µRAI: Securing Embedded Systems with Return Address Integrity

Naif Saleh Almakhdhub[1,4]

Abraham A. Clements[3]

Saurabh Bagchi[1]

Mathias Payer[2]

PURDUE UNIVERSITY [1]

EPFL [2]

Sandia National Laboratories [3]

جامعة الملك سعود King Saud University [4]

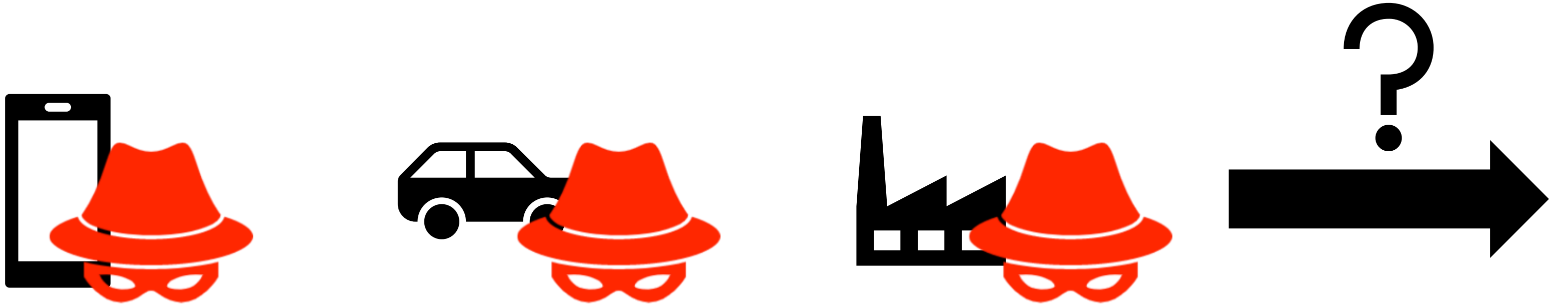
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND-XXXX

Current State of Security


SECURITYWEEK
INTERNET AND ENTERPRISE SECURITY NEWS, INSIGHTS & ANALYSIS

The Rise of ICS Malware: How Industrial Security Threats Are Becoming More Surgical [3]

By Barak Perelman on February 21, 2018



Target:
Embedded and IoT devices Running Microcontroller Systems (MCUS)

Attack:
Control-flow Hijacking

[1] https://www.wired.com/story/broadpwn-wi-fi-vulnerability-ios-android/
[2] https://keenlab.tencent.com/en/2020/01/02/exploiting-wifi-stack-on-tesla-model-s/
[3] https://www.securityweek.com/rise-ics-malware-how-industrial-security-threats-are-becoming-more-surgical

MCUS Challenges

Desktop		MCUS
✓ **Large virtual memory** (GBs)		✘ **Small physical memory** (MBs Flash, KBs RAM)
✓ **Basic defenses** (e.g., ASLR)		✘ **Basic defenses** (e.g., ASLR)
Larger code		✓ **Smaller code**
✓ **DEP**		✘ **DEP** (Disabled → Fixable)

Memory

Memory	Address
…	
Stack	0x08955555
…	
…	
…	
Heap	0x08
…	
…	
Data	0x08555111
…	
Code	0x08000000

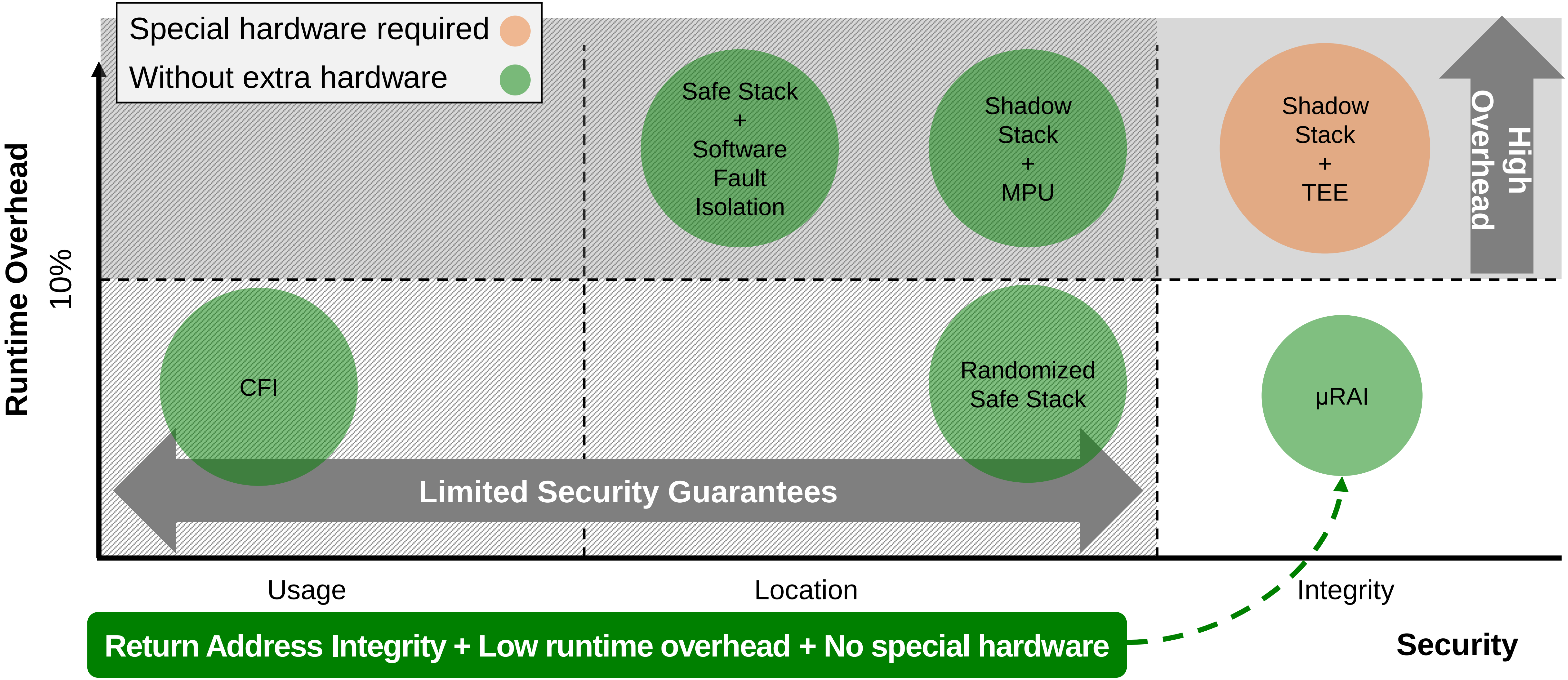
Flash (code)

Address	Flash
	…
	…
	…
	…
0x08000000	Code

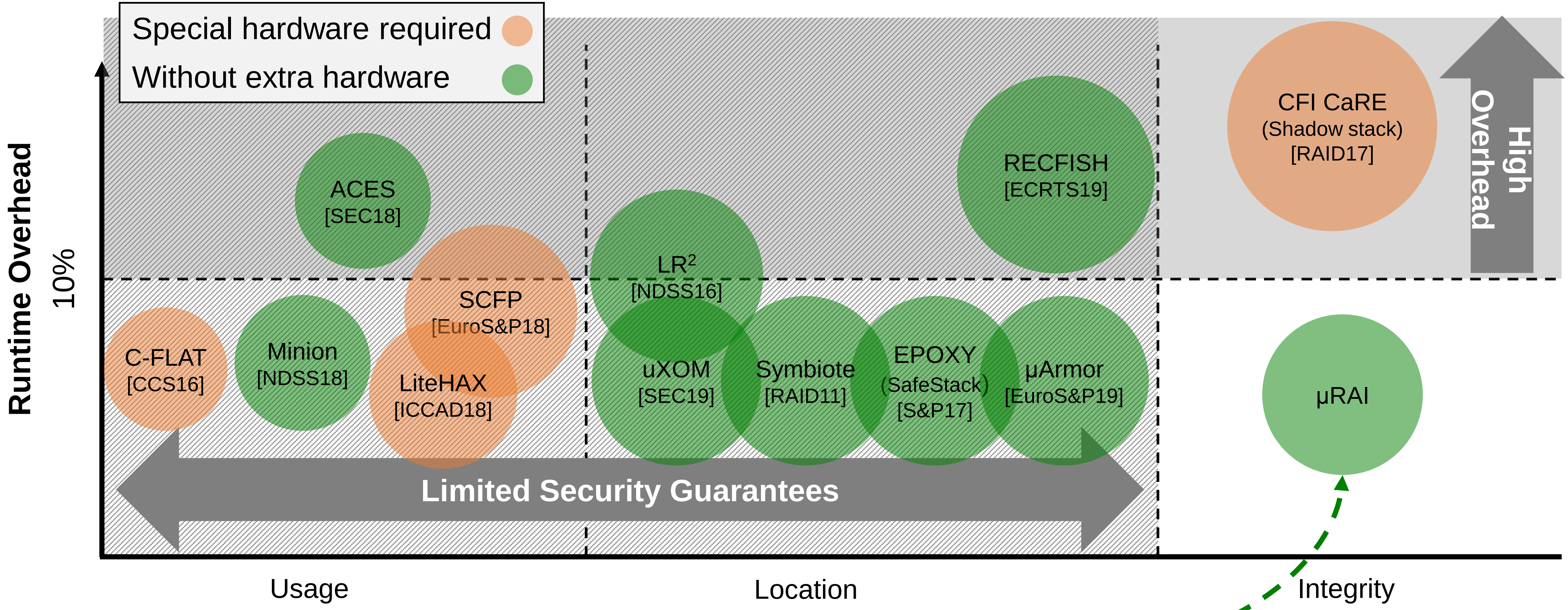
RAM (Data)

Address	RAM
0x02050000	Stack
	…
	Heap
	…
0x02000000	Data

MCUS Defenses for Return Addresses (Conceptual)



MCUS Defenses for Return Addresses (Related Work)



Return Address Integrity + Low runtime overhead + No special hardware

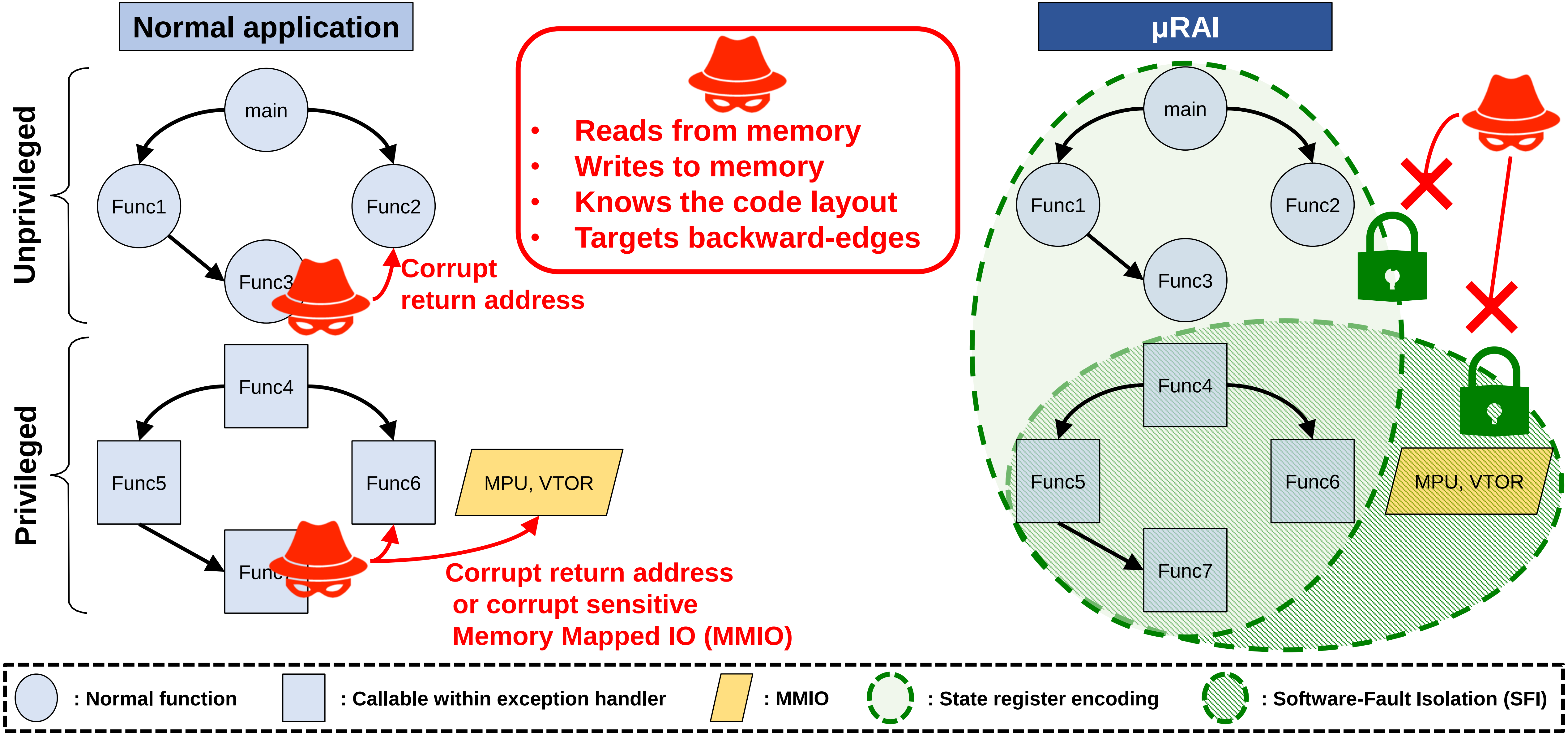
Return Address Integrity (RAI)

- **Every attack requires corrupting a return addresses by *overwriting* it**

RAI Property:

1. **Ensure the return address is never writable except by an authorized instruction**
2. **Return addresses are never pushed to the stack or any writable memory by an adversary**

Threat Model & µRAI Protection



μRAI: Overview

1. **Enforces the RAI property** → **State Register** +

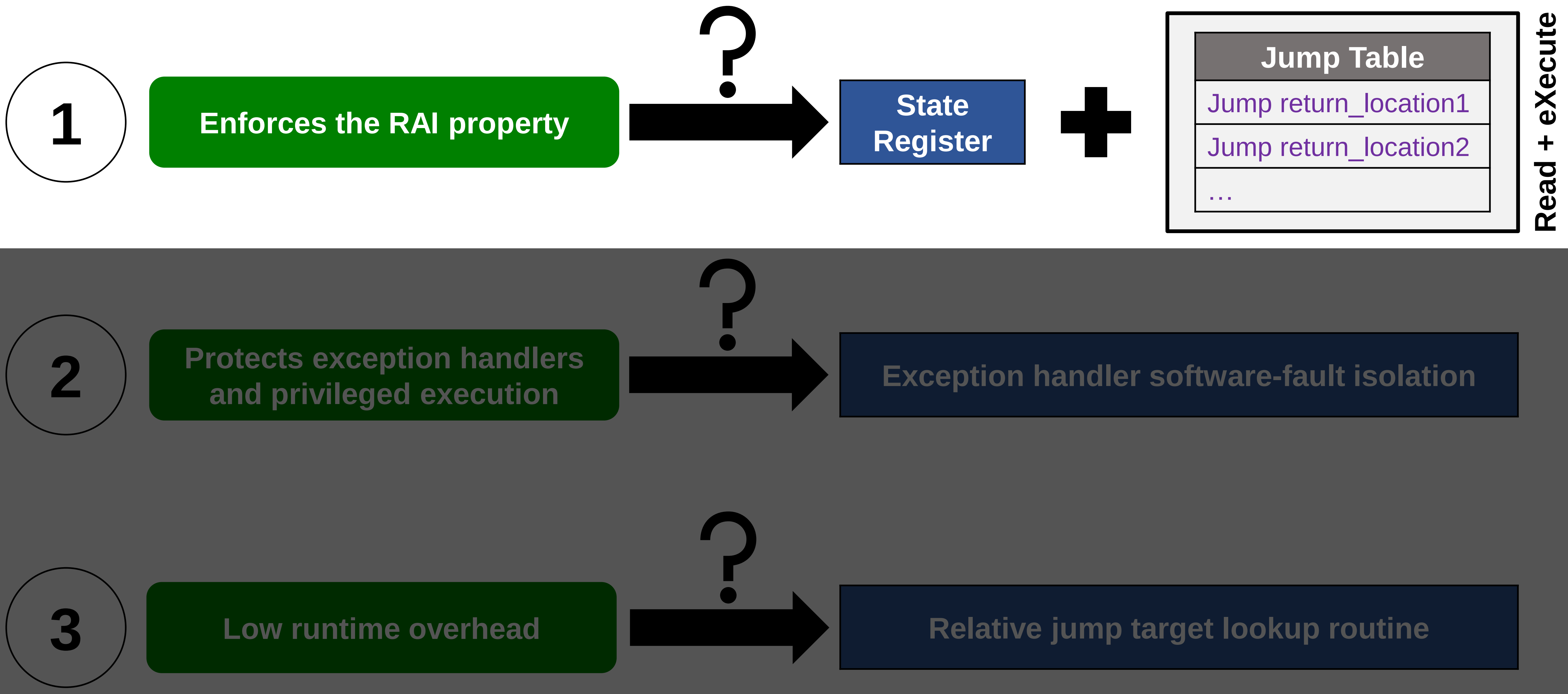
Jump Table
Jump return_location1
Jump return_location2
…

Read + eXecute

2. **Protects exception handlers and privileged execution** → **Exception handler software-fault isolation**

3. **Low runtime overhead** → **Relative jump target lookup routine**

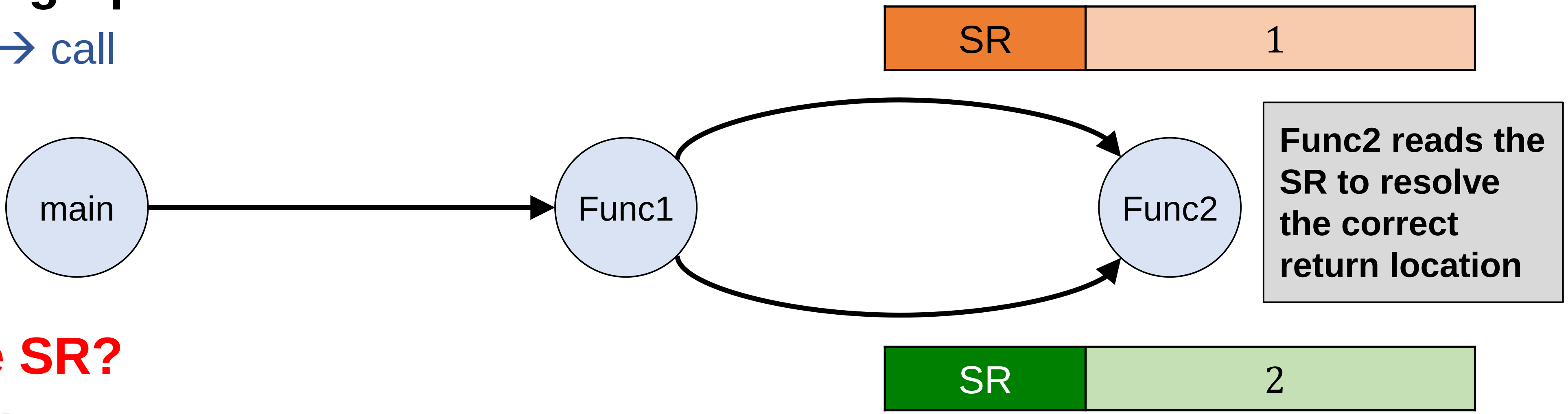
µRAI: Overview



µRAI and the State Register

- **State Register (SR):**
  - Can be any general-purpose register → exclusively used by µRAI
  - Never spilled → cannot be overwritten through a memory corruption
  - Does not contain a return address → encoded values to resolve the return location

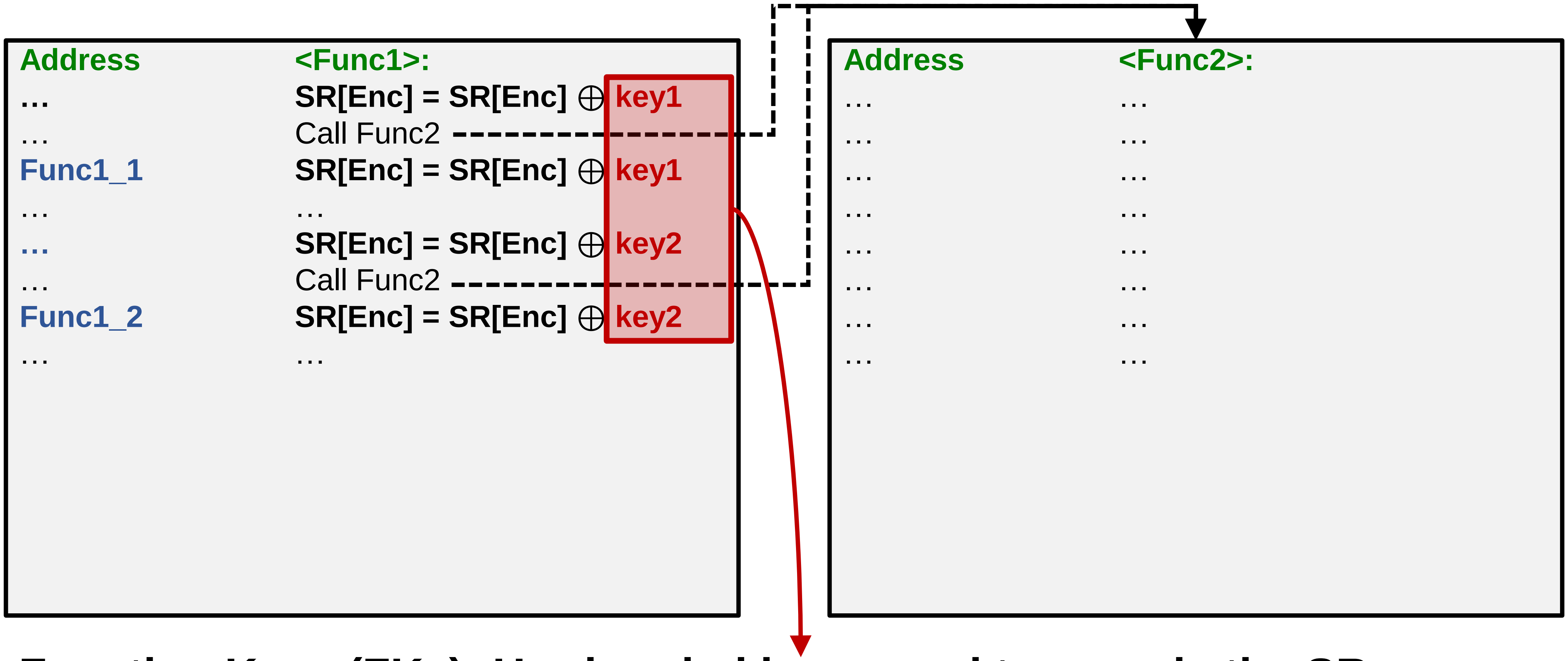
- **Example call graph:**
  - Each edge → call



- **How encode SR?**
- **An XOR chain**

µRAI: Terminology

SR[Recursive]	SR[Encoded]
0	C



- **Function Keys (FKs): Hard-coded keys used to encode the SR**

µRAI: Terminology

SR[Recursive]	SR[Encoded]
0	C

Address	<Func1>:
…	SR[Enc] = SR[Enc] ⊕ key1
…	Call Func2
Func1_1	SR[Enc] = SR[Enc] ⊕ key1
…	…
…	SR[Enc] = SR[Enc] ⊕ key2
…	Call Func2
Func1_2	SR[Enc] = SR[Enc] ⊕ key2
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	Jump ERROR

- **Function IDs (FIDs): Possible values of the SR for the function**

PURDUE UNIVERSITY EPFL Sandia National Laboratories جامعة الملك سعود King Saud University

µRAI: Terminology

SR[Recursive]	SR[Encoded]
0	C

Address	<Func1>:
…	SR[Enc] = SR[Enc] $\oplus$ key1
…	Call Func2
Func1_1	SR[Enc] = SR[Enc] $\oplus$ key1
…	…
…	SR[Enc] = SR[Enc] $\oplus$ key2
…	Call Func2
Func1_2	SR[Enc] = SR[Enc] $\oplus$ key2
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C $\oplus$ key1	Jump Func1_1
C $\oplus$ key2	Jump Func1_2
ELSE	Jump ERROR

- **Function Lookup Table (FLT): List of FIDs for the function**

µRAI: Transformation

SR[Recursive]	SR[Encoded]
0	C

Address	<Func1>:
…	SR[Enc] = SR[Enc] ⊕ key1
…	Call Func2
Func1_1	SR[Enc] = SR[Enc] ⊕ key1
…	…
…	SR[Enc] = SR[Enc] ⊕ key2
…	Call Func2
Func1_2	SR[Enc] = SR[Enc] ⊕ key2
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	Jump ERROR

- **Encode the SR and call Func2**

μRAI: Transformation

SR[Recursive]	**SR**[Encoded]
0	C⊕ **key1**

Address	<Func1>:
…	**SR[Enc] = SR[Enc] ⊕ key1**
…	Call Func2
Func1_1	**SR[Enc] = SR[Enc] ⊕ key1**
…	…
…	**SR[Enc] = SR[Enc] ⊕ key2**
…	Call Func2
Func1_2	**SR[Enc] = SR[Enc] ⊕ key2**
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	**Jump ERROR**

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	**Jump ERROR**

- **Func2 reads the SR and executes the corresponding direct jump**

µRAI: Transformation

SR[Recursive]	SR[Encoded]
0	C⊕ key1

Address	<Func1>:
…	SR[Enc] = SR[Enc] ⊕ key1
…	Call Func2
Func1_1	SR[Enc] = SR[Enc] ⊕ key1
…	…
…	SR[Enc] = SR[Enc] ⊕ key2
…	Call Func2
Func1_2	SR[Enc] = SR[Enc] ⊕ key2
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	Jump ERROR

- **Func2 returns correctly and the SR is decoded**

µRAI: Transformation

SR[Recursive]	SR[Encoded]
0	C

Address	<Func1>:
…	SR[Enc] = SR[Enc] ⊕ key1
…	Call Func2
Func1_1	SR[Enc] = SR[Enc] ⊕ key1
…	…
…	SR[Enc] = SR[Enc] ⊕ key2
…	Call Func2
Func1_2	SR[Enc] = SR[Enc] ⊕ key2
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	Jump ERROR

- **The previous SR value is restored**

μRAI: Transformation

SR[Recursive]	**SR**[Encoded]
0	C

Address	<Func1>:
…	**SR[Enc] = SR[Enc] $\oplus$ key1**
…	Call Func2
Func1_1	**SR[Enc] = SR[Enc] $\oplus$ key1**
…	…
…	**SR[Enc] = SR[Enc] $\oplus$ key2**
…	Call Func2
Func1_2	**SR[Enc] = SR[Enc] $\oplus$ key2**
…	…

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	**Jump ERROR**

Address	<Func2>:
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

Function ID (FID)	Return Target
C $\oplus$ key1	Jump Func1_1
C $\oplus$ key2	Jump Func1_2
ELSE	**Jump ERROR**

- **The same happens for other calls. Func1 can then return correctly**

µRAI: Overview

1 Enforces the RAI property → State Register +

Jump Table
Jump return_location1
Jump return_location2
…

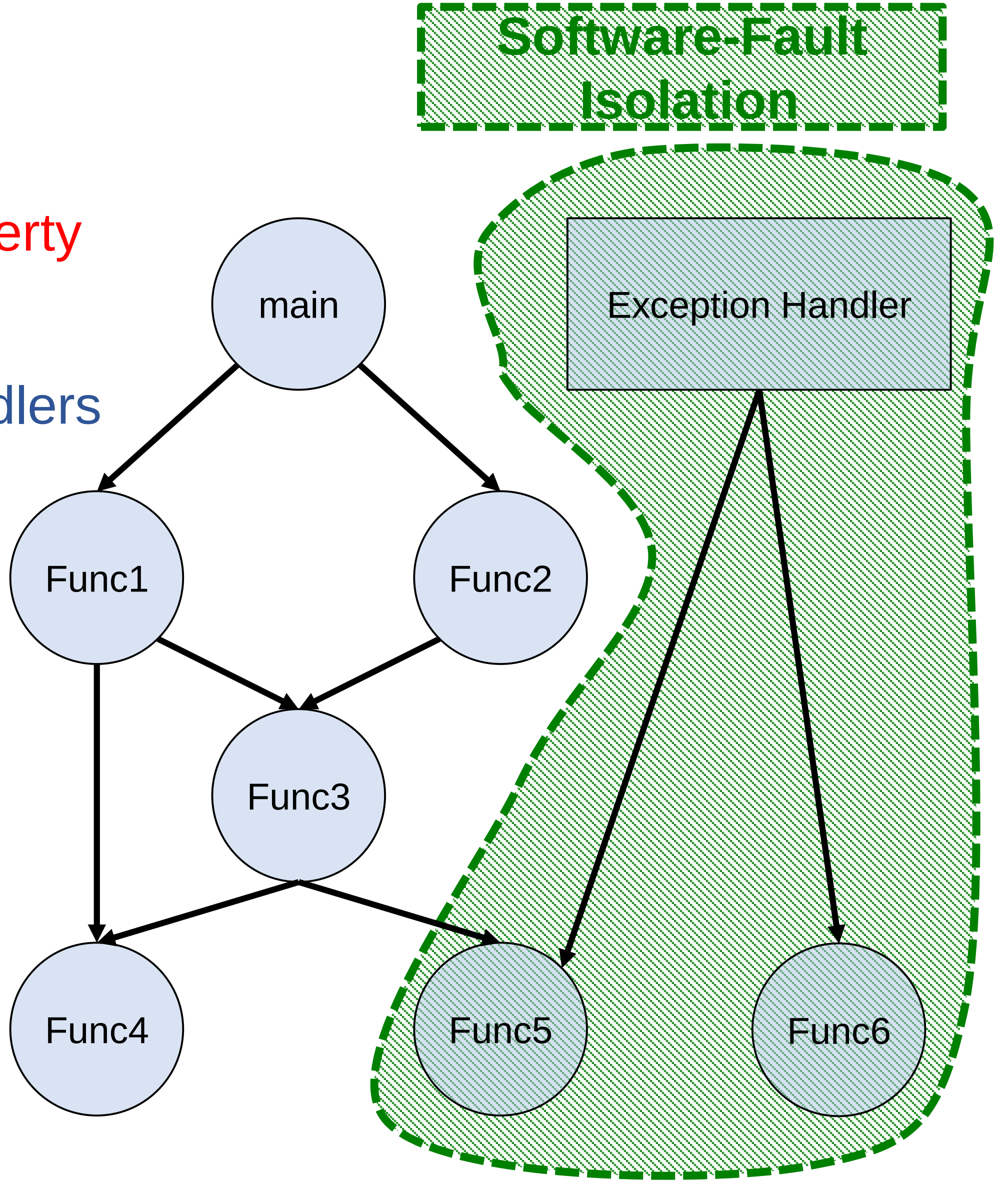
Read + eXecute

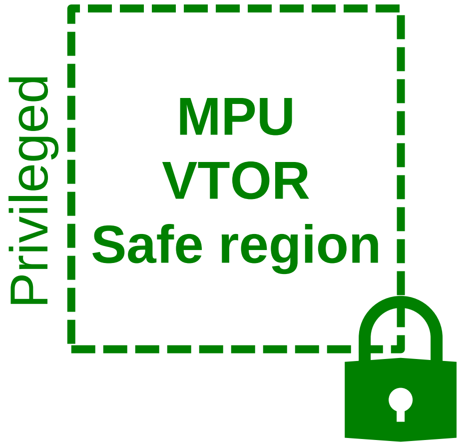
2 Protects exception handlers and privileged execution → Exception handler software-fault isolation

3 Low runtime overhead → Relative jump target lookup routine

µRAI: Enforce RAI for exception handlers

- **Exception handlers execute with privileges**
  - Can disable the MPU → enable code injection
  - Can corrupt exception stack frame → break RAI property
- **Solution:**
  - Apply SFI only to functions callable by exception handlers
  - Limit SFI overhead compared to full-SFI





µRAI: Overview

1

Enforces the RAI property

State Register

Jump Table
Jump return_location1
Jump return_location2
…

Read + eXecute

2

Protects exception handlers and privileged execution

Exception handler software-fault isolation

3

Low runtime overhead

?

Relative jump target lookup routine

Target Lookup Routine (TLR)

- **How can we find the correct direct jump in the FLT efficiently?**
  - Use a relative jump before the FLT
  - Resolve the correct return location efficiently regardless of FLT size

Address	<Func1>:
…	…
…	…
…	…
…	Jump PC+SR

FID	Return Target
1	Jump location1
2	Jump location2
3	Jump ERROR
4	Jump location4
5	Jump location5

Assume the correct return location is **location4**

Comparing all FID can be slow!

Use SR as an index of a jump table

Align the FLT
→ make SR = 4

µRAI: Overview

1. **Enforces the RAI property** → **State Register** + **Jump Table**

Jump Table
Jump return_location1
Jump return_location2
…

Read + eXecute

2. **Protects exception handlers and privileged execution** → **Exception handler software-fault isolation**

3. **Low runtime overhead** → **Relative jump target lookup routine**

Evaluation

- **Five MCUS applications on Cortex-M4:**
  - PinLock
  - FatFs_uSD
  - FatFs_RAM
  - LCD_uSD
  - Animation

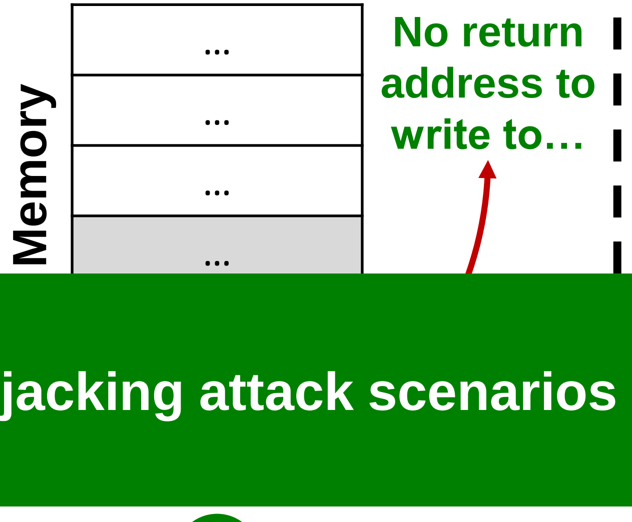
- **CoreMark benchmark[1]**
  - Standard MCUS performance benchmark

[1] EEMBC, "Coremark - industry-standard benchmarks for embedded systems," http://www.eembc.org/coremark

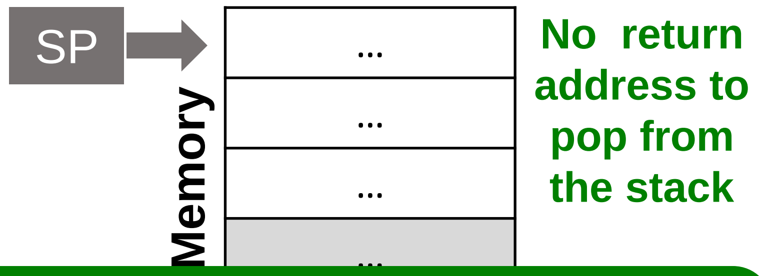
Security Evaluation Using PinLock: Unlock The Lock

Buffer overflow

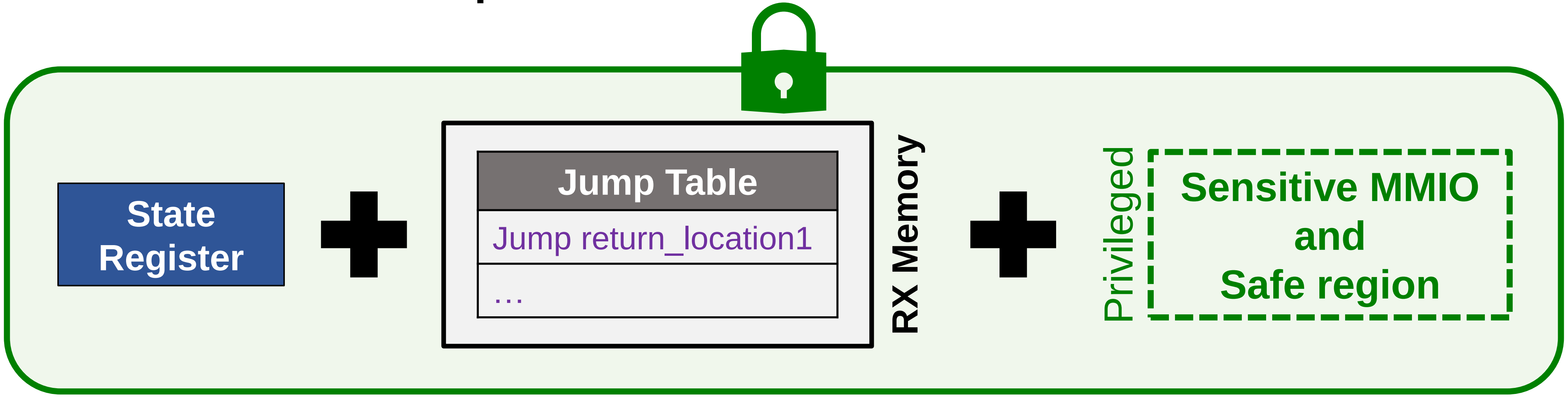
Arbitrary write



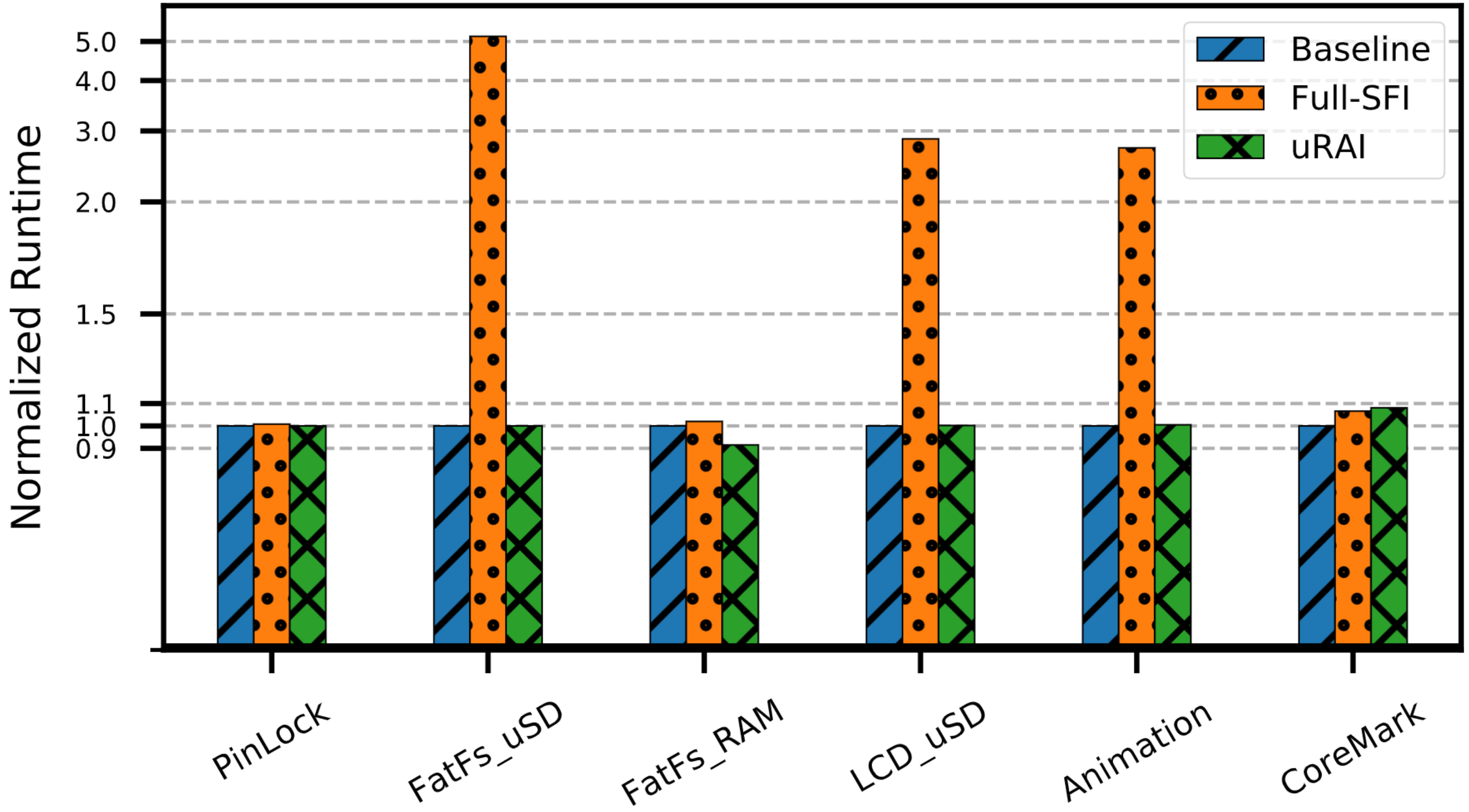
Stack pivot



✓ µRAI prevents all control-flow hijacking attack scenarios targeting return addresses



Performance results



- **Requiring full-SFI results in high overhead → average of 130.5%**
- **µRAI results in low overhead → average of 0.1%**

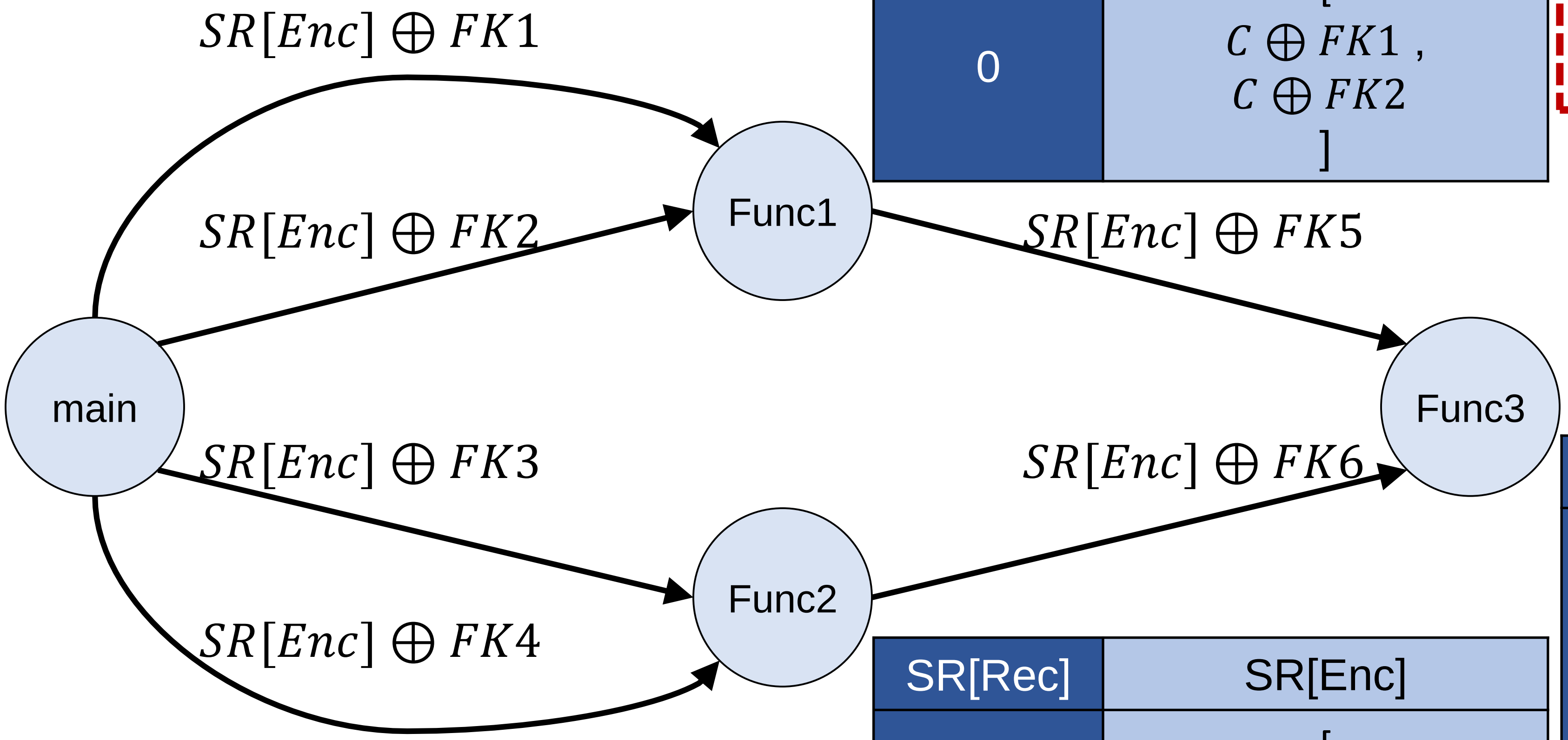
µRAI: Conclusion

- **Control-flow hijacking on MCUS is a threat**
- **µRAI secures MCUS against control-flow hijacking**
  - Enforces the RAI property for MCUS → protects backward edges
  - Complemented with type-based CFI → end-to-end code pointer protection
- **Presents a portable encoding scheme**
  - Does not require special hardware features (only a register and an MPU)
  - Applicable to other systems
- **Low runtime overhead**

https://github.com/embedded-sec/uRAI

Backup Slides

Challenge: Path Explosion



SR[Rec]	SR[Enc]
0	C

SR[Rec]	SR[Enc]
0	[$C \oplus FK1$, $C \oplus FK2$]

SR[Rec]	SR[Enc]
0	[$C \oplus FK3$, $C \oplus FK4$]

- **Func3 has 2 call sites**
- **FLT is size is 4!**

SR[Rec]	SR[Enc]
0	[$C \oplus FK1 \oplus FK5$, $C \oplus FK2 \oplus FK5$, $C \oplus FK3 \oplus FK6$, $C \oplus FK4 \oplus FK6$]

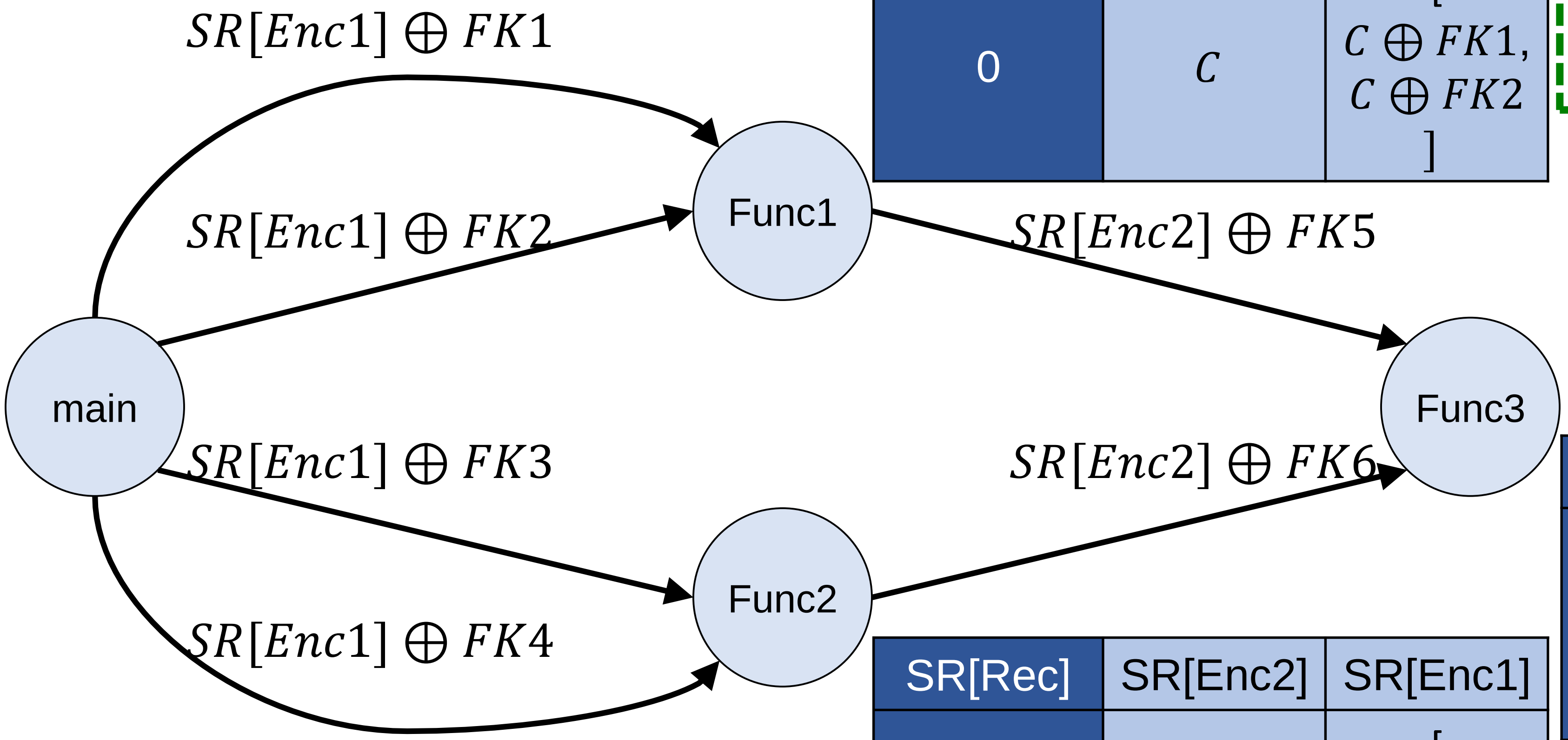
Path Explosion Solution: Segmentation

- **State register segmentation**

Recursion counter (Higher N bits)			Encoded value (Lower 32-N bits)			
Segment 1	...	Segment K	Segment M	…	Segment 2	Segment 1

- **Functions only use the bits in their assigned segment.**

Path Explosion Solution: Segmentation



SR[Rec]	SR[Enc2]	SR[Enc1]
0	C	C

SR[Rec]	SR[Enc2]	SR[Enc1]
0	C	[$C \oplus FK1$, $C \oplus FK2$]

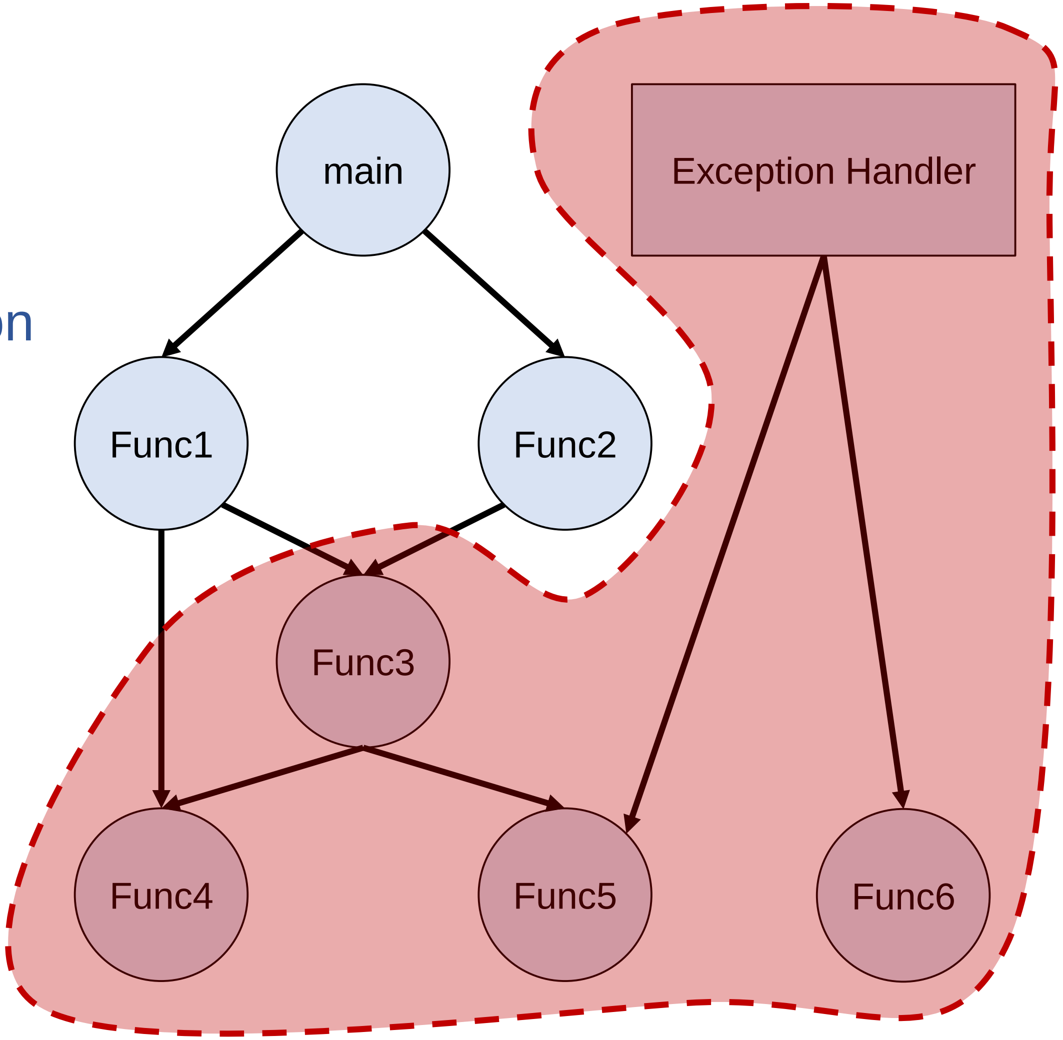
SR[Rec]	SR[Enc2]	SR[Enc1]
0	C	[$C \oplus FK3$, $C \oplus FK4$]

- **Segmentation reduces FLT from 4 to 2**

SR[Rec]	SR[Enc2]	SR[Enc1]
0	[$C \oplus FK5$, $C \oplus FK6$,]	Not used

µRAI: Scalability

- **What if no more values can be found for the SR?**
- **Solution:**
  - Partition the call graph if no solution is found
  - Entering a new partition
    →Save and reset the SR to a privileged safe region
  - Returning to a previous partition
    → Restore the SR



Store Instructions Protected with EH-SFI

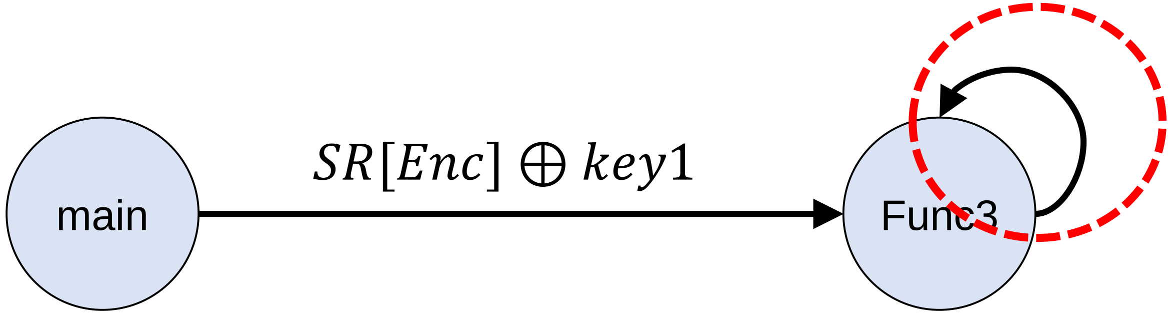
App	# of Store instruction			
	Static	Total	(Static/Total)%	Dynamic
PinLock	56	516	10.9	7
FatFs_uSD	99	1,802	5.5	906K
FatFs_RAM	7	1,116	0.6	7
LCD_uSD	99	2,814	3.5	48K
Animation	99	2,760	3.6	66K
CoreMark	56	1,024	5.5	7

SR Layout: Recursion

- **The SR has two parts:**
  - ENC: Encoded value
  - REC: Recursion counter

- **Cannot use XOR with recursion**
- **Collision occurs with existing values Func1**

$\rightarrow \boldsymbol{SR} \oplus ANY\ KEY \oplus ANY\ KEY = \boldsymbol{SR}$



$\boldsymbol{SR[Rec]} + 1$ **before the call**
$\boldsymbol{SR[Rec]} - 1$ **after return**
If $\boldsymbol{SR[Rec]} > 0$ **→ recursion**

SR[Rec]	SR[Enc]
0	C

SR[Rec]	SR[Enc]
0	$C \oplus key1$

µRAI: Handling recursion

SR[Rec]	SR[Enc]
0	C

```
Address        <Func1>:
…              SR[Enc] = SR[Enc] ⊕ key1
…              Call Func2
Func1_1        SR[Enc] = SR[Enc] ⊕ key1
…              …
…              SR[Enc] = SR[Enc] ⊕ key2
…              Call Func2
Func1_2        SR[Enc] = SR[Enc] ⊕ key2
…              …
```

Function ID (FID)	Return Target
C	Jump return_location1
ELSE	Jump ERROR

```
Address        <Func2>:
…              SR[Rec]++
…              Call Func2
Func2_1         SR[Rec]--
…              …
…              IF 0 < SR[Rec] < MAX_REC
…                         Jump Func2_1
…              ELSE
…              …
```

Function ID (FID)	Return Target
C ⊕ key1	Jump Func1_1
C ⊕ key2	Jump Func1_2
ELSE	Jump ERROR

- **If recursive → use a counter (*recursion is discouraged in MCUS*)**