Leveraging Micro-architectural Side Channel Information to Efficiently Enhance Program Control Flow Integrity

Chen Liu and Chengmo Yang

Electrical and Computer Engineering
University of Delaware

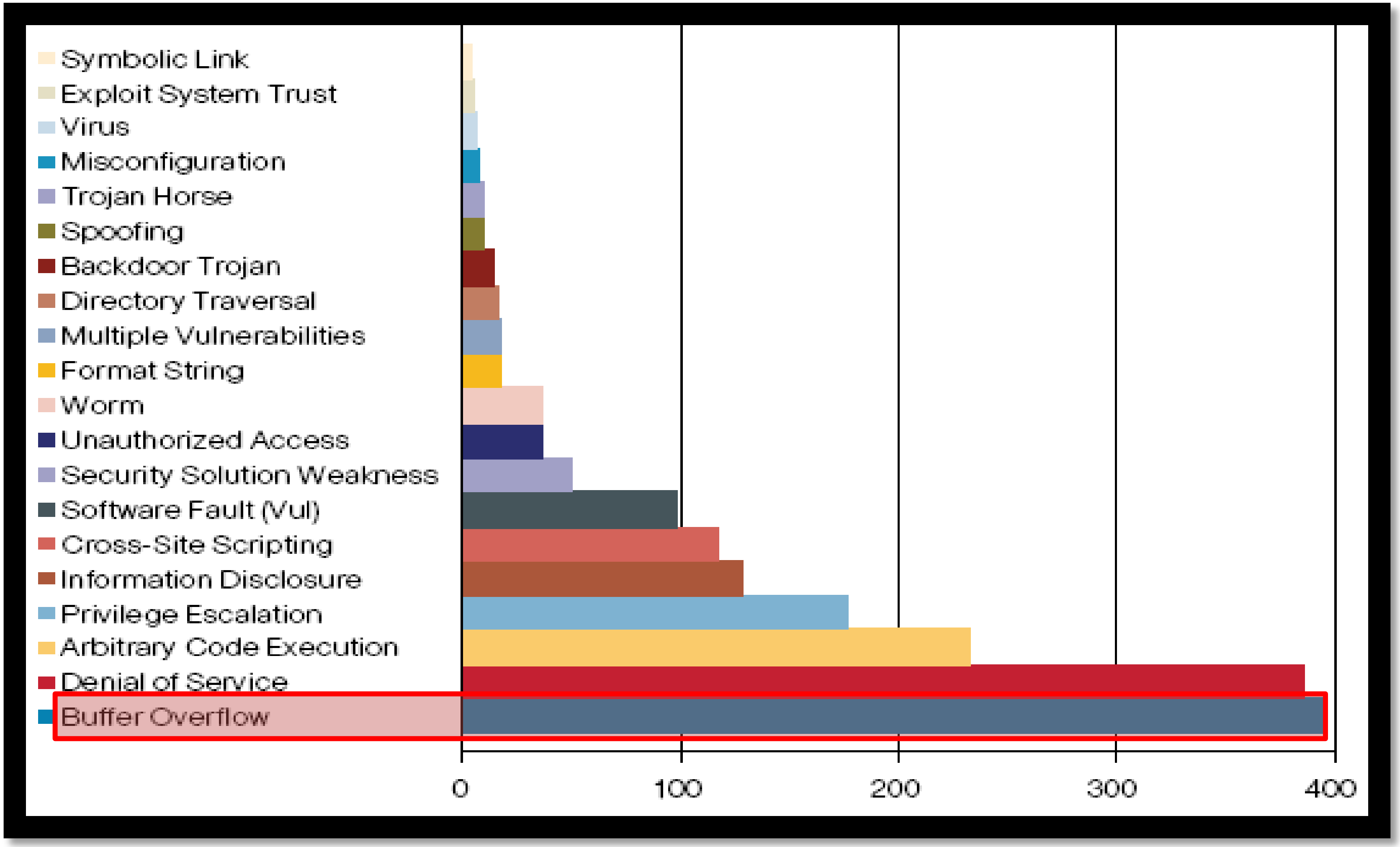
10/8/2014

Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

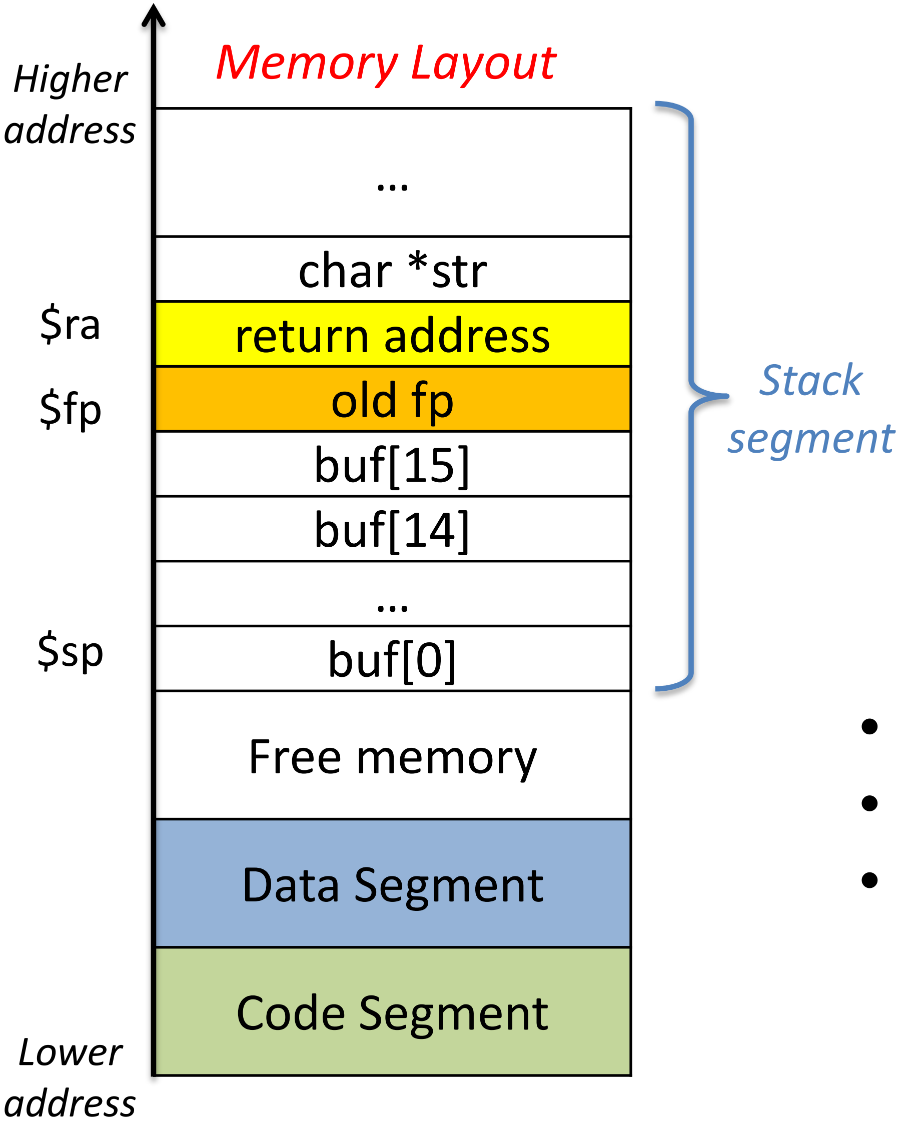
Computer System Security

Most Severe Threat – Buffer Overflow



Top 20 threats and vulnerabilities, January - October 2007

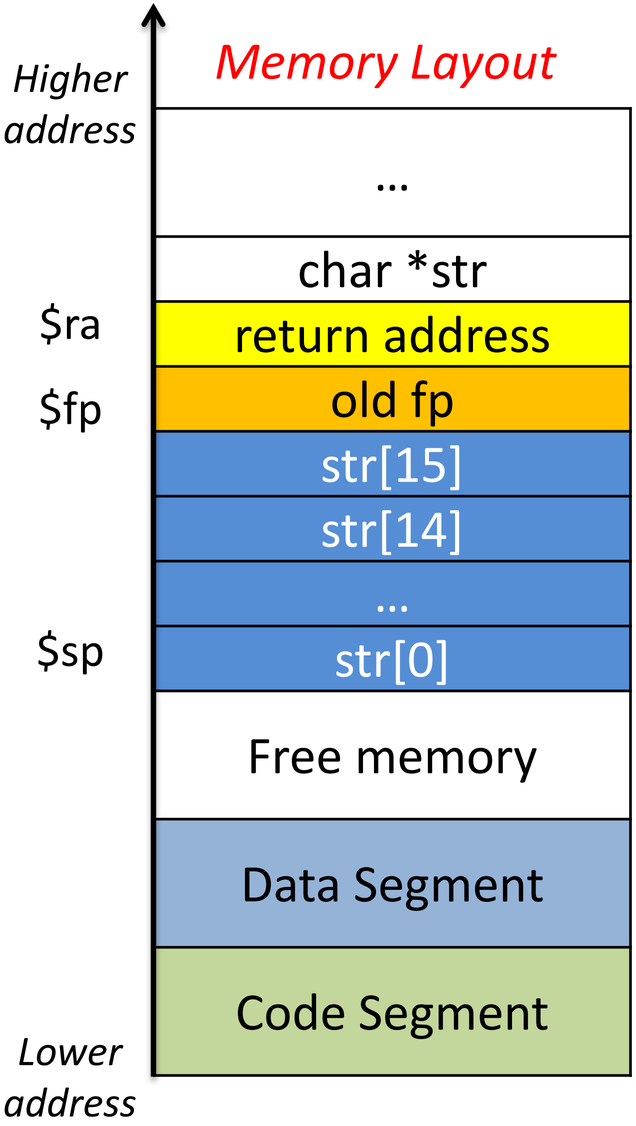
Stack Buffer Overflow – Hijack Return Address



```
int BOF (char *str) {
    char buf[16];
    strcpy(buf, str);
    return 1;
}
```

- $ra: Return address pointer
- $fp: Frame pointer
- $sp: Stack pointer

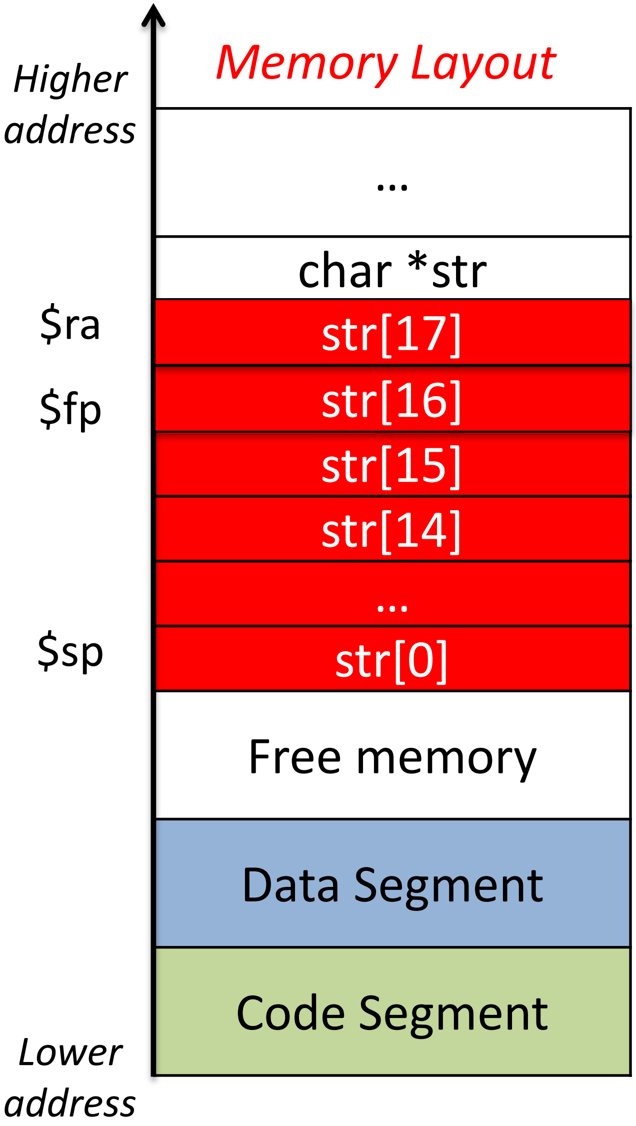
Stack Buffer Overflow – Hijack Return Address



```
int BOF (char *str) {
    char buf[16];
    strcpy(buf, str);
    return 1;
}
```

- If sizeof(str)<=16, no problem

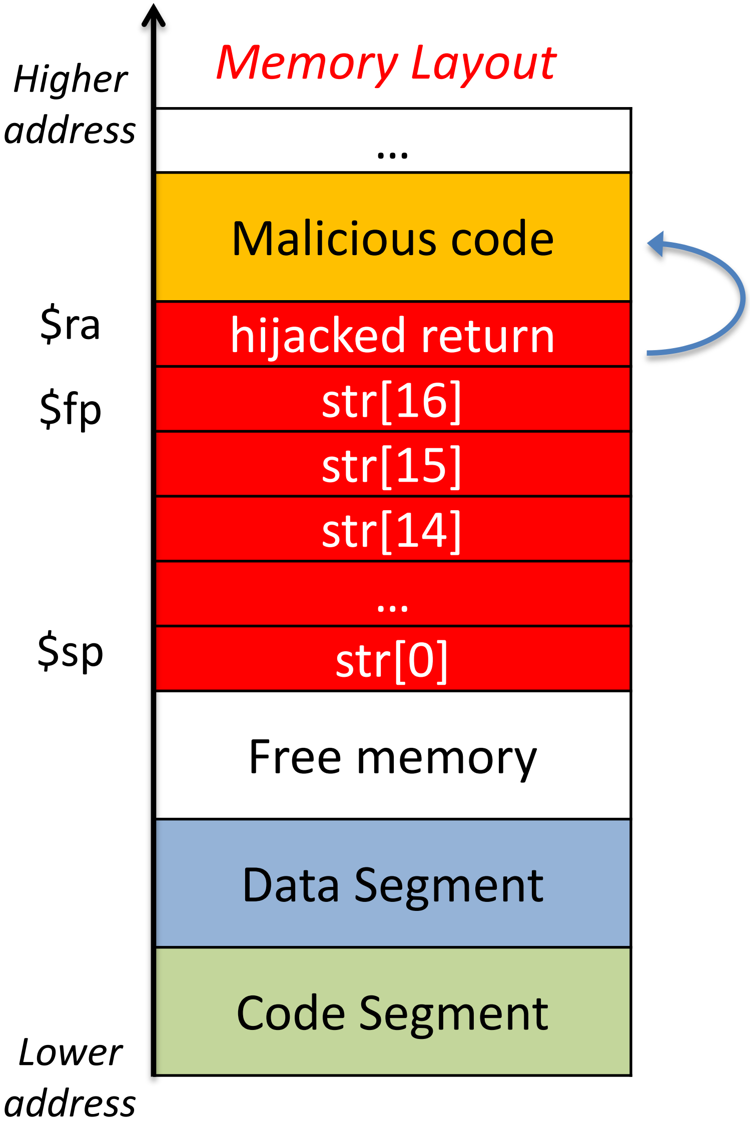
Stack Buffer Overflow – Hijack Return Address



```
int BOF (char *str) {
    char buf[16];
    strcpy(buf, str);
    return 1;
}
```

- If sizeof(str)<=16, no problem
- If sizeof(str)>16, adjacent entries will be overwritten
- **Return address could be hijacked**
- Three types of BOF based attacks:
  1. Code injection
  2. Return-to-libc
  3. Return-oriented programming

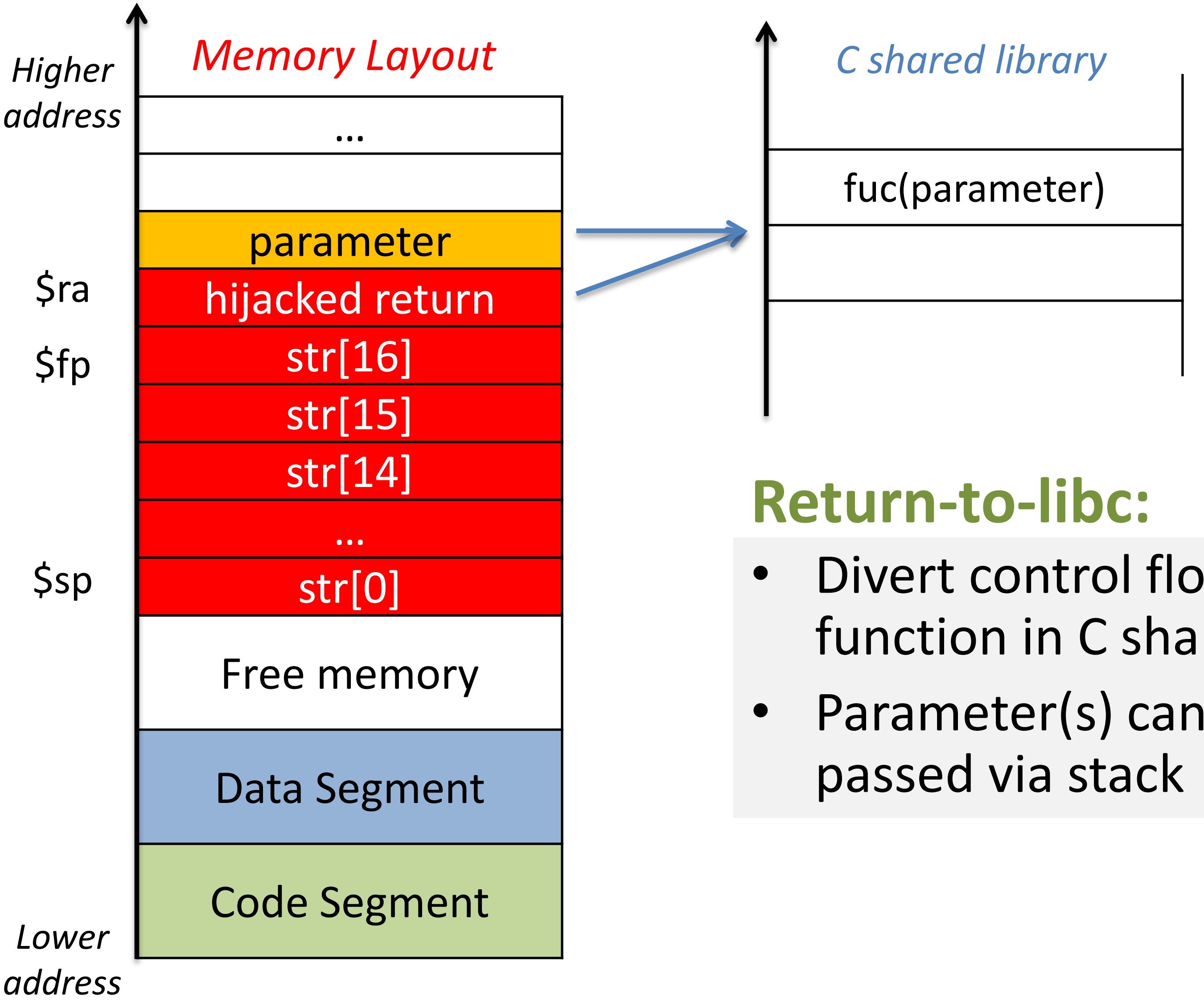
Buffer Overflow (BOF) – Code Injection



Code Injection:

- Put malicious code in str[]
- Write str[] in stack
- Direct return address to the malicious code

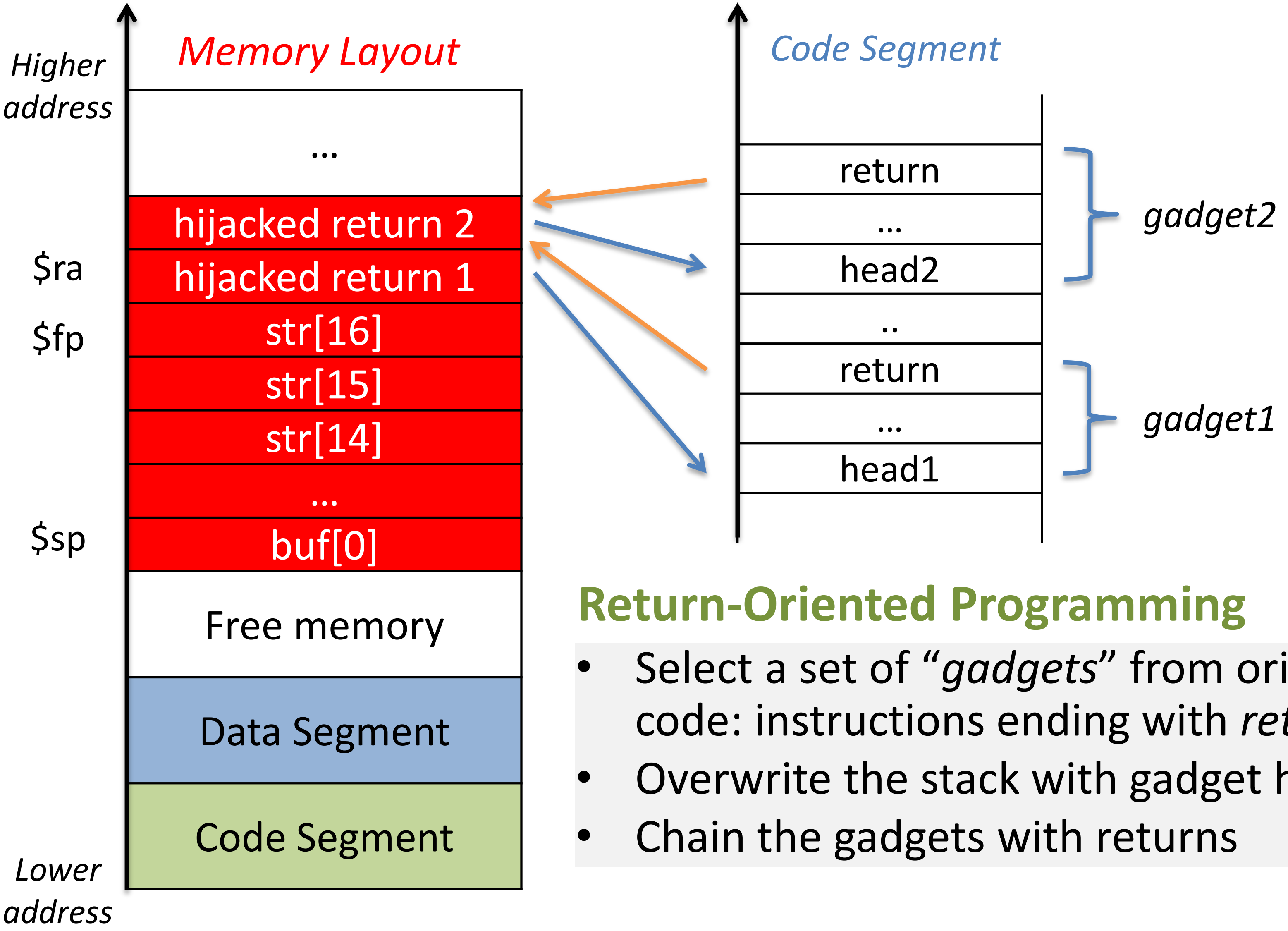
Buffer Overflow (BOF) – Return-to-libc



Return-to-libc:

- Divert control flow to a function in C shared library
- Parameter(s) can also be passed via stack

Buffer Overflow (BOF) – Return-Oriented Programming



Return-Oriented Programming

- Select a set of "*gadgets*" from original code: instructions ending with *return*.
- Overwrite the stack with gadget heads
- Chain the gadgets with returns

Existing Countermeasures

NX bit

- Pages cannot be both writable and executable at the same time

StackGuard

- Places a *canary* in between local variables and frame pointer

Address randomization

- Adds a random offset to each page/segment

Control flow checking

- Compares runtime control flow with profiled control flow

Not suitable for embedded systems

- Software-based, sizable overhead
- Need to change compiler
- Cannot defend against all three types of BOF based attacks

Our solution: online attack detection by monitoring micro-architectural events

- Hardware based, low overhead
- Achieve high detection accuracy through hardware enhancement
- Can handle all three types of BOF based attacks

Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

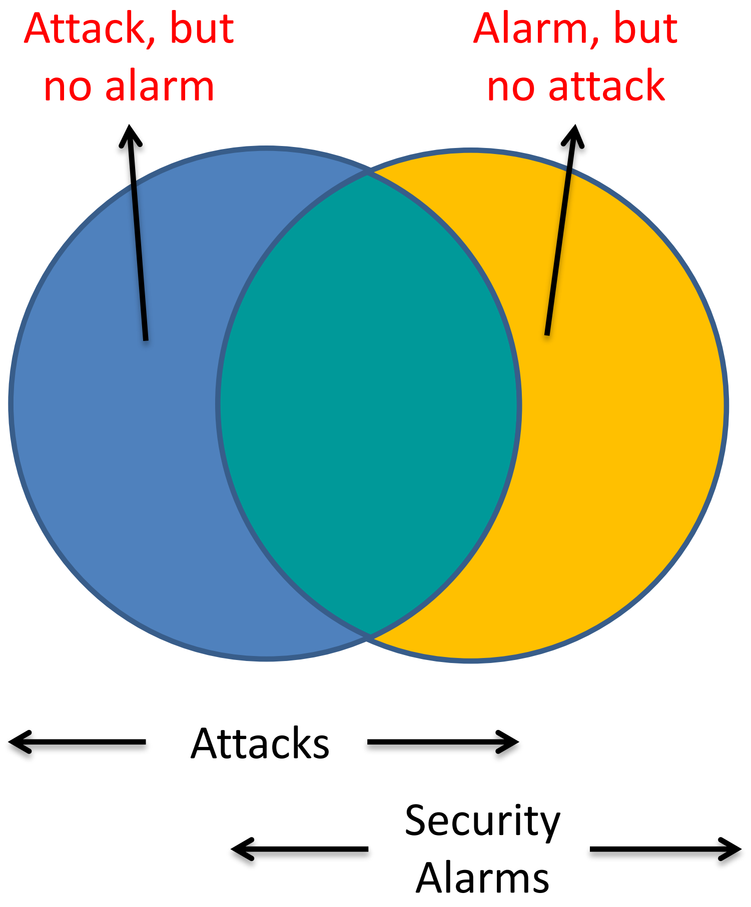
High-quality Detection Scheme for Embedded Systems

LOW COST

- Low runtime cost
- Low design cost

HIGH ACCURACY

- Low false positive rate
- Low false negative rate



High-quality Detection Scheme for Embedded Systems

LOW COST

- Low runtime cost
- Low design cost

HIGH ACCURACY

- Low false positive rate
- Low false negative rate

Design requirements

- Monitor events that are highly correlated with BOF attacks
- Leverage existing performance-driven micro-architectural modules for security purpose
- Further extend those modules to improve detection accuracy

High-quality Detection Scheme for Embedded Systems

LOW COST

- Low runtime cost
- Low design cost

HIGH ACCURACY

- Low false positive rate
- Low false negative rate

Three critical questions:

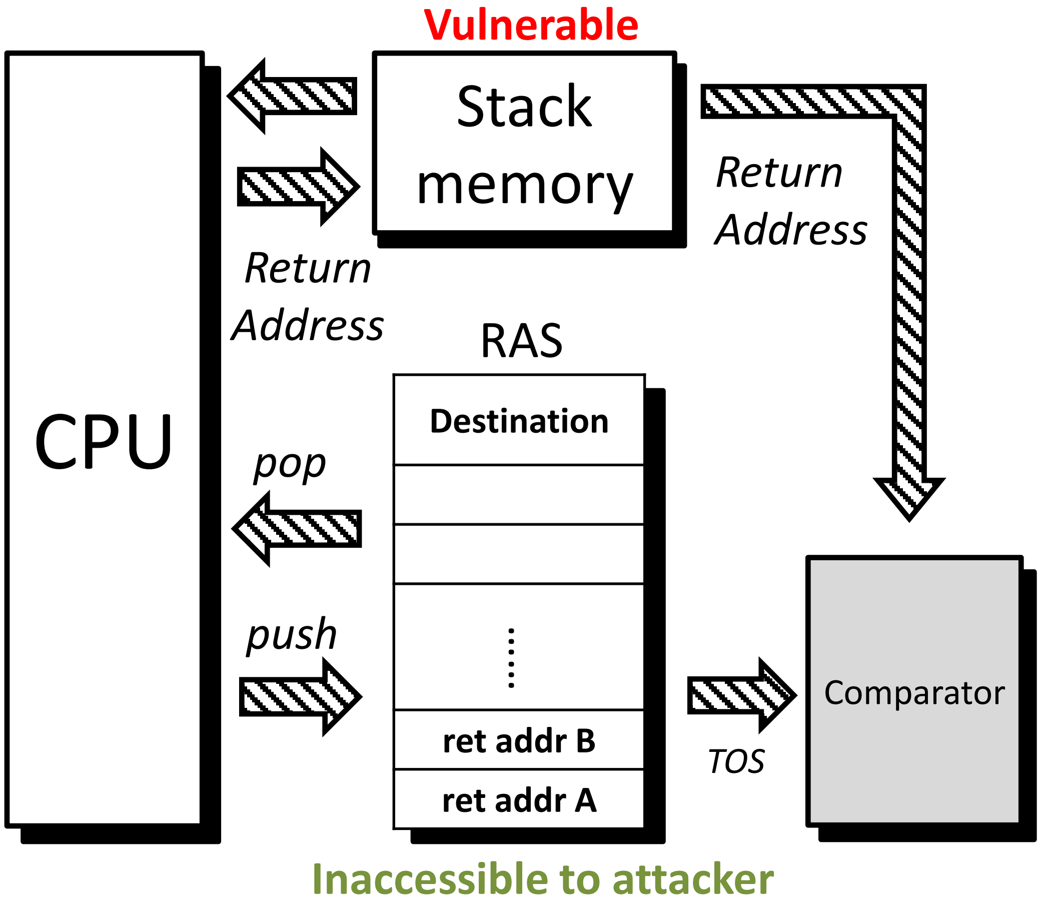
- Which events to monitor?
- How to further enhance accuracy?
- What is the overall alarm condition?

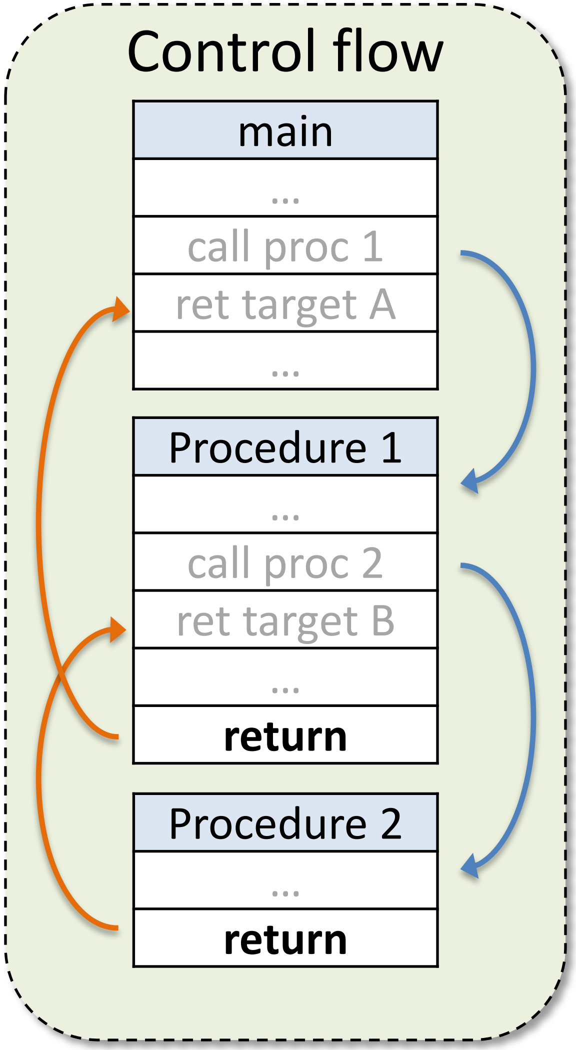
Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

Event 1: Return address stack (RAS) mis-prediction

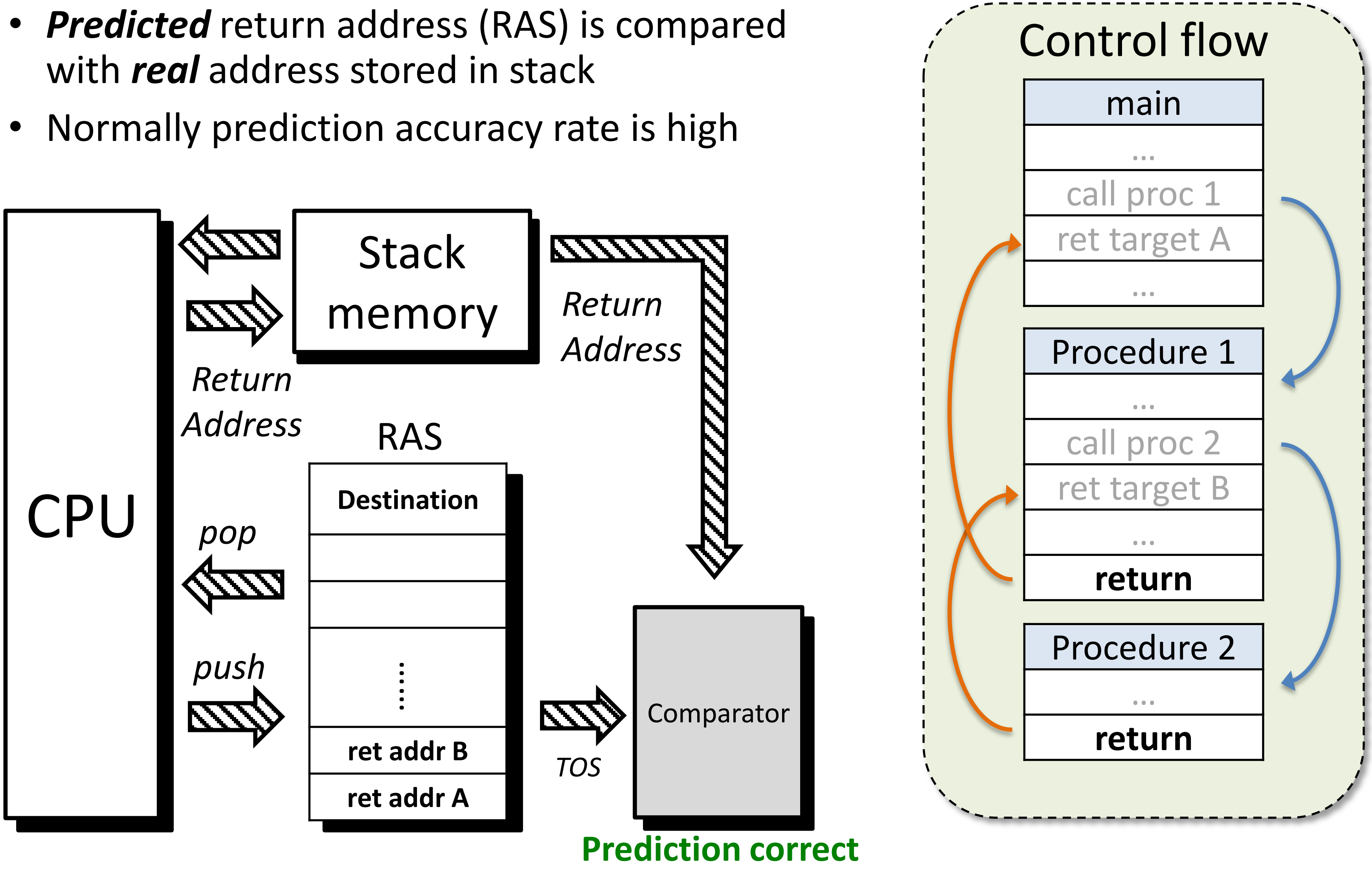
- Modern processors use RAS to improve pipeline performance
- Return address predicted at instruction fetch stage





Event 1: Return address stack (RAS) mis-prediction

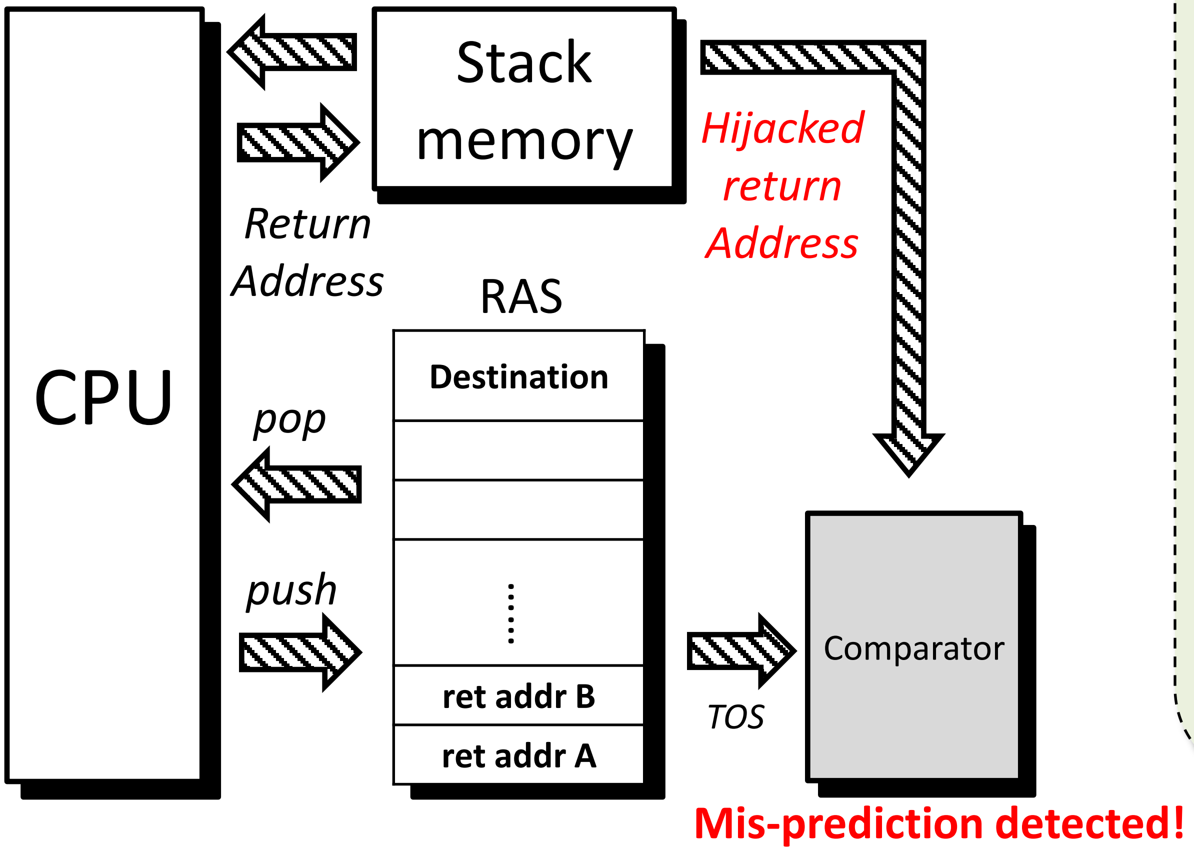
- ***Predicted*** return address (RAS) is compared with ***real*** address stored in stack
- Normally prediction accuracy rate is high

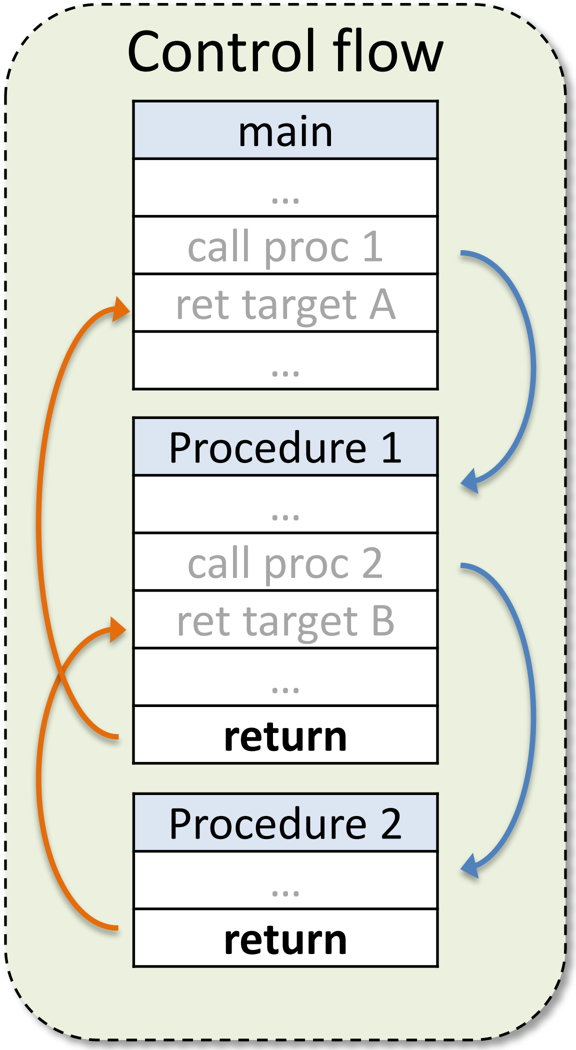


Prediction correct

Event 1: Return address stack (RAS) mis-prediction

- ***Predicted*** return address (RAS) is compared with ***real*** address stored in stack
- When there is attack, mis-prediction occurs

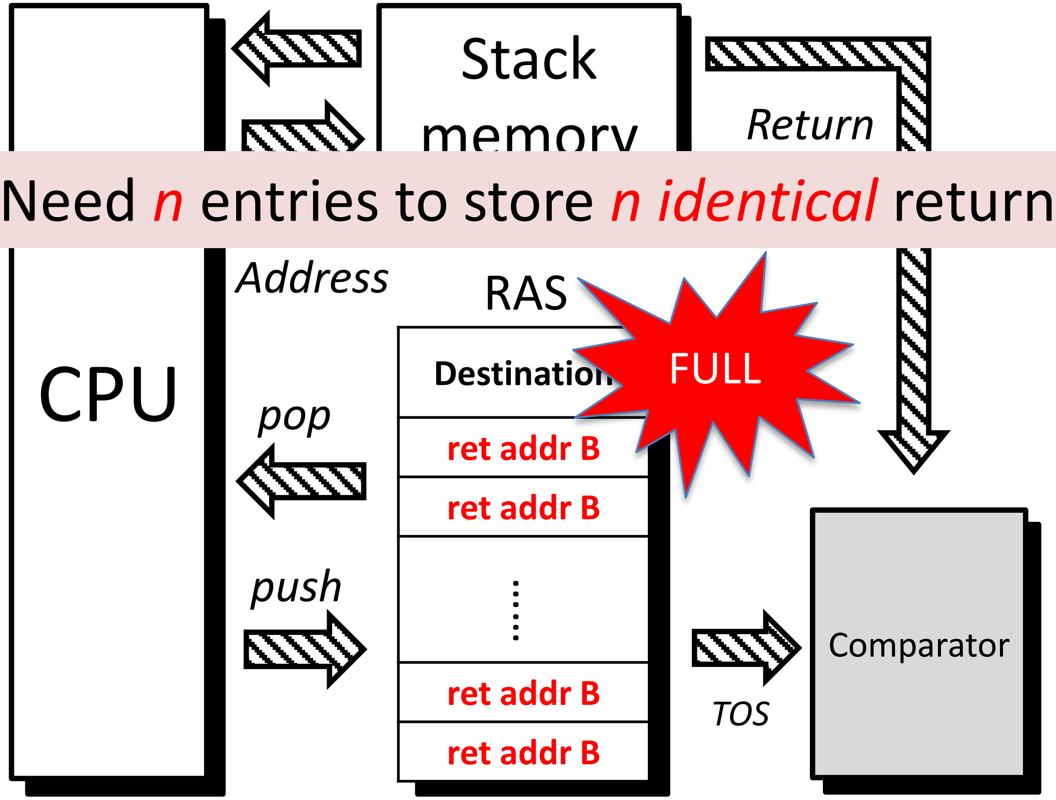




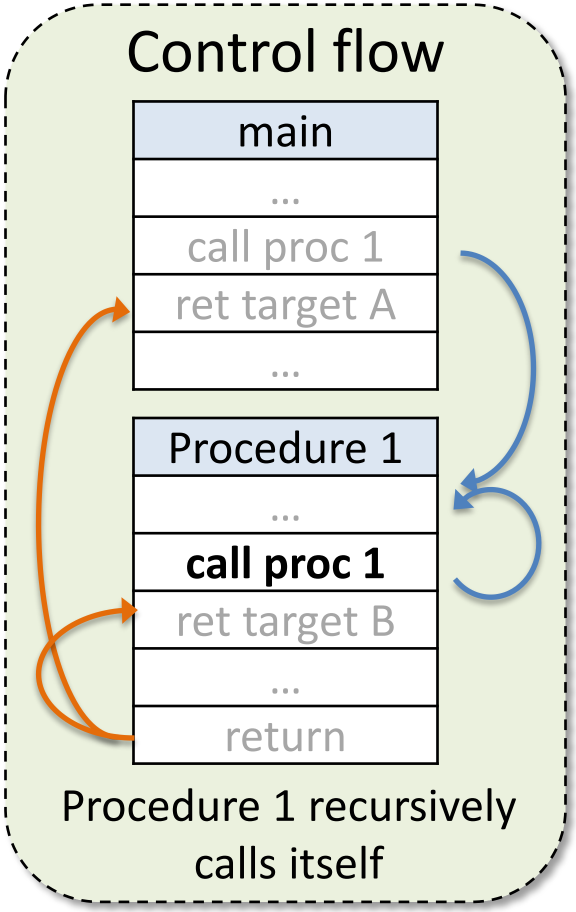
Event 1: Possible **False Positives**

When there is no attack, RAS mis-prediction may occur due to

- Non-LIFO control flow, uncommon
- Size limitation of RAS, upon recursive procedure calls



Need *n* entries to store *n identical* returns

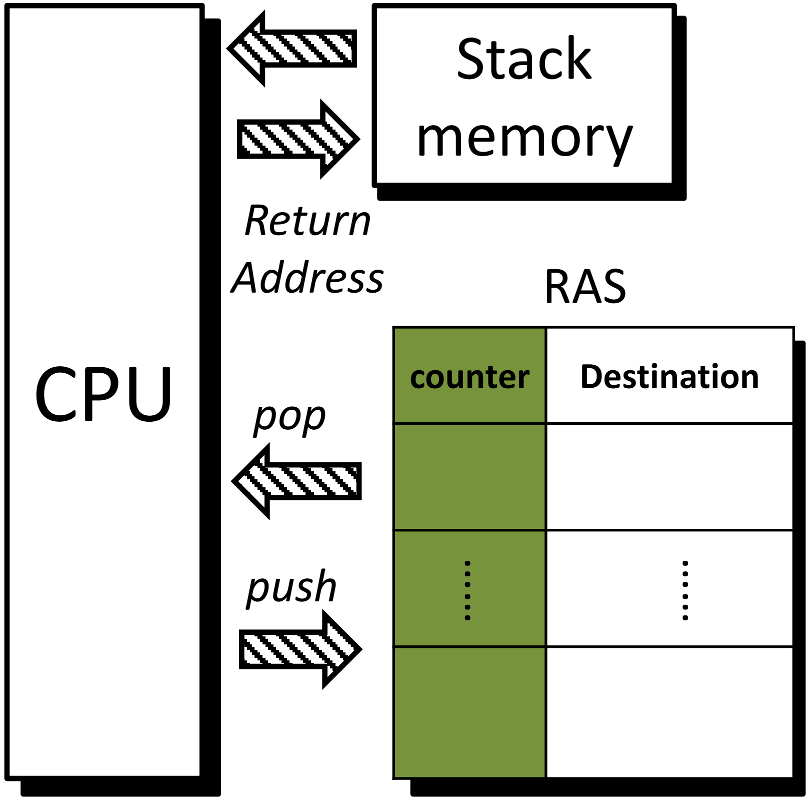


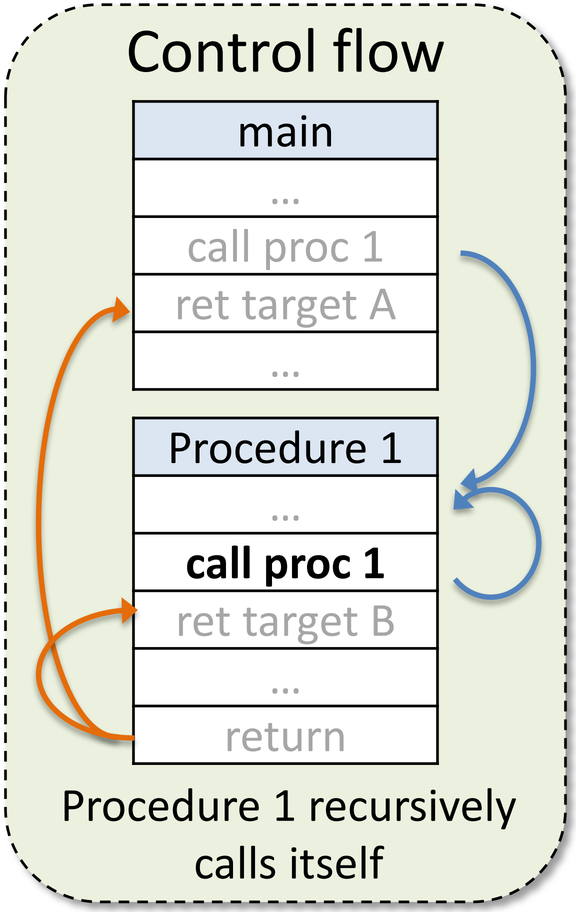
Procedure 1 recursively calls itself

Event 1: Enhance RAS with **Call Counter**

Idea: add a recursive call counter to each RAS entry

- Initially, counter=0



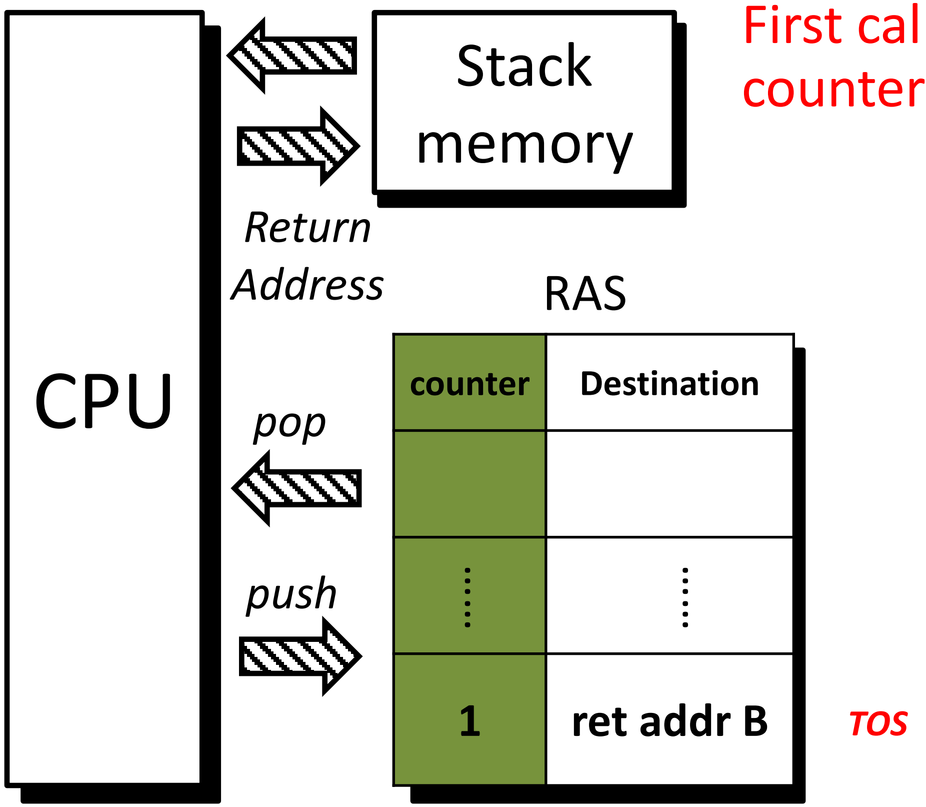


Procedure 1 recursively calls itself

Event 1: Enhance RAS with **Call Counter**

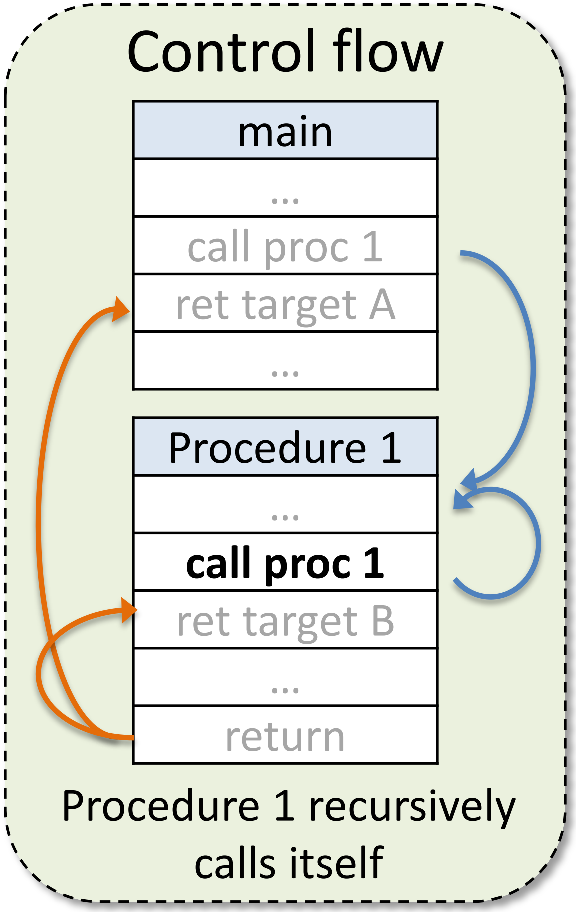
Idea: add a recursive call counter to each RAS entry

- Initially, counter=0
- Upon a call, if TOS = new return address, counter++



counter	Destination
⋮	⋮
1	**ret addr B**

First call, counter=1

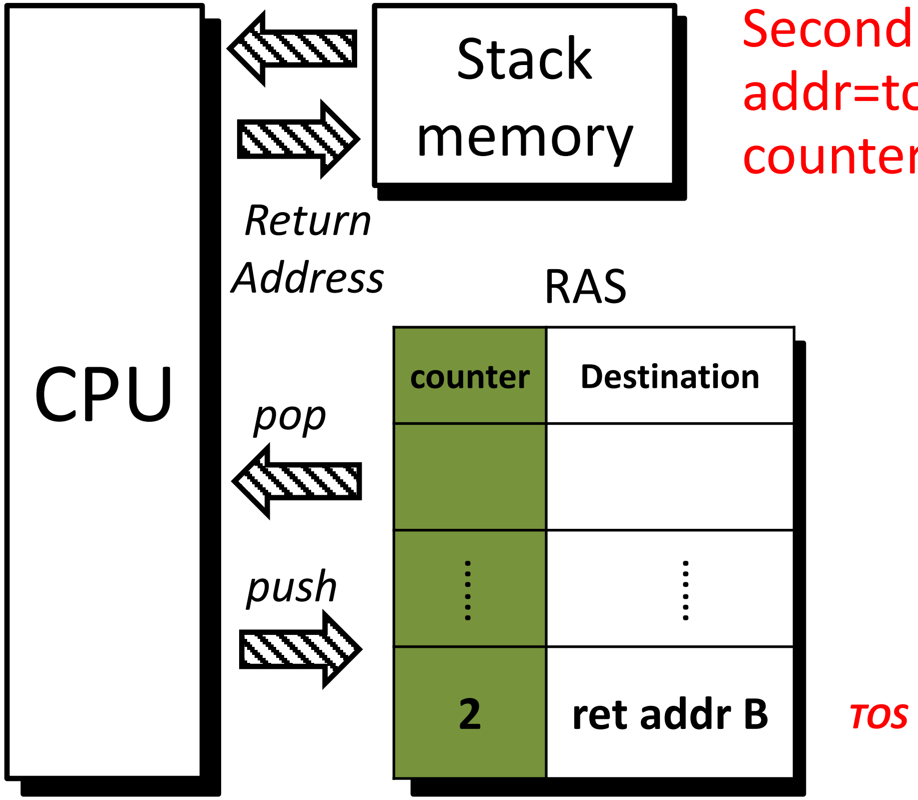


Procedure 1 recursively calls itself

Event 1: Enhance RAS with **Call Counter**

Idea: add a recursive call counter to each RAS entry

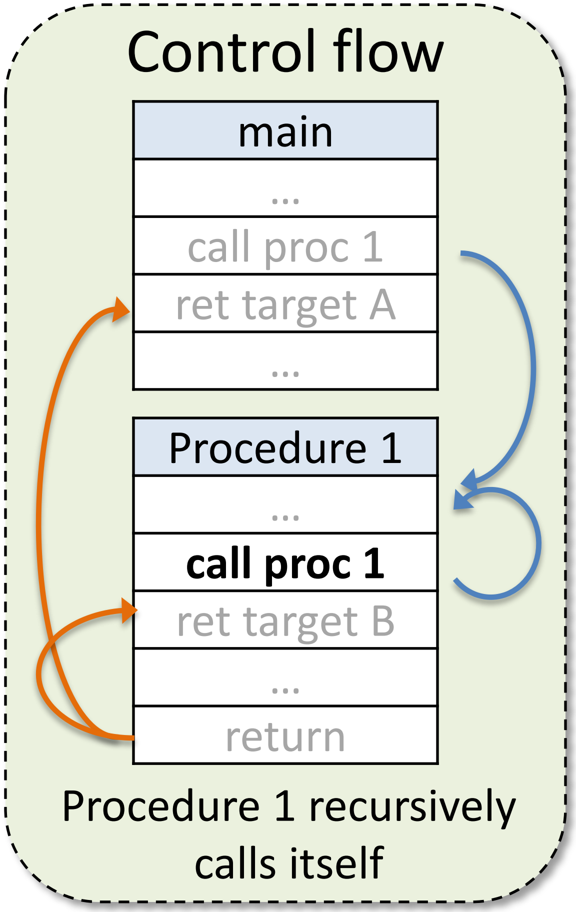
- Initially, counter=0
- Upon a call, if TOS = new return address, counter++



RAS

counter	Destination
⋮	⋮
2	**ret addr B**

Second call,
addr=tos,
counter=2

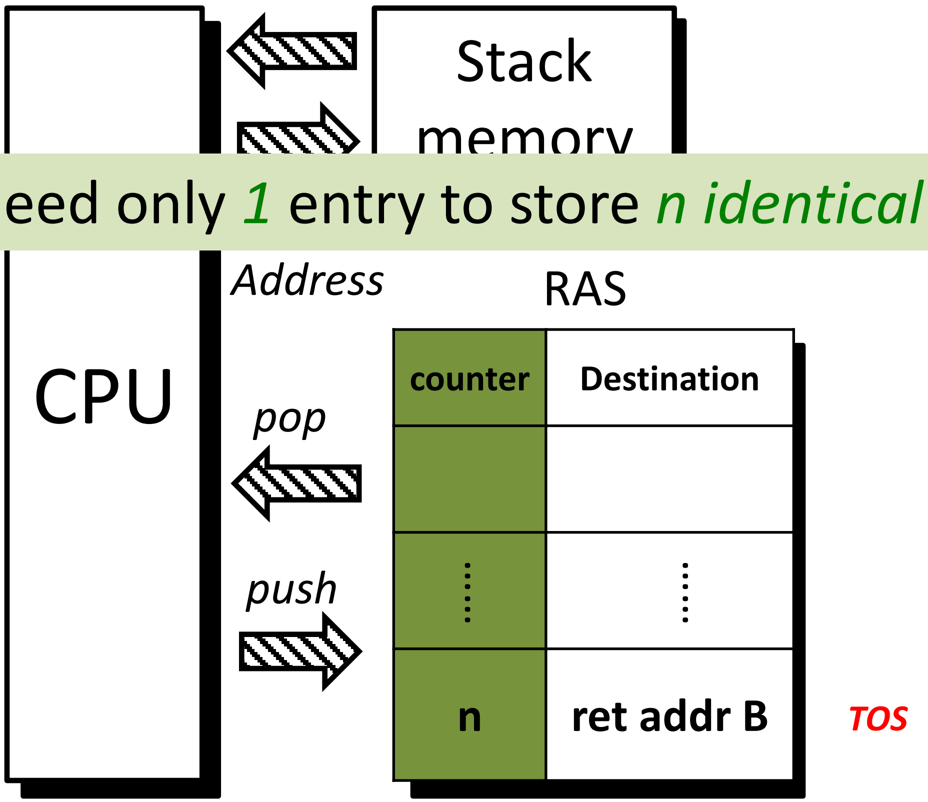


Procedure 1 recursively calls itself

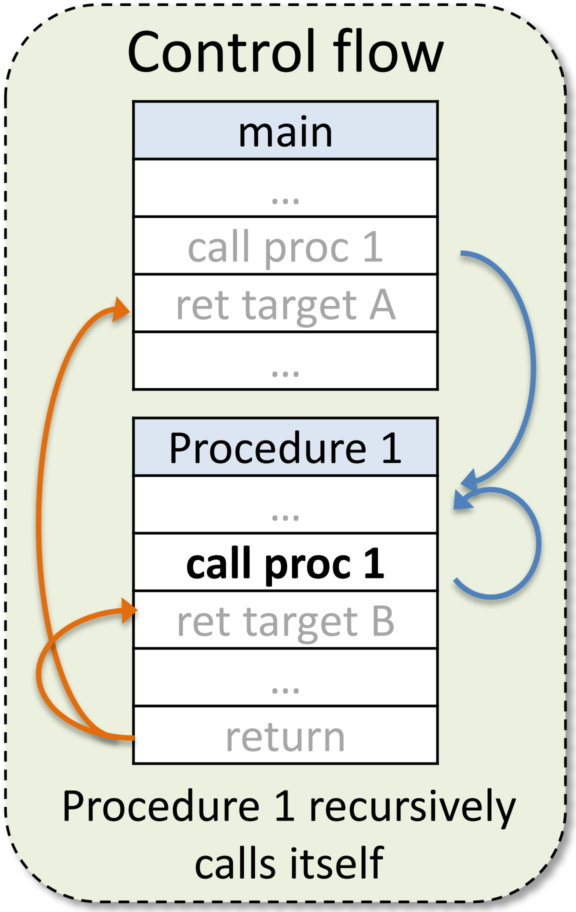
Event 1: Enhance RAS with **Call Counter**

Idea: add a recursive call counter to each RAS entry

- Initially, counter=0
- Upon a call, if TOS = new return address, counter++



Need only *1* entry to store *n identical* returns

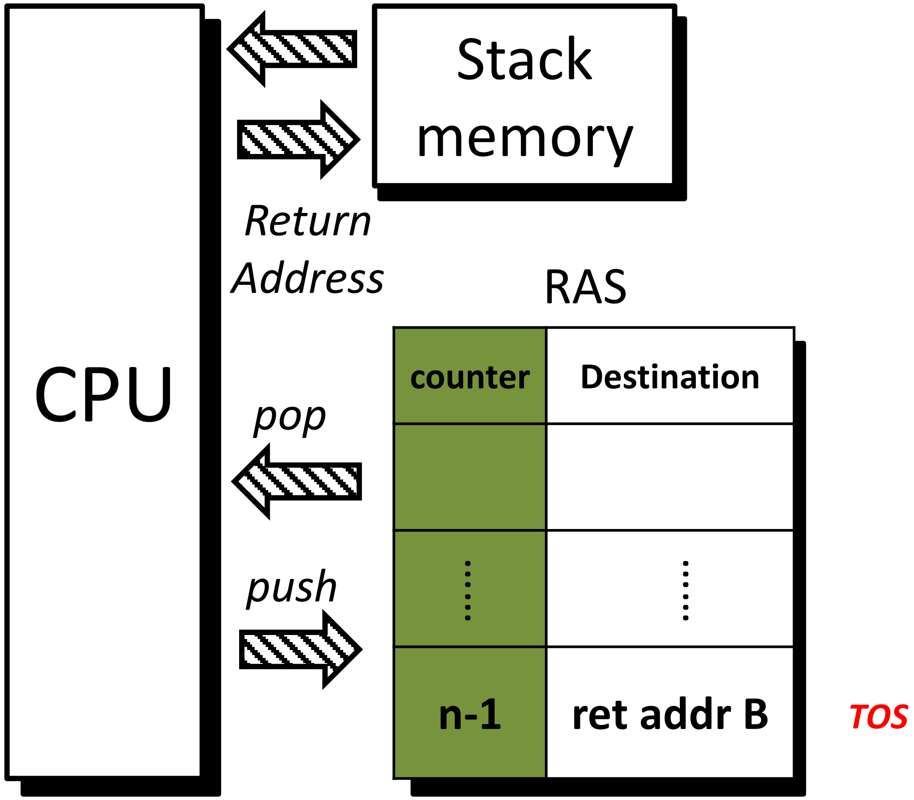


Procedure 1 recursively calls itself

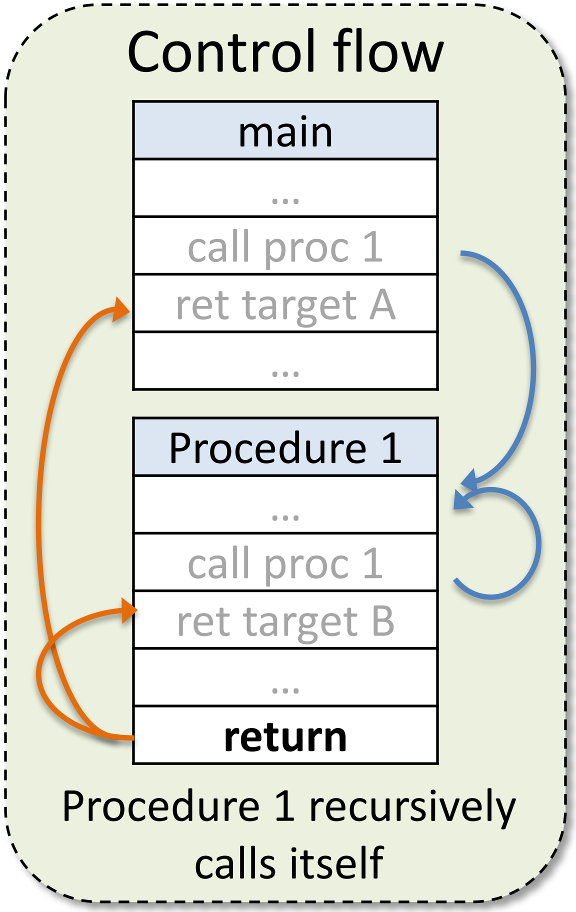
Event 1: Enhance RAS with **Call Counter**

Idea: add a recursive call counter to each RAS entry

- Upon a return, counter--



counter	Destination
⋮	⋮
n-1	ret addr B



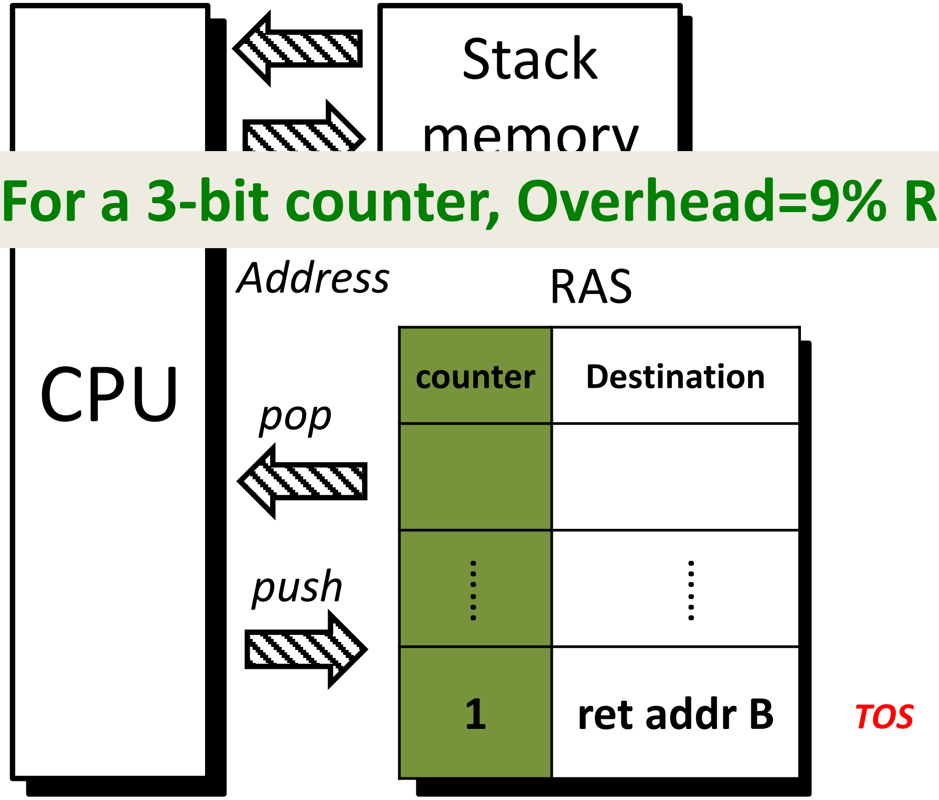
Procedure 1 recursively calls itself

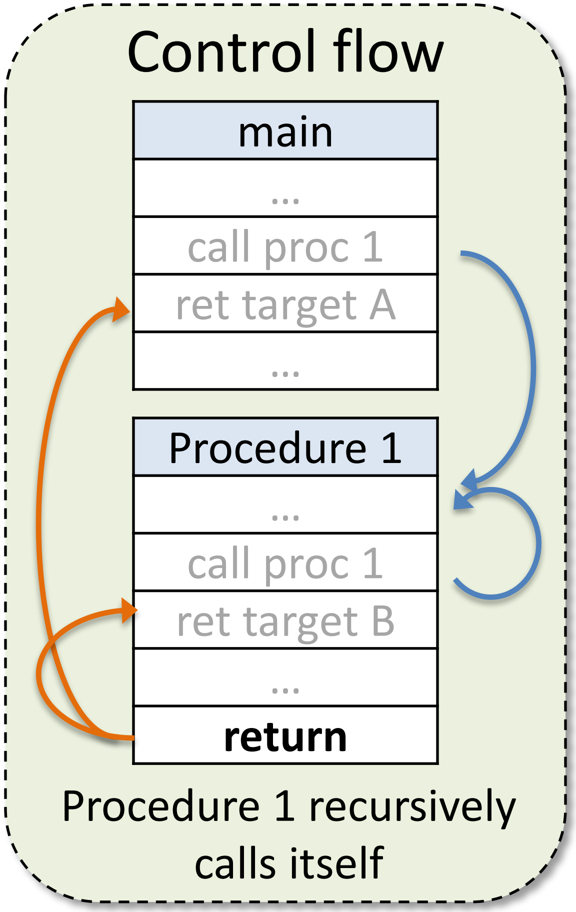
Event 1: Enhance RAS with **Call Counter**

Idea: add a recursive call counter to each RAS entry

- Upon a return, counter--
- Pop stack when counter = 0

For a 3-bit counter, Overhead=9% RAS size





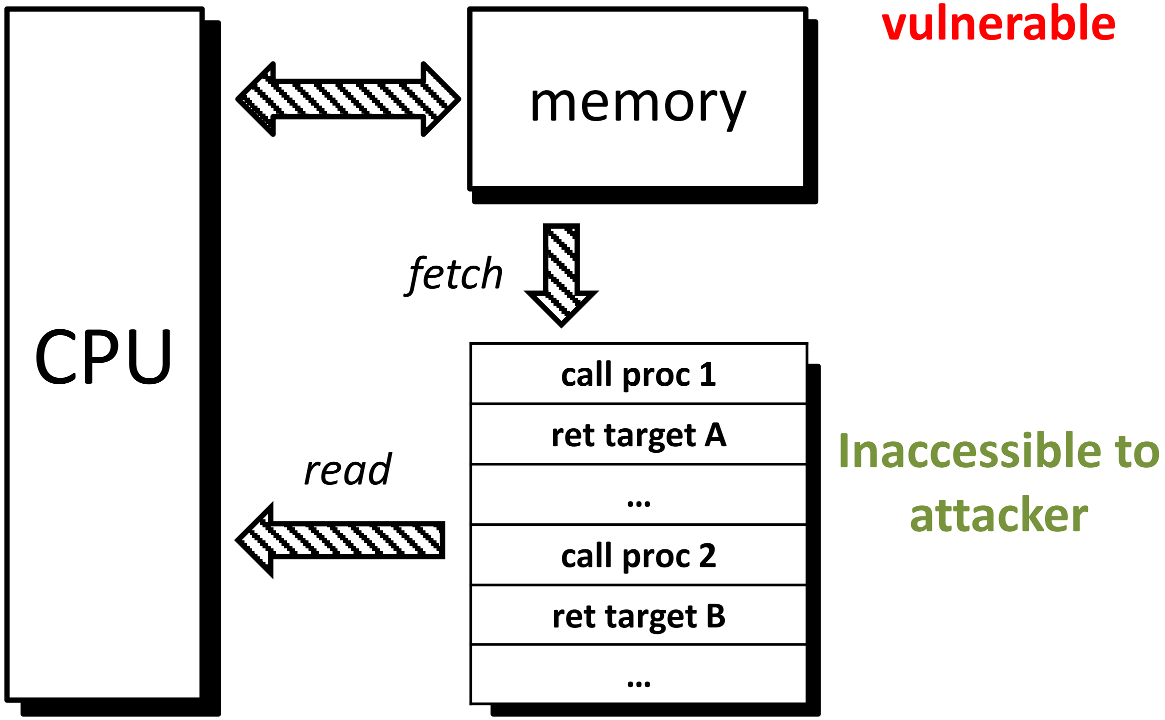
Procedure 1 recursively calls itself

Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

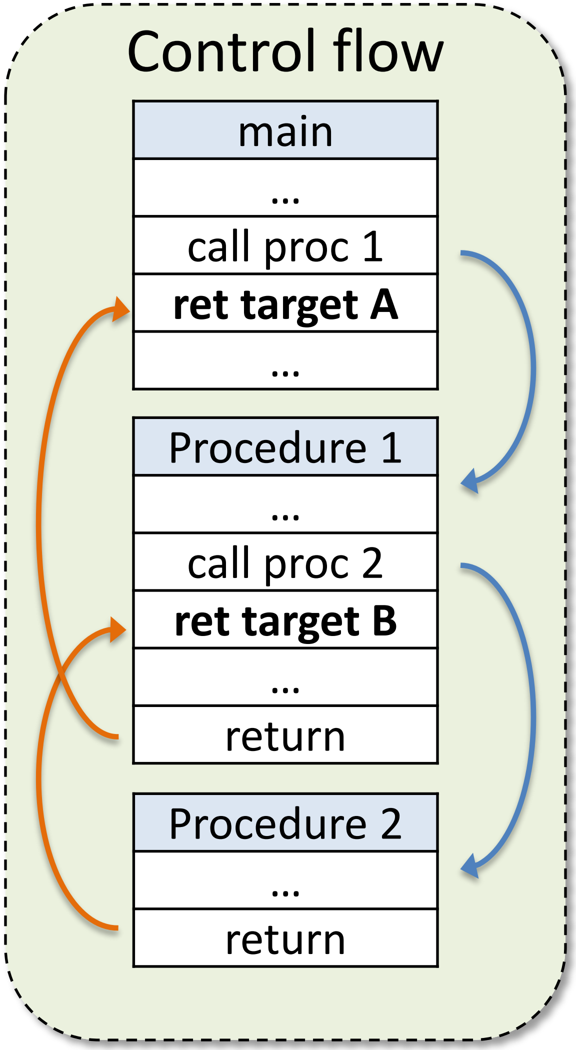
Event 2: I-cache Miss of Return Target

- Modern processors use I-cache to speed up instruction fetch
- Usually very low miss rate due high locality



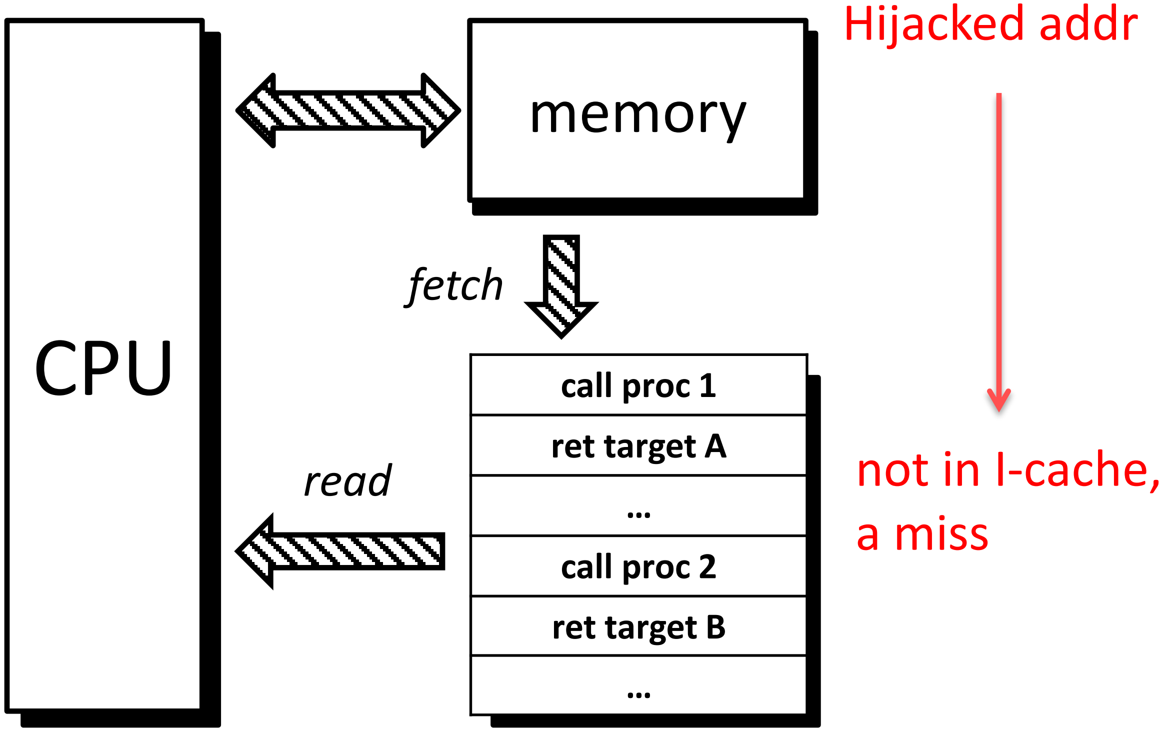
vulnerable

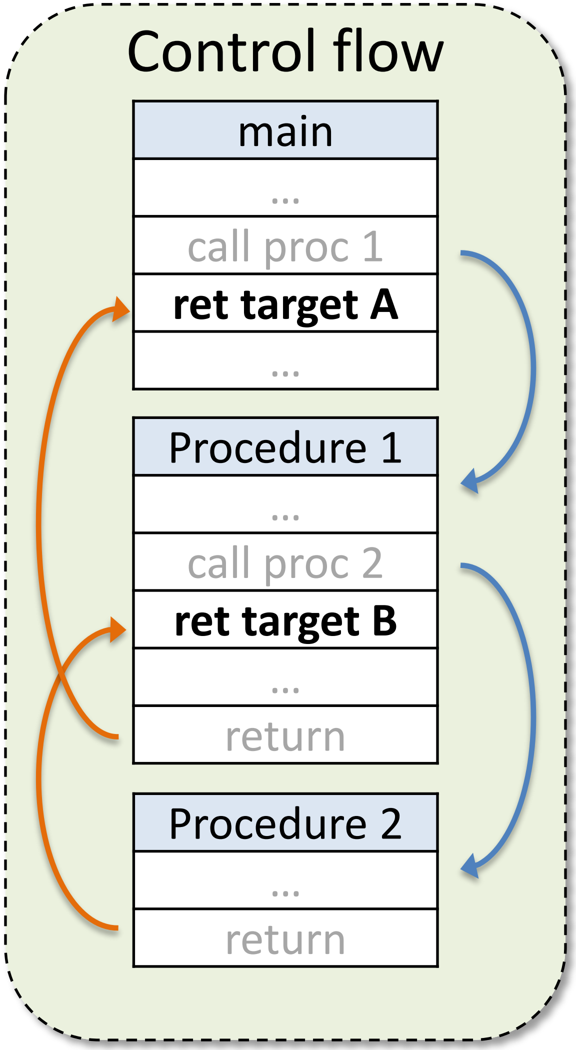
Inaccessible to attacker



Event 2: I-cache Miss of Return Target

- **Can be used to detect BOF attack:** the malicious return target is not in I-cache

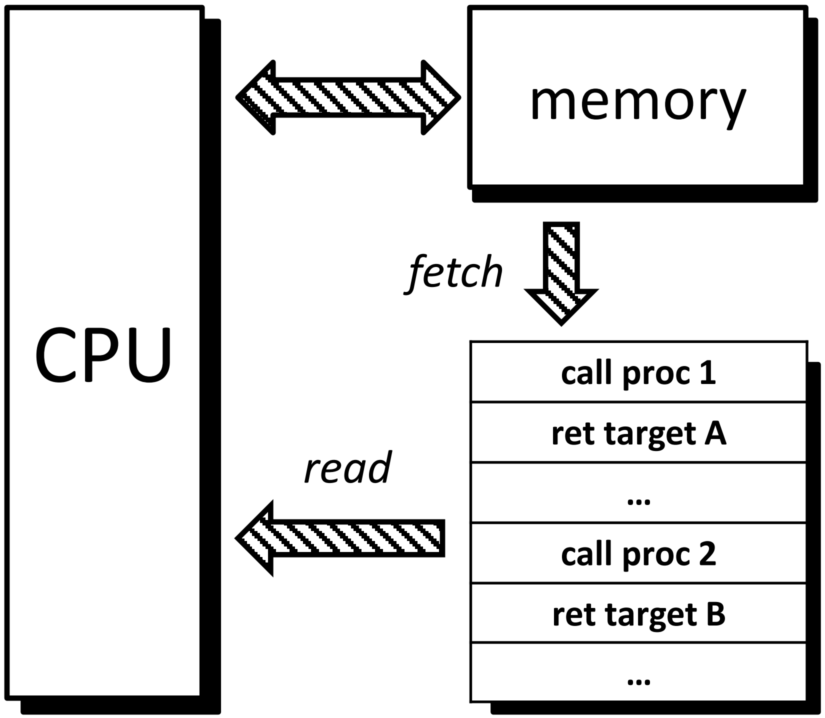




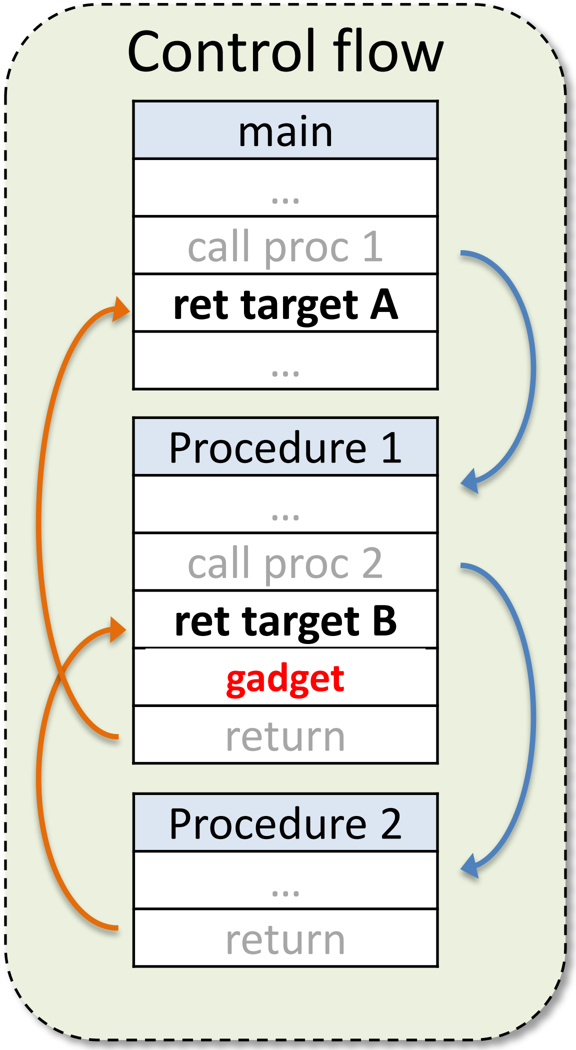
Event 2: Possible **False Negatives**

An attack can bypass detection if the malicious return target is in I-cache

- E.g., for ROP, gadgets may be recently accessed and hence placed in I-cache



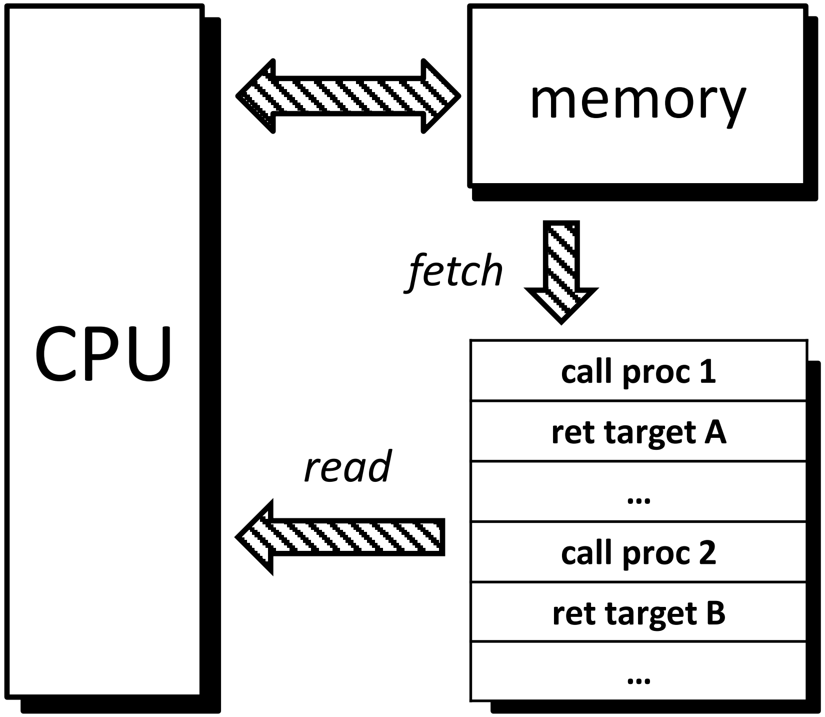
I-Cache hit!
False negative



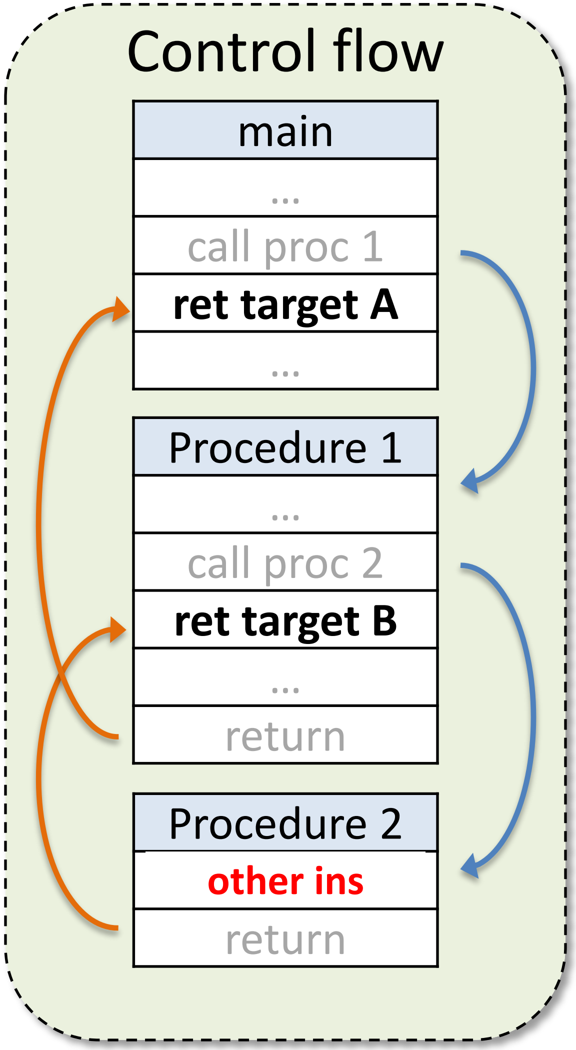
Event 2: Possible **False Positives**

When there is no attack, I-cache miss occurs if a valid return address is not in I-cache

- Either not fetched, or replaced by other instructions



I-cache miss!
False positive

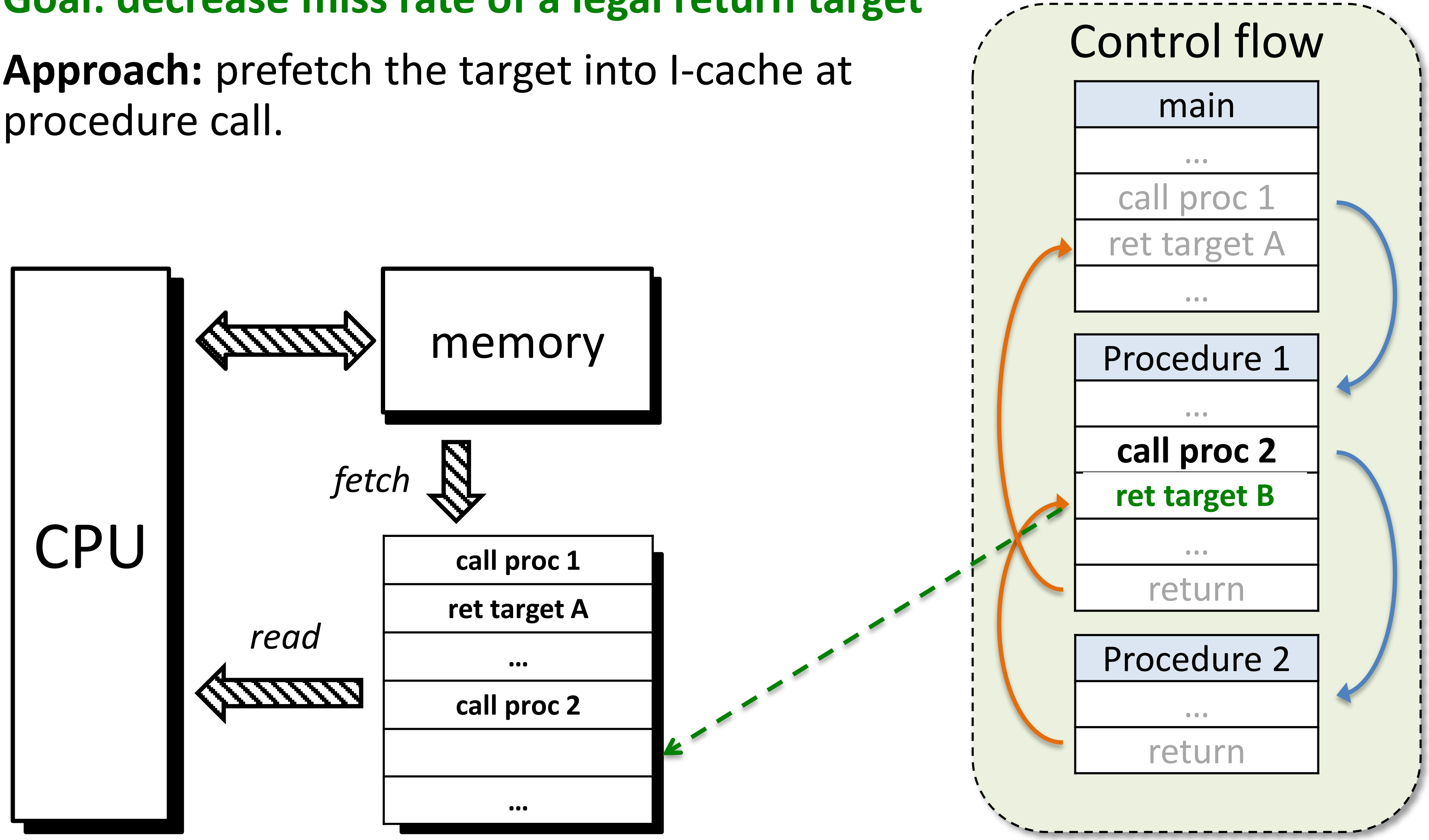


UNIVERSITY *of* DELAWARE

Event 2: Enhance I-cache with **Prefetching**

Goal: decrease miss rate of a legal return target

Approach: prefetch the target into I-cache at procedure call.

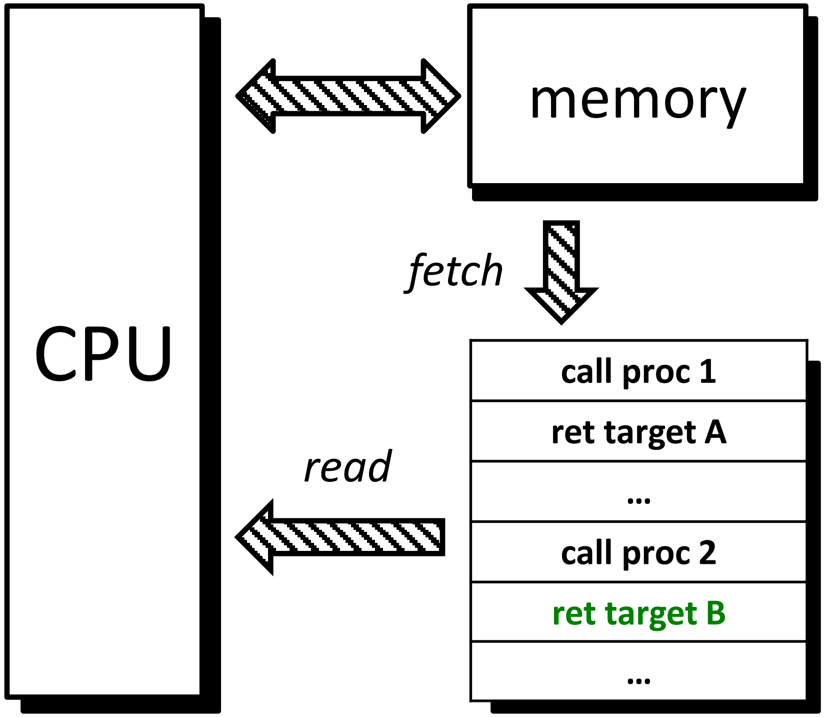


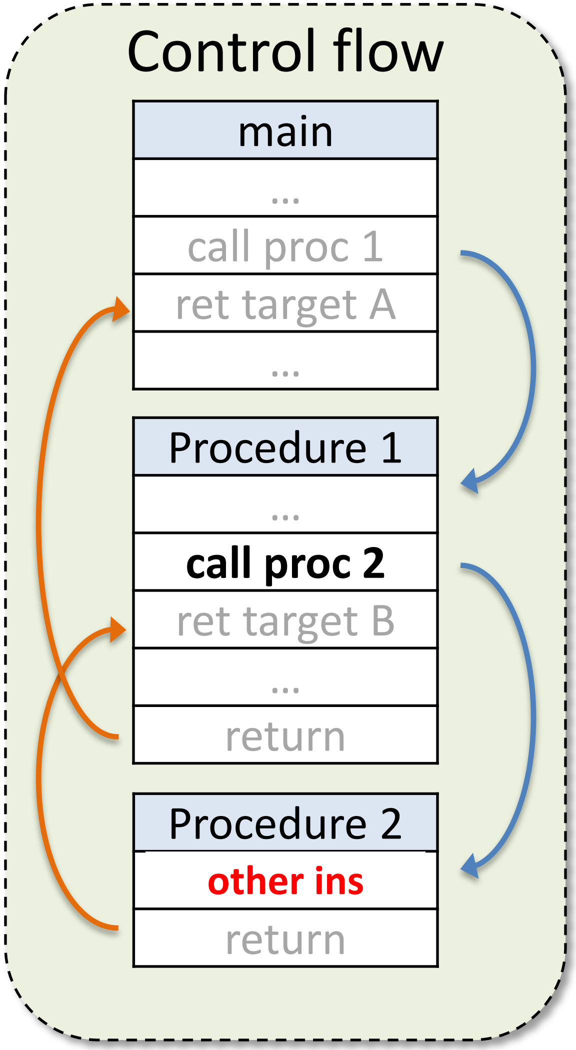
Event 2: Enhance I-cache with **Prefetching**

Goal: decrease miss rate of a legal return target

Approach: prefetch the target into I-cache at procedure call.

However, this target may still being replaced by other instructions before being used.

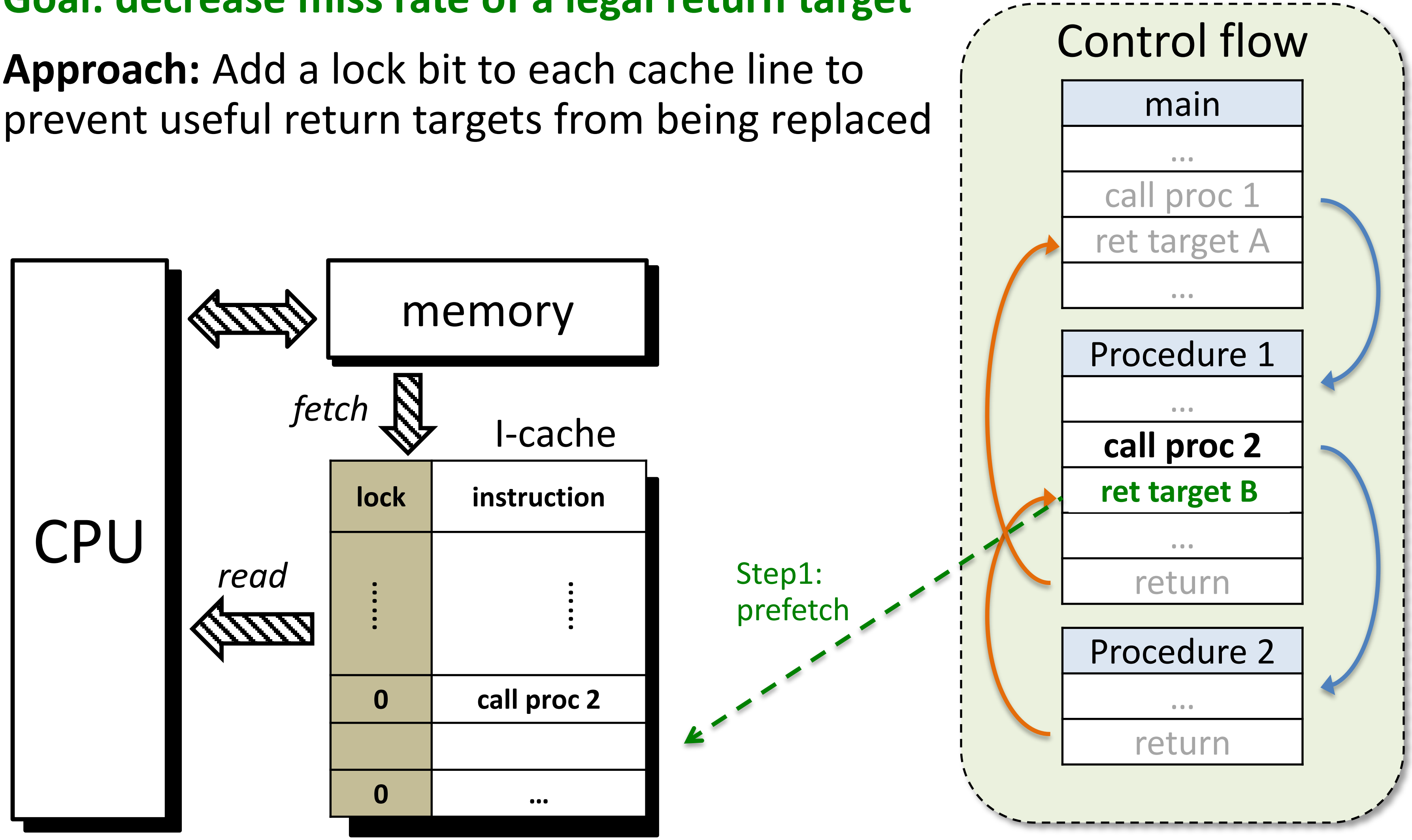




Event 2: Enhance I-cache with **Cache-line Locking**

Goal: decrease miss rate of a legal return target

Approach: Add a lock bit to each cache line to prevent useful return targets from being replaced

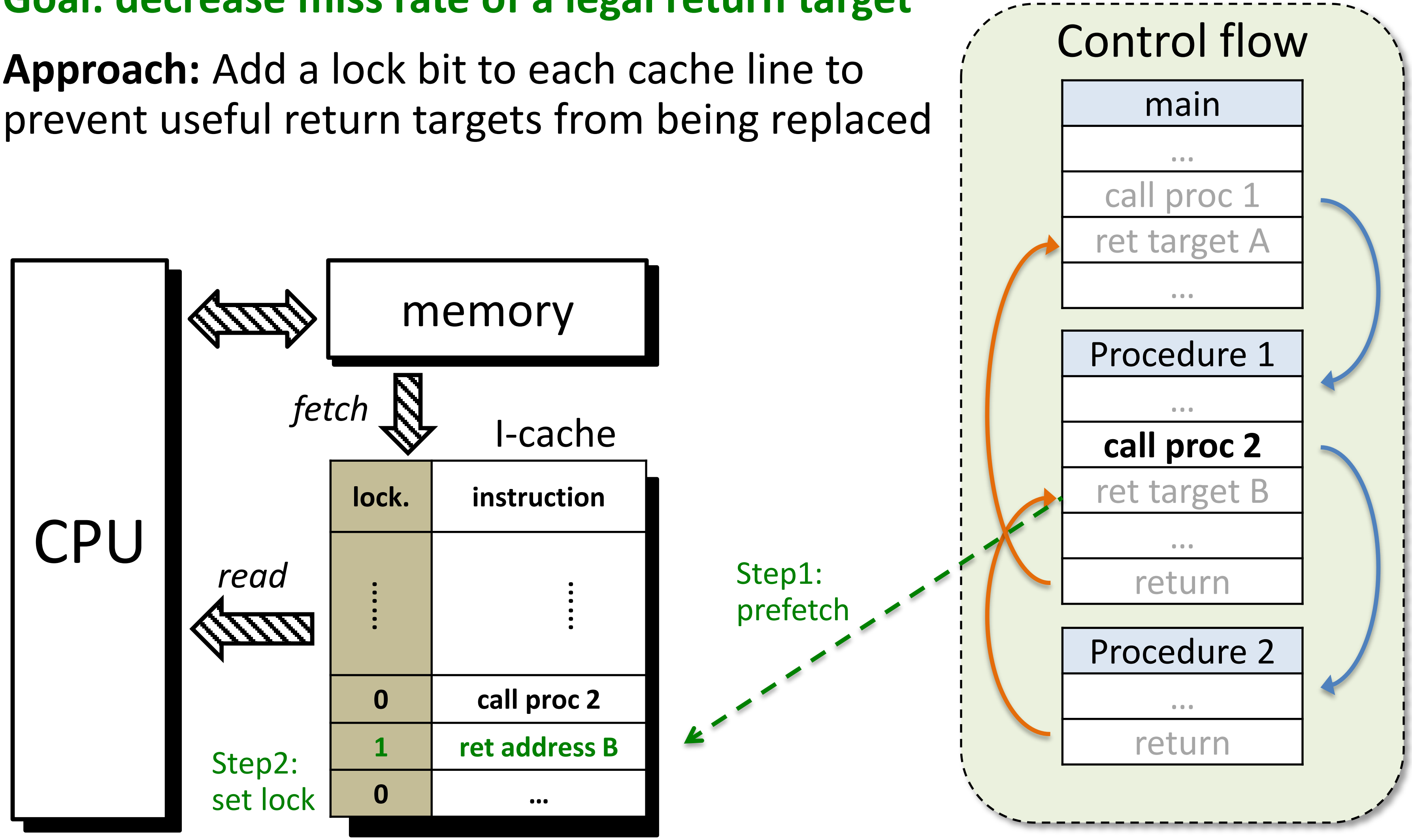


lock	instruction
⋮	⋮
0	call proc 2
0	…

Event 2: Enhance I-cache with **Cache-line Locking**

Goal: decrease miss rate of a legal return target

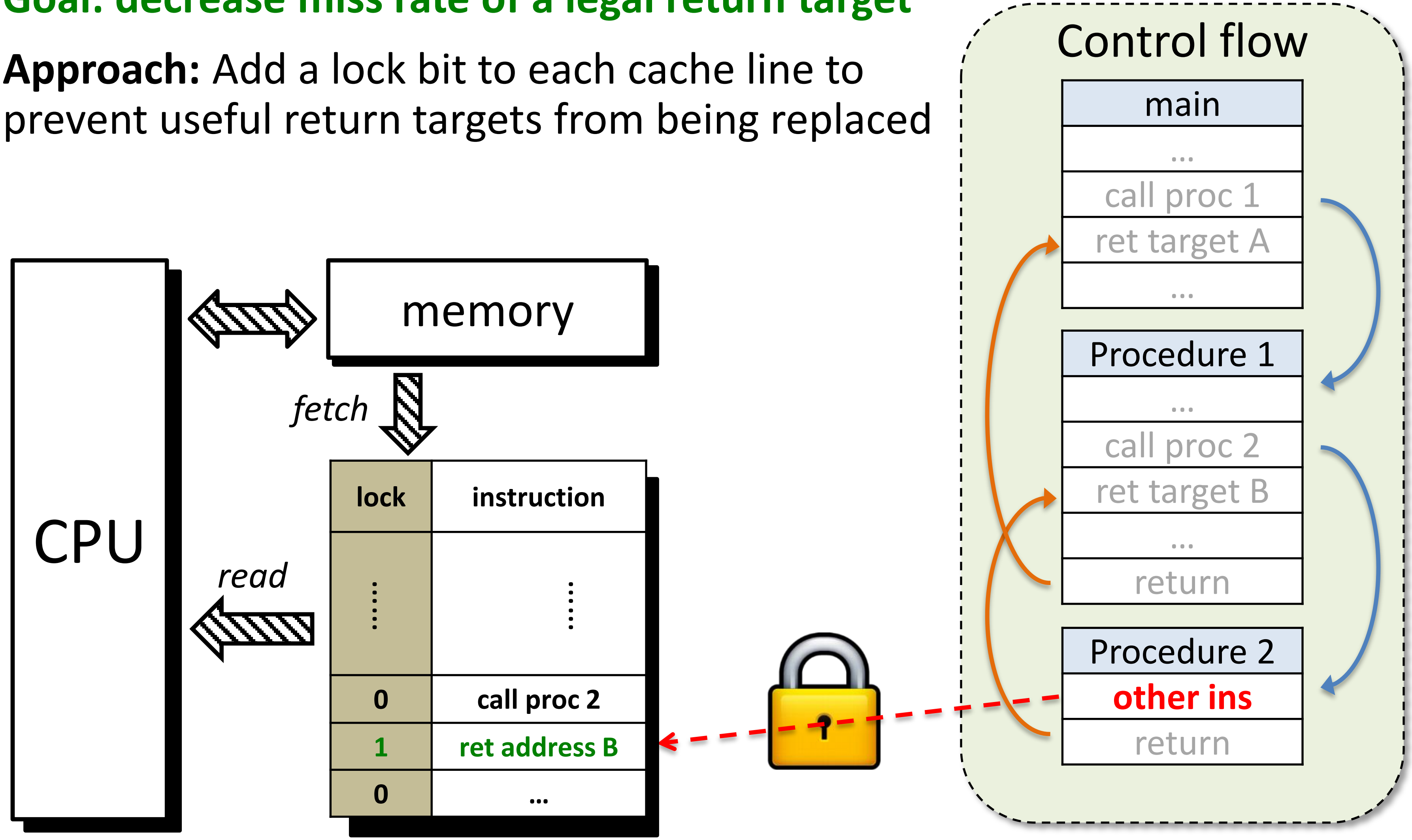
Approach: Add a lock bit to each cache line to prevent useful return targets from being replaced



Event 2: Enhance I-cache with **Cache-line Locking**

Goal: decrease miss rate of a legal return target

Approach: Add a lock bit to each cache line to prevent useful return targets from being replaced

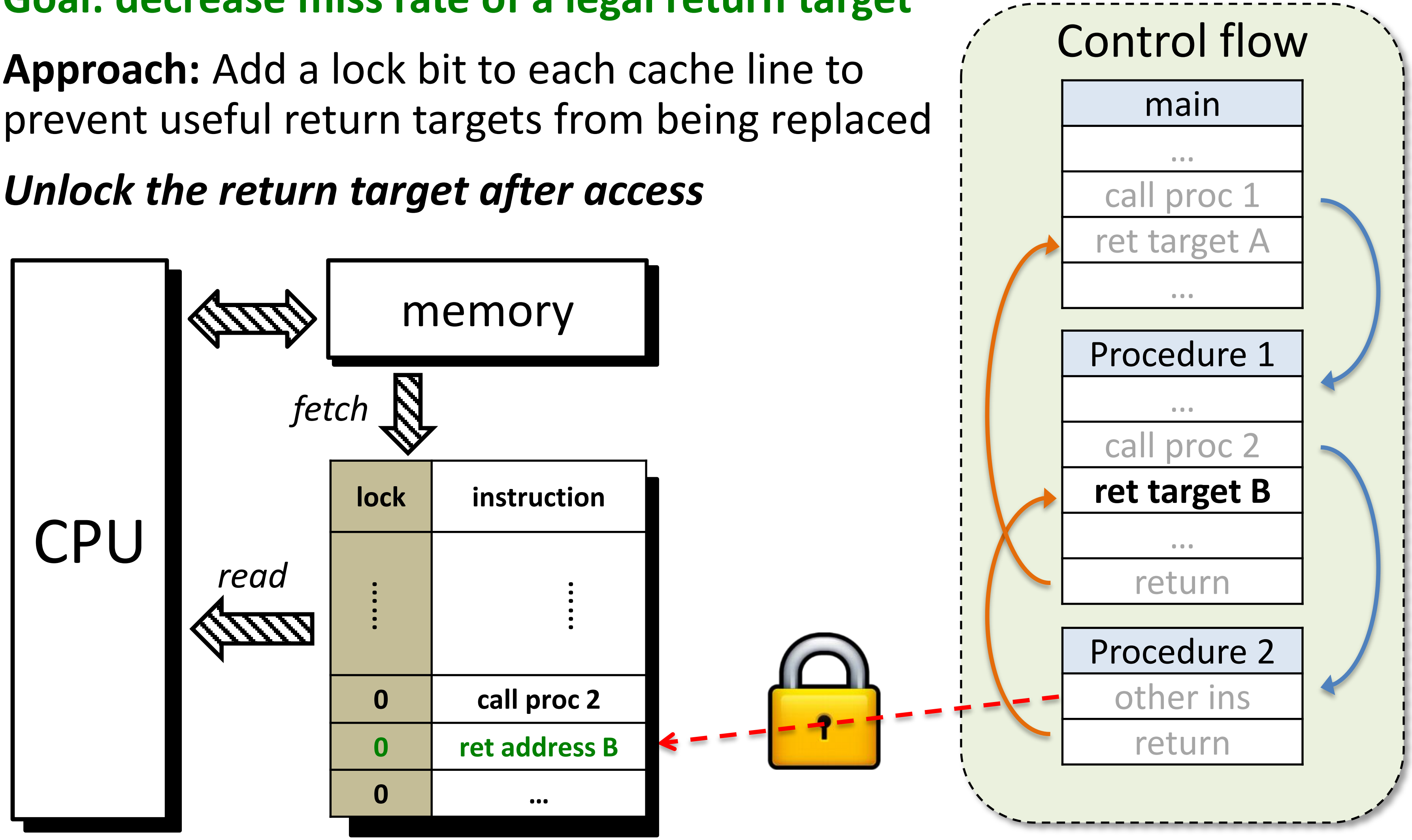


Event 2: Enhance I-cache with **Cache-line Locking**

Goal: decrease miss rate of a legal return target

Approach: Add a lock bit to each cache line to prevent useful return targets from being replaced

Unlock the return target after access



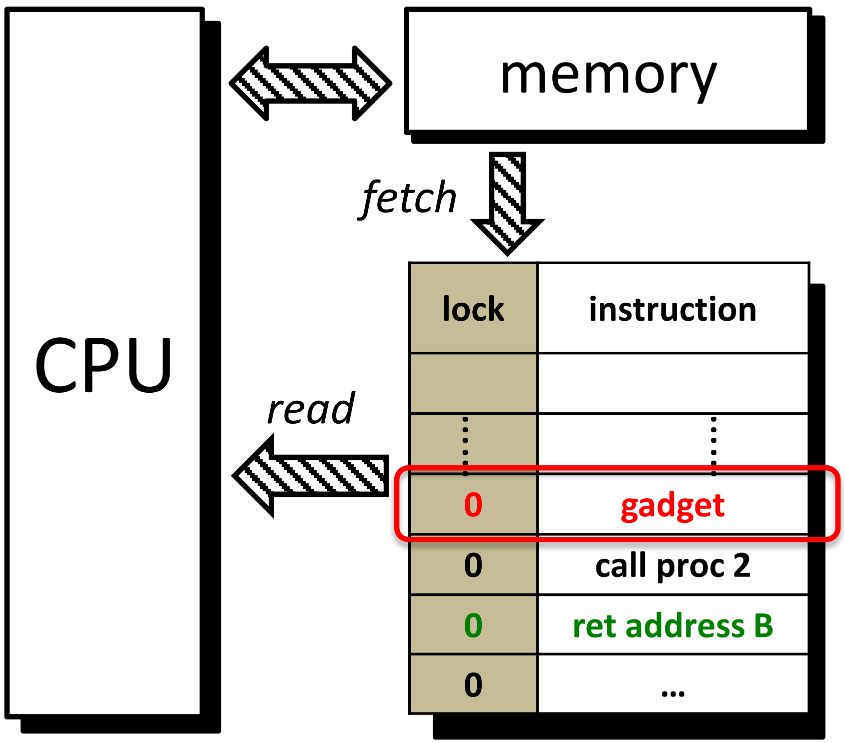
lock	instruction
⋮	⋮
0	call proc 2
0	ret address B
0	…

Event 2: Enhance I-cache with **Cache-line Locking**

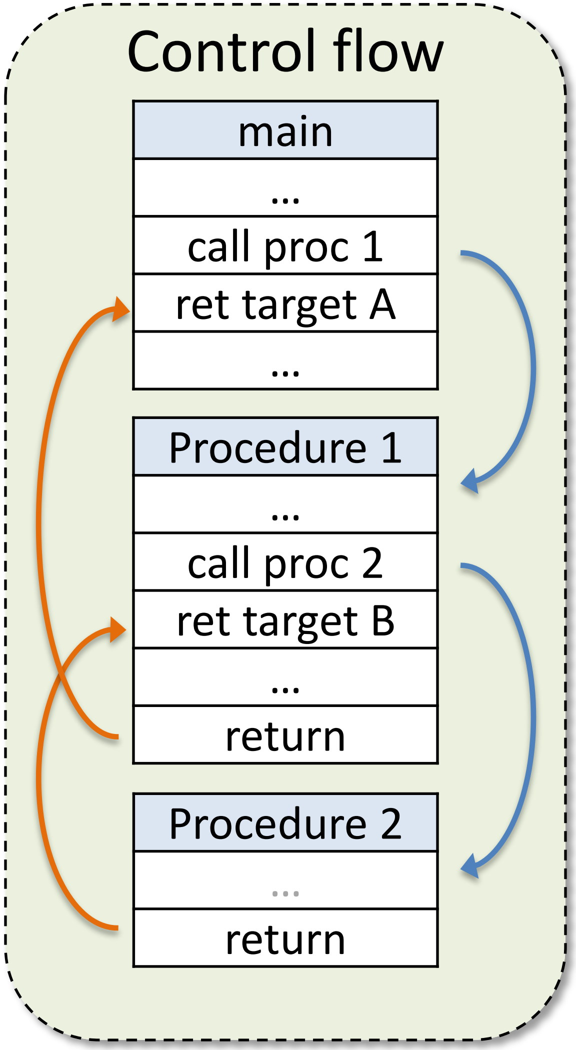
Cache locking also reduces false negative rate!

- E.g., for ROP, even if a gadget is in I-cache, its lock will not be set

return to: gadget



lock	instruction
⋮	⋮
0	gadget
0	call proc 2
0	ret address B
0	…



Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

Answers to the Three Critical Questions

1. Which events to monitor?

- Event 1: RAS mis-prediction of return addresses
- Event 2: return target missed in I-cache **OR** return target not locked in I-cache

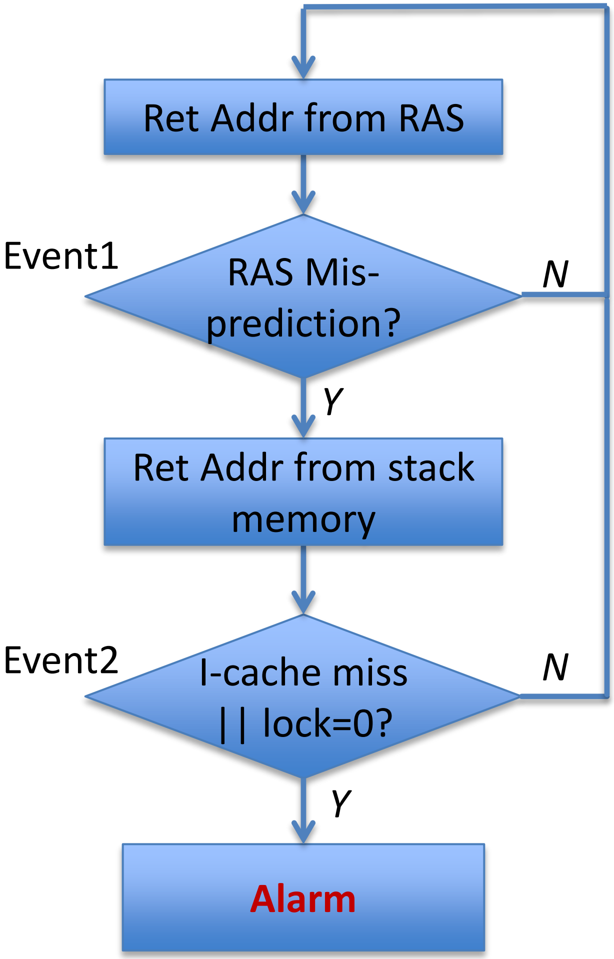
2. How do further improve prediction accuracy?

HW Enhancement	False positive	False negative
RAS call counter	↓	
I-cache prefetching	↓	
I-cache locking	↓	↓

3. What is the alarm condition?

- **Jointly monitoring both events**

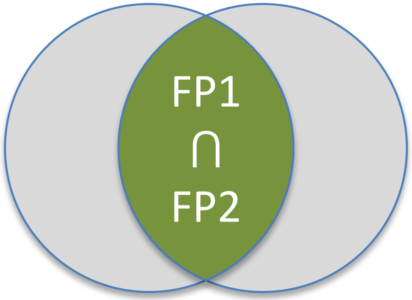
Jointly Monitoring Both Events



Monitoring Flowchart

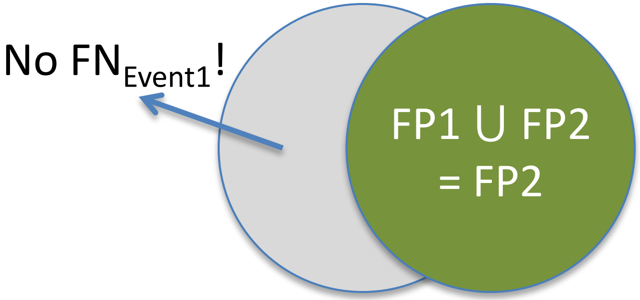
False Positive (FP): no attack, false alarm
False Negative (FN): has attack, no alarm

$$FP_{joint} = FP_{Event1} \cap FP_{Event2}$$



Reduced!

$$FN_{joint} = FN_{Event1} \cup FN_{Event2}$$



Unchanged!

Outline

- Background: stack buffer overflow
- Motivation: need for low cost & accurate detection scheme
- Micro-architectural event monitoring:
  - Event 1: Return Address Stack (RAS) mis-prediction
  - Event 2: instruction cache misses
  - Alarm condition
- Experimental evaluation

Experimental Setup

- ***Schemes to evaluate***
  1. Original benchmark (no protection)
  2. Monitoring only RAS
  3. Jointly monitoring both RAS and I-cache
  4. Jointed events with hardware enhancement
- ***Attacks simulated***
  Code injection: malicious code in stack segment
  Return-to-libc: malicious code in C library
  ROP: chain malicious gadgets in original code
- ***Benchmarks*** From SPEC2000, MiBench, and Mediabench
- ***Simulator*** **SimpleScalar:**
  cycle accurate, micro-architectural level

RAS size	8 entries
RAS call counter	3 bits
L1 I-cache sets	512
L1 I-cache block size	32
L1 I-cache associativity	2

Reduction in False Positive Count

- Results collected by running detection scheme without performing attack

	RAS-only	Jointed events		Jointed+HW enhanced	
	Count	Count	Reduction	Count	Reduction
art	14	4	71%	0	100%
crafty	96174	15259	84%	3944	96%
dijkstra	781	0	100%	0	100%
fft	22	11	50%	0	100%
galgel	111	38	66%	1	99%
gcc	143981	38060	74%	8244	94%
gzip	32	3	91%	0	100%
jpeg	17	9	47%	0	100%
mcf	13728	2	100%	97	99%
mpeg2	28	13	54%	0	100%
Average			**74%**		**99%**

- Both the jointly monitoring strategy and the hardware enhancements effectively reduce false positives

False Negative Rate Evaluation

Methodology

Run detection scheme together with the benchmark, perform attacks randomly in the following way:

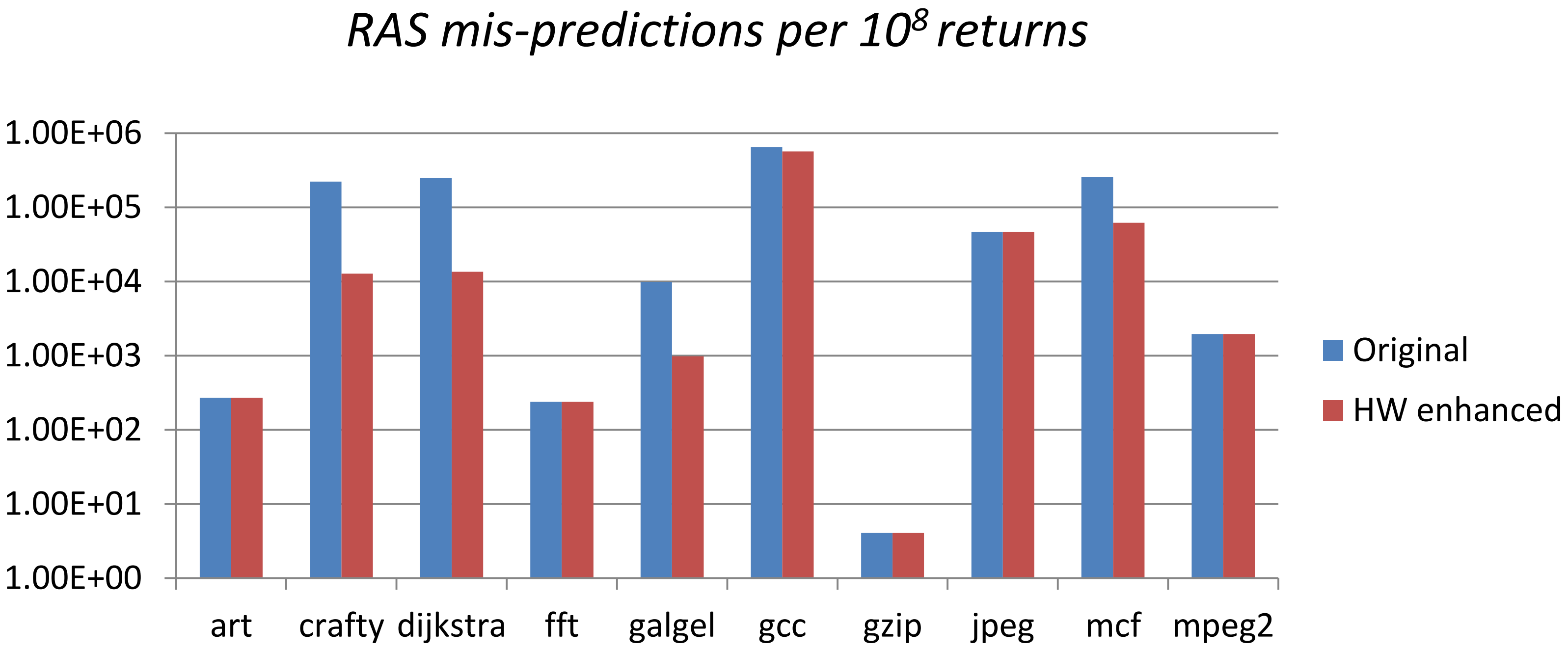
- **Code injection**
  Hijacked return address = randomly picked address in **stack** segment
- **return-to-libc**
  Hijacked return address = randomly picked address in **code** segment except for the current program text
- **return-oriented programming (ROP)**
  Chain of gadgets = 31 addresses randomly picked from program code
  Perform detection on these gadget heads one by one,
  Report an attack if any gadget head triggers the alarm condition.

False Negative Rate Evaluation

	Code injection	Return-to-libc	Return-oriented programming		
			RAS-only	Joint-event	Jointed+ HW enhanced
art	0	0	0	5.11E-28	4.49E-101
crafty	0	0	0	1.23E-35	7.07E-103
dijkstra	0	0	0	4.36E-21	1.07E-77
fft	0	0	0	2.93E-14	1.10E-92
galgel	0	0	0	1.09E-49	0
gcc	0	0	0	5.76E-56	9.84E-117
gzip	0	0	0	6.22E-26	7.39E-114
jpeg	0	0	0	1.50E-34	0
mcf	0	0	0	3.74E-23	1.24E-94
mpeg2	0	0	0	2.95E-23	1.30E-79

- Except for ROP, no other false negatives observed
- The false negative rates of ROP are extremely low since it is very hard for all the gadgets to escape from triggering the alarm

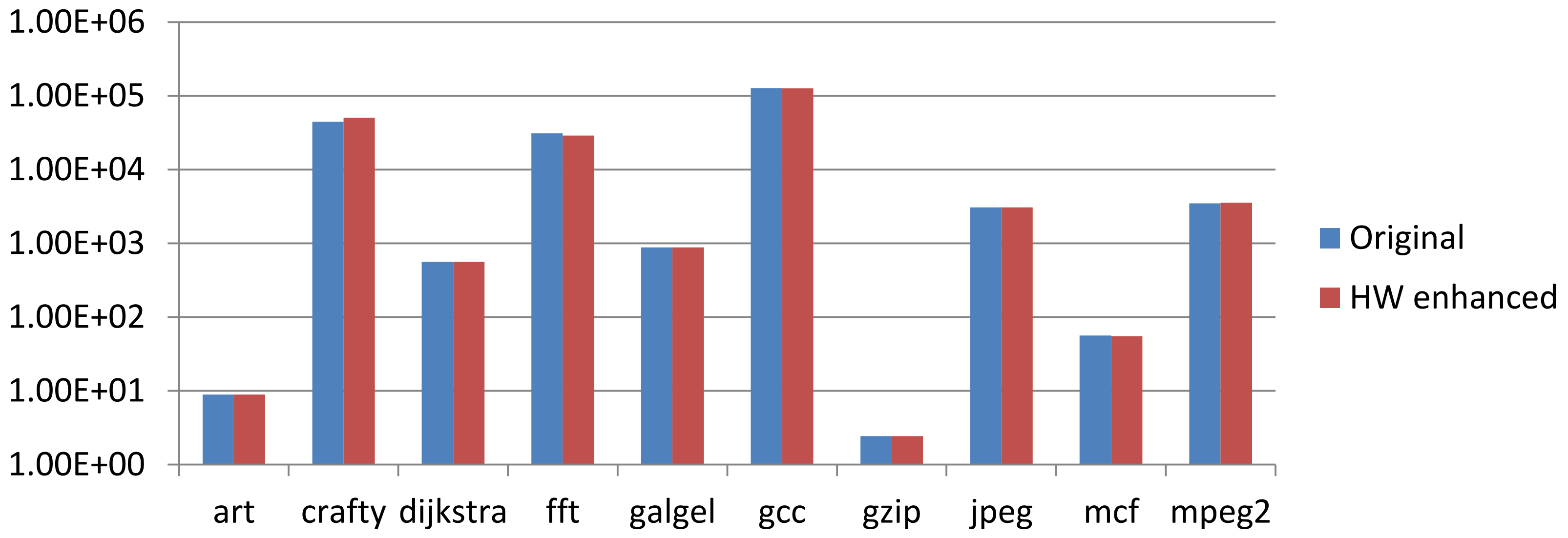
Impact on RAS Mis-prediction Rates



- Programs originally with many RAS mis-predictions benefit significantly from the call counter – these mis-predictions are caused by recursive procedure calls

Impact on I-cache Miss Rate

I-cache misses per 10^7 instructions



- Prefetching and cache-line locking may:
  - reduce miss rate – misses of useful return targets are eliminated
  - increase miss rate – locked lines may have conflicts with hot lines
- Overall, the two enhancements have negligible impact on I-cache misses

Summary

- Stack buffer overflow is a common cyber security vulnerability
- Our approach: monitor micro-architectural events at runtime to detect such attacks.
  - Event 1: RAS mis-prediction of a return address
  - Event 2: I-cache miss of a return target **OR** the target is not locked in I-cache
- Our approach works well in embedded systems:
  - Low cost: hardware-based scheme requires little runtime overhead; reuse performance-driven enhancements for security purpose
  - High accuracy: jointly event monitoring flow and the proposed hardware enhancements eliminate most of the false positives and false negatives

Thank you!

Questions?