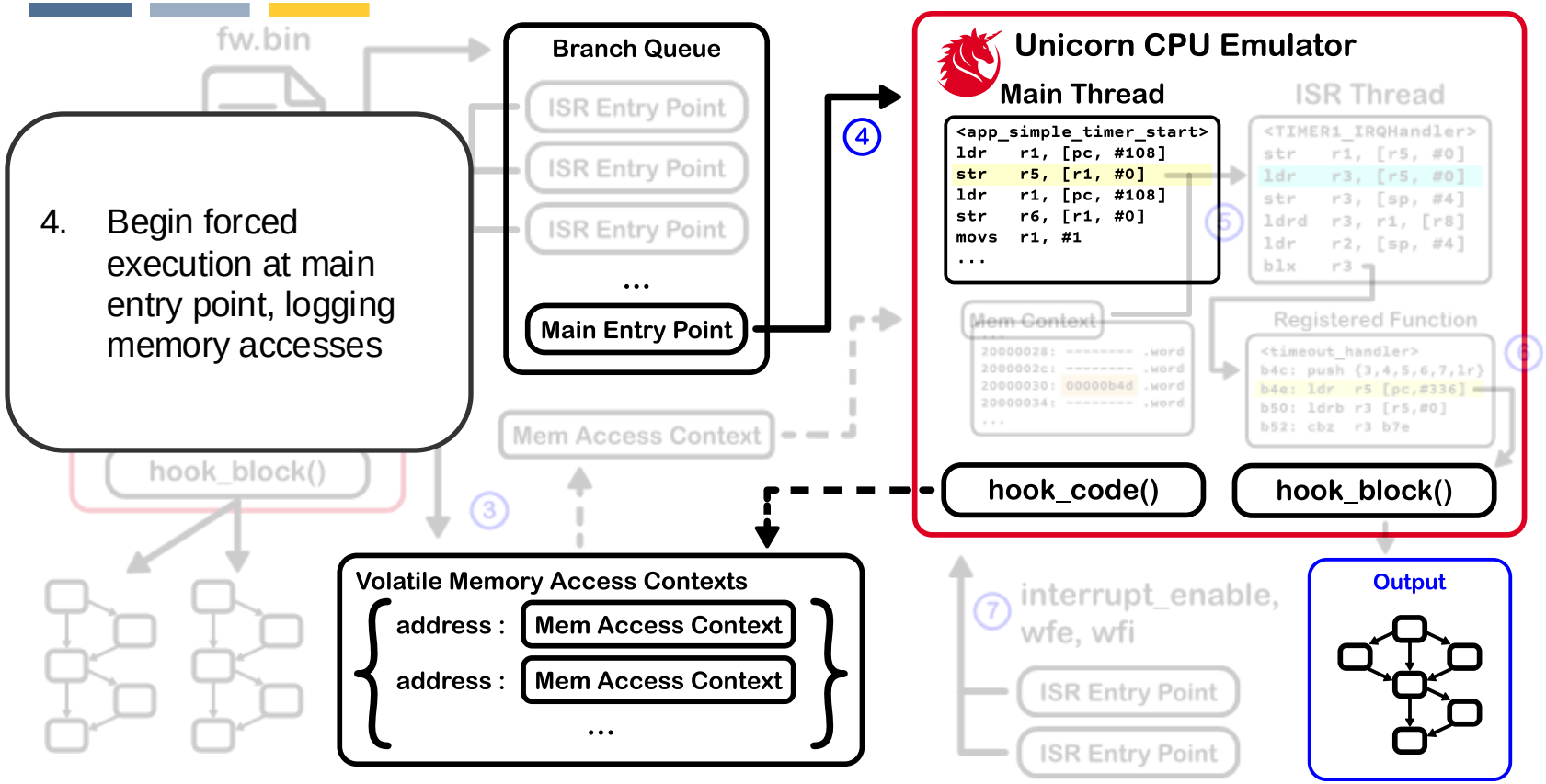

fw.bin
Branch Queue
ISR Entry Point
ISR Entry Point
ISR Entry Point
...
Main Entry Point
Unicorn
CPU Emulator
hook_code()
hook_block()
Unicorn CPU Emulator
Main Thread
ISR Thread
Mem Context
Registered Function
Volatile Memory Acc
address :
address :
Mem Access Context
ISR Entry Point
ISR Entry Point
2. Begin forced execution on each known ISR entry point in the vector table;
Record any memory accesses as volatile addresses

Method



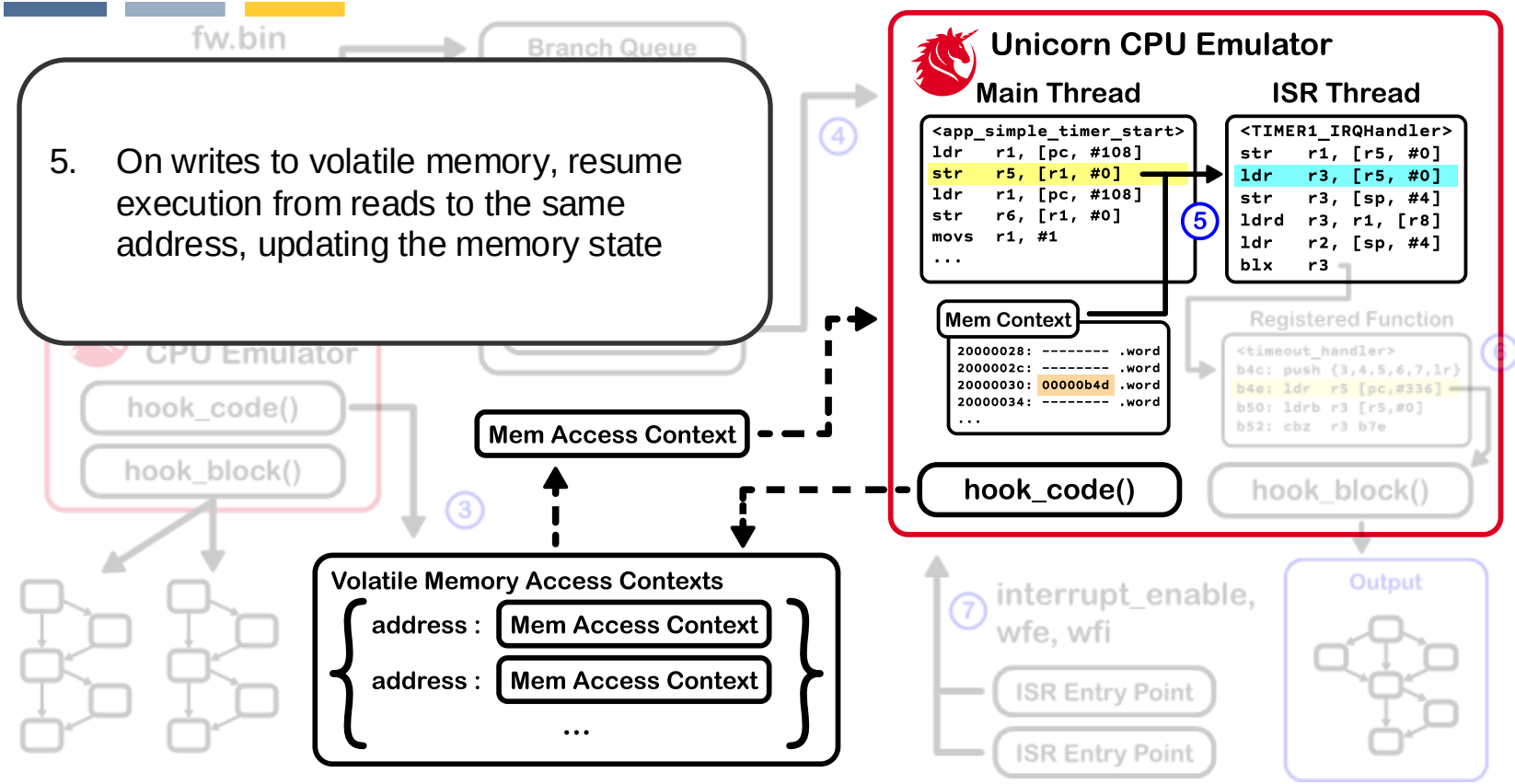
3. Backup emulator execution state at each volatile memory read in ISR contexts for later restore

Method


fw.bin
4. Begin forced execution at main entry point, logging memory accesses
Branch Queue
ISR Entry Point
ISR Entry Point
ISR Entry Point
...
Main Entry Point
4
Unicorn CPU Emulator
Main Thread
<app_simple_timer_start>
ldr r1, [pc, #108]
str r5, [r1, #0]
ldr r1, [pc, #108]
str r6, [r1, #0]
movs r1, #1
...
ISR Thread
<TIMER1_IRQHandler>
str r1, [r5, #0]
ldr r3, [r5, #0]
str r3, [sp, #4]
ldrd r3, r1, [r8]
ldr r2, [sp, #4]
blx r3
5
Mem Context
Registered Function
<timeout_handler>
b4c: push {3,4,5,6,7,lr}
b4e: ldr r5 [pc,#336]
b50: ldrb r3 [r5,#0]
b52: cbz r3 b7e
6
hook_code()
hook_block()
Mem Access Context
hook_block()
3
Volatile Memory Access Contexts
address : Mem Access Context
address : Mem Access Context
...
7
interrupt_enable, wfe, wfi
ISR Entry Point
ISR Entry Point
Output

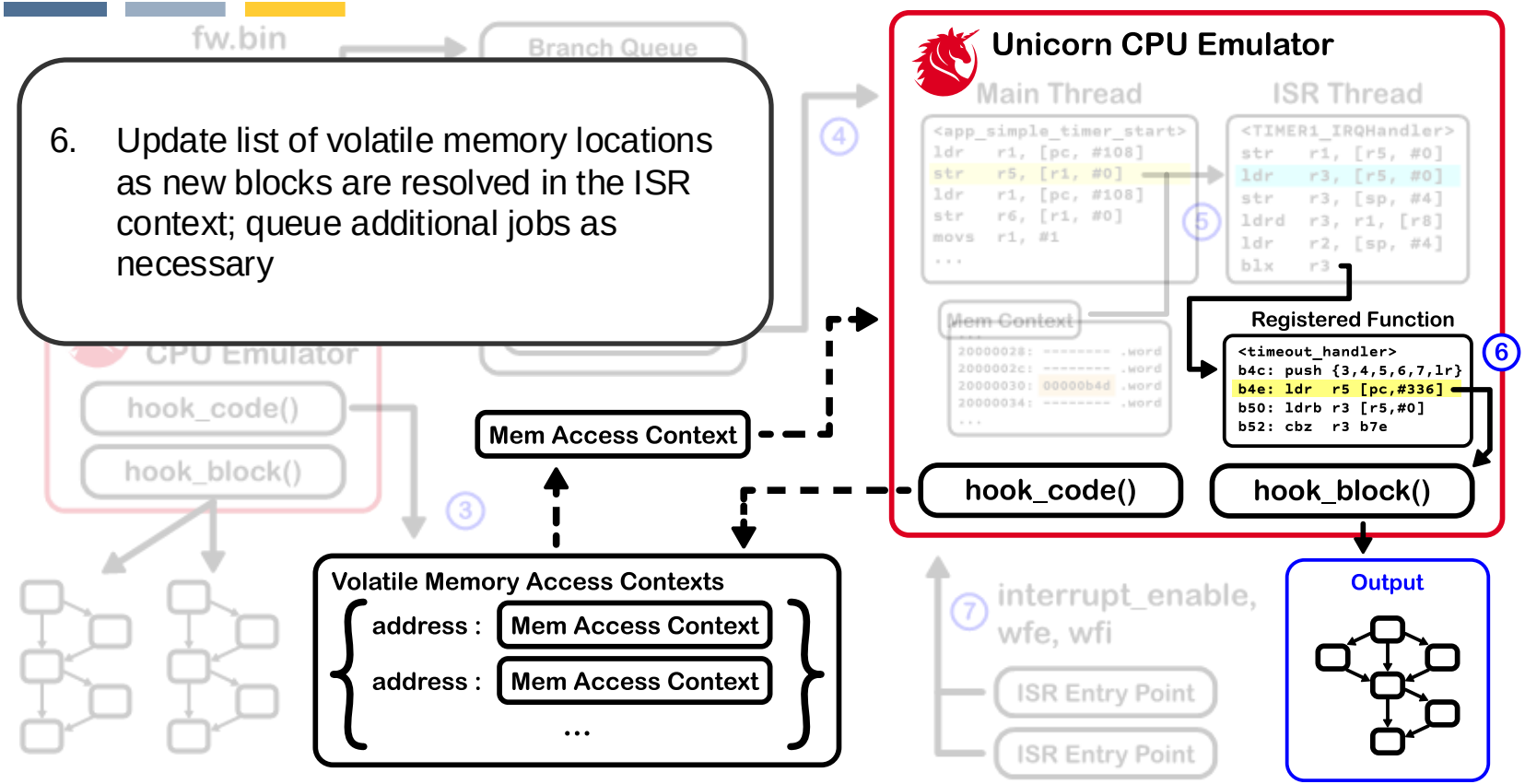
Method

5. On writes to volatile memory, resume execution from reads to the same address, updating the memory state

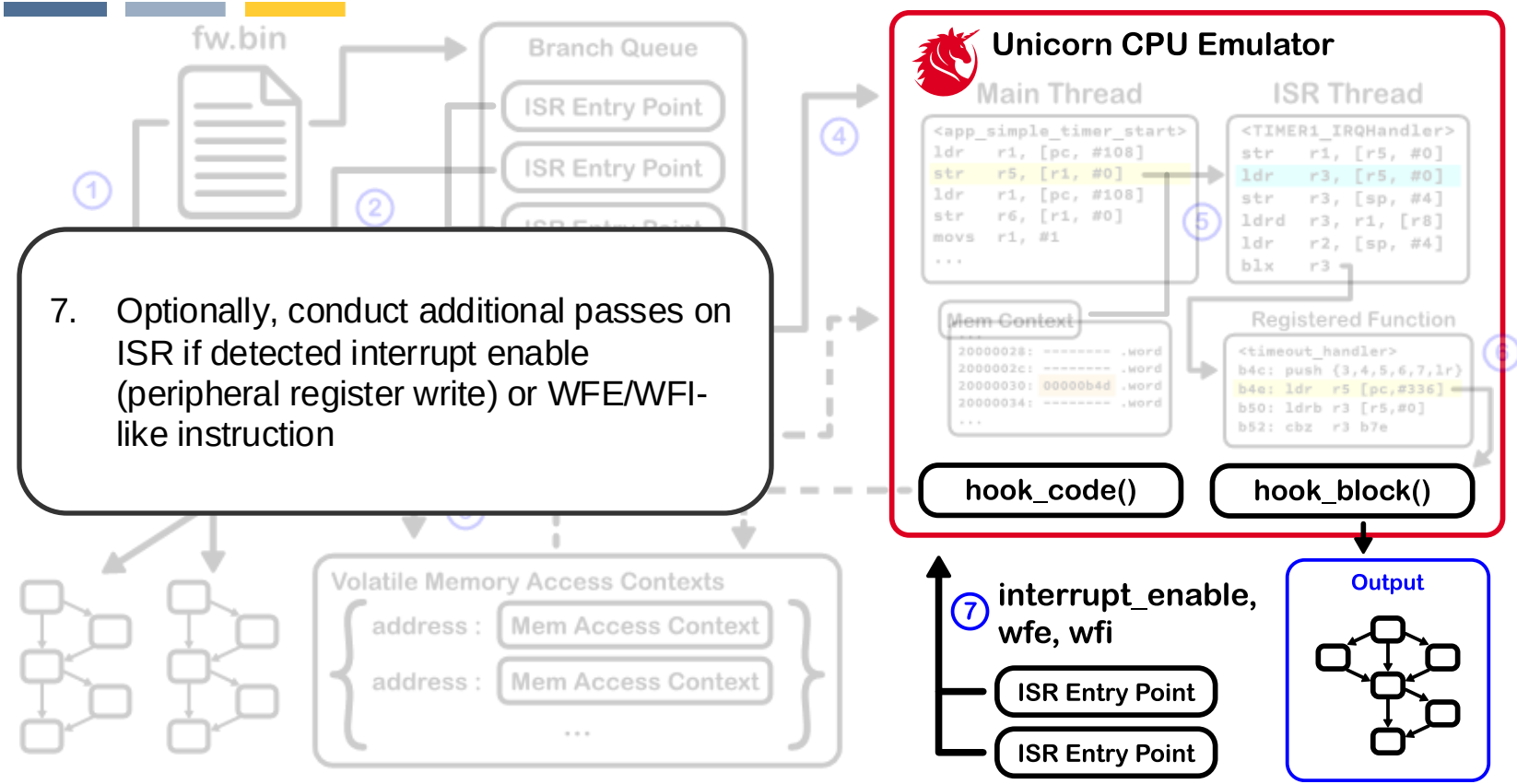


Method

6. Update list of volatile memory locations as new blocks are resolved in the ISR context; queue additional jobs as necessary



Method


fw.bin
Branch Queue
ISR Entry Point
ISR Entry Point
Unicorn CPU Emulator
Main Thread
ISR Thread
<app_simple_timer_start>
ldr r1, [pc, #108]
str r5, [r1, #0]
ldr r1, [pc, #108]
str r6, [r1, #0]
movs r1, #1
<TIMER1_IRQHandler>
str r1, [r5, #0]
ldr r3, [r5, #0]
str r3, [sp, #4]
ldrd r3, r1, [r8]
ldr r2, [sp, #4]
blx r3
Mem Context
Registered Function
<timeout_handler>
b4c: push {3,4,5,6,7,lr}
b4e: ldr r5 [pc,#336]
b50: ldrb r3 [r5,#0]
b52: cbz r3 b7e
hook_code()
hook_block()
7. Optionally, conduct additional passes on ISR if detected interrupt enable (peripheral register write) or WFE/WFI-like instruction
Volatile Memory Access Contexts
address : Mem Access Context
address : Mem Access Context
7 interrupt_enable, wfe, wfi
ISR Entry Point
ISR Entry Point
Output

Evaluation: Callback Resolution

- FFXE fully recovers callback edges to all handlers in our test set
  - Other tools typically fail to resolve these edges
- Nordic nRF5 SDK
  - 8 samples x 4 optimization levels = 32 binaries
  - 24 manually identified callback functions

Table 1: Registered Function Resolution Comparison of CFG Recovery Methods.

Firmware		angr_emu	angr_fast	**ffxe**	fxe	ghidra
gpiote	-O0	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O1	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
	-O2	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
	-O3	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
i2s	-O0	0/0/4	0/4/4	**9/4/4**	0/0/4	0/4/4
	-O1	0/0/4	0/4/4	**10/4/4**	0/0/4	0/3/4
	-O2	0/0/4	0/4/4	**12/4/4**	0/0/4	0/3/4
	-O3	0/0/4	0/4/4	**12/4/4**	0/0/4	0/3/4
saadc	-O0	0/0/2	0/2/2	**4/2/2**	0/0/2	0/2/2
	-O1	0/0/2	0/2/2	**4/2/2**	0/0/2	0/2/2
	-O2	0/0/2	1/2/2	**4/2/2**	0/0/2	0/2/2
	-O3	0/0/2	1/2/2	**4/2/2**	0/0/2	0/2/2
simple_timer	-O0	0/0/2	0/2/2	**2/2/2**	0/0/2	0/2/2
	-O1	0/0/2	0/2/2	**2/2/2**	0/0/2	0/1/2
	-O2	0/0/2	1/2/2	**2/2/2**	0/0/2	0/1/2
	-O3	0/0/2	1/2/2	**2/2/2**	0/0/2	0/1/2
spi	-O0	0/0/5	0/5/5	**7/5/5**	0/0/5	0/5/5
	-O1	0/0/5	0/5/5	**8/5/5**	0/0/5	0/4/5
	-O2	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
	-O3	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
timer	-O0	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O1	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O2	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O3	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
twi_sensor	-O0	0/0/5	0/5/5	**7/5/5**	0/0/5	0/5/5
	-O1	0/0/5	0/5/5	**8/5/5**	0/0/5	0/4/5
	-O2	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
	-O3	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
uart	-O0	1/1/4	0/4/4	**9/4/4**	0/0/4	0/4/4
	-O1	0/0/4	0/4/4	**10/4/4**	0/0/4	0/4/4
	-O2	0/0/4	2/4/4	**12/4/4**	0/0/4	0/2/4
	-O3	0/0/4	2/4/4	**12/4/4**	0/0/4	0/2/4

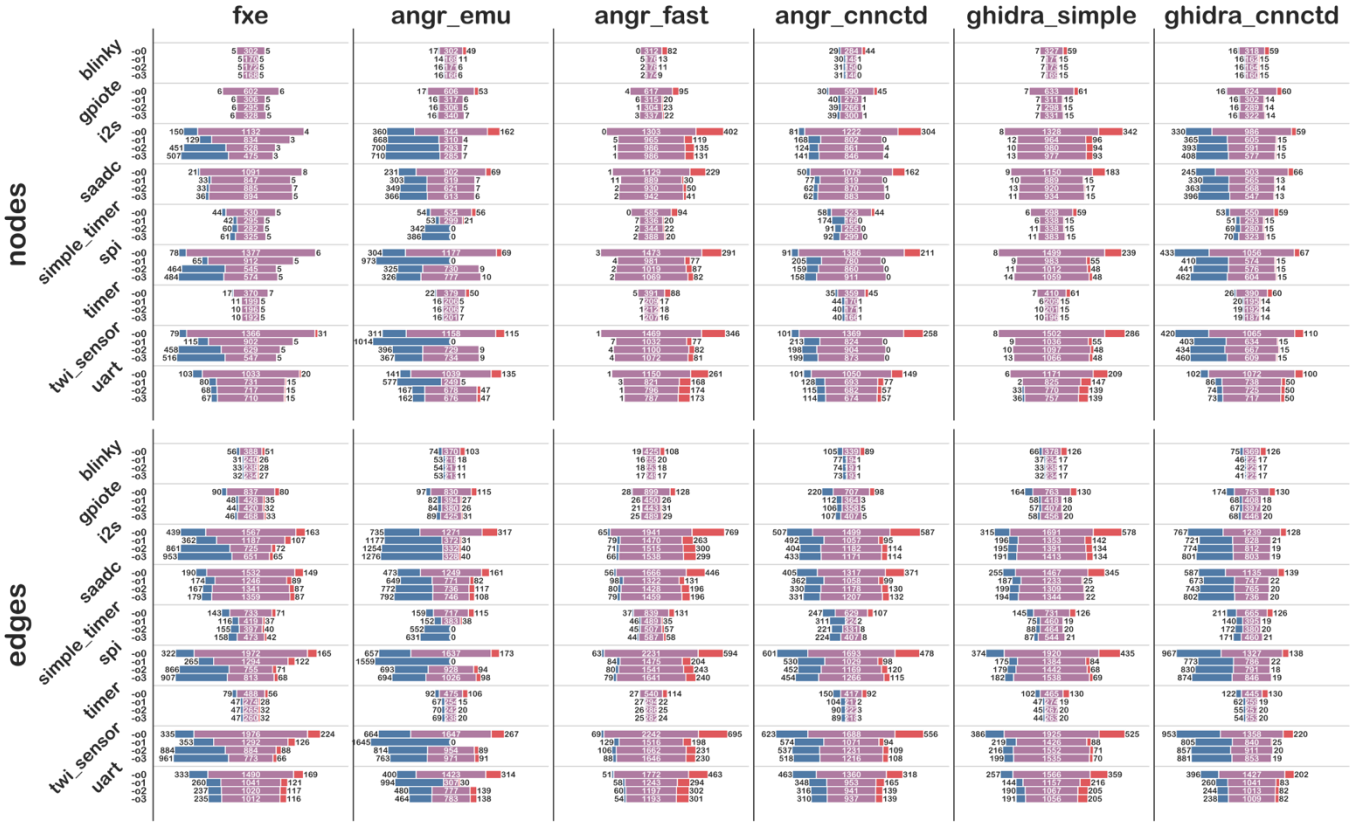
Table elements: (# of found edges to registered function)/(# of found registered function entry blocks)/(# of known registered functions).

Evaluation: Overall (Test Set)

FFXE-only (blue)

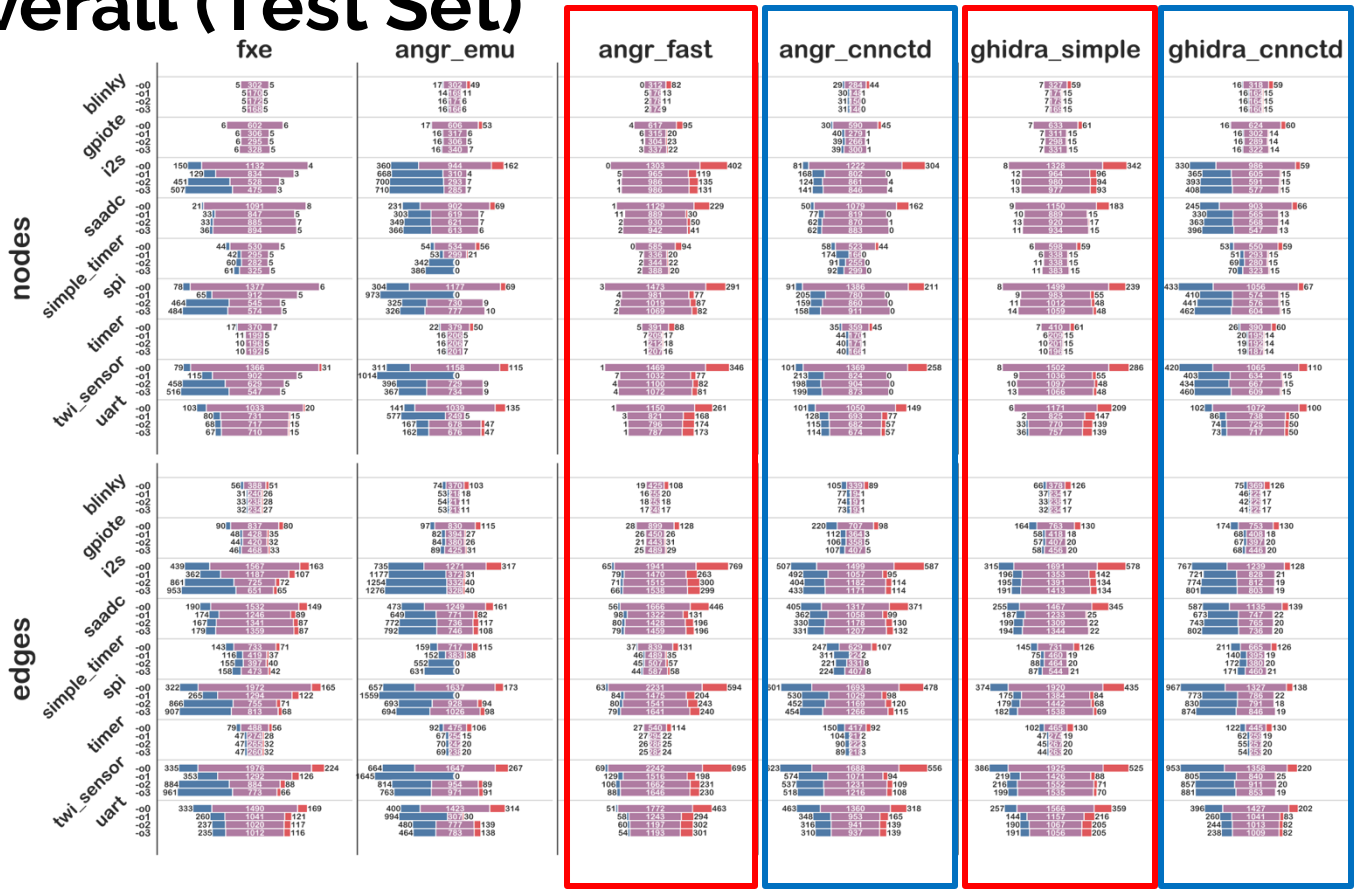
Other engine (red)

Overlap (purple)

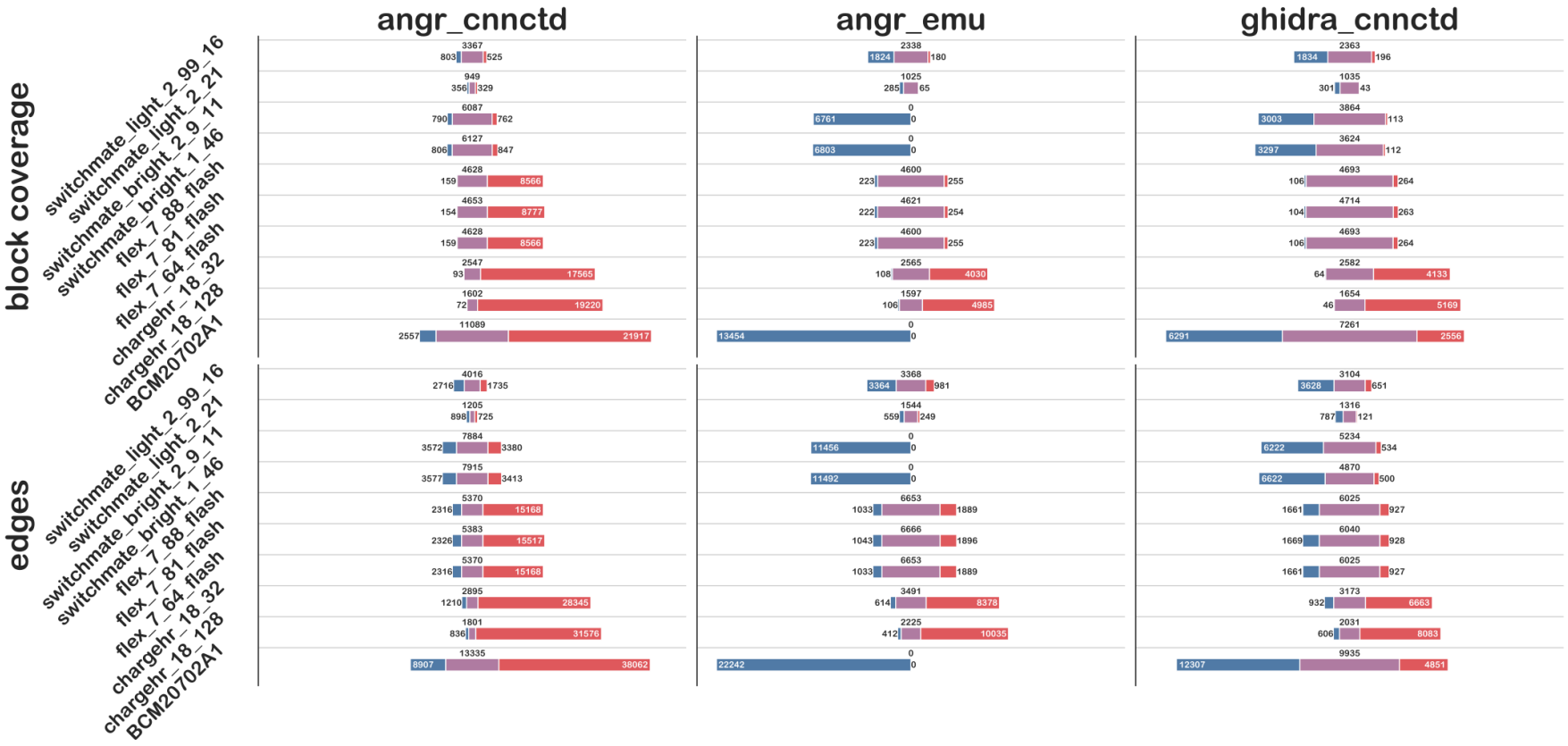


Evaluation: Overall (Test Set)

Complementary coverage for reachable graphs



Evaluation: Overall (Real-World)



Conclusion

- FFXE designed to resolve edges to interrupt callback functions
  - Common occurrence in bare-metal firmware SDKs
- Evaluated FFXE's ability to resolve these edges against existing tools
  - All other tools failed to locate such edges in our test set
- Demonstrated FFXE provides complementary coverage to existing tools
  - Ghidra, angr's static and dynamic methods, original FXE

GitHub: https://github.com/rchtsang/ffxe