Summer School on Trusted and Trustworthy Computing
24 September 2015, Bukarest, Germany

The Beast in Your Memory: Modern Exploitation Techniques and Defenses

Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi

CASED, Technische Universität Darmstadt

Intel Collaborative Research Institute for Secure Computing at TU Darmstadt, Germany

http://trust.cased.de/

Motivation

- Sophisticated, complex
- Various of different developers
- Native Code

Large attack surface for runtime attacks

[Úlfar Erlingsson, Low-level Software Security: Attacks and Defenses, TR 2007]

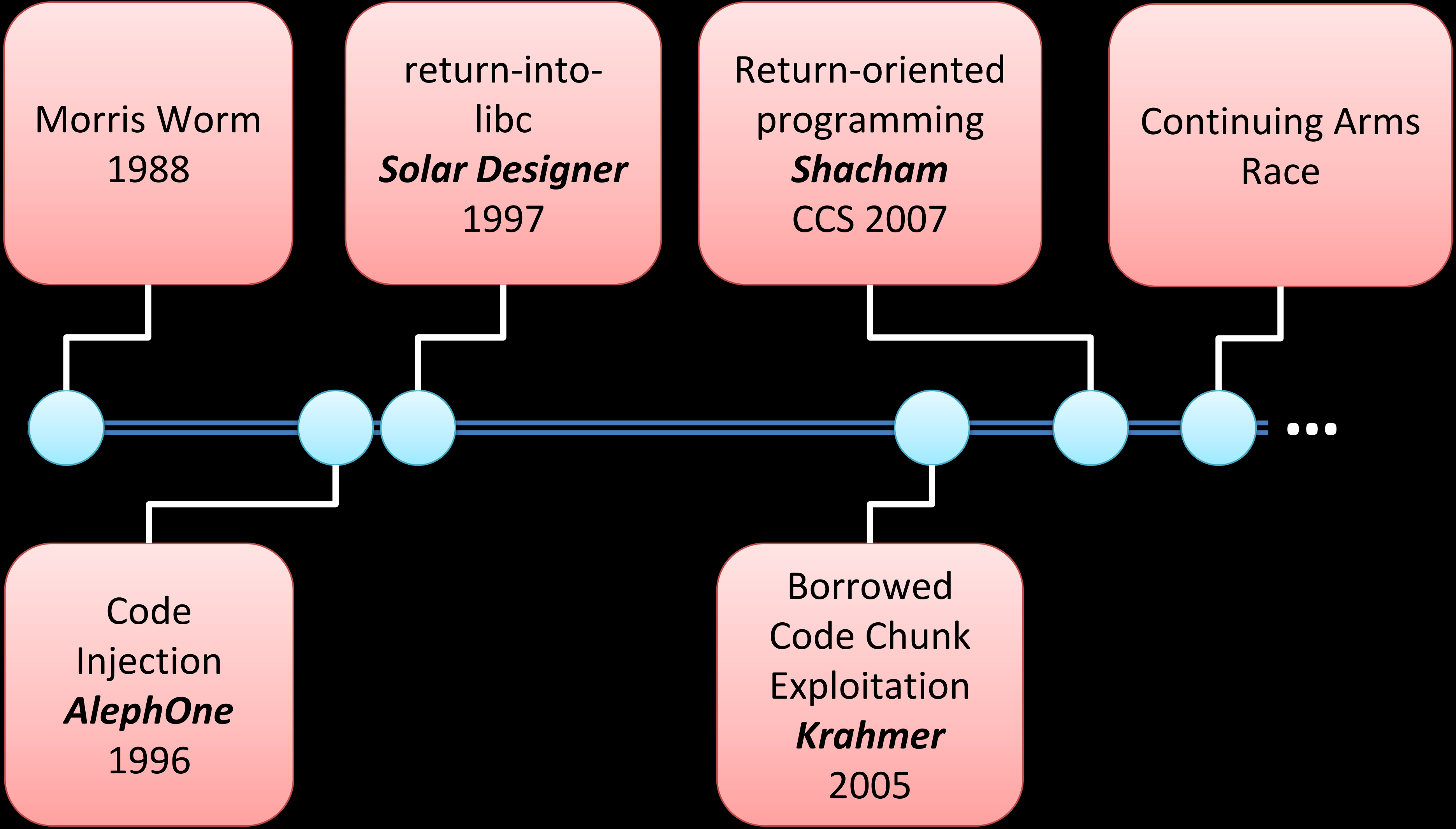
Introduction

- Vulnerabilities
  - Programs continuously suffer from program bugs, e.g., a buffer overflow
  - Memory errors
  - CVE statistics; zero-day

In this tutorial

- Runtime Attack
  - Exploitation of program vulnerabilities to perform malicious program actions
  - Control-flow attack; runtime exploit

Three Decades of Runtime Attacks



Morris Worm
1988

Code Injection
AlephOne
1996

return-into-libc
Solar Designer
1997

Borrowed Code Chunk Exploitation
Krahmer
2005

Return-oriented programming
Shacham
CCS 2007

Continuing Arms Race

Are these attacks relevant?

Relevance and Impact

High Impact of Attacks

- Web browsers repeatedly exploited in pwn2own contests
- Zero-day issues exploited in Stuxnet/Duqu [Microsoft, BH 2012]
- iOS jailbreak

Industry Efforts on Defenses

- Microsoft EMET (Enhanced Mitigation Experience Toolkit) includes a ROP detection engine
- Microsoft Control Flow Guard (CFG) in Windows 10
- Google's compiler extension VTV (vitual table verification)

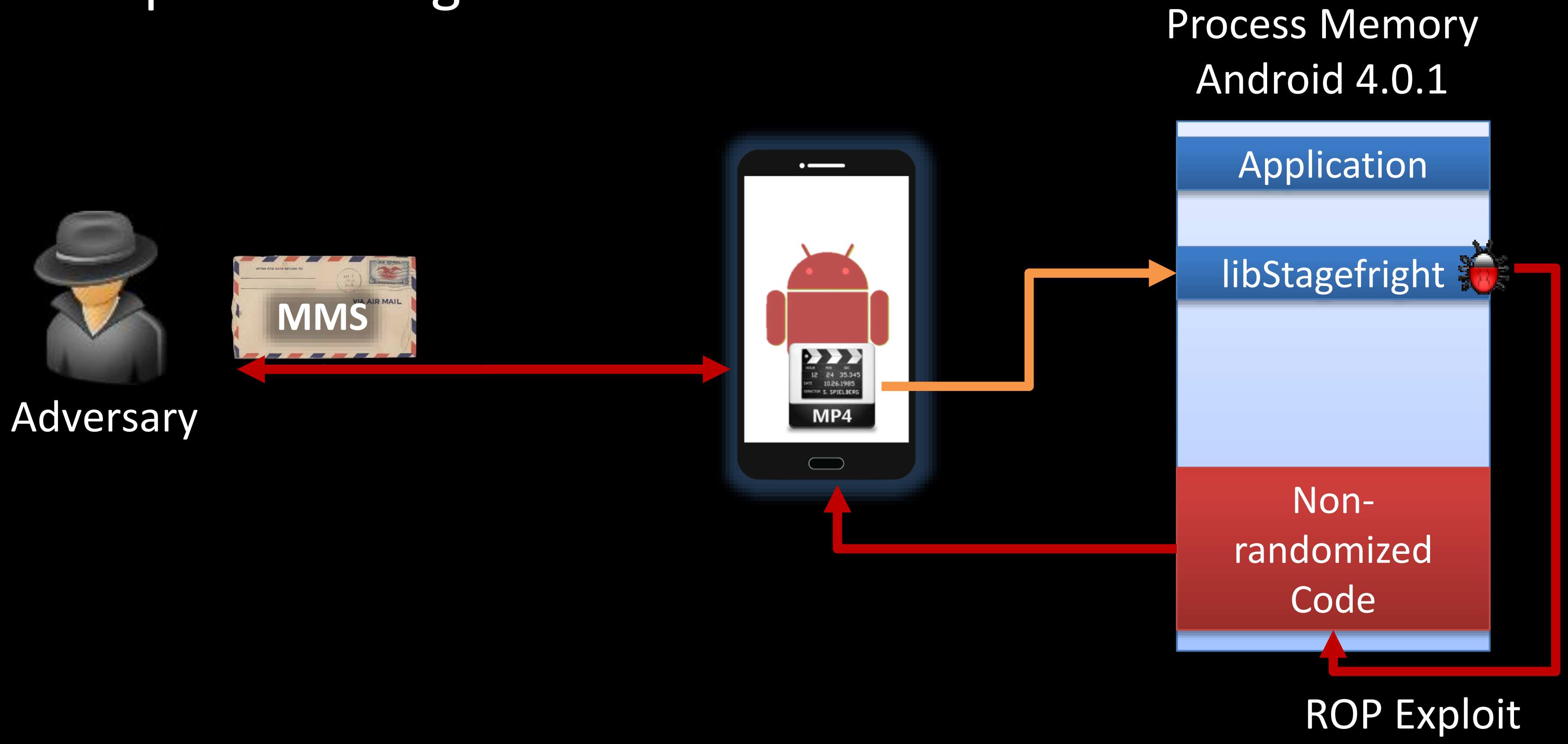
Hot Topic of Research

- A large body of recent literature on attacks and defenses

Stagefright [Drake, BlackHat 2015]

These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices

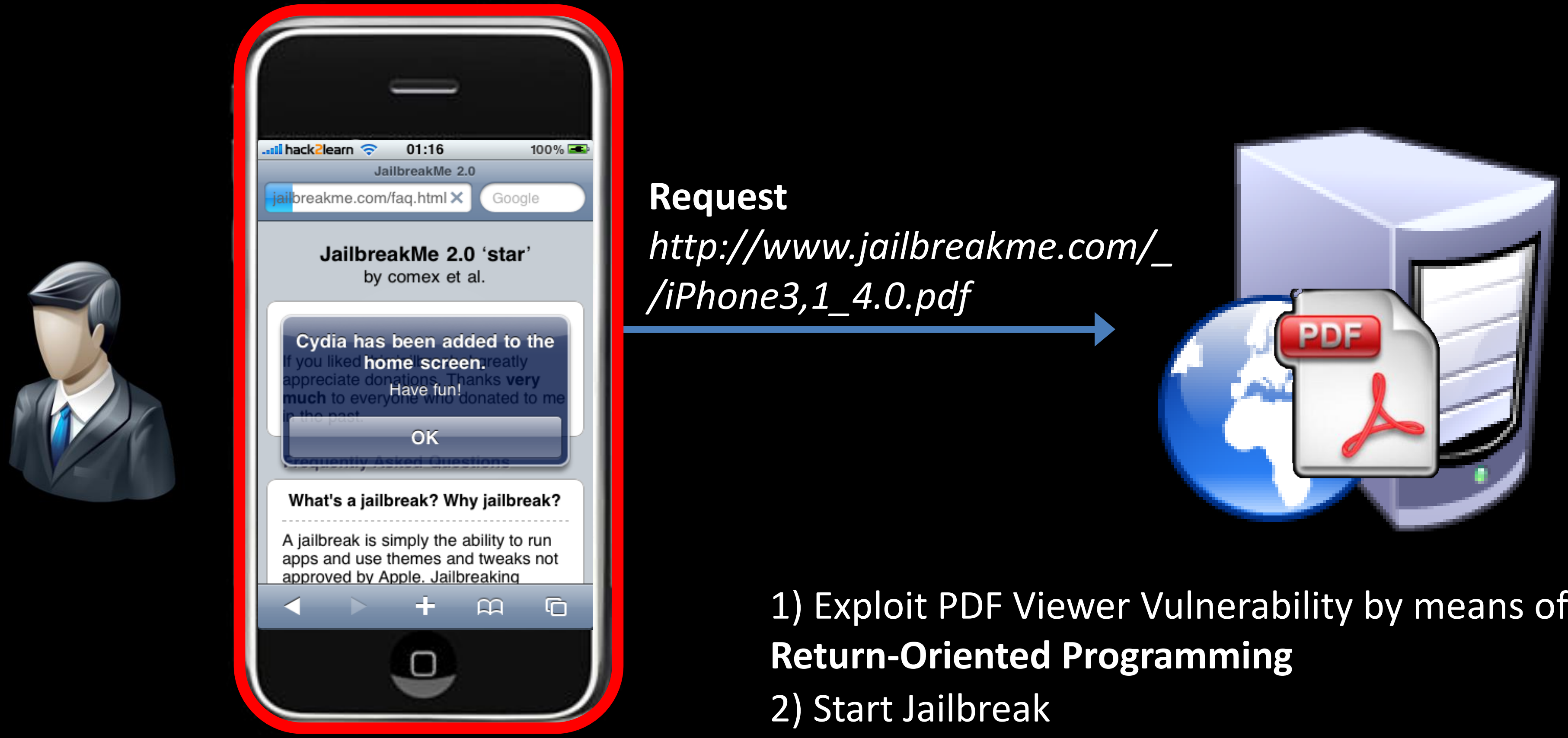
Zimperium Blog



But runtime exploits have also some “good” side-effects

Apple iPhone Jailbreak

Disable signature verification and escalate privileges to root



JailbreakMe 2.0 'star'
by comex et al.

Cydia has been added to the home screen.
Have fun!

Frequently Asked Questions

What's a jailbreak? Why jailbreak?

A jailbreak is simply the ability to run apps and use themes and tweaks not approved by Apple. Jailbreaking

Request
http://www.jailbreakme.com/_
/iPhone3,1_4.0.pdf

1) Exploit PDF Viewer Vulnerability by means of **Return-Oriented Programming**

2) Start Jailbreak

3) Download required system files

4) Jailbreak Done

Outline of This Lecture

BASICS

- What is a runtime attack?
- Why today‘s attacks use code reuse?

CODE-REUSE ATTACKS

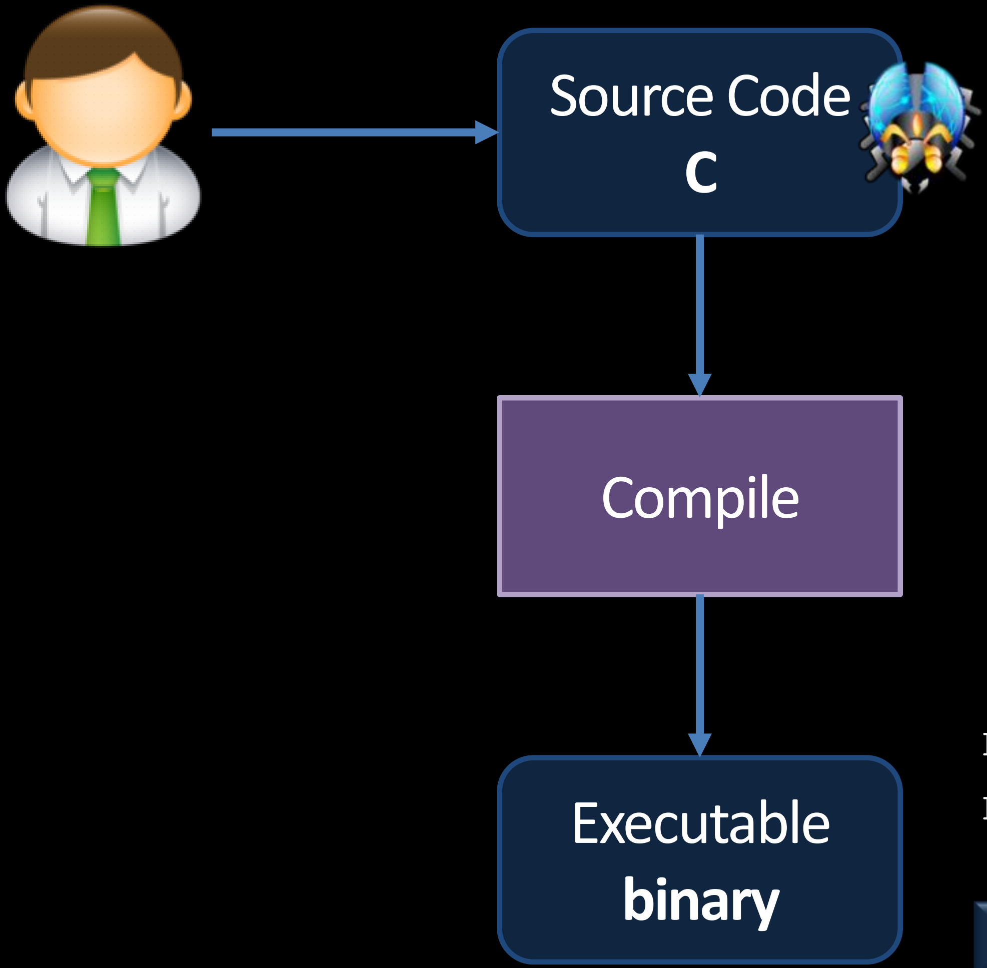
- What is return-oriented programming (ROP) and how does it work?

CURRENT SECURITY RESEARCH

- Can code randomization (ASLR) help?
- How do control-flow integrity (CFI) solutions such as Microsoft EMET or kBouncer aim at preventing ROP?
- Can the latest CFI solutions be bypassed? What‘s next?

BASICS
What is a runtime attack ?

Big Picture: Program Compilation



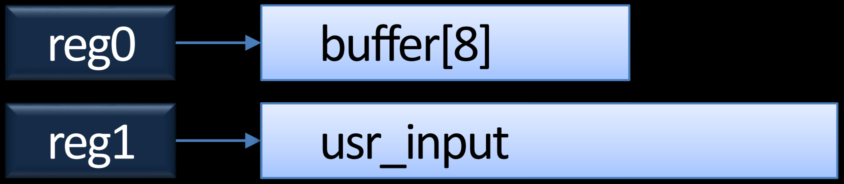
Source Code
C

Compile

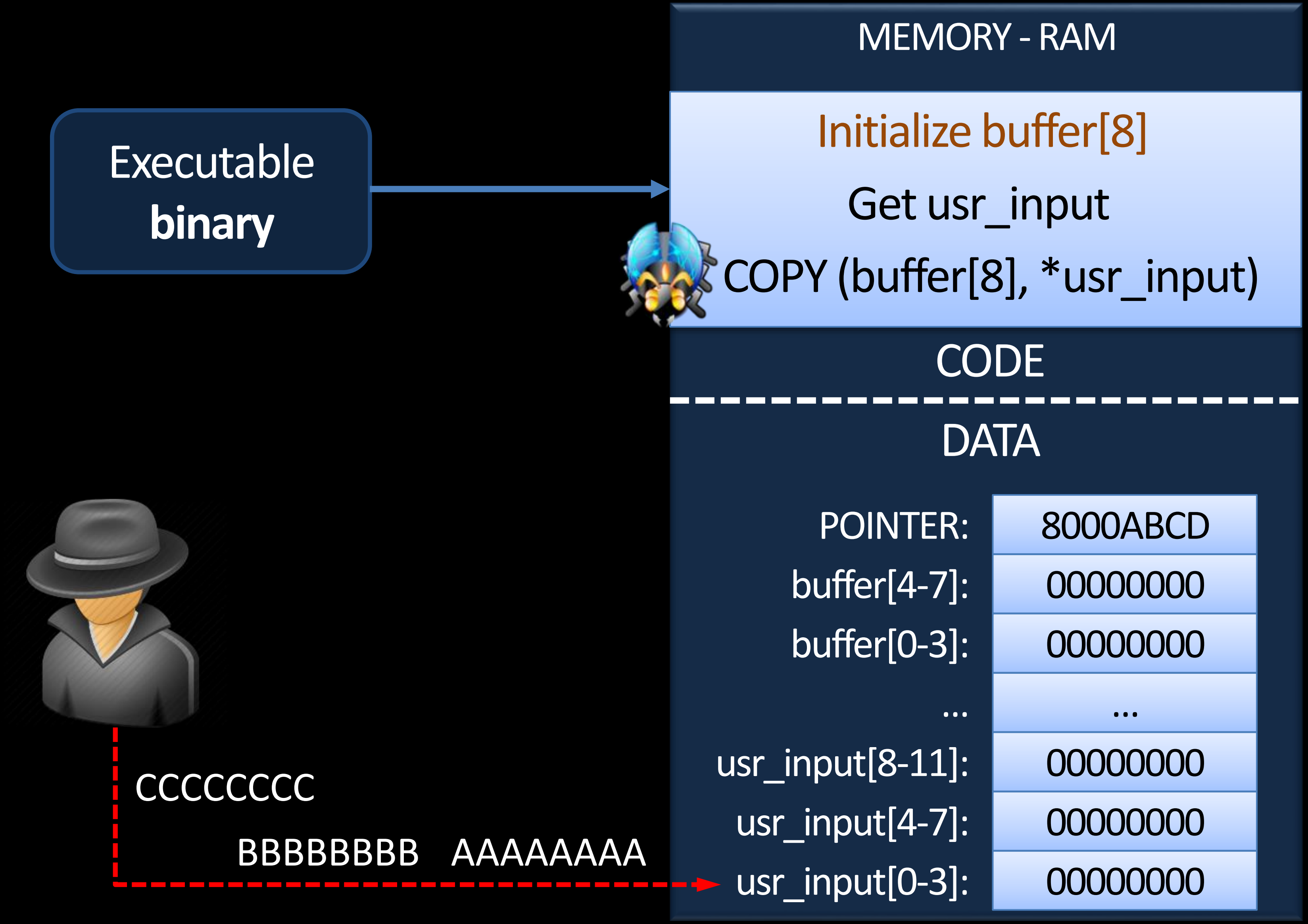
Executable
binary

*COPY (buffer[8], *usr_input)*

```
mov reg0[0-3], reg1[0-3]
mov reg0[4-n], reg1[4-n]
```



Big Picture: Program Execution (1/3)



Executable **binary**

MEMORY - RAM

Initialize buffer[8]

Get usr_input

COPY (buffer[8], *usr_input)

CODE

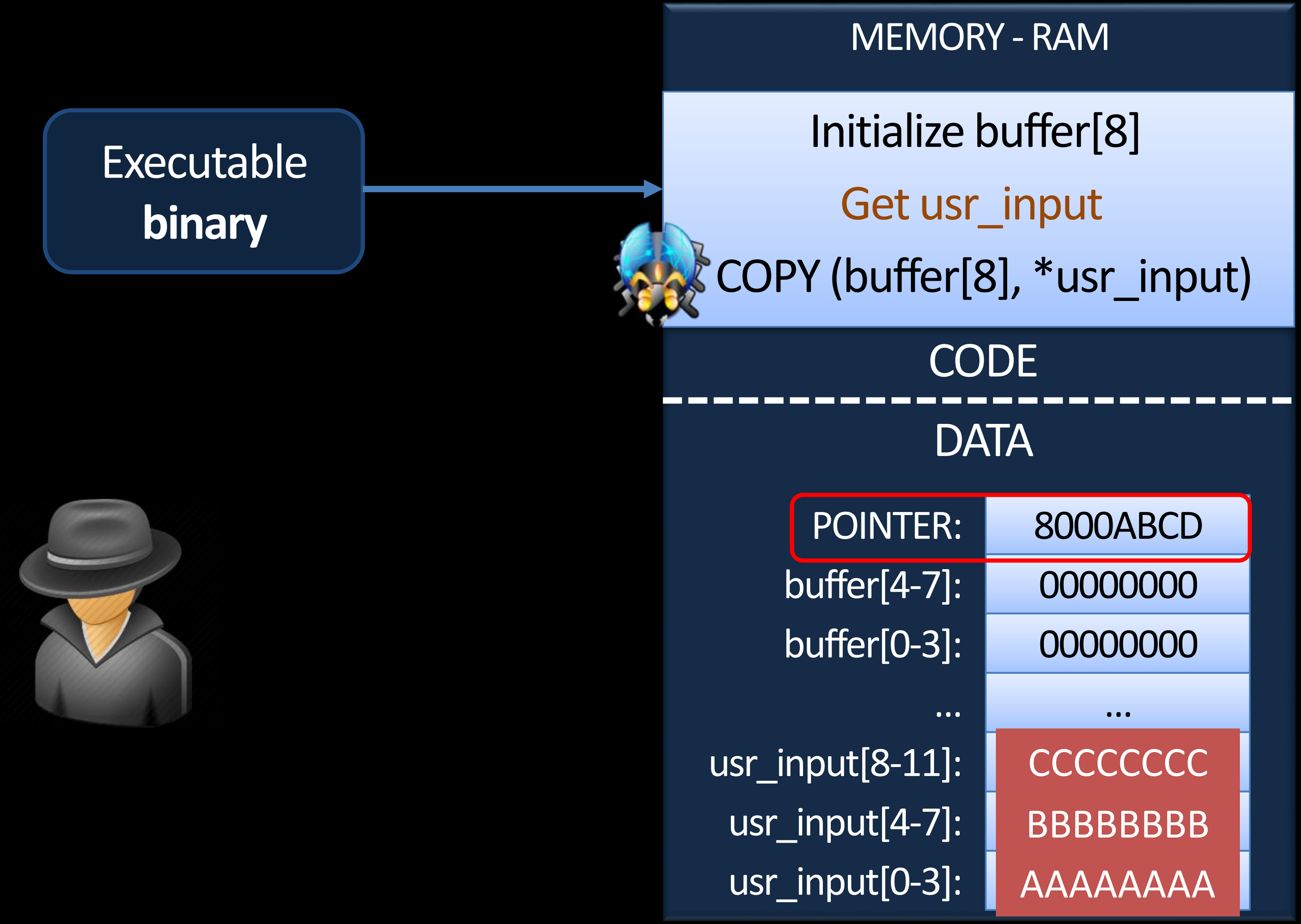
DATA

POINTER:	8000ABCD
buffer[4-7]:	00000000
buffer[0-3]:	00000000
…	…
usr_input[8-11]:	00000000
usr_input[4-7]:	00000000
usr_input[0-3]:	00000000

CCCCCCCC

BBBBBBBB AAAAAAAA

Big Picture: Program Execution (2/3)



Executable **binary**

MEMORY - RAM

Initialize buffer[8]

Get usr_input

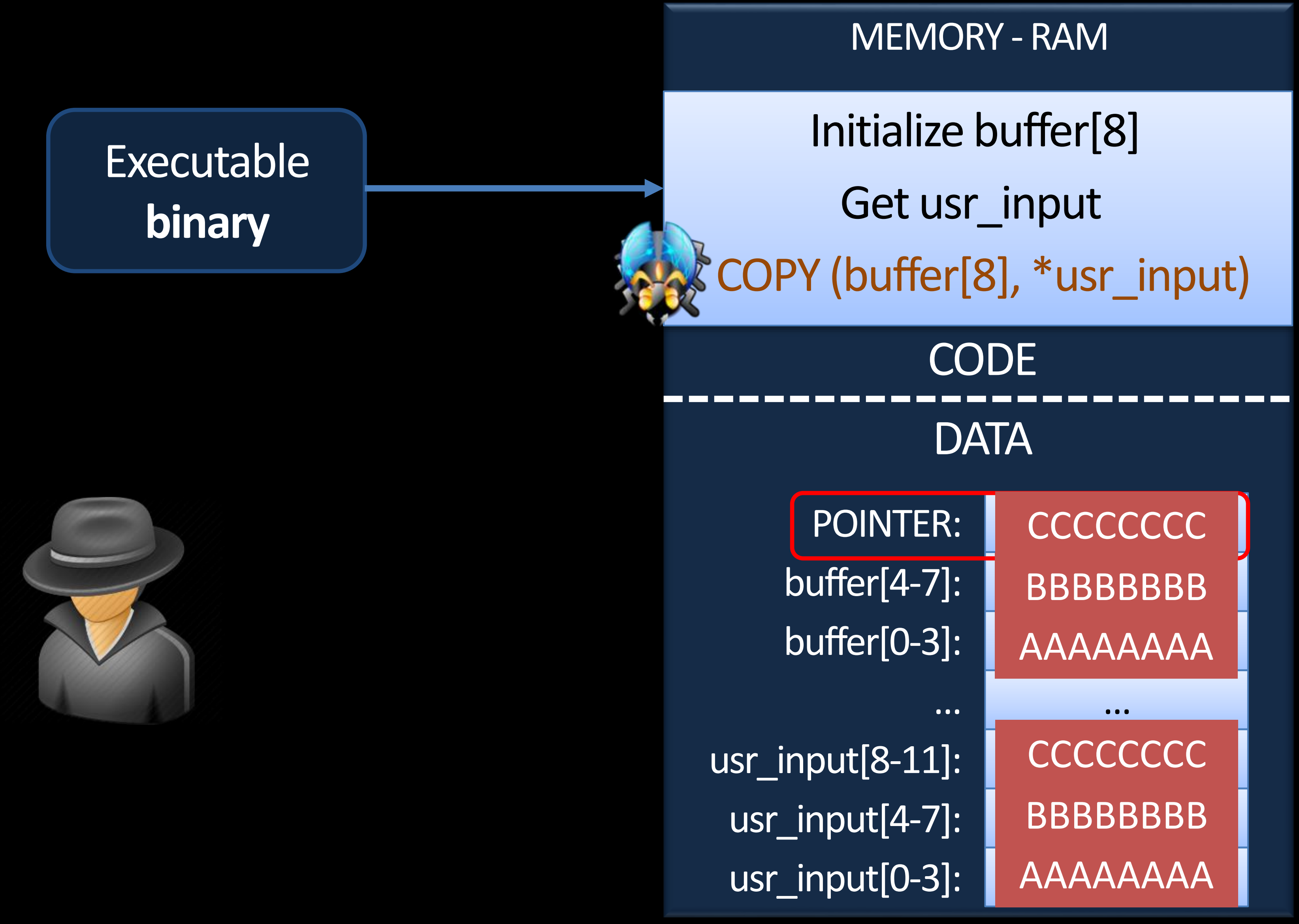
COPY (buffer[8], *usr_input)

CODE

DATA

POINTER:	8000ABCD
buffer[4-7]:	00000000
buffer[0-3]:	00000000
…	…
usr_input[8-11]:	CCCCCCCC
usr_input[4-7]:	BBBBBBBB
usr_input[0-3]:	AAAAAAAA

Big Picture: Program Execution (3/3)



Executable **binary**

MEMORY - RAM

Initialize buffer[8]

Get usr_input

COPY (buffer[8], *usr_input)

CODE

DATA

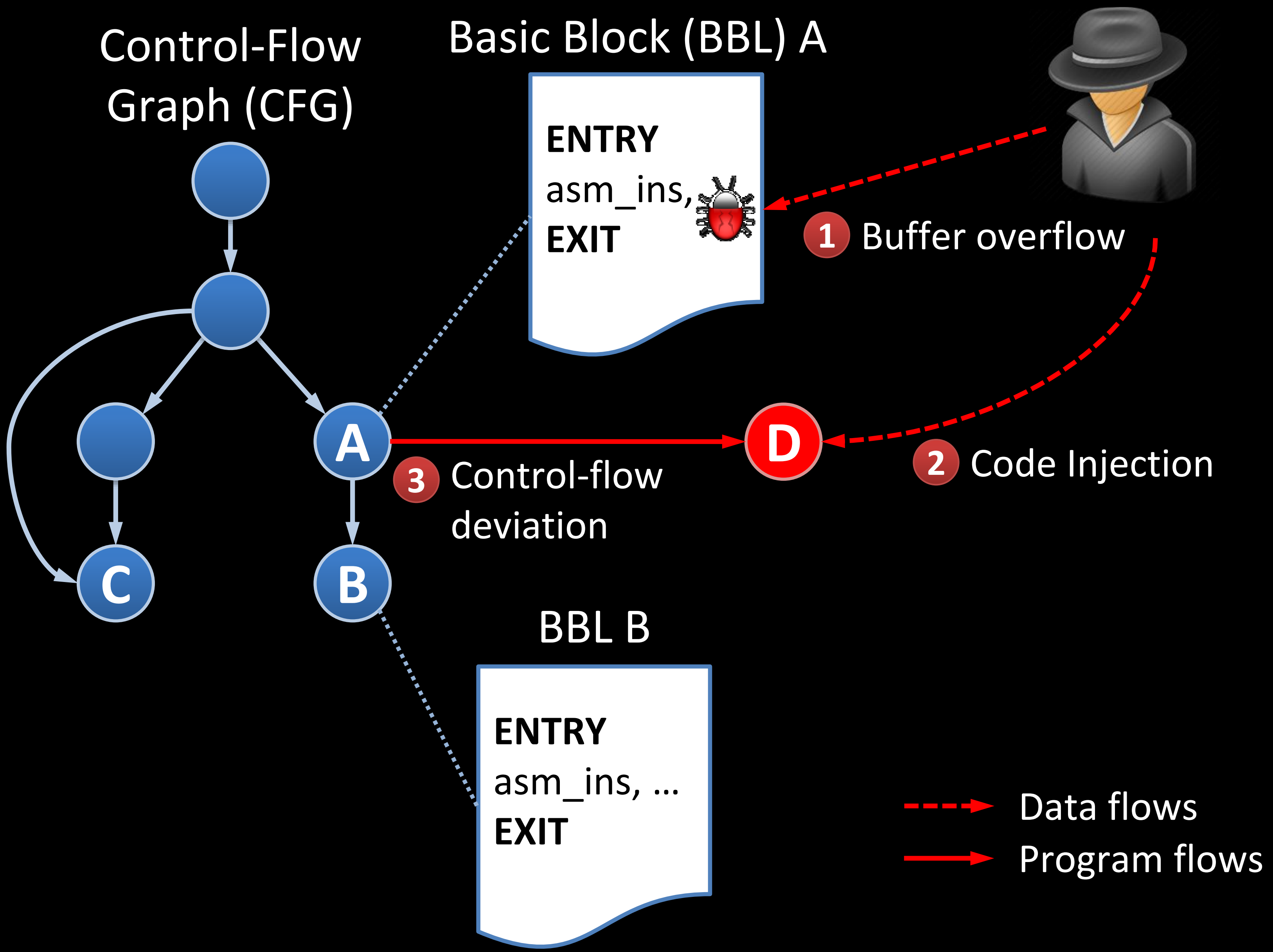
POINTER:	CCCCCCCC
buffer[4-7]:	BBBBBBBB
buffer[0-3]:	AAAAAAAA
…	…
usr_input[8-11]:	CCCCCCCC
usr_input[4-7]:	BBBBBBBB
usr_input[0-3]:	AAAAAAAA

Observations

- There are several observations
  1. A programming error leads to a program-flow deviation
  2. Missing bounds checking
     - Languages like C, C++, or assembler do not automatically enforce bounds checking on data inputs
  3. An adversary can provide inputs that influence the program flow

- What are the consequences?

General Principle of Code Injection Attacks



Control-Flow Graph (CFG)

Basic Block (BBL) A

ENTRY
asm_ins,
EXIT

1 Buffer overflow

2 Code Injection

3 Control-flow deviation

A

B

C

D

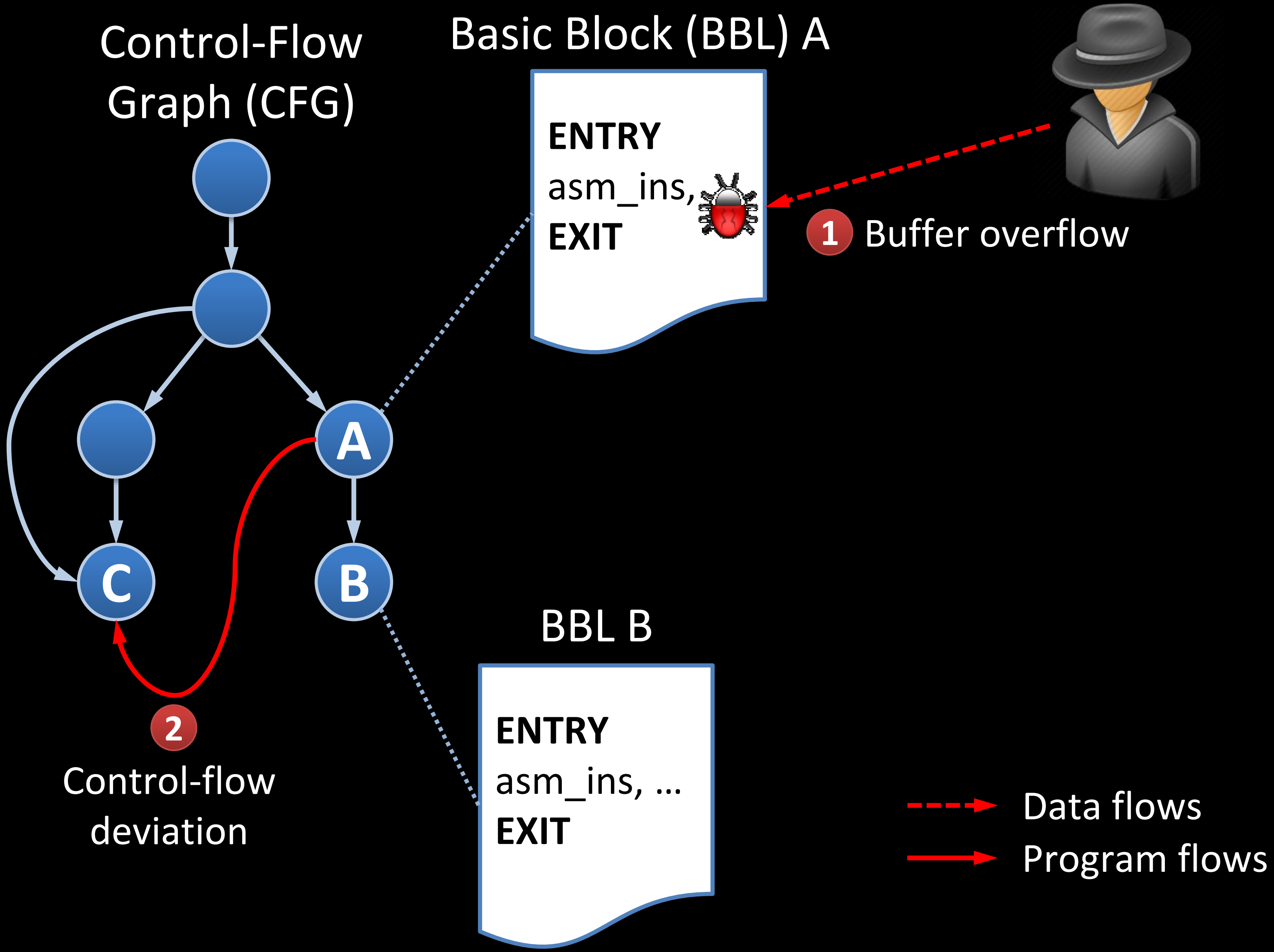
BBL B

ENTRY
asm_ins, …
EXIT

Data flows

Program flows

General Principle of Code Reuse Attacks



Control-Flow Graph (CFG)

Basic Block (BBL) A

ENTRY
asm_ins, …
EXIT

1 Buffer overflow

BBL B

ENTRY
asm_ins, …
EXIT

2 Control-flow deviation

Data flows

Program flows

Code Injection vs. Code Reuse

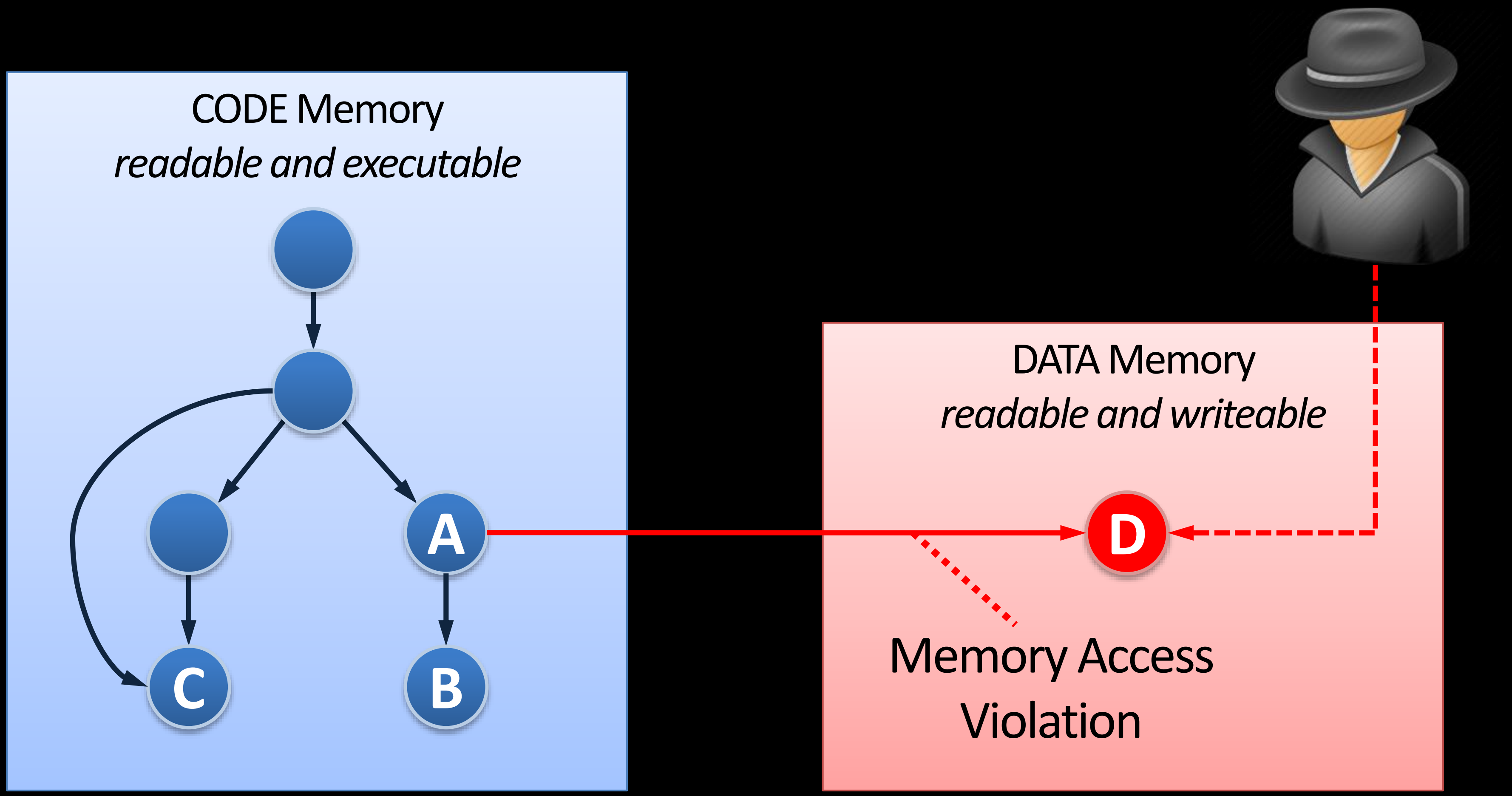
- Code Injection – *Adding a new node to the CFG*
  - Adversary can execute arbitrary malicious code
    - open a remote console (classical shellcode)
    - exploit further vulnerabilities in the OS kernel to install a virus or a backdoor
- Code Reuse – *Adding a new path to the CFG*
  - Adversary is limited to the code nodes that are available in the CFG
  - Requires reverse-engineering and static analysis of the code base of a program

BASICS

Code injection is more powerful; so why are attacks today typically using code reuse?

Data Execution Prevention (DEP)

- Prevent execution from a writeable memory (data) area

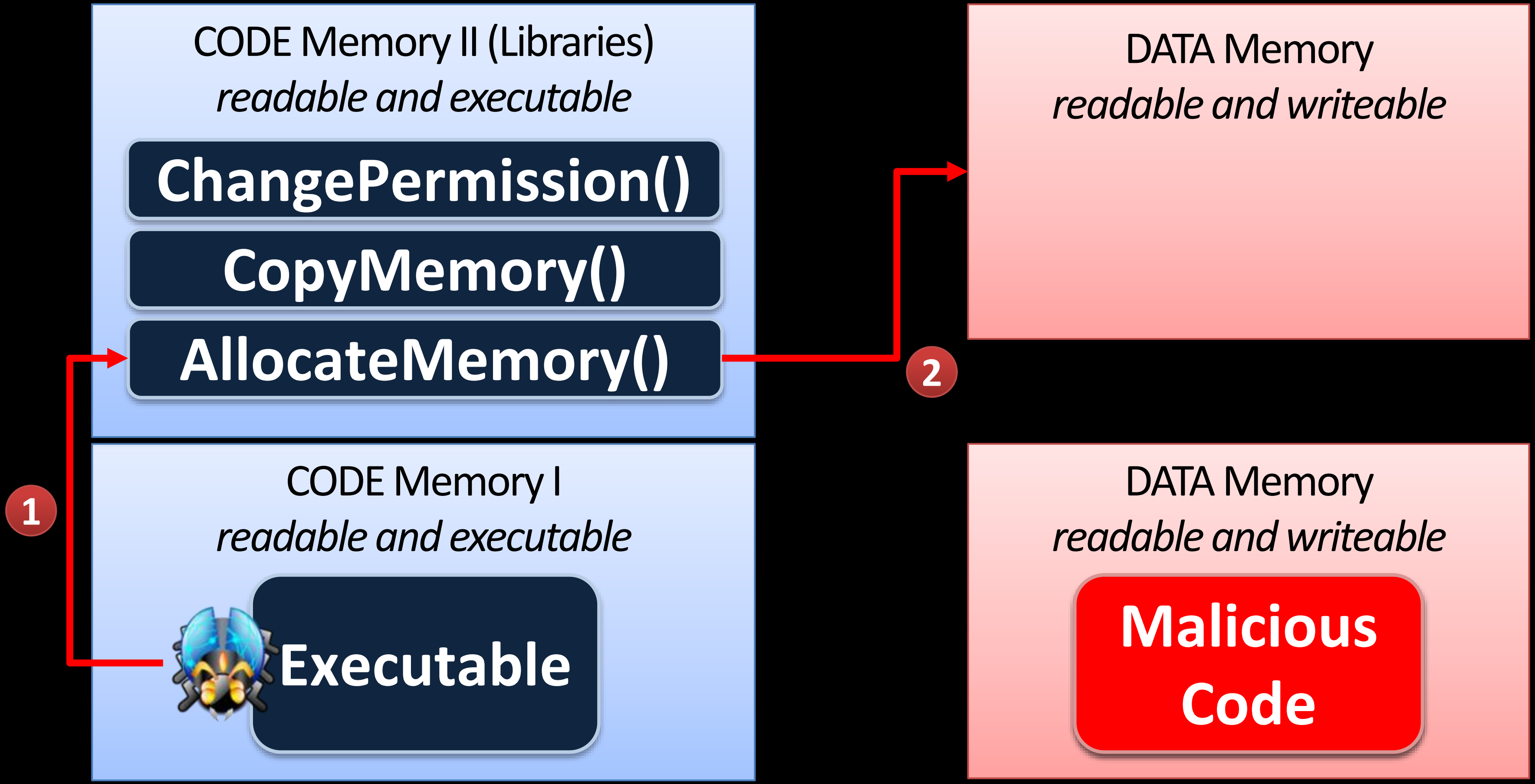


Data Execution Prevention (DEP) cntd.

- Implementations
  - Modern OSes enable DEP by default (Windows, Linux, iOS, Android, Mac OSX)
  - Intel, AMD, and ARM feature a special No-Execute bit to facilitate deployment of DEP
- Side Note
  - There are other notions referring to the same principle
    - $W \oplus X$ – Writeable XOR eXecutable
    - Non-executable memory

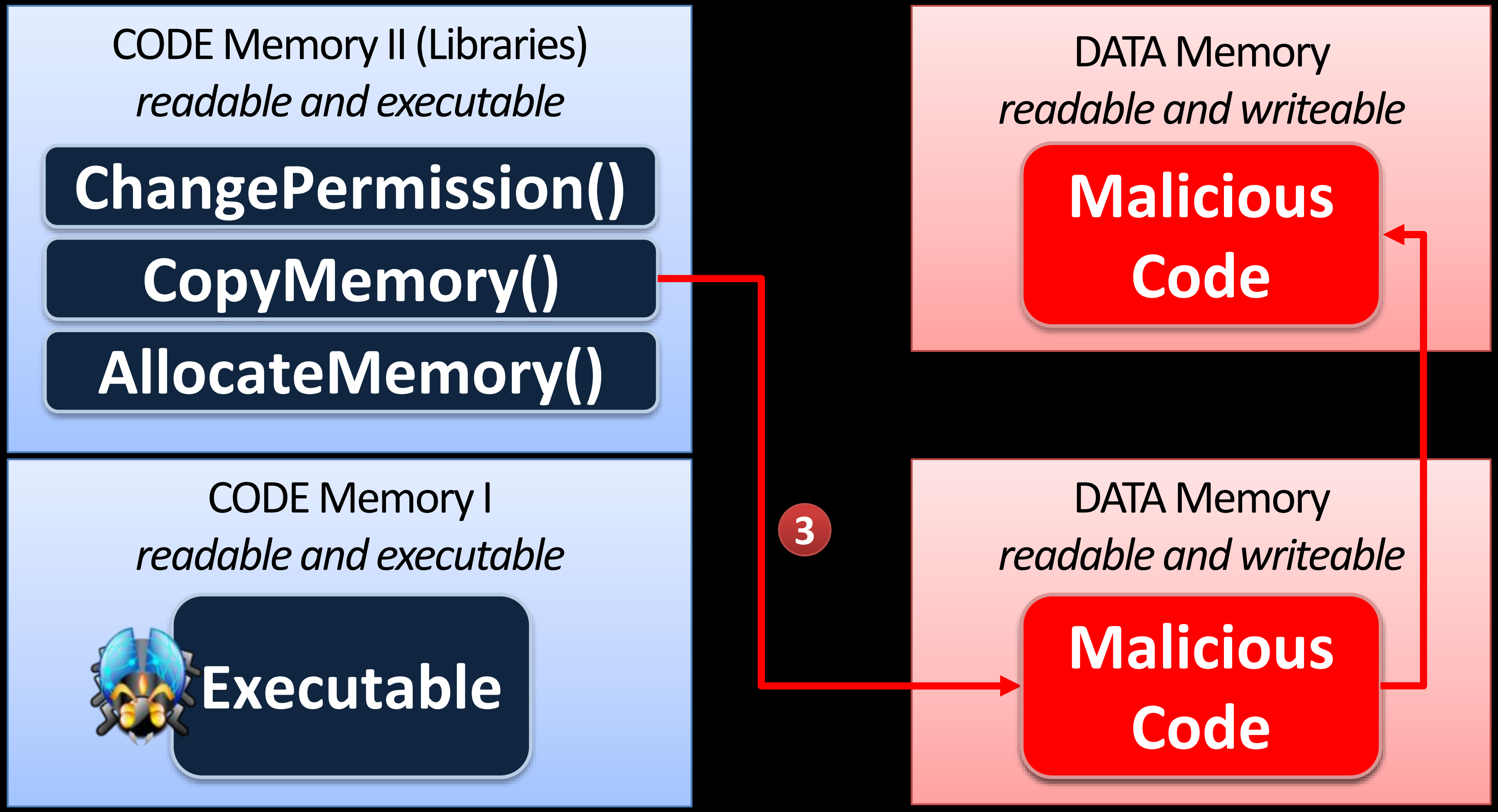
Hybrid Exploits (1/3)

- Today‘s attacks combine code reuse with code injection



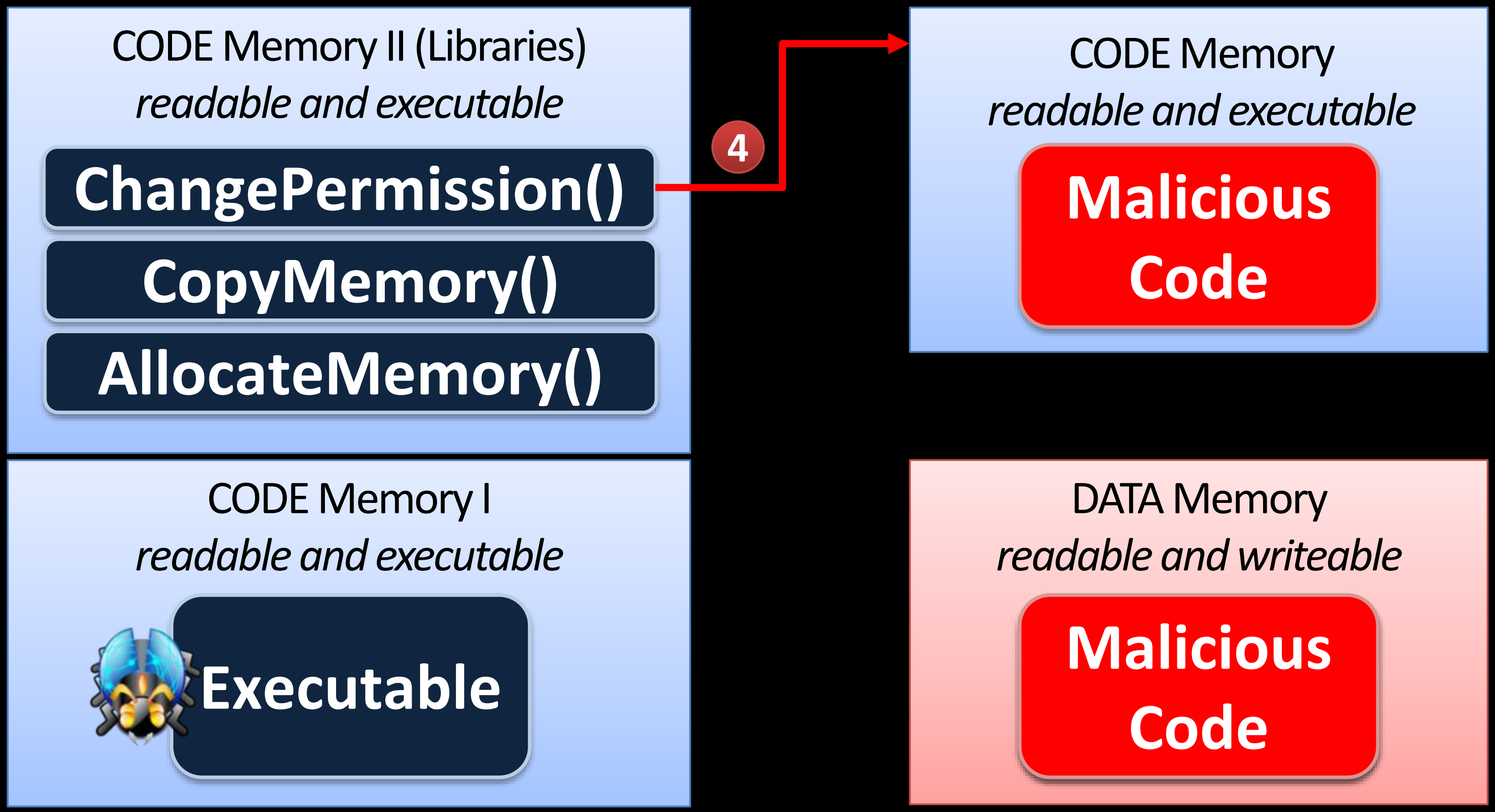
Hybrid Exploits (2/3)

- Today‘s attacks combine code reuse with code injection



Hybrid Exploits (3/3)

- Today‘s attacks combine code reuse with code injection



CODE Memory II (Libraries)

readable and executable

ChangePermission()

CopyMemory()

AllocateMemory()

4

CODE Memory

readable and executable

Malicious Code

CODE Memory I

readable and executable

Executable

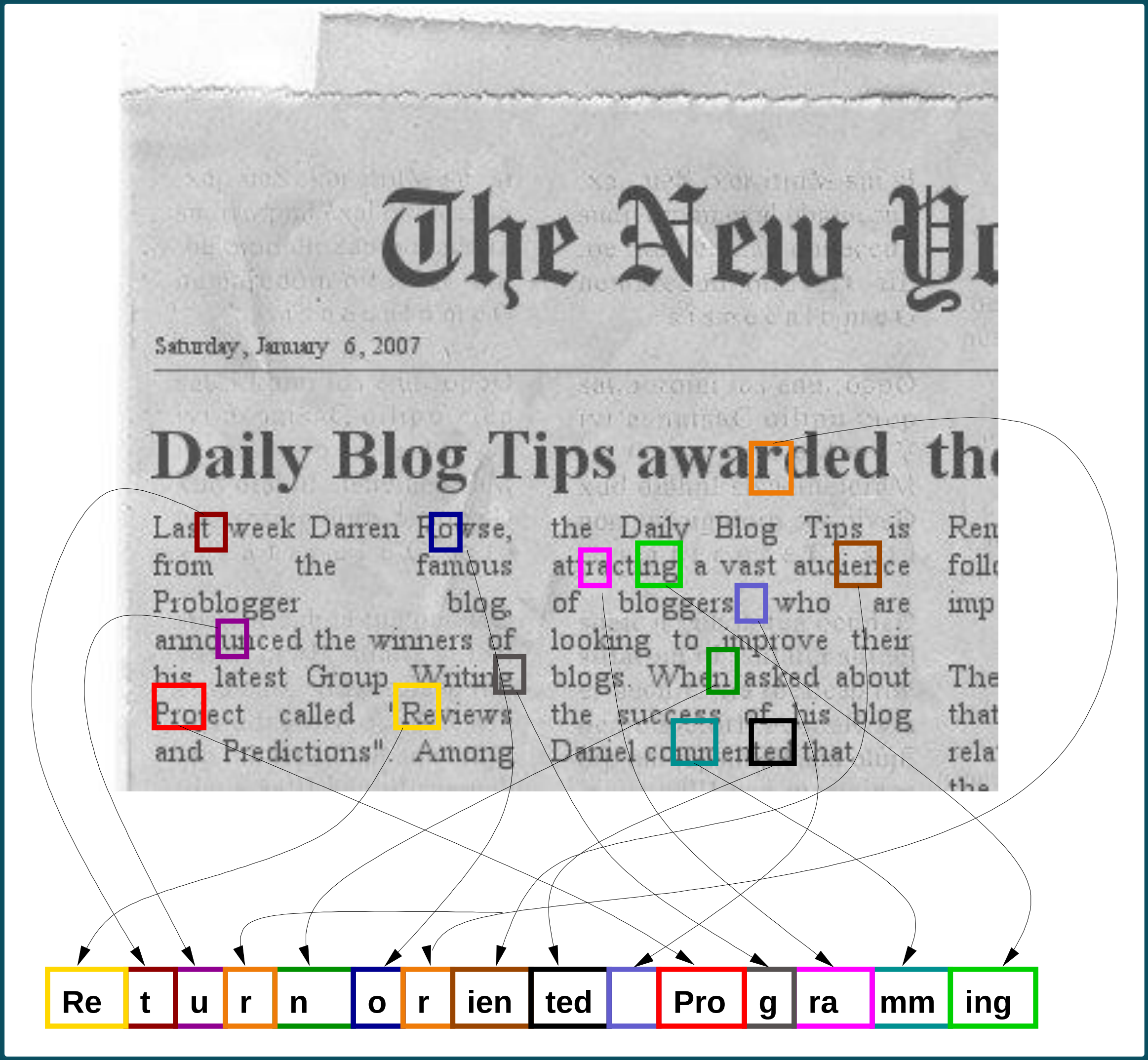
DATA Memory

readable and writeable

Malicious Code

CODE-REUSE ATTACKS
What is ROP and how does it work?

The Big Picture



The New Y

Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among

the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Ren
foll
imp

The
that
rela
the

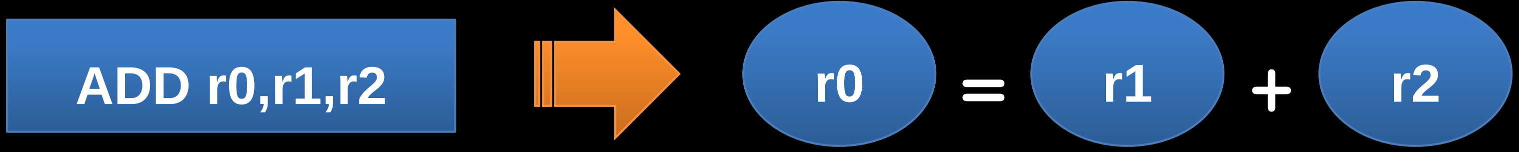
Selected background on ARM registers, stack layout, and calling convention

ARM Overview

- ARM stands for Advanced RISC Machine
- Main application area: Mobile phones, smartphones (Apple iPhone, Google Android), music players, tablets, and some netbooks
- Advantage: Low power consumption
- Follows RISC design
  - Mostly single-cycle execution
  - Fixed instruction length
  - Dedicated load and store instructions
- ARM features XN (eXecute Never) Bit

ARM Overview

- Some features of ARM
    - Conditional Execution
    - Two Instruction Sets
        - ARM (32-Bit)
            - The traditional instruction set
        - THUMB (16-Bit)
            - Suitable for devices that provide limited memory space
        - The processor can exchange the instruction set on-the-fly
        - Both instruction sets may occur in a single program
    - 3-Register-Instruction Set
        - **instruction** *destination, source, source*

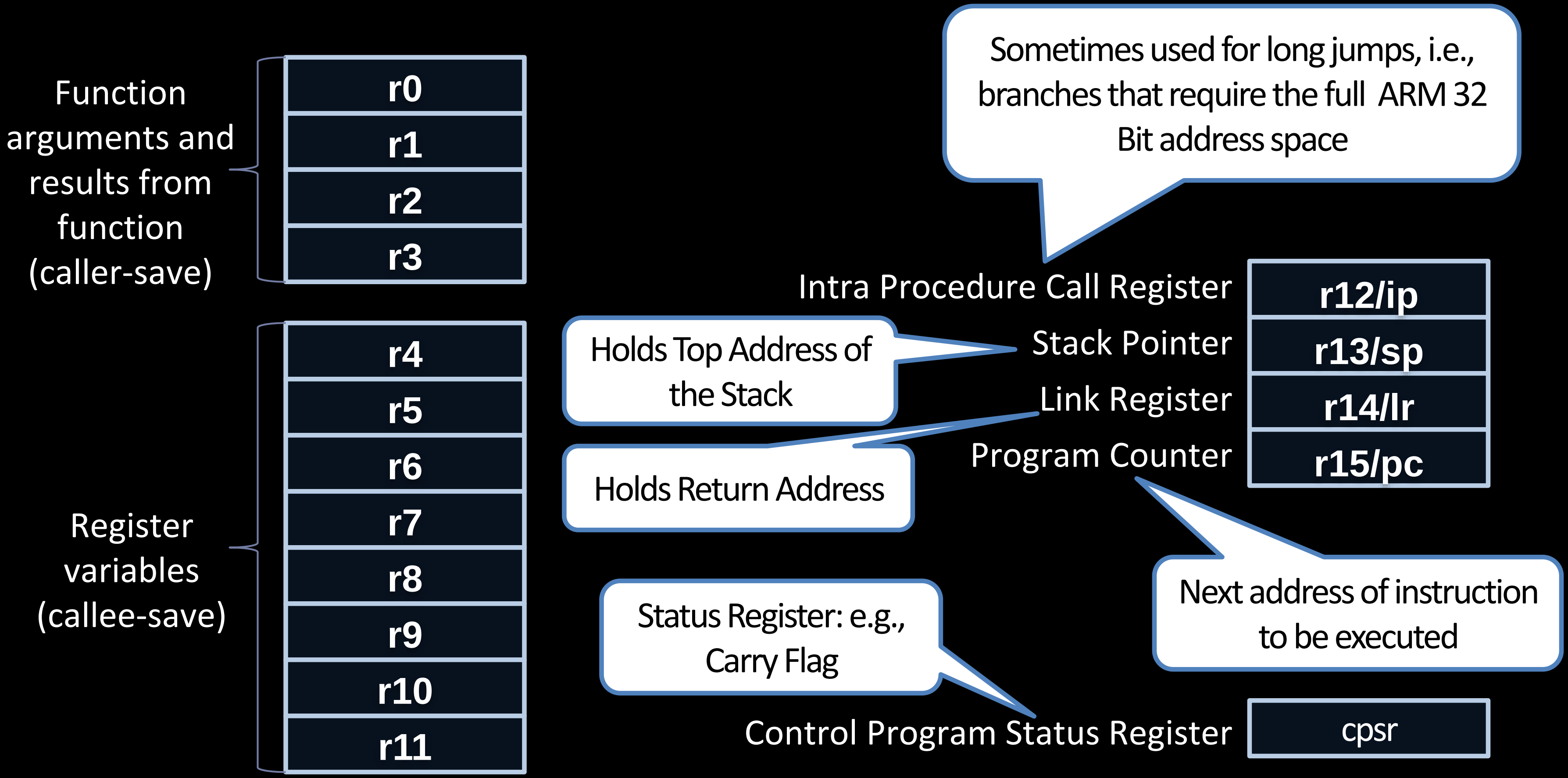


ADD r0,r1,r2

r0 = r1 + r2

ARM Registers

- ARM‘s 32 Bit processor features 16 registers
- All registers r0 to r15 are directly accessible



Function arguments and results from function (caller-save)

r0

r1

r2

r3

Register variables (callee-save)

r4

r5

r6

r7

r8

r9

r10

r11

Sometimes used for long jumps, i.e., branches that require the full ARM 32 Bit address space

Intra Procedure Call Register r12/ip

Stack Pointer r13/sp

Link Register r14/lr

Program Counter r15/pc

Holds Top Address of the Stack

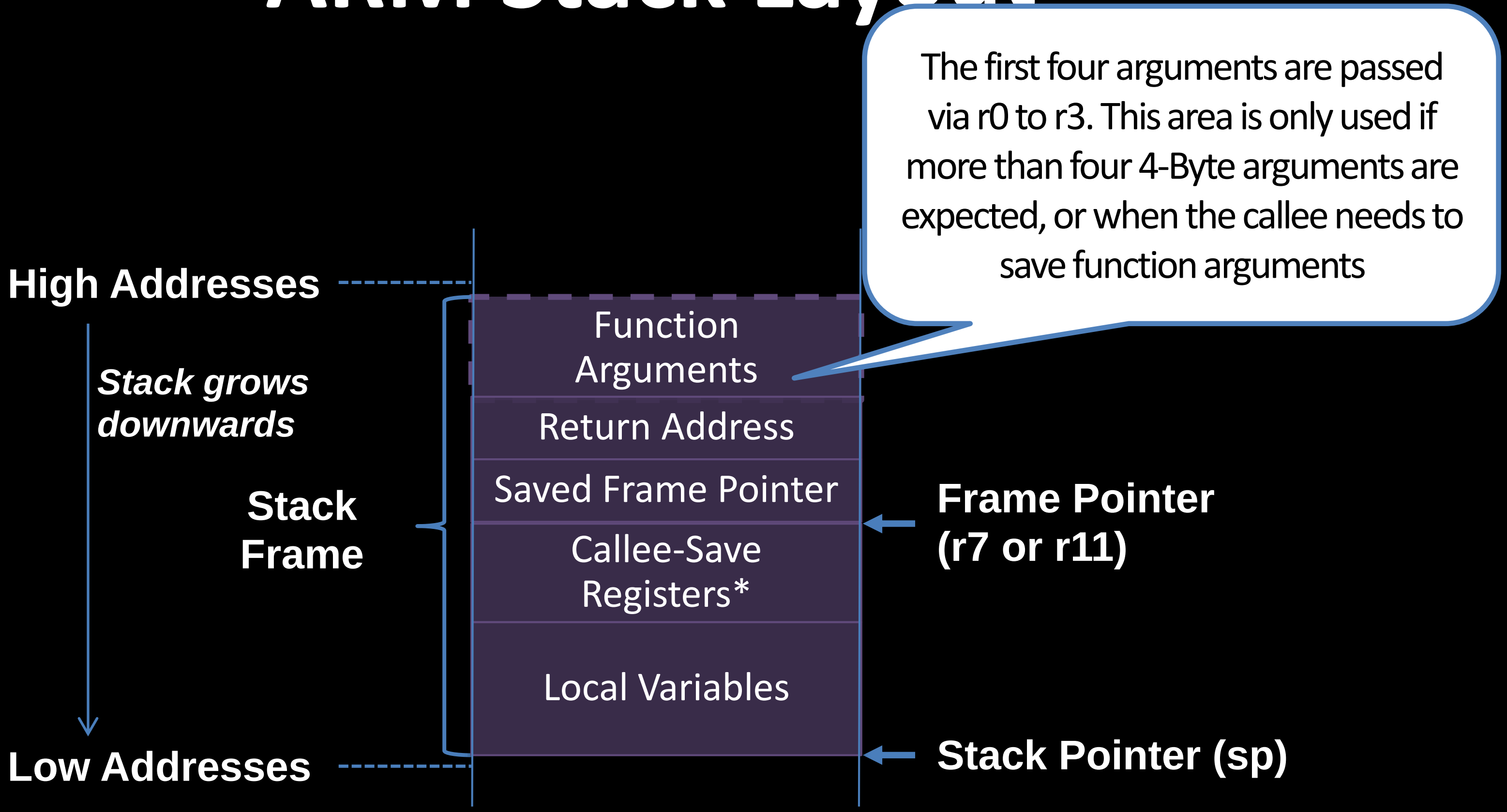
Holds Return Address

Next address of instruction to be executed

Status Register: e.g., Carry Flag

Control Program Status Register cpsr

ARM Stack Layout



The first four arguments are passed via r0 to r3. This area is only used if more than four 4-Byte arguments are expected, or when the callee needs to save function arguments

* Note that a subroutine does not always store all callee-save registers (r4 to r11); instead it stores those registers that it really uses/changes

Function Calls on ARM

Branch with Link

BL addr

- Branches to addr, and stores the return address in link register lr/r14
- The return address is simply the address that follows the BL instruction

Branch with Link and eXchange instruction set

BLX addr|reg

- Branches to addr|reg, and stores the return address in lr/r14
- This instruction allows the exchange between ARM and THUMB
  - ARM->THUMB: LSB=1
  - THUMB->ARM: LSB=0

Function Returns on ARM

Branch with eXchange instruction set

BX lr

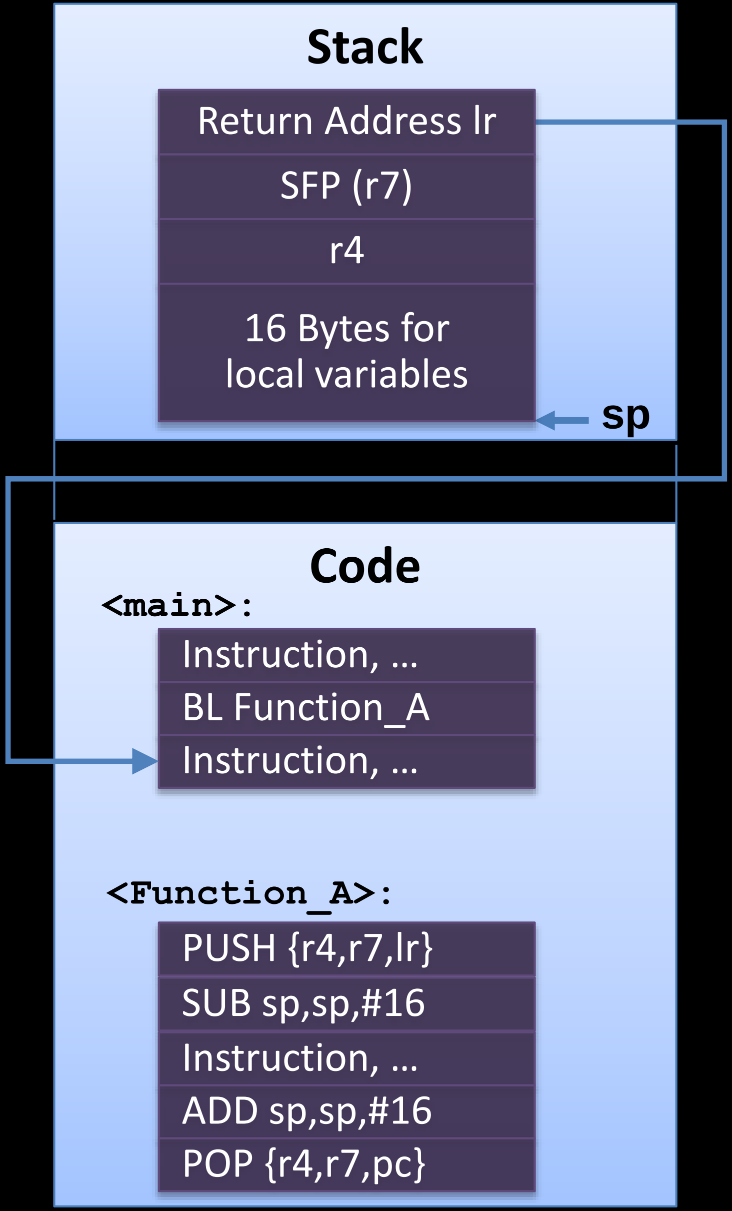
- Branches to the return address stored in the link register lr
- Register-based return for leaf functions

POP {pc}

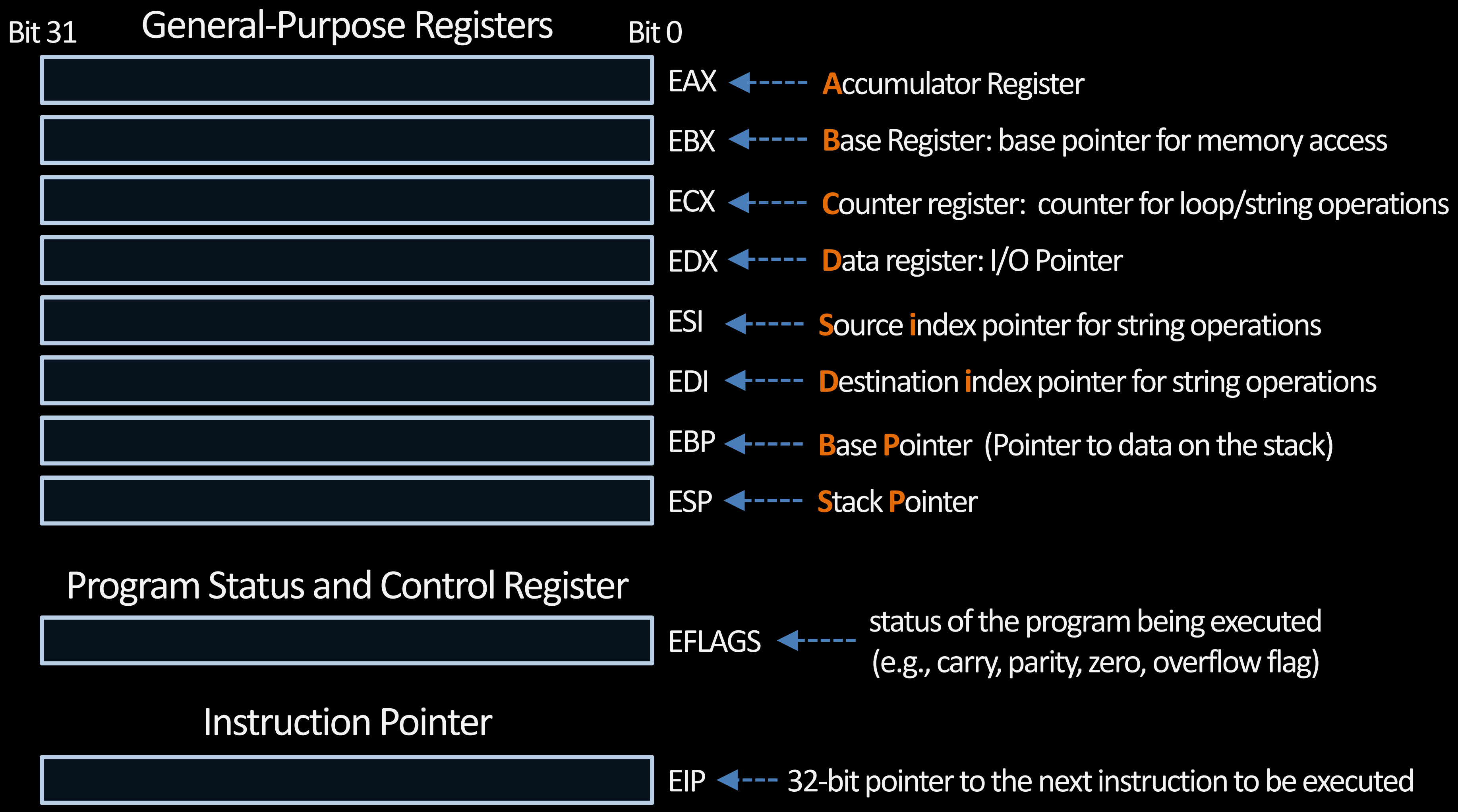
- Pops top of the stack into the program counter pc/r15
- Stack-based return for non-leaf functions

THUMB Example for Calling Convention

- Function Call: BL Function_A
  - The BL instruction automatically loads the return address into the link register lr
- Function Prologue 1: PUSH {r4,r7,lr}
  - Stores callee-save register r4, the frame pointer r7, and the return address lr on the stack
- Function Prologue 2: SUB sp,sp,#16
  - Allocates 16 Bytes for local variables on the stack
- Function Body: Instructions, …
- Function Epilogue 2: ADD sp,sp,#16
  - Reallocates the space for local variables
- Function Epilogue 2: POP {r4,r7,pc}
  - The POP instruction pops the callee-save register r4, the saved frame pointer r7, and the return address off the stack which is loaded it into the program counter pc
  - Hence, the execution will continue in the main function



General System and Application Programming Registers



Bit 31 General-Purpose Registers Bit 0

EAX ← Accumulator Register

EBX ← Base Register: base pointer for memory access

ECX ← Counter register: counter for loop/string operations

EDX ← Data register: I/O Pointer

ESI ← Source index pointer for string operations

EDI ← Destination index pointer for string operations

EBP ← Base Pointer (Pointer to data on the stack)

ESP ← Stack Pointer

Program Status and Control Register

EFLAGS ← status of the program being executed (e.g., carry, parity, zero, overflow flag)

Instruction Pointer

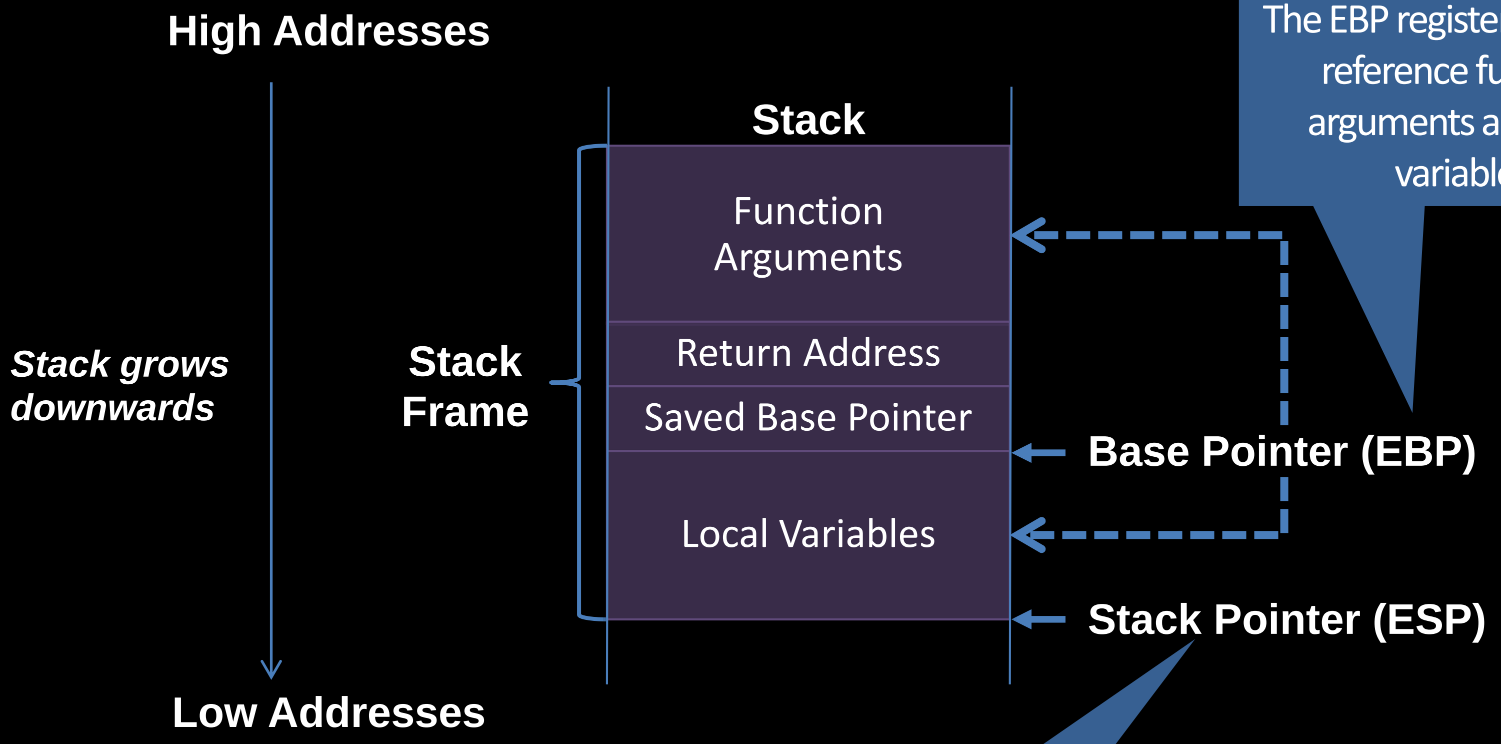
EIP ← 32-bit pointer to the next instruction to be executed

Source: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture

http://download.intel.com/products/processor/manual/253665.pdf

Stack Frame

Each function is associated with one stack frame on the stack

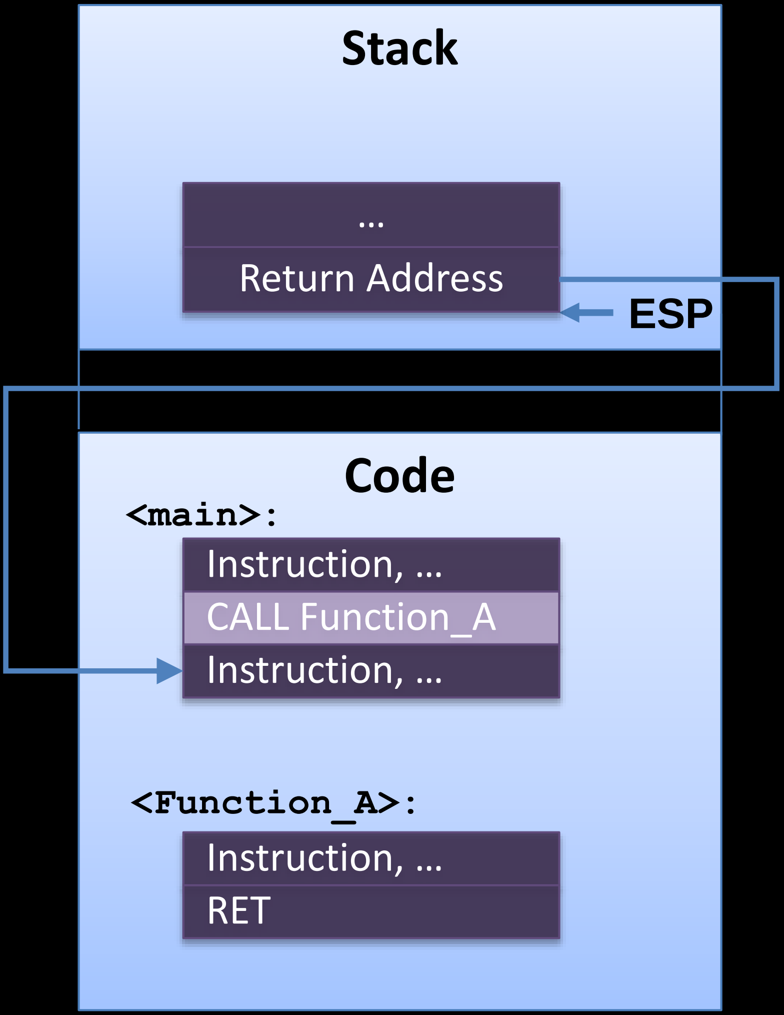


The EBP register is used to reference function arguments and local variables

The ESP register holds the stack pointer and always points to the last element on the stack

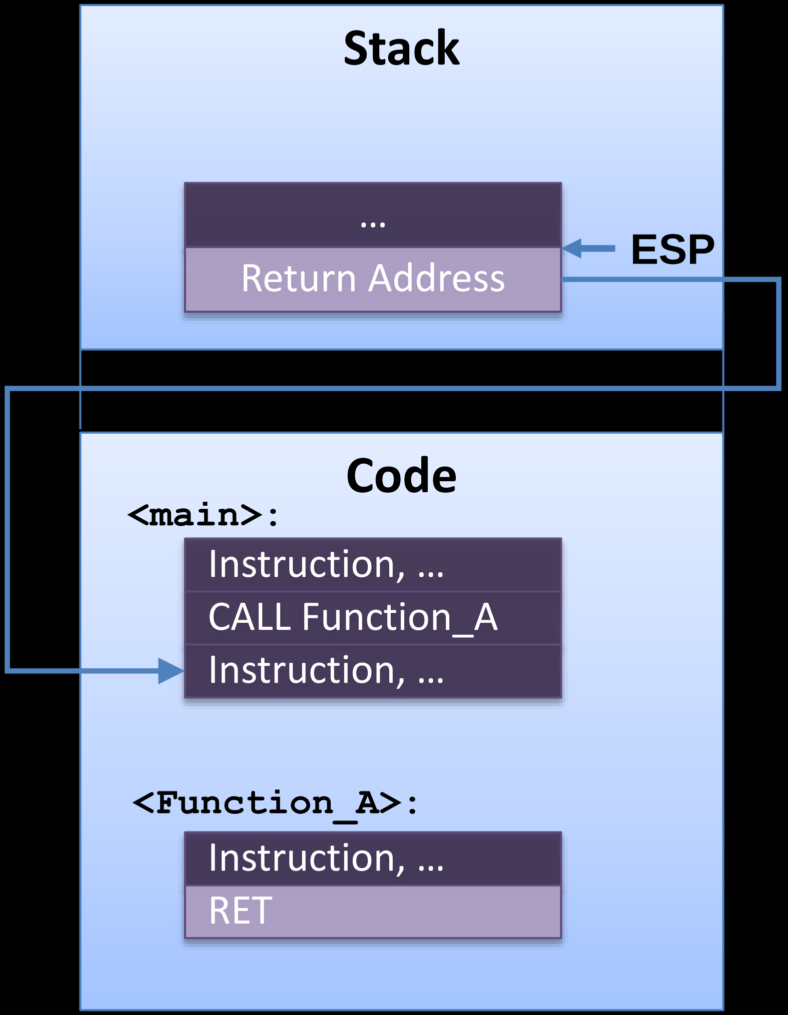
Calling Convention (on Intel x86)

- Function call performed via the x86 CALL instruction
    - E.g., CALL Function_A
    - The CALL instruction automatically pushes the return address on the stack, while the return address simply points to the instruction preceding the call



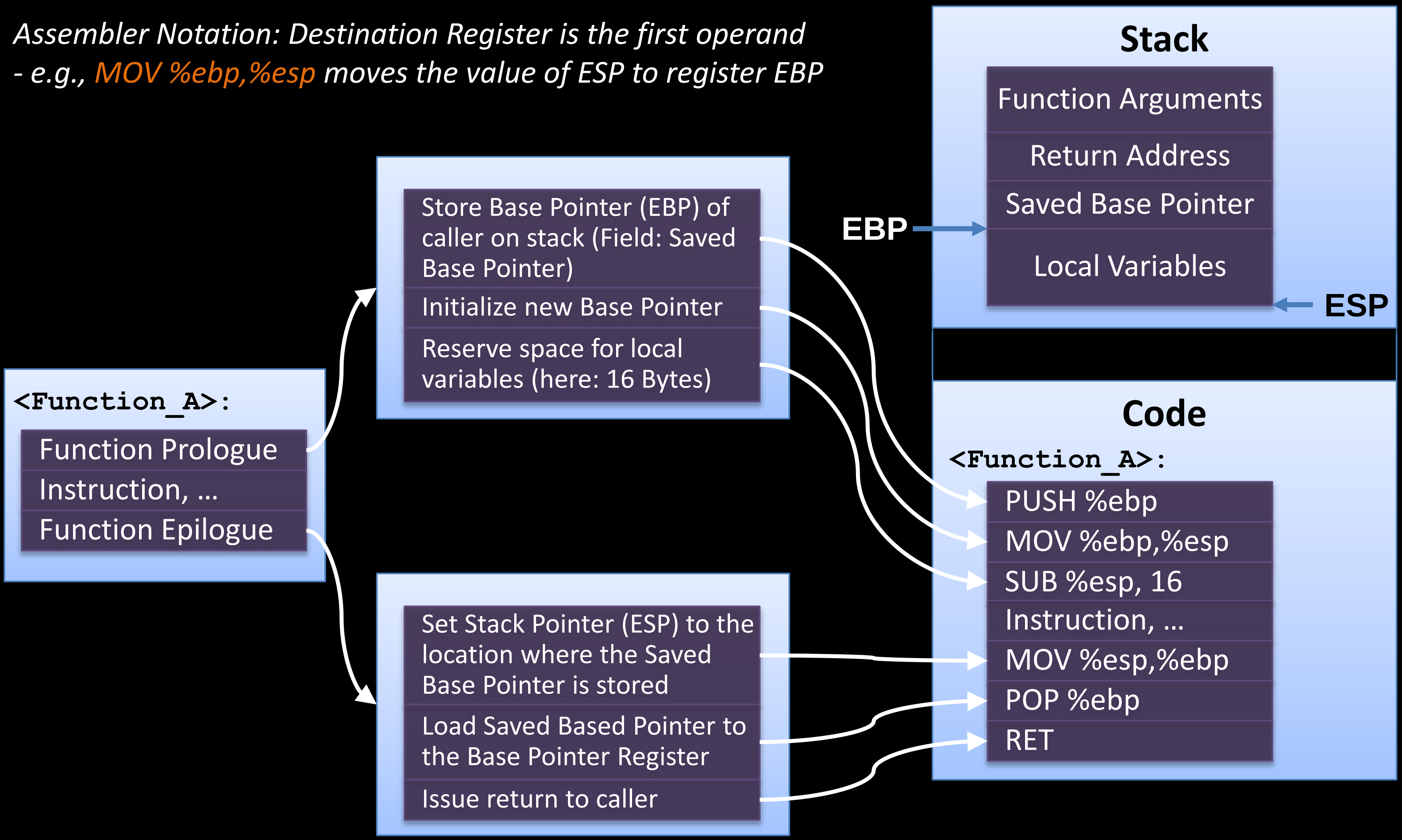
Calling Convention (on Intel x86)

- Function return is performed via the x86 RET instruction
  - The RET instruction pops the return address off the stack and loads it into the instruction pointer (EIP)
  - Hence, the execution will continue in the main function



Function Prologue and Epilogue by Example

Assembler Notation: Destination Register is the first operand
- e.g., MOV %ebp,%esp moves the value of ESP to register EBP



`<Function_A>:`

- Function Prologue
- Instruction, …
- Function Epilogue

Function Prologue:

- Store Base Pointer (EBP) of caller on stack (Field: Saved Base Pointer)
- Initialize new Base Pointer
- Reserve space for local variables (here: 16 Bytes)

Function Epilogue:

- Set Stack Pointer (ESP) to the location where the Saved Base Pointer is stored
- Load Saved Based Pointer to the Base Pointer Register
- Issue return to caller

Stack

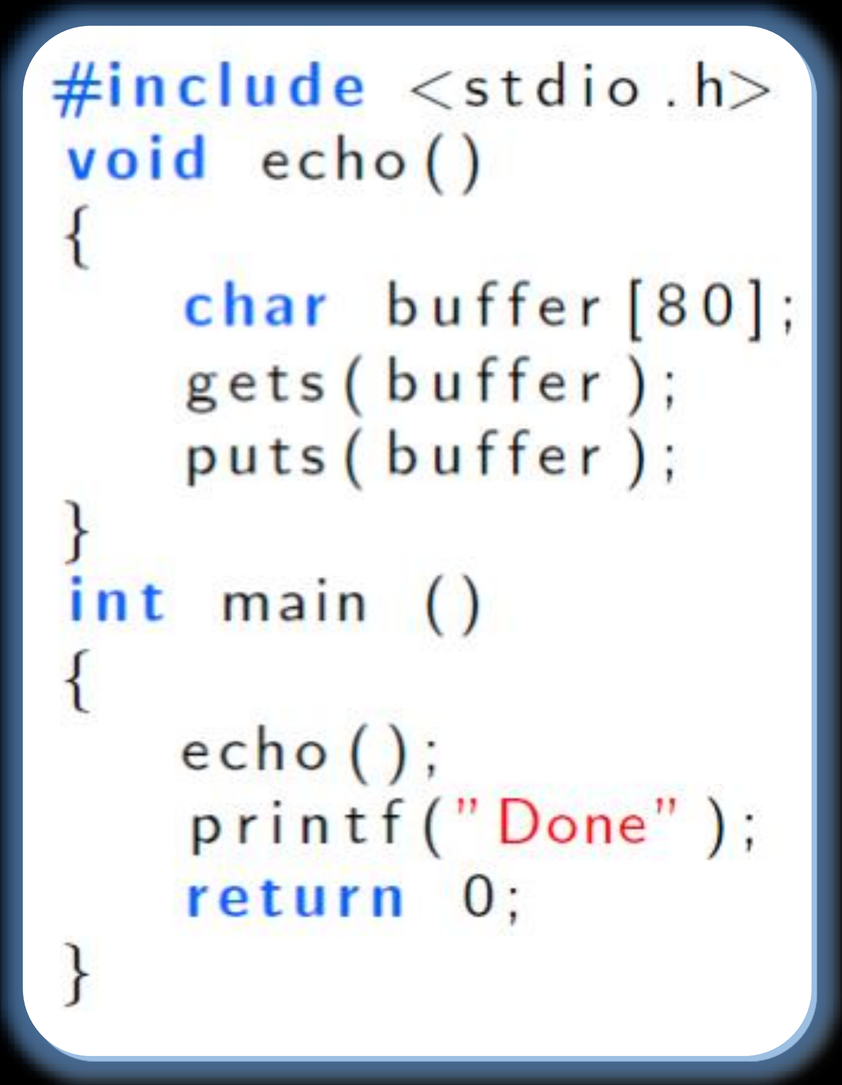
- Function Arguments
- Return Address
- Saved Base Pointer
- Local Variables

Code

```
<Function_A>:
PUSH %ebp
MOV %ebp,%esp
SUB %esp, 16
Instruction, …
MOV %esp,%ebp
POP %ebp
RET
```

Let‘s go back to runtime attacks

Running Example

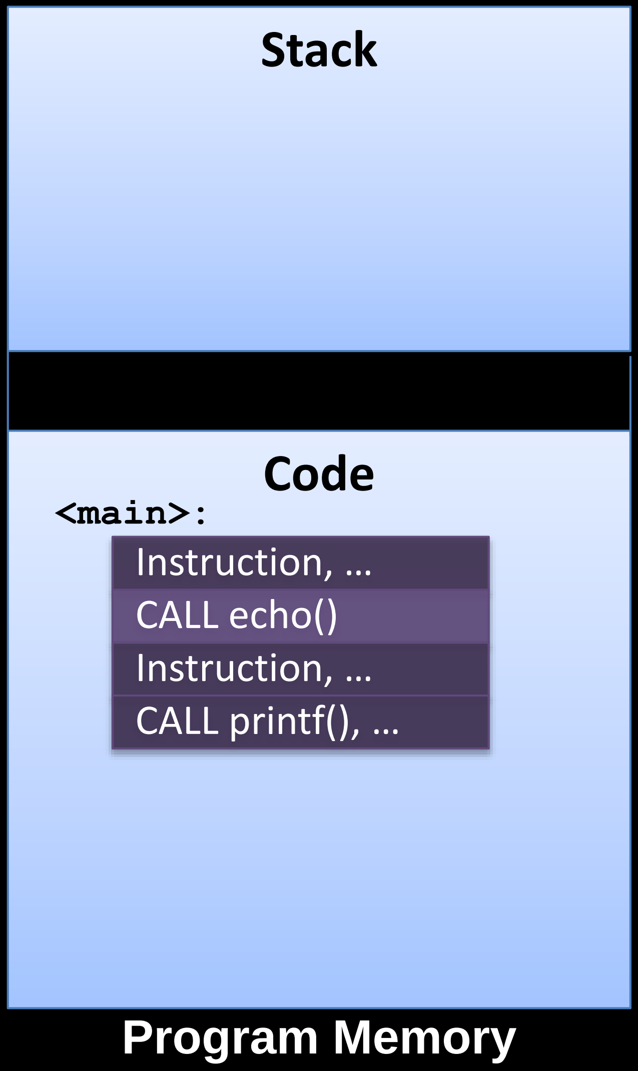


```c
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main ()
{
    echo();
    printf("Done");
    return 0;
}
```

Launching a code injection attack against the vulnerable program

Call to subroutine echo()

Adversary



Stack

Code

<main>:

Instruction, …

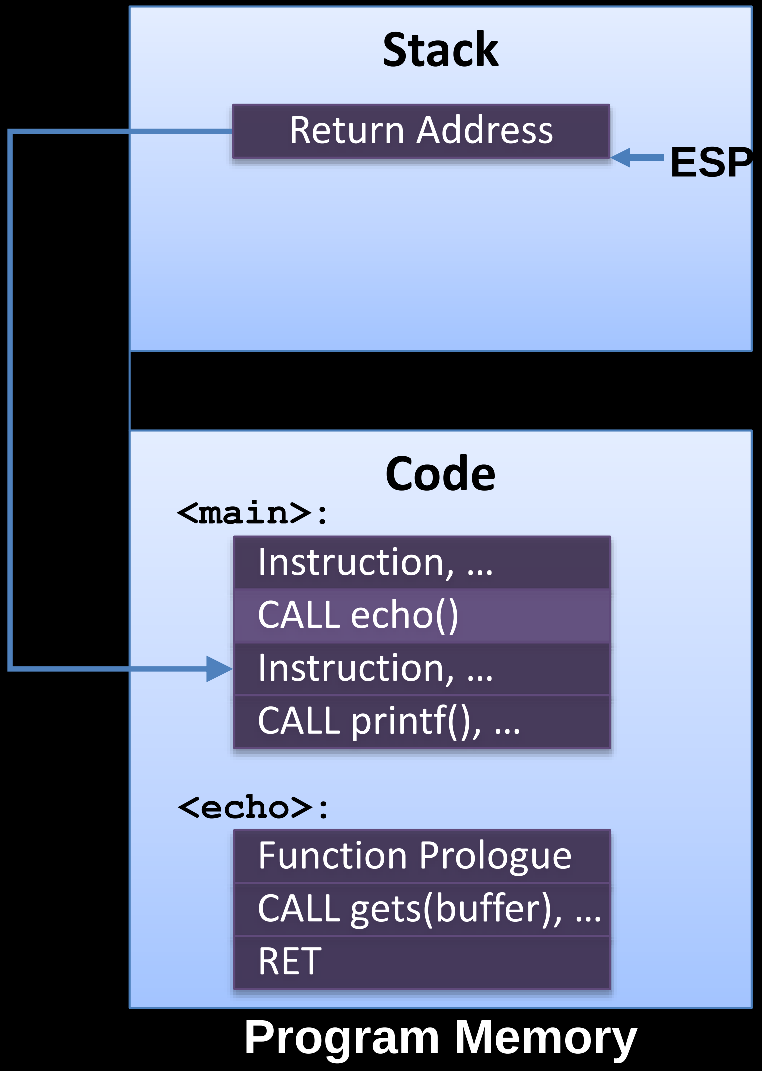
CALL echo()

Instruction, …

CALL printf(), …

Program Memory

CALL instruction pushes return address onto the Stack



Stack

Return Address

ESP

Code

`<main>:`

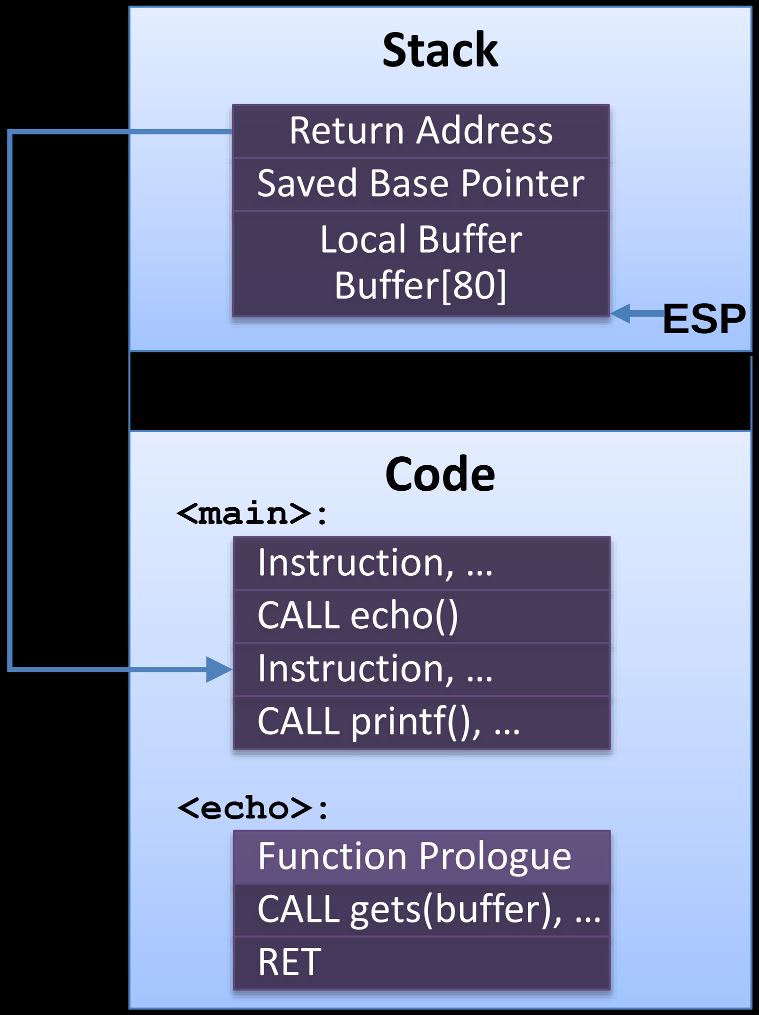
- Instruction, …
- CALL echo()
- Instruction, …
- CALL printf(), …

`<echo>:`

- Function Prologue
- CALL gets(buffer), …
- RET

Program Memory

Function prologue of echo() gets executed



Stack

Return Address

Saved Base Pointer

Local Buffer
Buffer[80]

ESP

Code

`<main>:`

Instruction, …

CALL echo()

Instruction, …

CALL printf(), …

`<echo>:`

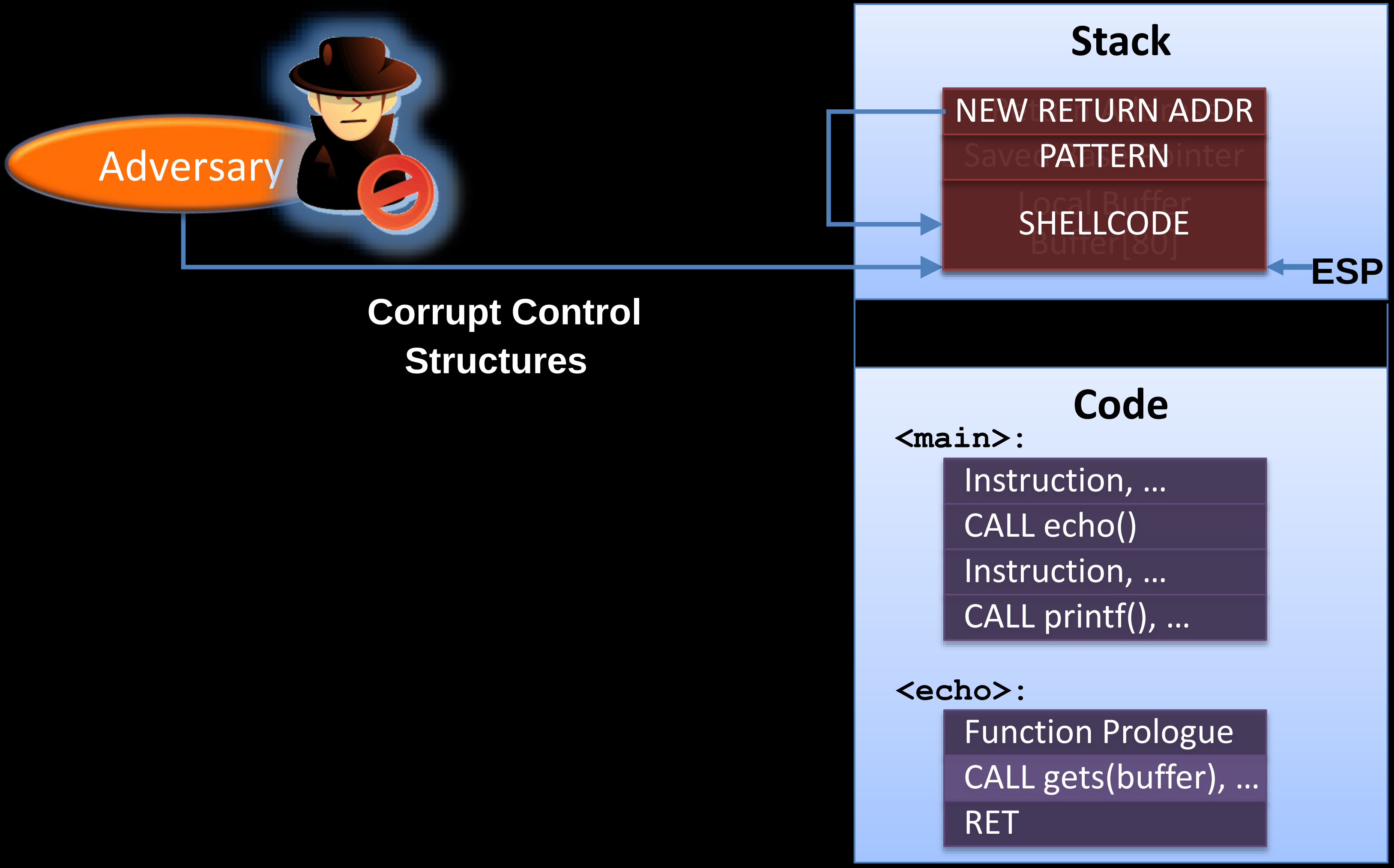
Function Prologue

CALL gets(buffer), …

RET

Program Memory

Subroutine call to gets()



Corrupt Control Structures

Stack

NEW RETURN ADDR
PATTERN
SHELLCODE

Code

`<main>:`

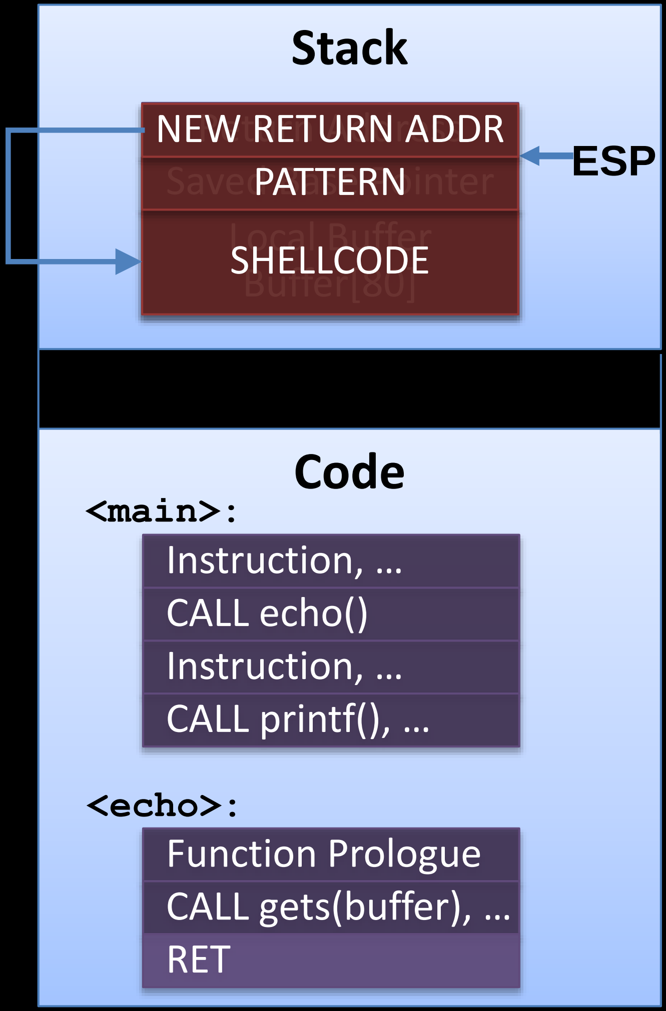
Instruction, …
CALL echo()
Instruction, …
CALL printf(), …

`<echo>:`

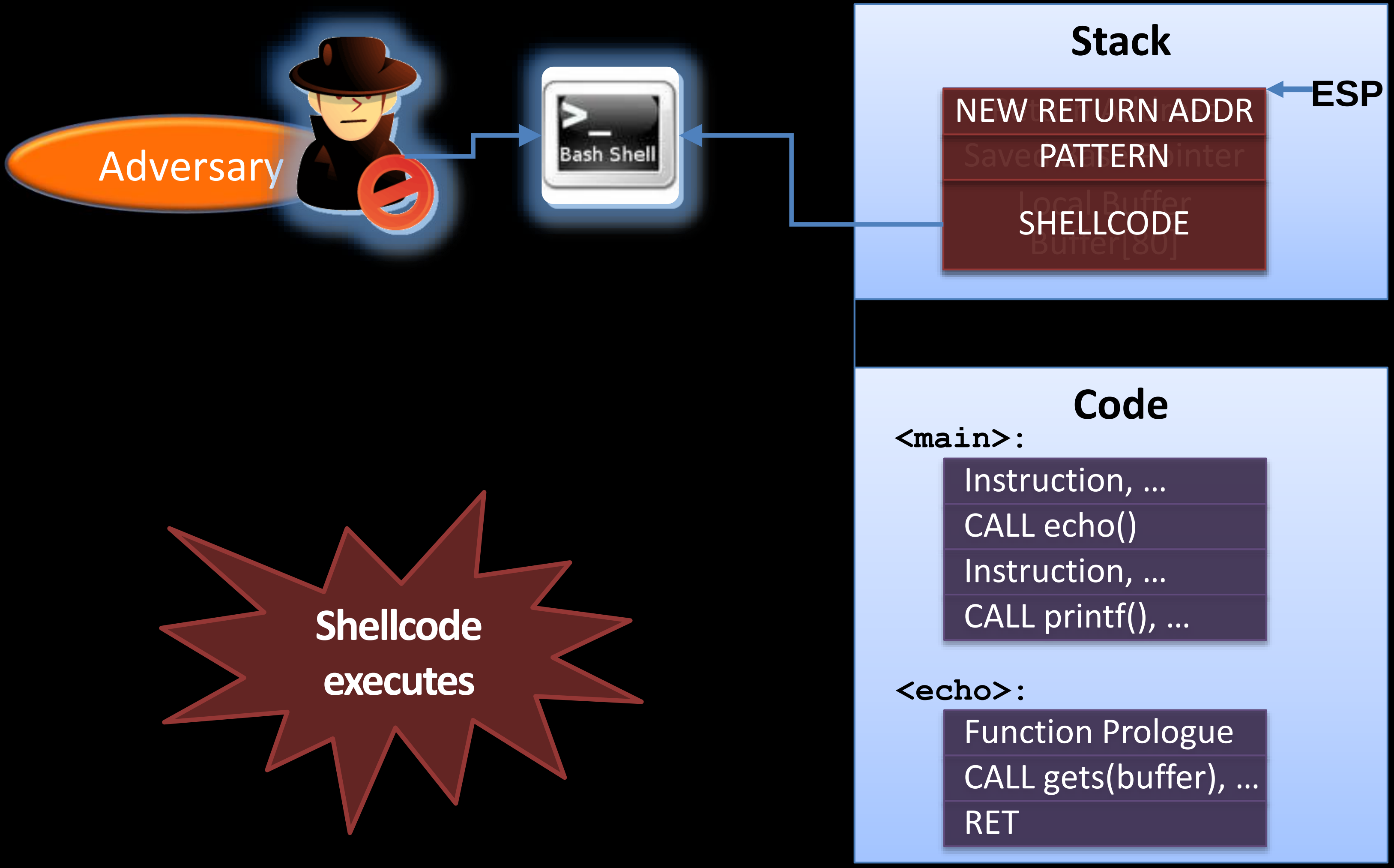
Function Prologue
CALL gets(buffer), …
RET

Program Memory

echo() now returns!



Program Memory



Adversary

Bash Shell

Stack

ESP

NEW RETURN ADDR

PATTERN

SHELLCODE

Code

`<main>:`

Instruction, …

CALL echo()

Instruction, …

CALL printf(), …

`<echo>:`

Function Prologue

CALL gets(buffer), …

RET

Shellcode executes

Program Memory

Code Injection on ARM

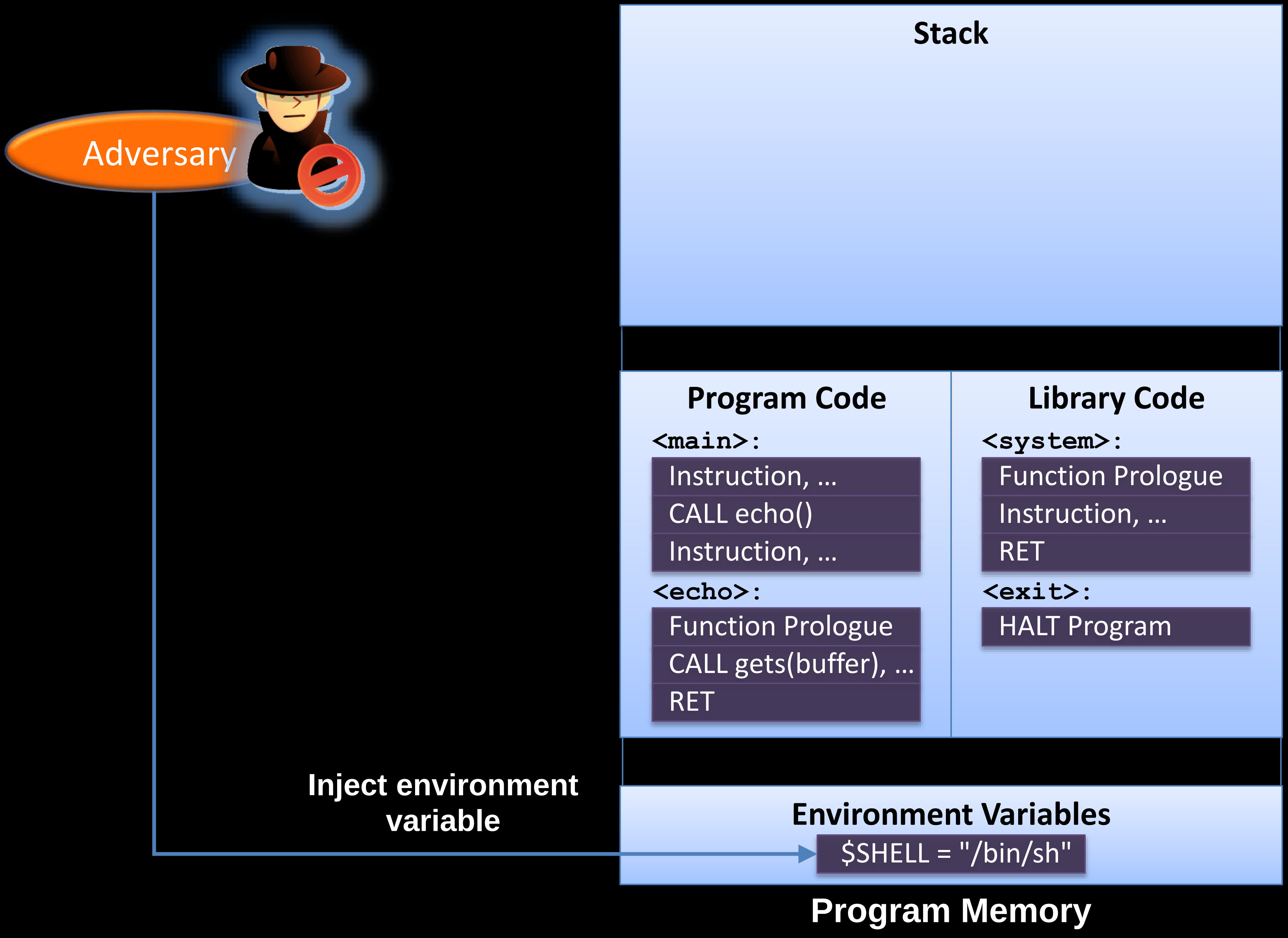
- Same attack strategy
- Implementation differences
  - BLX/BL instruction used for function call
  - Function prologue pushes the return address and the callee-save registers on the stack

Code-Reuse Attacks

It started with return-into-libc

[Solar Designer, http://insecure.org/sploits/linux.libc.return.lpr.sploit.html 1997]

- Basic idea of return-into-libc
  - Redirect execution to functions in shared libraries
  - Main target is UNIX C library libc
    - Libc is linked to nearly every Unix program
    - Defines system calls and other basic facilities such as open(), malloc(), printf(), system(), execve(), etc.
  - Attack example: system (“/bin/sh”), exit()



Adversary

Stack

Program Code

<main>:

Instruction, …

CALL echo()

Instruction, …

<echo>:

Function Prologue

CALL gets(buffer), …

RET

Library Code

<system>:

Function Prologue

Instruction, …

RET

<exit>:

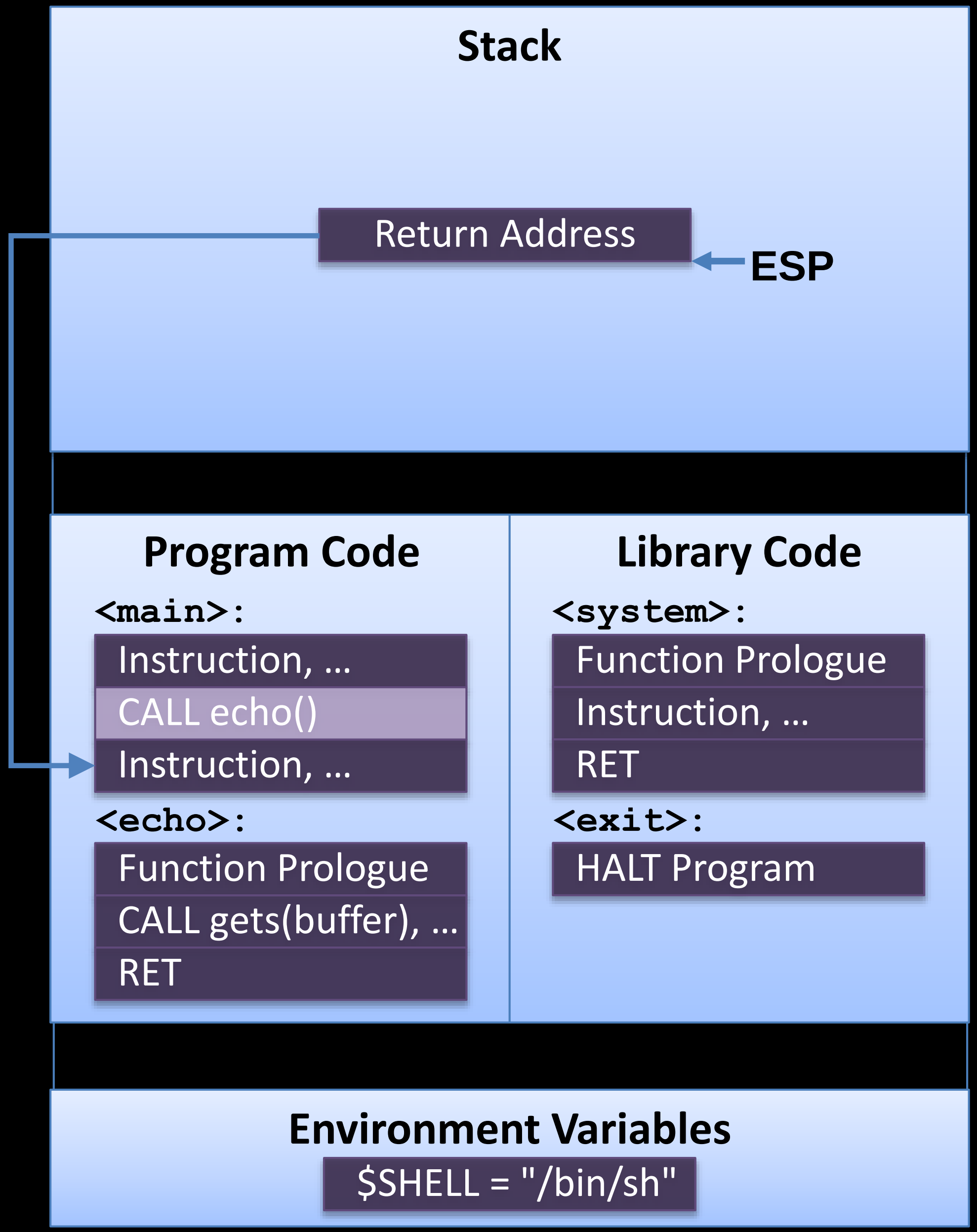
HALT Program

Inject environment variable

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Stack

Return Address

ESP

Program Code

```
<main>:
```

Instruction, …

CALL echo()

Instruction, …

```
<echo>:
```

Function Prologue

CALL gets(buffer), …

RET

Library Code

```
<system>:
```

Function Prologue

Instruction, …

RET

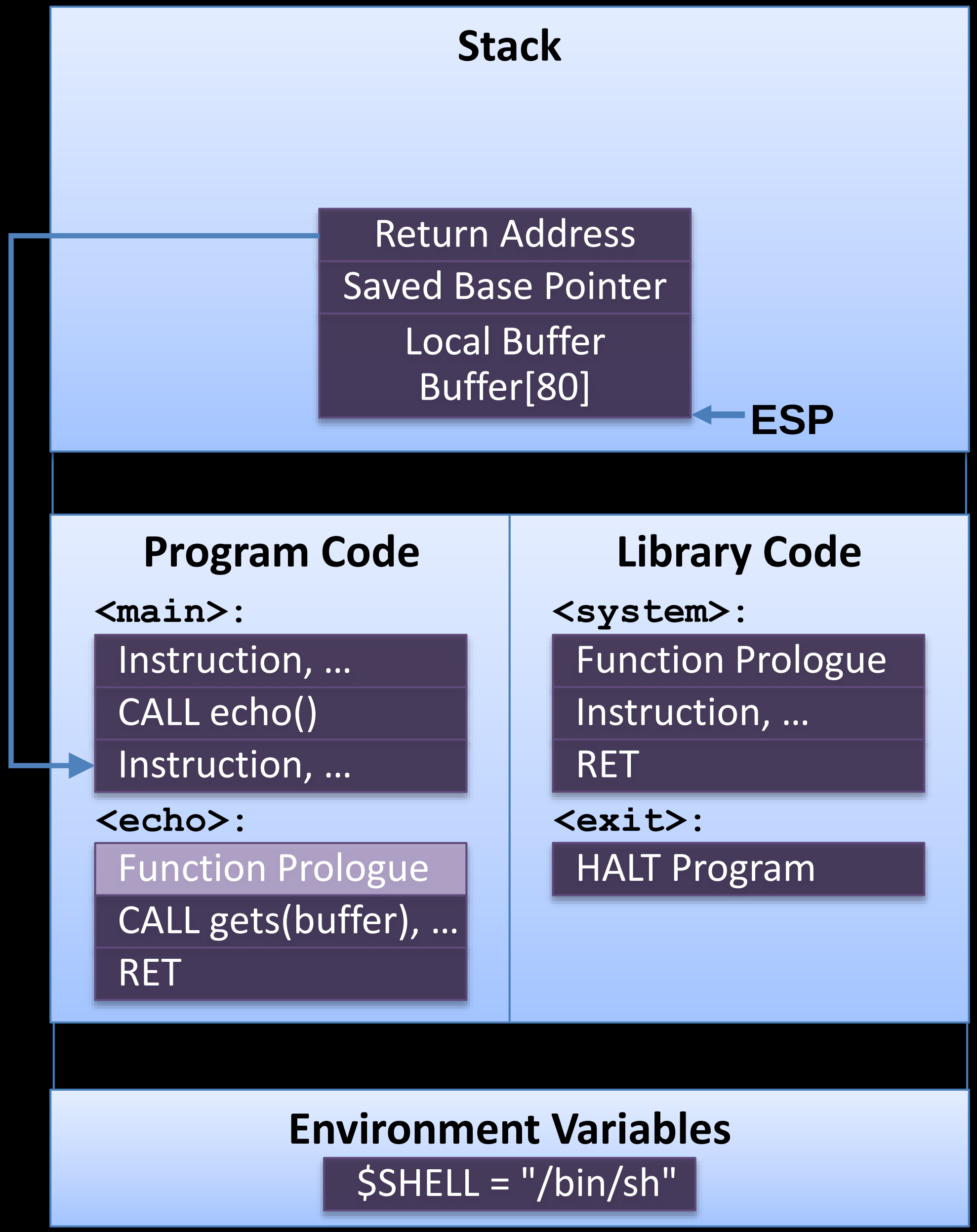
```
<exit>:
```

HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Stack

Return Address

Saved Base Pointer

Local Buffer
Buffer[80]

ESP

Program Code

```
<main>:
```

Instruction, …

CALL echo()

Instruction, …

```
<echo>:
```

Function Prologue

CALL gets(buffer), …

RET

Library Code

```
<system>:
```

Function Prologue

Instruction, …

RET

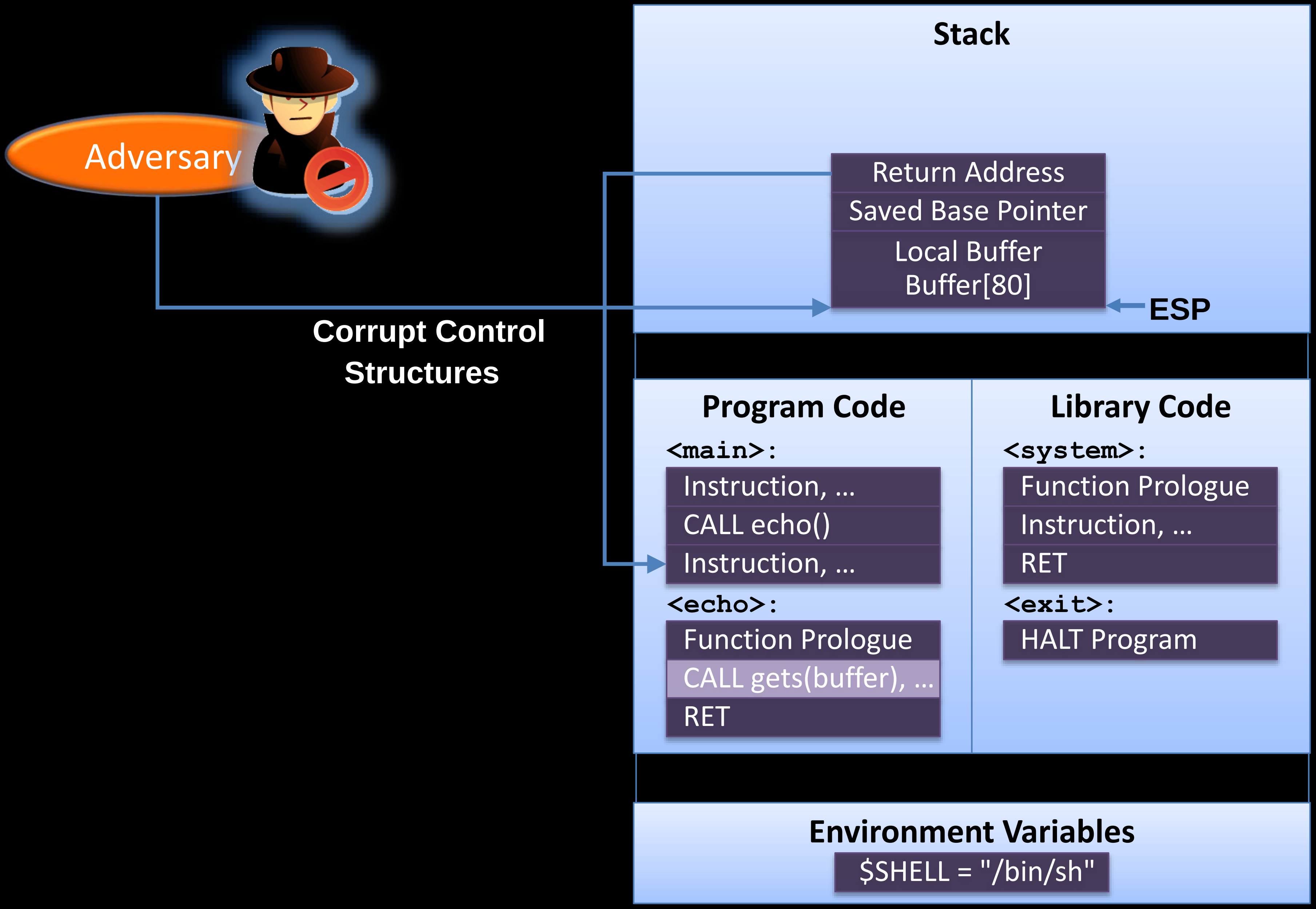
```
<exit>:
```

HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Adversary

Corrupt Control Structures

Stack

Return Address

Saved Base Pointer

Local Buffer
Buffer[80]

ESP

Program Code

<main>:

Instruction, …

CALL echo()

Instruction, …

<echo>:

Function Prologue

CALL gets(buffer), …

RET

Library Code

<system>:

Function Prologue

Instruction, …

RET

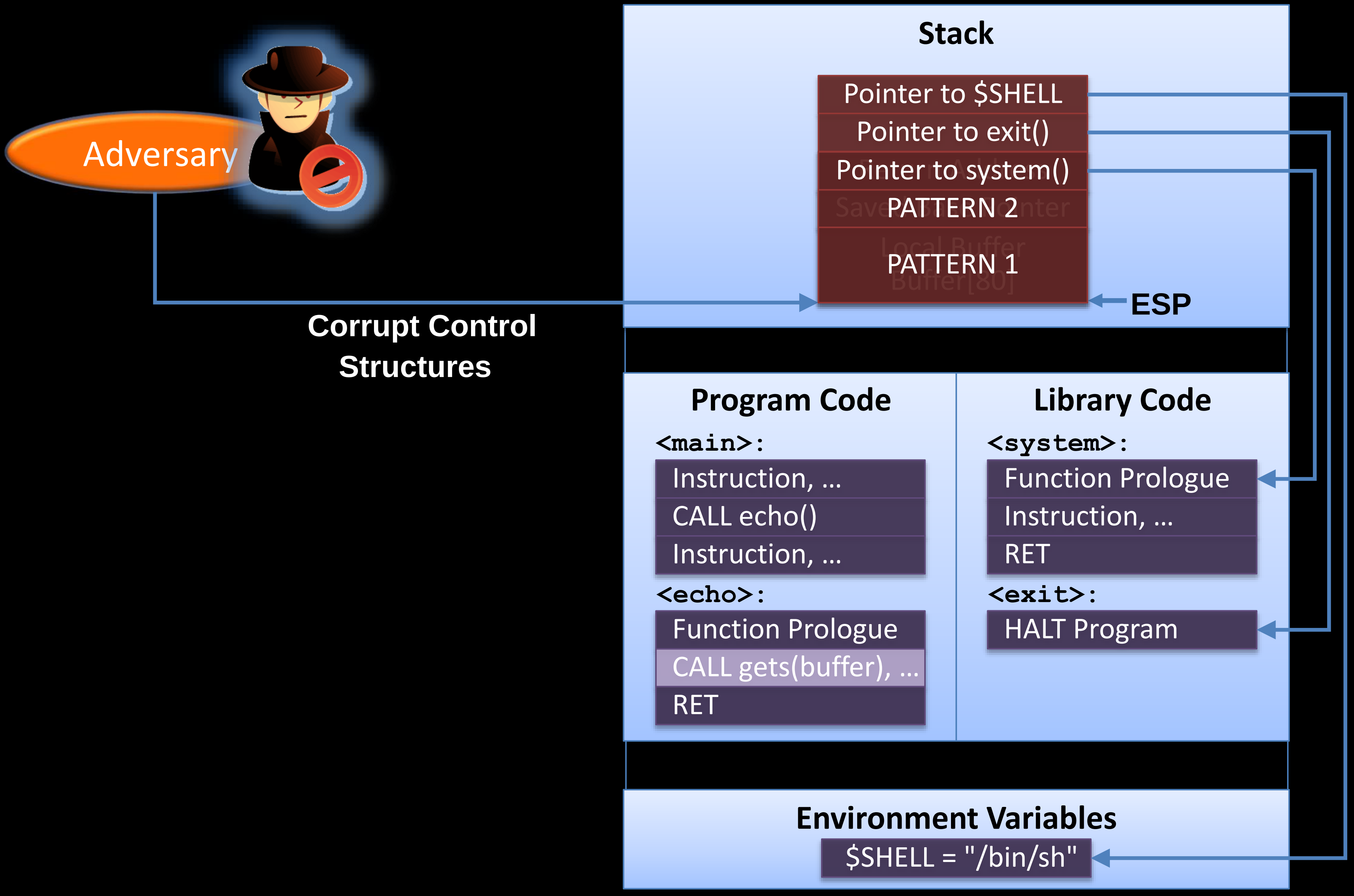
<exit>:

HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Adversary

Corrupt Control Structures

Stack

Pointer to $SHELL

Pointer to exit()

Pointer to system()

PATTERN 2

PATTERN 1

ESP

Program Code

<main>:

- Instruction, …
- CALL echo()
- Instruction, …

<echo>:

- Function Prologue
- CALL gets(buffer), …
- RET

Library Code

<system>:

- Function Prologue
- Instruction, …
- RET

<exit>:

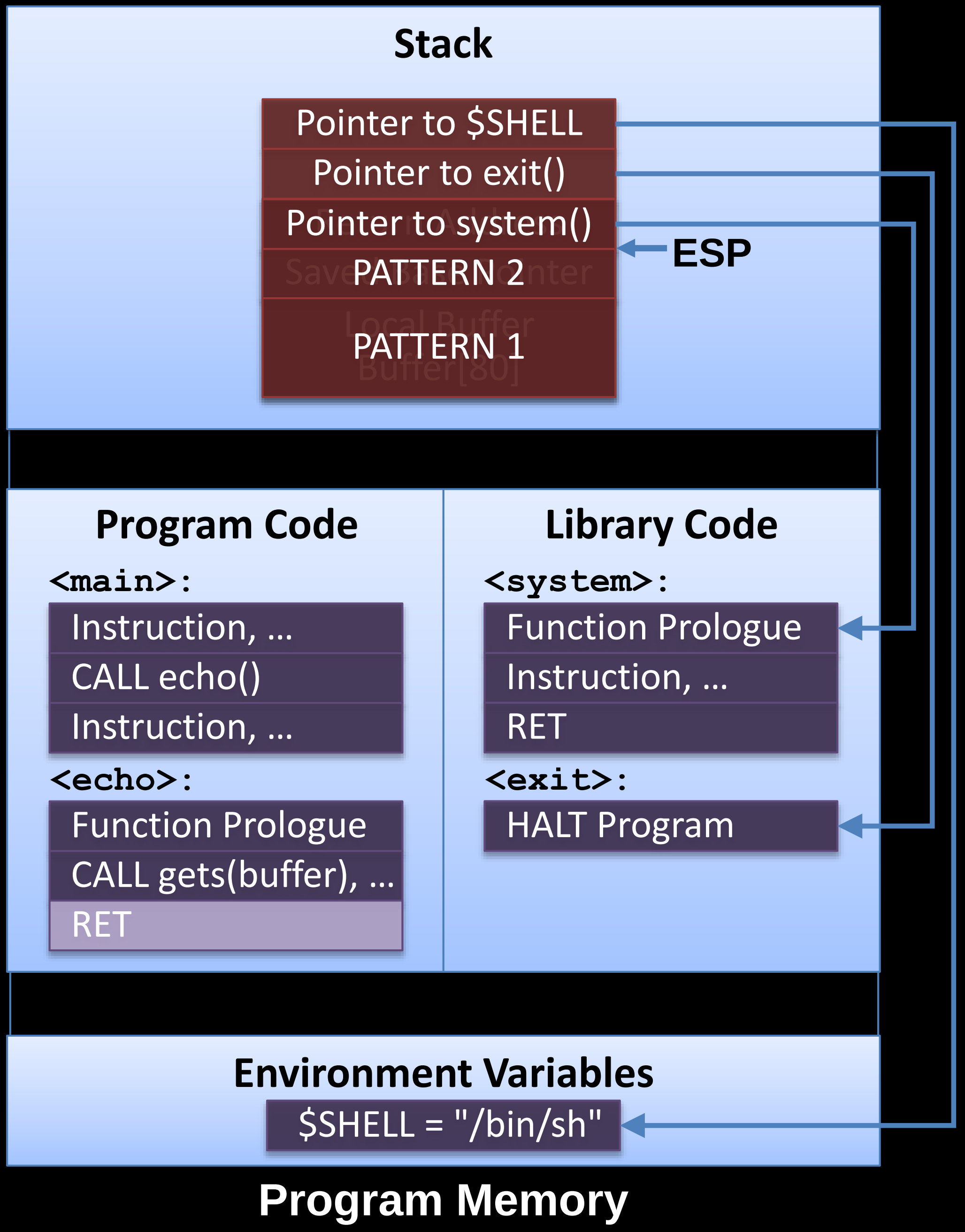
- HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory

echo() now returns!



Stack

Pointer to $SHELL

Pointer to exit()

Pointer to system()

ESP

PATTERN 2

PATTERN 1

Program Code

```
<main>:
```

Instruction, …

CALL echo()

Instruction, …

```
<echo>:
```

Function Prologue

CALL gets(buffer), …

RET

Library Code

```
<system>:
```

Function Prologue

Instruction, …

RET

```
<exit>:
```

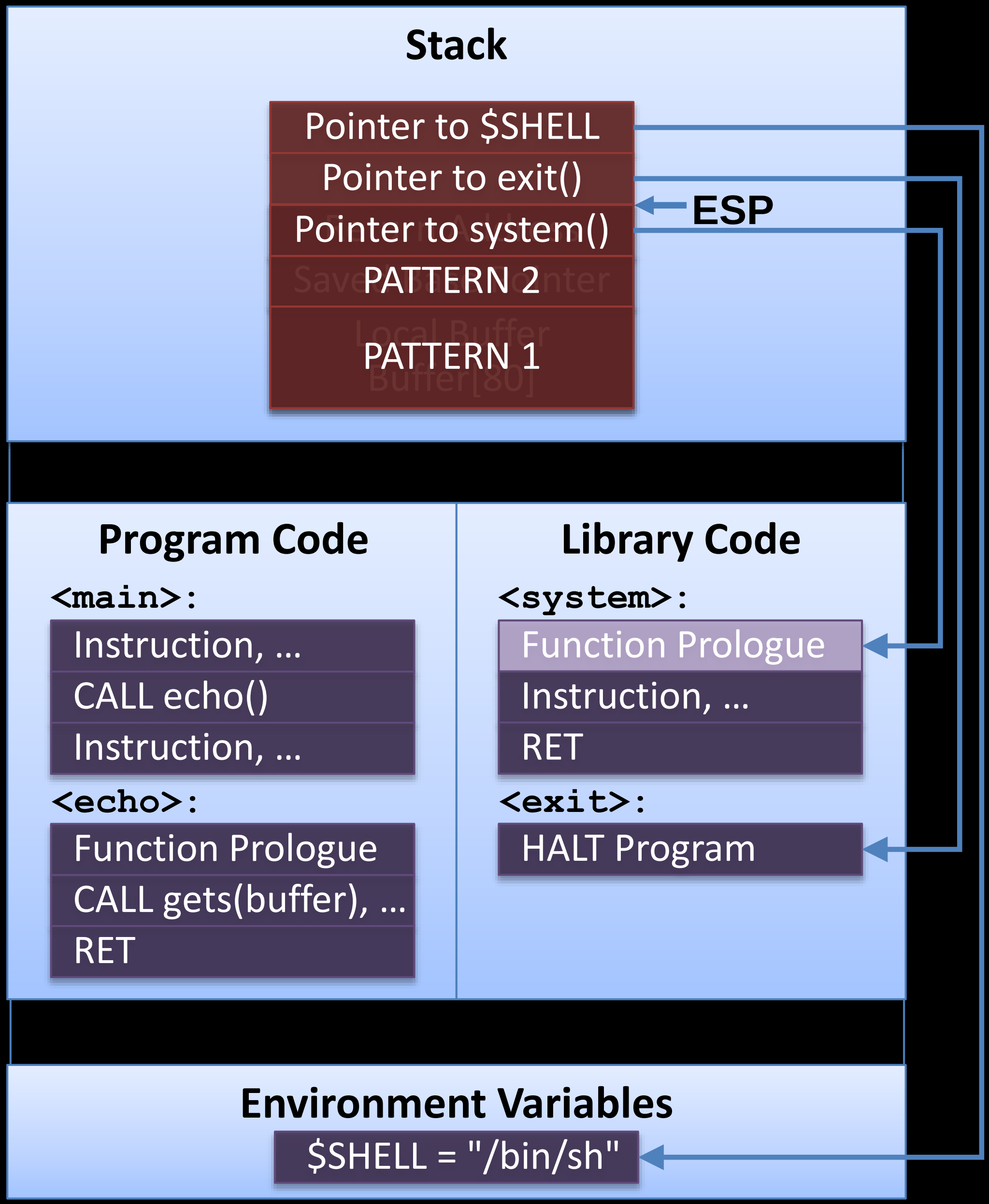
HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory

Adversary



Stack

Pointer to $SHELL

Pointer to exit()

ESP

Pointer to system()

PATTERN 2

PATTERN 1

Program Code

```
<main>:
```

Instruction, …

CALL echo()

Instruction, …

```
<echo>:
```

Function Prologue

CALL gets(buffer), …

RET

Library Code

```
<system>:
```

Function Prologue

Instruction, …

RET

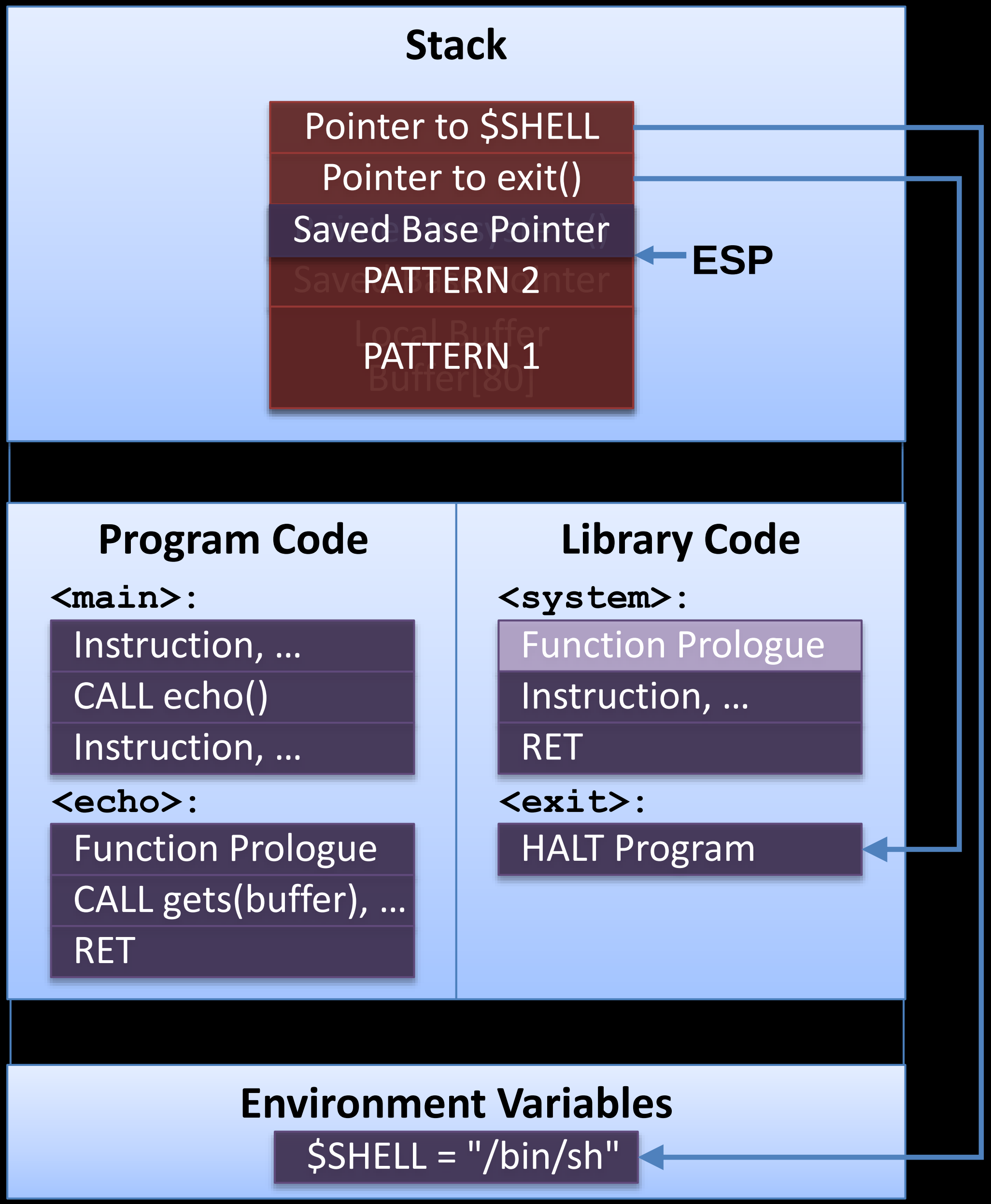
```
<exit>:
```

HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Stack

- Pointer to $SHELL
- Pointer to exit()
- Saved Base Pointer
- PATTERN 2
- PATTERN 1

Save Pointer

Local Buffer

Buffer[80]

ESP

Program Code

`<main>:`

- Instruction, …
- CALL echo()
- Instruction, …

`<echo>:`

- Function Prologue
- CALL gets(buffer), …
- RET

Library Code

`<system>:`

- Function Prologue
- Instruction, …
- RET

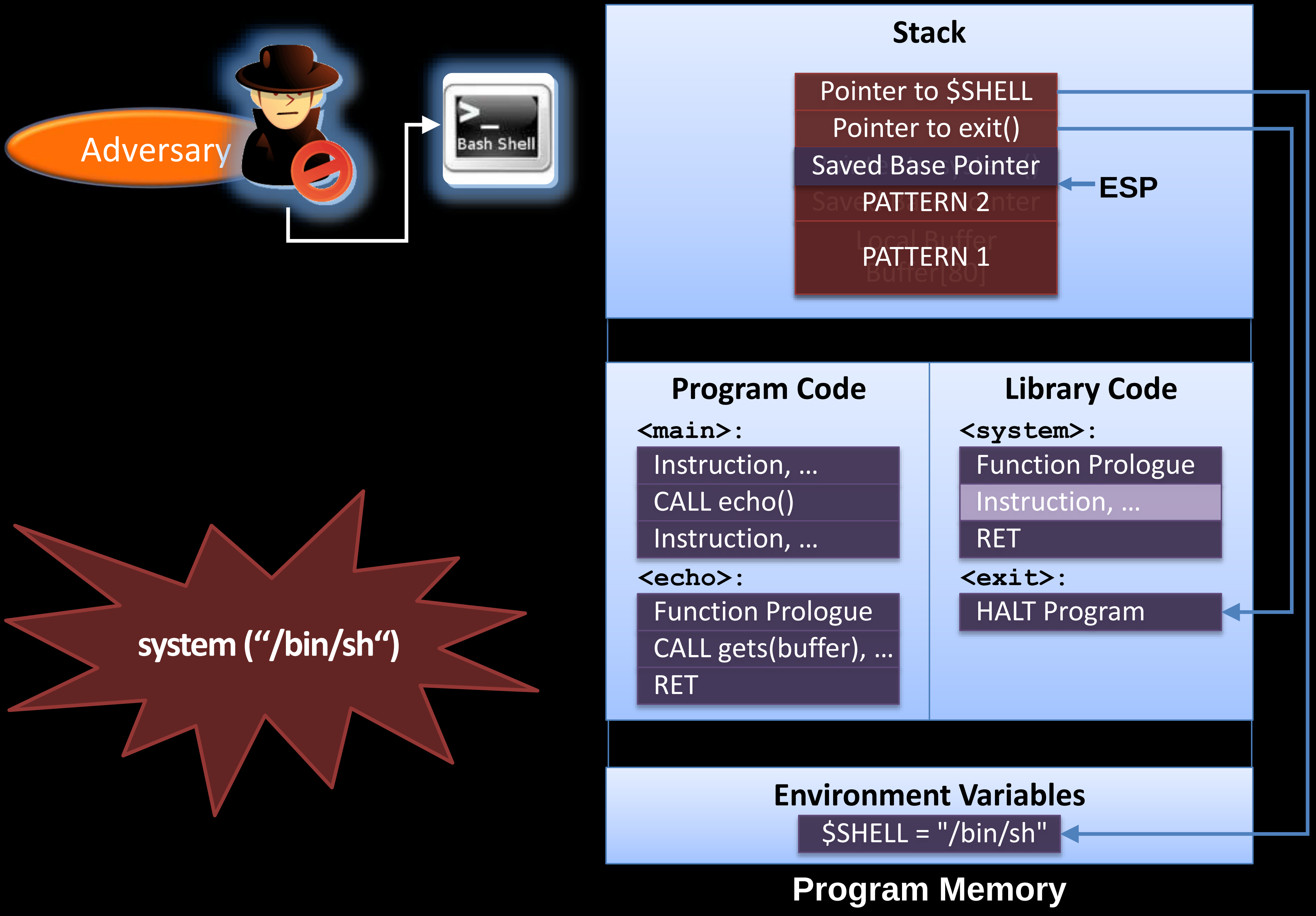
`<exit>:`

- HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Adversary

Bash Shell

system (“/bin/sh“)

Stack

Pointer to $SHELL

Pointer to exit()

Saved Base Pointer

PATTERN 2

PATTERN 1

ESP

Program Code

```
<main>:
```

Instruction, …

CALL echo()

Instruction, …

```
<echo>:
```

Function Prologue

CALL gets(buffer), …

RET

Library Code

```
<system>:
```

Function Prologue

Instruction, …

RET

```
<exit>:
```

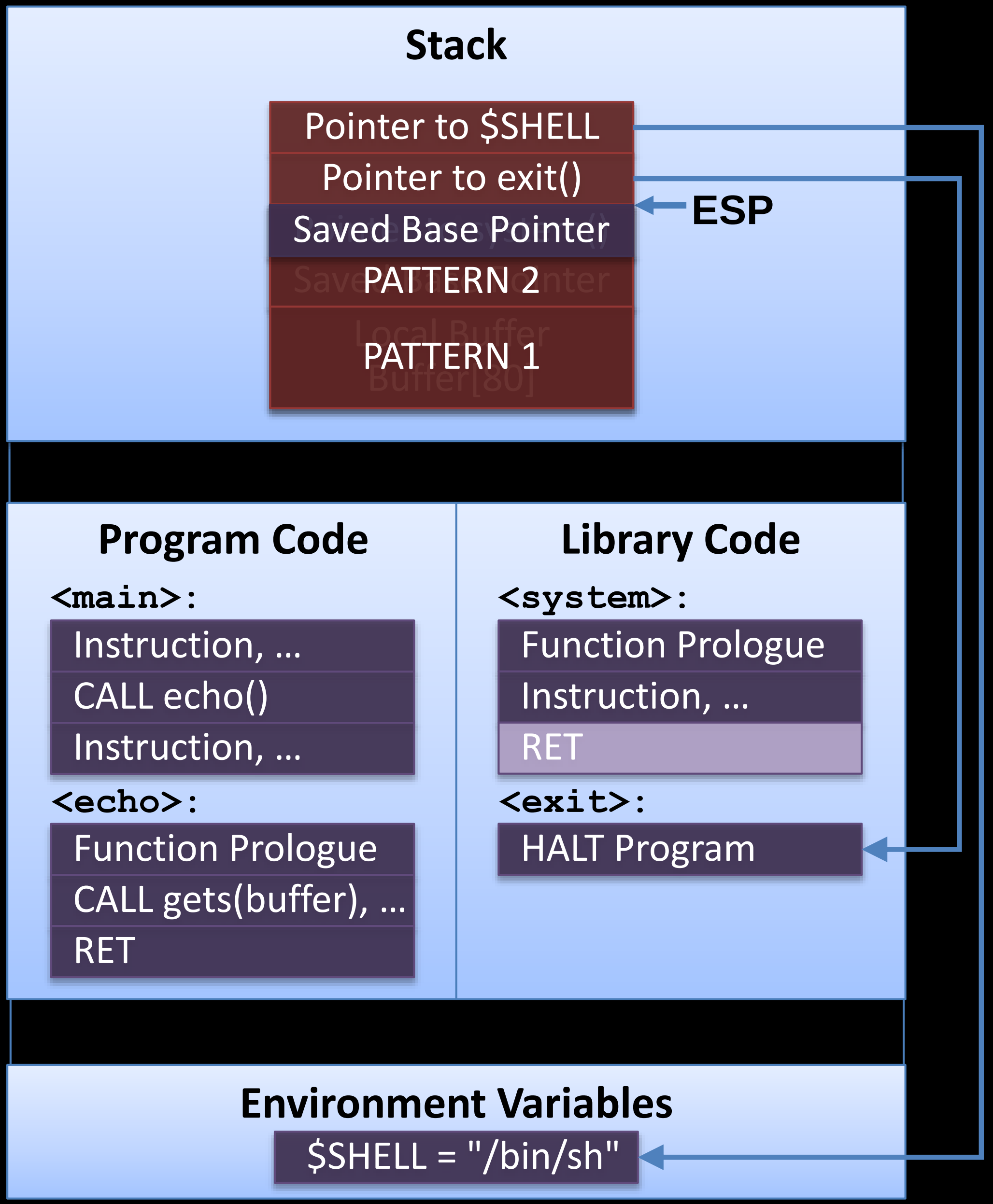
HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory

system ("/bin/sh") returning



Stack

- Pointer to $SHELL
- Pointer to exit()
- Saved Base Pointer
- PATTERN 2
- PATTERN 1

ESP

Program Code

`<main>:`

- Instruction, …
- CALL echo()
- Instruction, …

`<echo>:`

- Function Prologue
- CALL gets(buffer), …
- RET

Library Code

`<system>:`

- Function Prologue
- Instruction, …
- RET

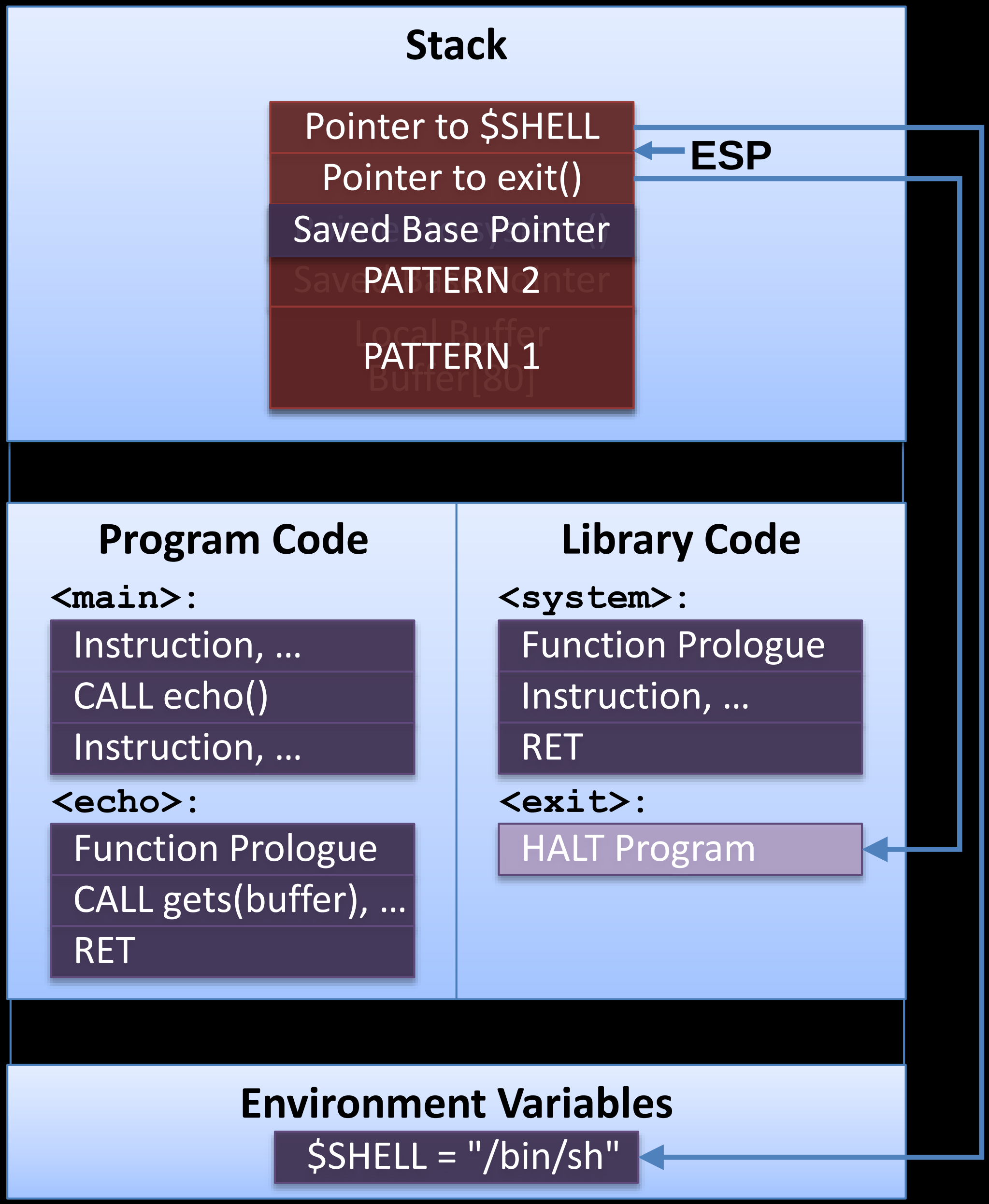
`<exit>:`

- HALT Program

Environment Variables

$SHELL = "/bin/sh"

Program Memory



Stack

Pointer to $SHELL

Pointer to exit()

ESP

Saved Base Pointer

PATTERN 2

PATTERN 1

Program Code

`<main>:`

Instruction, …

CALL echo()

Instruction, …

`<echo>:`

Function Prologue

CALL gets(buffer), …

RET

Library Code

`<system>:`

Function Prologue

Instruction, …

RET

`<exit>:`

HALT Program

Environment Variables

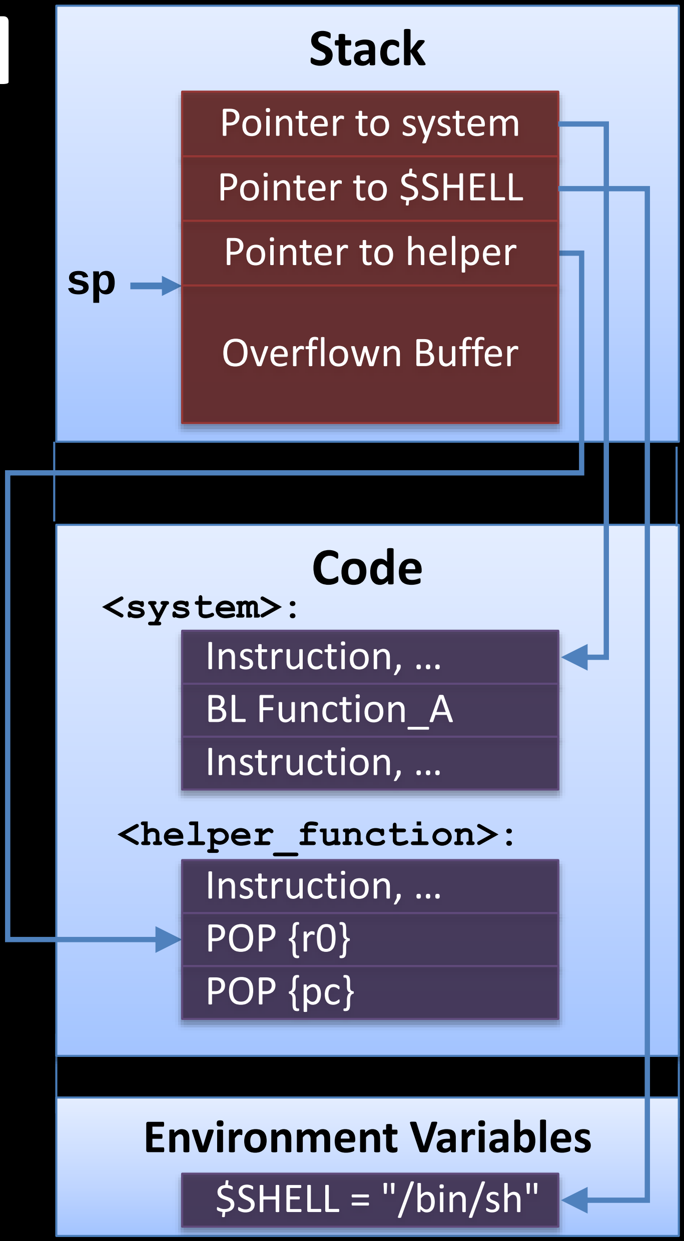
$SHELL = "/bin/sh"

Program Memory

Program terminates

return-into-libc on ARM

- The first four function arguments are passed via registers
- Hence, how do we initialize the arguments before calling system() ?
  - We return to an instruction sequence that loads the argument from the stack



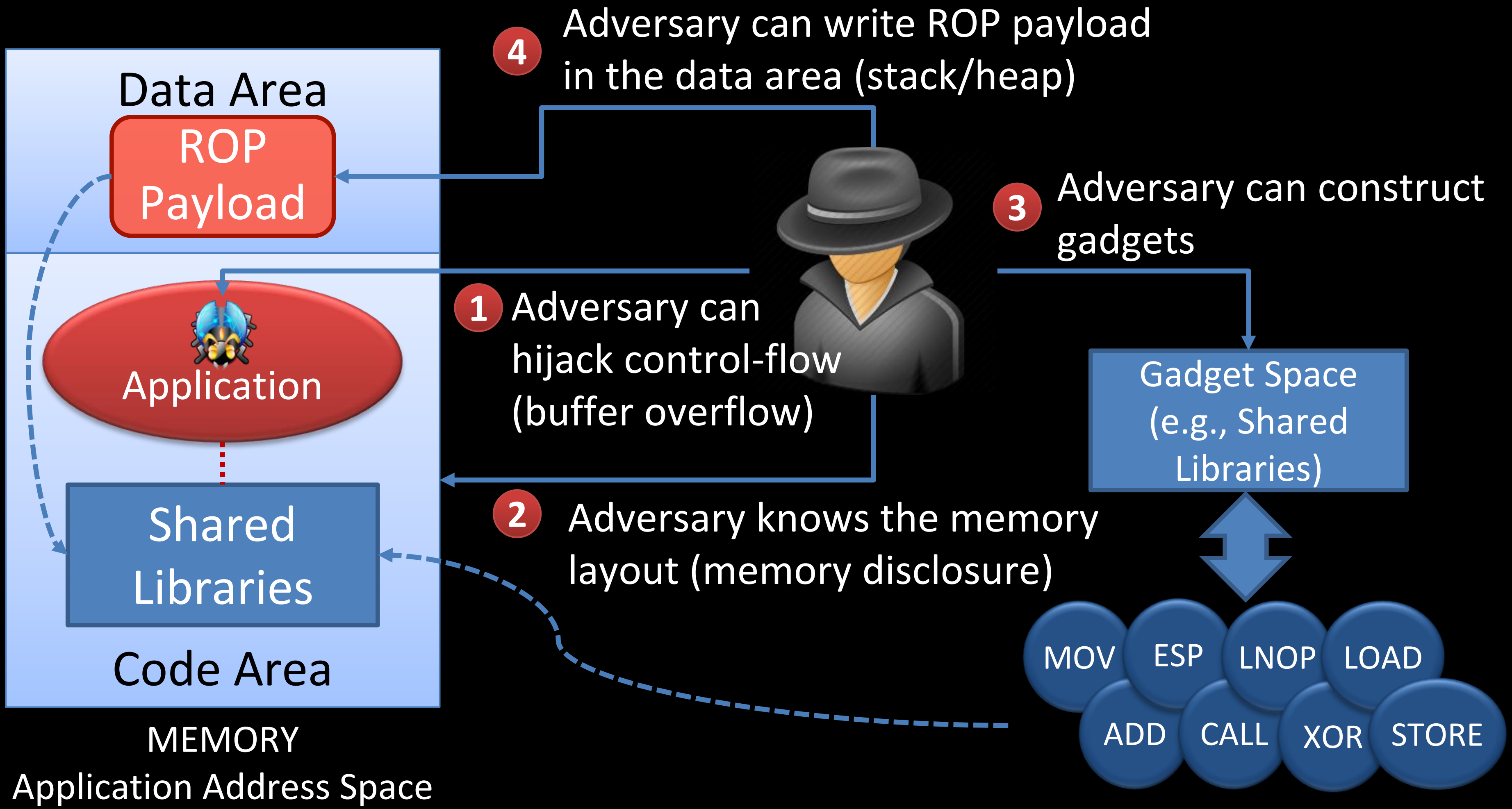
Limitations

- No branching, i.e., no arbitrary code execution

- Critical functions can be eliminated or wrapped

Generalization of return-into-libc attacks:
return-oriented programming (ROP)
[Shacham, ACM CCS 2007]

ROP Adversary Model/Assumption



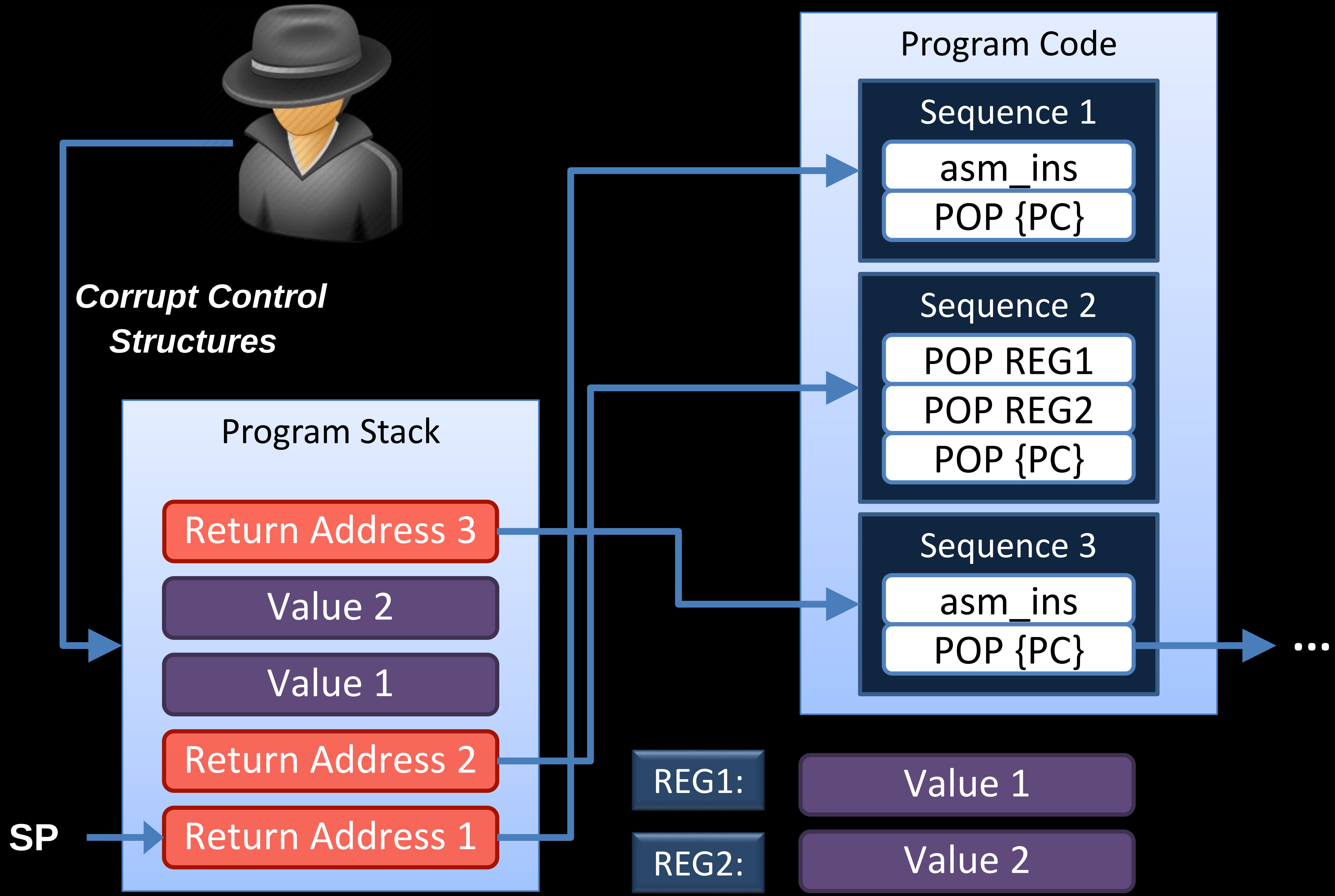
4 Adversary can write ROP payload in the data area (stack/heap)

3 Adversary can construct gadgets

1 Adversary can hijack control-flow (buffer overflow)

2 Adversary knows the memory layout (memory disclosure)

ROP Attack Technique: Overview



Corrupt Control Structures

Program Stack

Return Address 3

Value 2

Value 1

Return Address 2

Return Address 1

SP

Program Code

Sequence 1

asm_ins

POP {PC}

Sequence 2

POP REG1

POP REG2

POP {PC}

Sequence 3

asm_ins

POP {PC}

…

REG1: Value 1

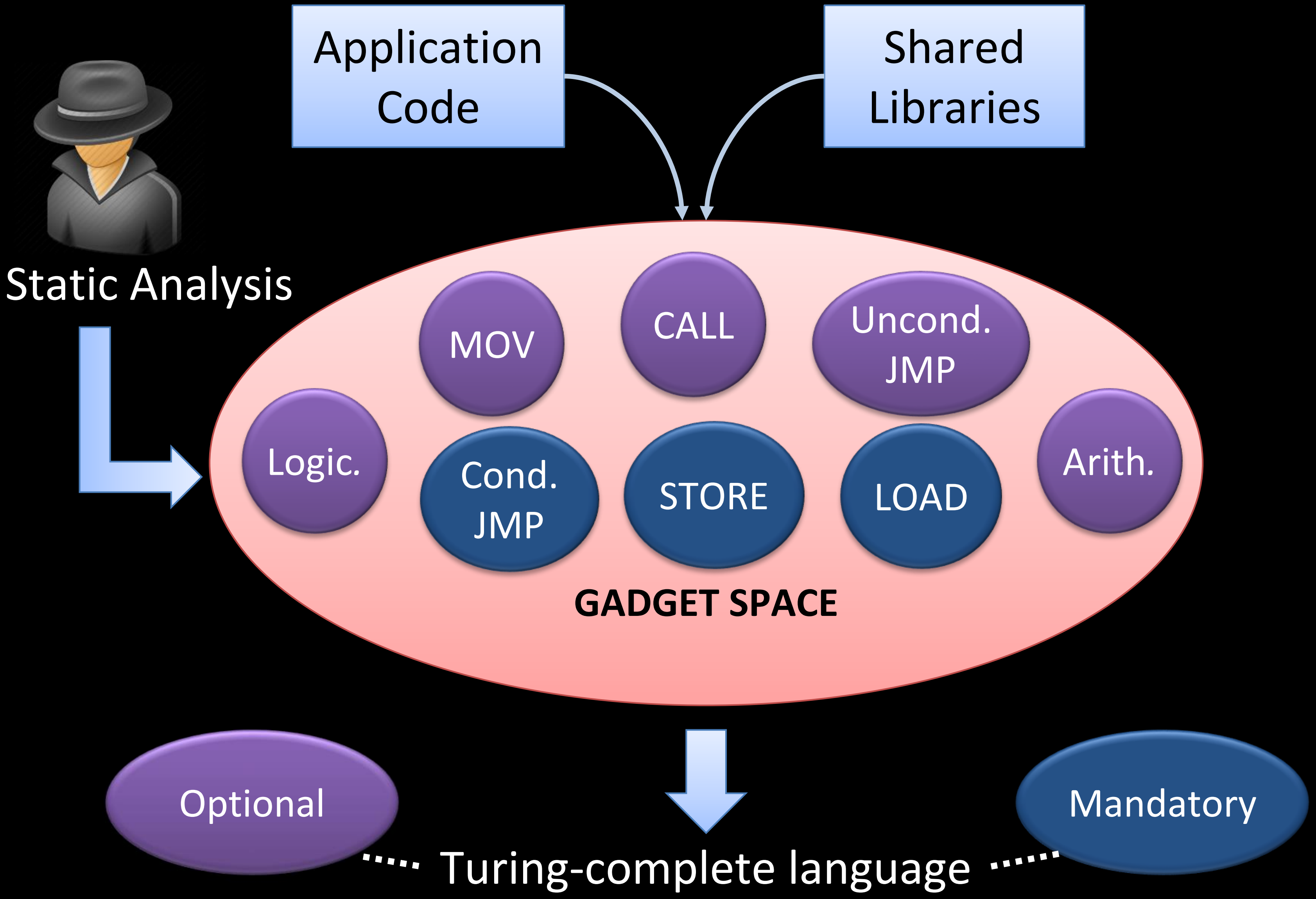
REG2: Value 2

Summary of Basic Idea

- Perform arbitrary computation with return-into-libc techniques
- Approach
  - Use small instruction sequences (e.g., of libc) instead of using whole functions
  - Instruction sequences range from 2 to 5 instructions
  - All sequences end with a return (POP{PC}) instruction
  - Instruction sequences are chained together to a gadget
  - A gadget performs a particular task (e.g., load, store, xor, or branch)
  - Afterwards, the adversary enforces his desired actions by combining the gadgets

Special Aspects of ROP

Code Base and Turing-Completeness



Application Code

Shared Libraries

Static Analysis

MOV

CALL

Uncond. JMP

Logic.

Cond. JMP

STORE

LOAD

Arith.

GADGET SPACE

Optional

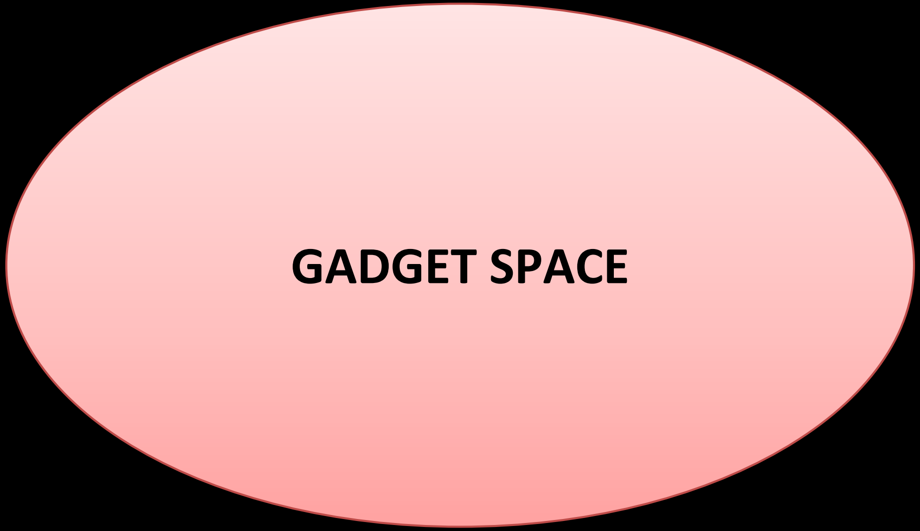
Mandatory

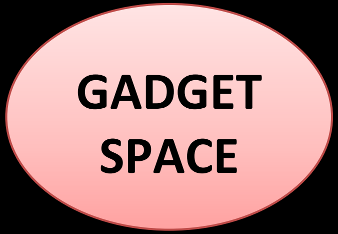
Turing-complete language

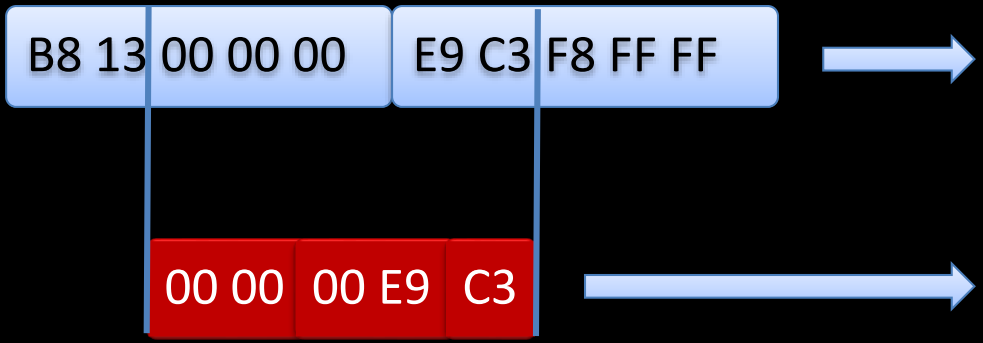
Gadget Space on Different Architectures

Architectures with no memory alignment, e.g., Intel x86

Architectures with memory alignment, e.g., SPARC, ARM







Intended Code

```
mov $0x13,%eax
jmp 3aae9
```

Unintended Code

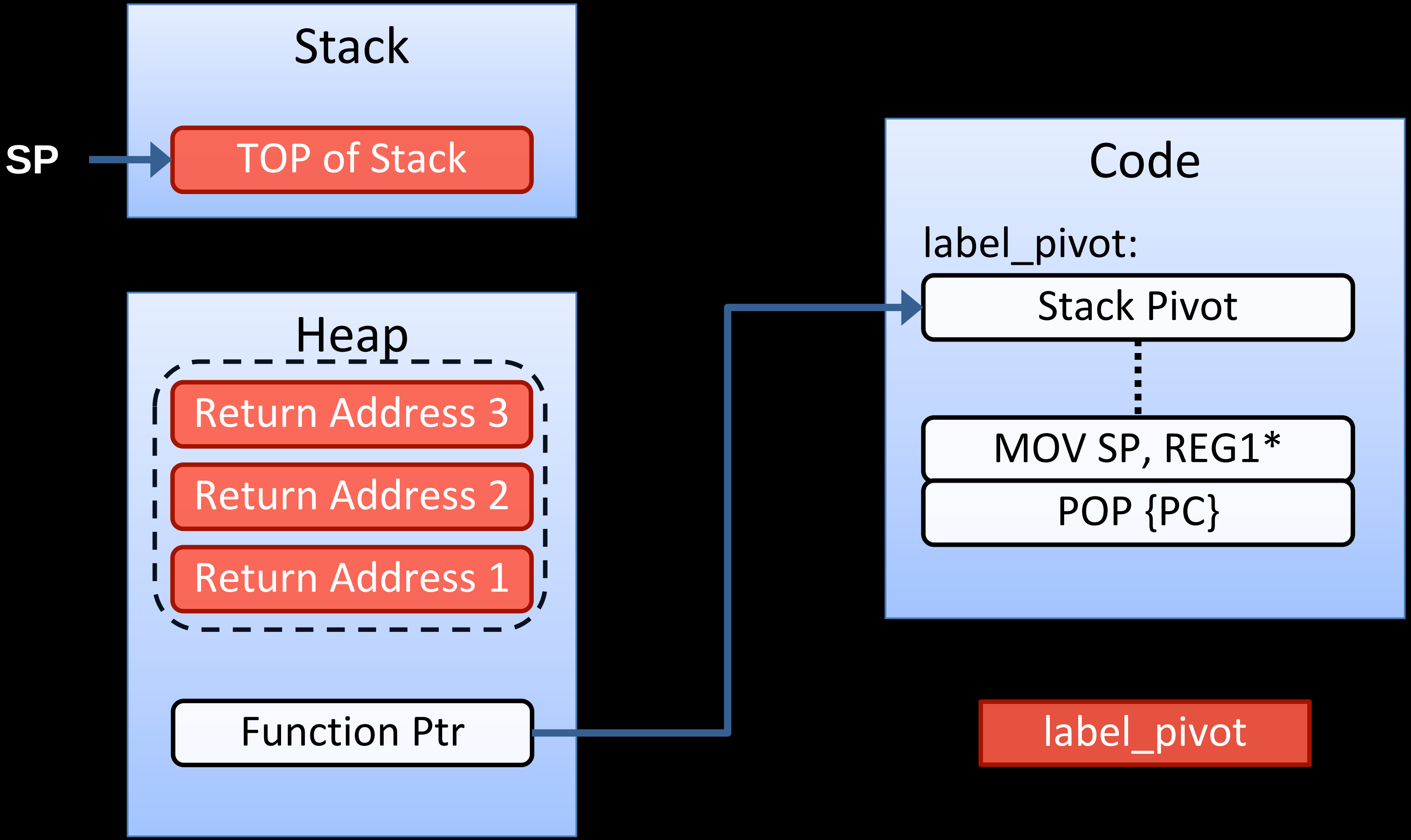
```
add %al,(%eax)
add %ch,%cl
ret
```

Stack Pivot

[Zovi, RSA Conference 2010]

- Stack pointer plays an important role
  - It operates as an instruction pointer in ROP attacks
- Challenge
  - In order to launch a ROP exploit based on a heap overflow, we need to set the stack pointer to point to the heap
  - This is achieved by a stack pivot

Stack Pivot in Detail



*REG1 is controlled by the adversary and holds beginning of ROP payload

ROP Variants

- Motivation: return address protection (shadow stack)
  - Validate every return (intended and unintended) against valid copies of return addresses
    [Davi et al., AsiaCCS 2011]
- Exploit indirect jumps and calls
  - ROP without returns
    [Checkoway et al., ACM CCS 2010]

CURRENT RESEARCH

SELECTED

1997
ret2libc
Solar Designer

2001
Advanced ret2libc
Nergal

2005
Borrowed Code Chunk Exploitation
Krahmer

2007
ROP on x86
Shacham (CCS)

2008
ROP on SPARC
Buchanan et al (CCS)

ROP on Atmel AVR
Francillon et al (CCS)

2009
ROP Rootkits
Hund et al (USENIX)

ROP on PowerPC
FX Lindner (BlackHat)

ROP on ARM/iOS
Miller et al (BlackHat)

2010
ROP without Returns
Checkoway et al (CCS)

Practical ROP
Zovi (RSA Conference)

Pwn2Own (iOS/IE)
Iozzo et al / Nils

2011/
2012
Real-World Exploits

2013
JIT-ROP
Snow et al (IEEE S&P)

2014
Blind ROP
Bittau et al (IEEE S&P)

Out-Of-Control
Göktas et al (IEEE S&P)

Stitching Gadgets
Davi et al (USENIX)

Flushing Attacks
Schuster et al (RAID)

ROP is Dangerous
Carlini et al (USENIX)

Our Work & Involvement

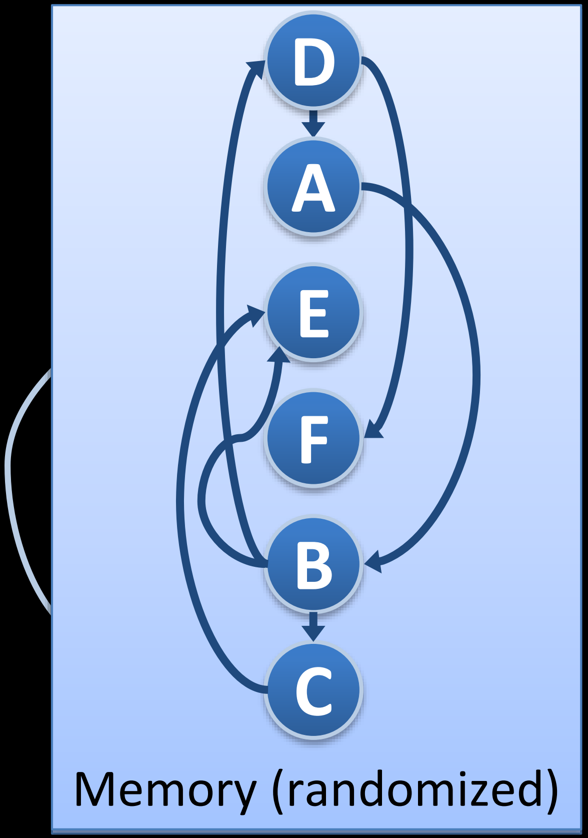
- **Attacks**
  - Return-Oriented Programming without Returns [CCS 2010]
  - Privilege Escalation Attacks on Android [ISC 2010]
  - Just-In-Time Return-oriented Programming (JIT-ROP) [IEEE S&P 2013, Best Student Paper] & [BlackHat USA 2013]
  - Stitching the Gadgets [USENIX Security 2014] & [BlackHat USA 2014]
  - COOP [IEEE Security & Privacy 2015]
  - Losing Control [CCS 2015]
- **Detection & Prevention**
  - ROPdefender [AsiaCCS 2011]
  - Mobile Control-Flow Integrity (MoCFI) [NDSS 2012]
  - XIFER: Fine-Grained ASLR [AsiaCCS 2013]
  - Filtering ROP Payloads [RAID 2013]
  - Isomeron [NDSS 2015]
  - Readactor [IEEE Security & Privacy 2015]
  - HAFIX: Fine-Grained CFI in Hardware [DAC 2014, DAC 2015]
  - Readactor++ [CCS 2015]

In this lecture

Main Defense Techniques

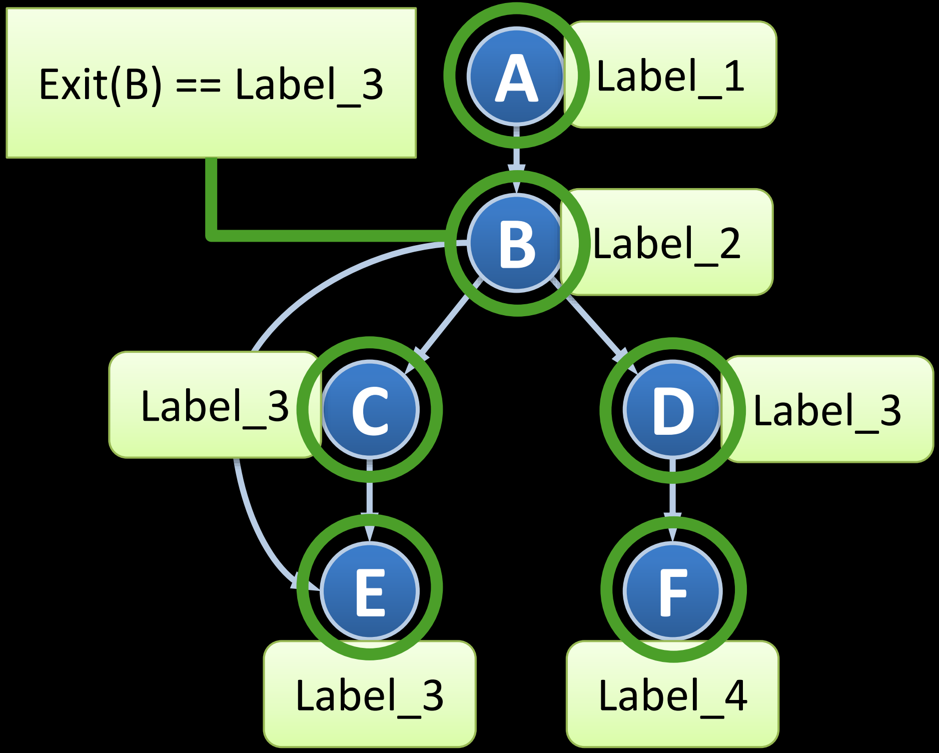
(Fine-grained) Code Randomization

[Cohen 1993 & Larsen et al., SoK IEEE S&P 2014]



Control-Flow Integrity (CFI)

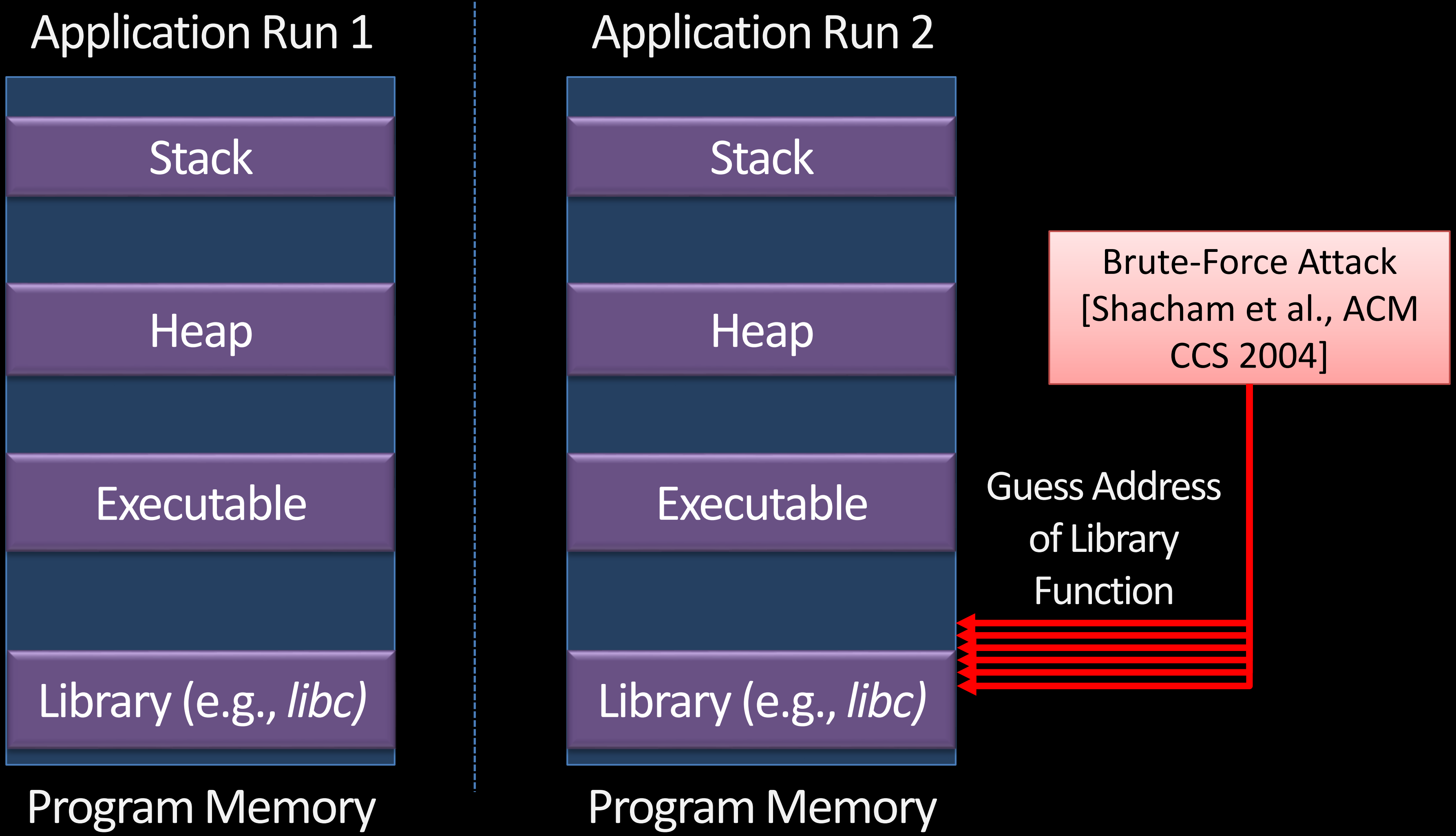
[Abadi et al., CCS 2005 & TISSEC 2009]



ASLR – Address Space Layout Randomization

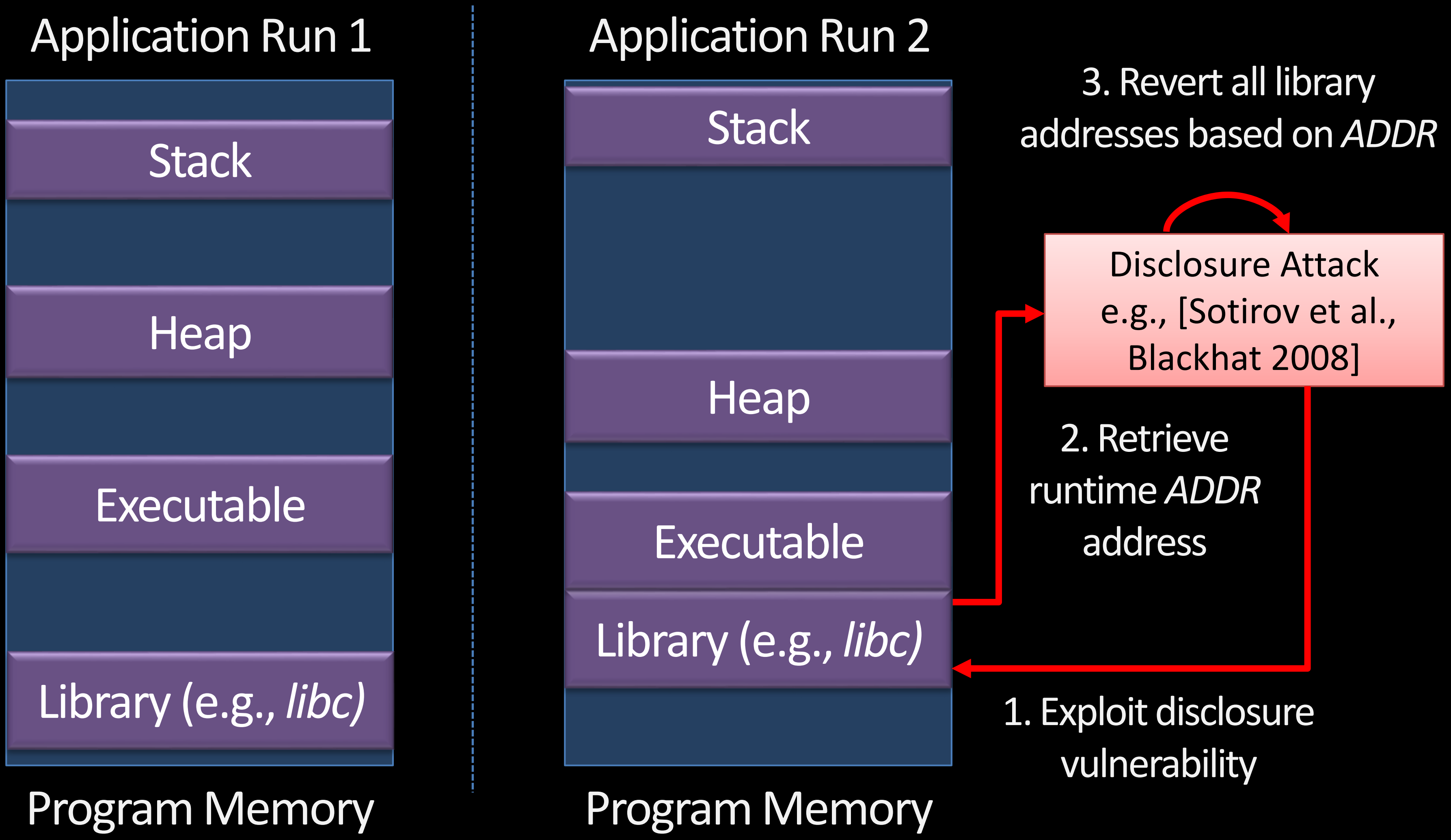
Basics of Code Randomization

- ASLR randomizes the base address of code/data segments

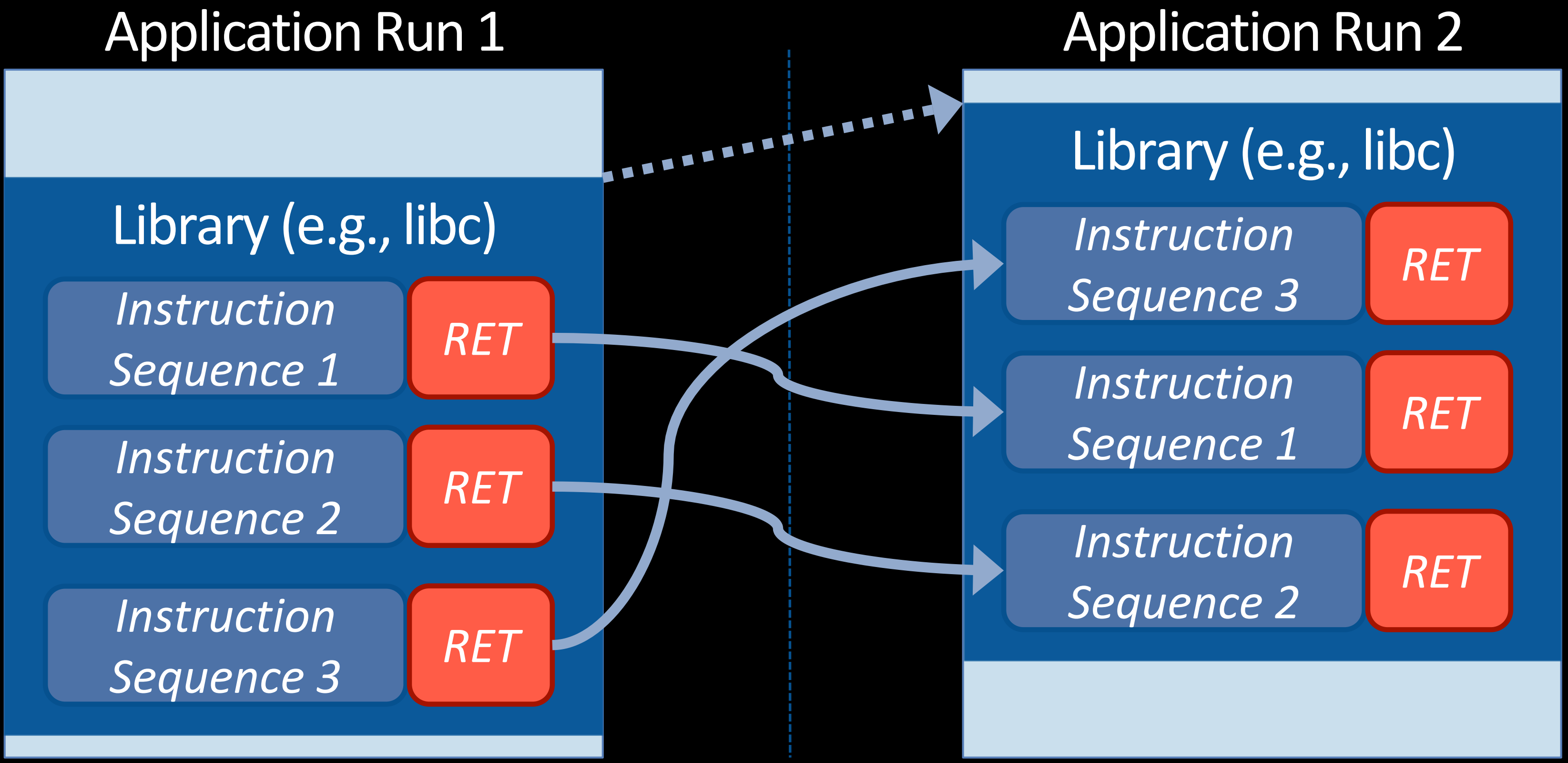


Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments


Application Run 1
Stack
Heap
Executable
Library (e.g., *libc*)
Program Memory
Application Run 2
Stack
Heap
Executable
Library (e.g., *libc*)
Program Memory
3. Revert all library addresses based on *ADDR*
Disclosure Attack e.g., [Sotirov et al., Blackhat 2008]
2. Retrieve runtime *ADDR* address
1. Exploit disclosure vulnerability

Fine-Grained ASLR



- **ORP** [Pappas et al., IEEE S&P 2012]: Instruction reordering/substitution within a BBL
- **ILR** [Hiser et al., IEEE S&P 2012]: Randomizing each instruction‘s location
- **STIR** [Wartell et al., ACM CCS 2012] & **XIFER** [with Davi et al., AsiaCCS 2013]: Permutation of BBLs

Does Fine-Grained ASLR Provide a Viable Defense in the Long Run?

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization

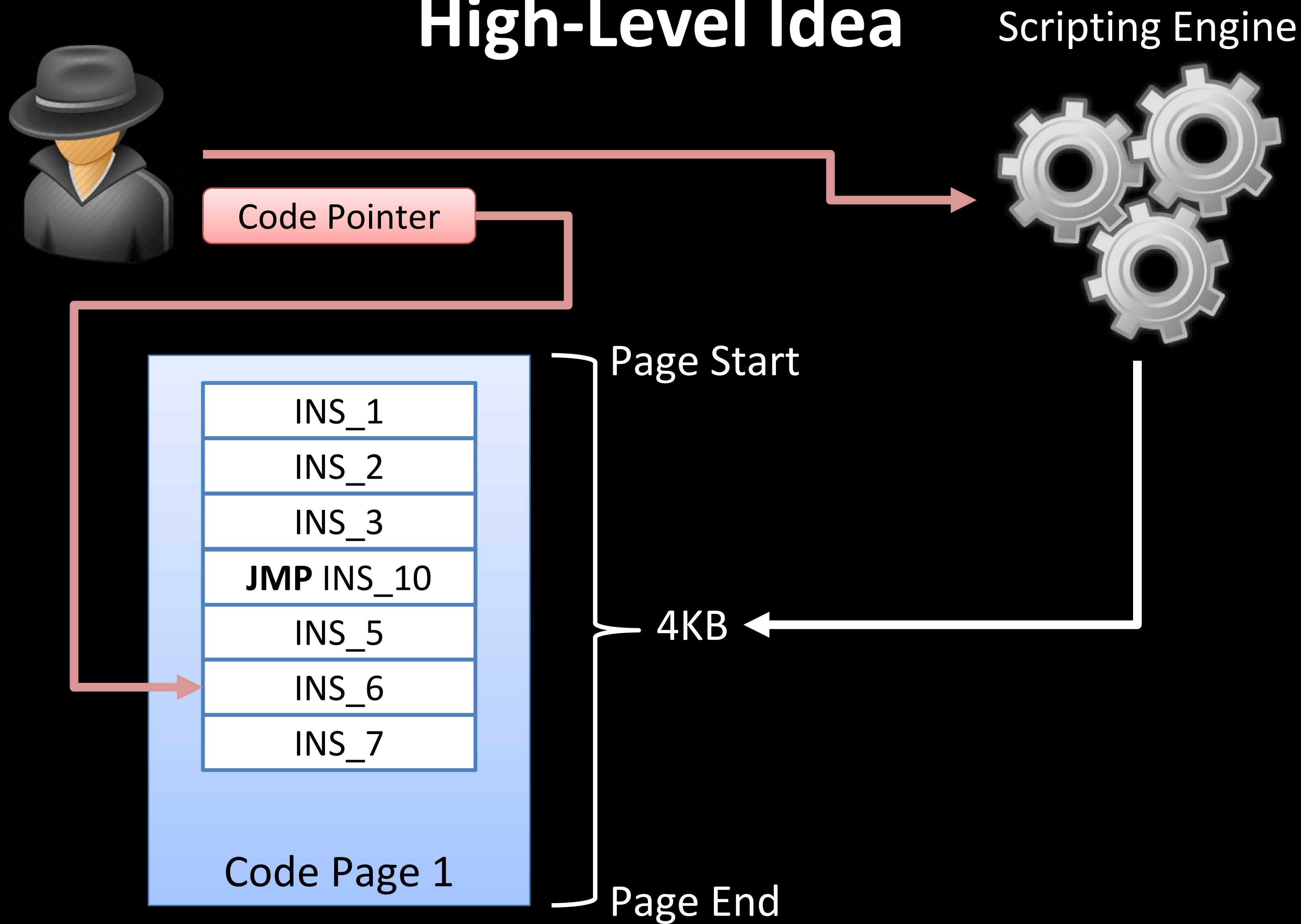
IEEE Security and Privacy Best Student Paper 2013

Kevin Z. Snow (UNC Chapel Hill), Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose (UNC Chapel Hill), Ahmad-Reza Sadeghi

Contributions

1. A novel ROP attack that undermines fine-grained ASLR
2. We show that memory disclosures are far more damaging than previously believed
3. A prototype exploit framework that demonstrates one instantiation of our idea, called JIT-ROP

High-Level Idea



Scripting Engine

Code Pointer

INS_1

INS_2

INS_3

JMP INS_10

INS_5

INS_6

INS_7

Code Page 1

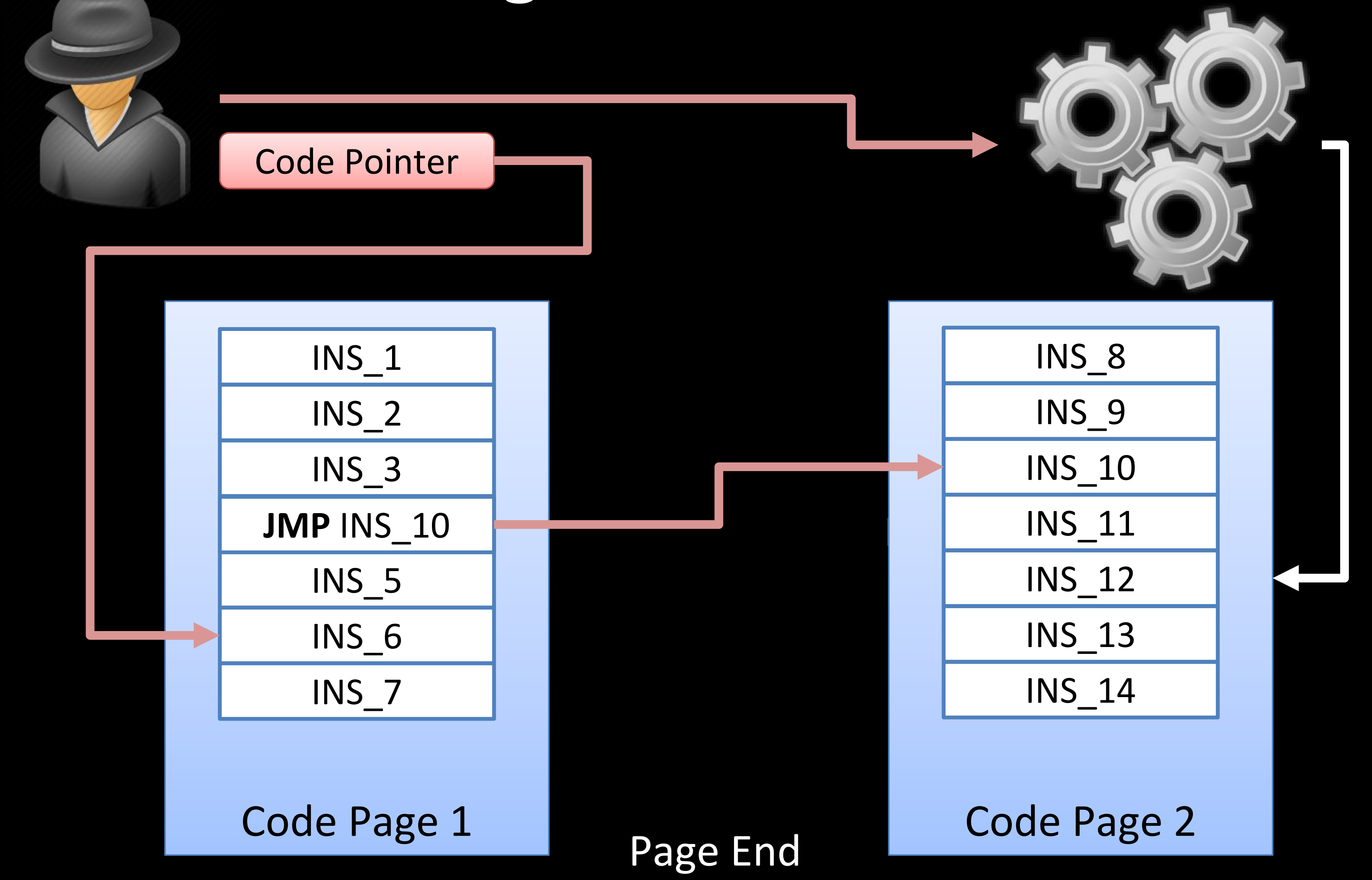
Page Start

4KB

Page End

High-Level Idea

Scripting Engine



Code Pointer

INS_1
INS_2
INS_3
JMP INS_10
INS_5
INS_6
INS_7

Code Page 1

INS_8
INS_9
INS_10
INS_11
INS_12
INS_13
INS_14

Code Page 2

Page End

Applying JIT-ROP to Internet Explorer 8

- We applied JIT-ROP to a real-world vulnerability in IE 8
  - CVE-2012-1876: Heap overflow vulnerability
  - Within 7 seconds, our attack harvested code pages, identified and constructed useful ROP gadgets, and finally build and executed the payload

For more evaluation results and details check out our paper and BlackHat USA 2013 slides

Possible Defenses

Execute-only memory

Software-based: Execute-no-Read

[Backes et al., ACM CCS 2014]

Hardware-based: Readactor

[with Crane et al., IEEE S&P 2015]

Execution-path randomization

Isomeron

[Davi et al., NDSS 2015]

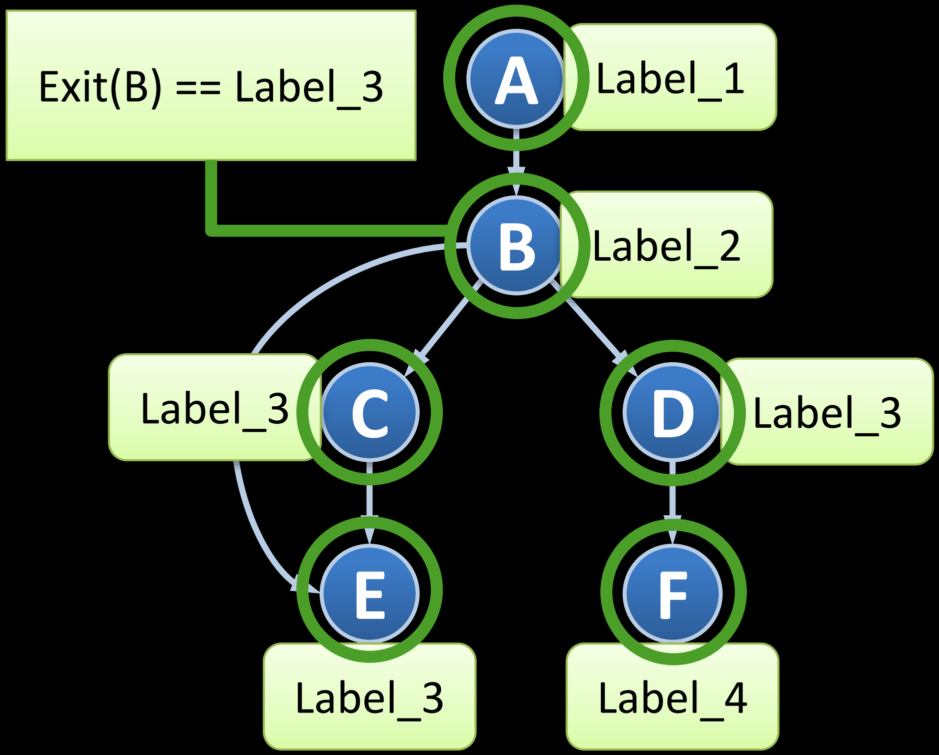
Control-flow Integrity (CFI)

CFI does not rely on any randomization key

Control-Flow Integrity (CFI)

[Abadi et al., CCS 2005 & TISSEC 2009]

A general defense against code-reuse attacks



CFI Defense Literatur

SELECTED

2002	**Program Shepherding** *Kiriansky et al. (USENIX Sec.)*
2005	**Control-Flow Integrity (CFI)** *Abadi et al. (CCS 2005)*
2006	**XFI** *Abadi et al. (OSDI)* **Architectural Support for CFI** *Budiu et al. (ASID)*
2010	**HyperSafe** *Wang et al. (IEEE S&P)*
2011	**CFI and Data Sandboxing** *Zeng et al (CCS)* **Control-Flow Locking** *Bletch et al. (ACSAC)*
2012	**Branch Regulation** *Kayaalp et al (ISCA)* **Mobile CFI** *Davi et al. (NDSS)*
2013	**Control-Flow Restrictor** *Pewny et al (ACSAC)* **kBouncer** *Pappas et al. (USENIX Sec.)* **bin-CFI** *Zhang et al. (USENIX Sec.)* **CCFIR** *Zhang et al. (IEEE S&P)*
2014	**ROPecker** *Cheng et al. (NDSS)* **Forward-Edge CFI** *Tice et al. (USENIX Sec.)* **SAFEDISPATCH** *Jang et al. (NDSS)* **Modular CFI** *Niu et al. (PLDI)* **RockJIT** *Niu et al. (CCS)* **HAFIX** *Davi et al. (DAC)*

Which Instructions to Protect?

Returns

- **Purpose**: Return to calling function
- **CFI Relevance**: Return address located on stack

Indirect Jumps

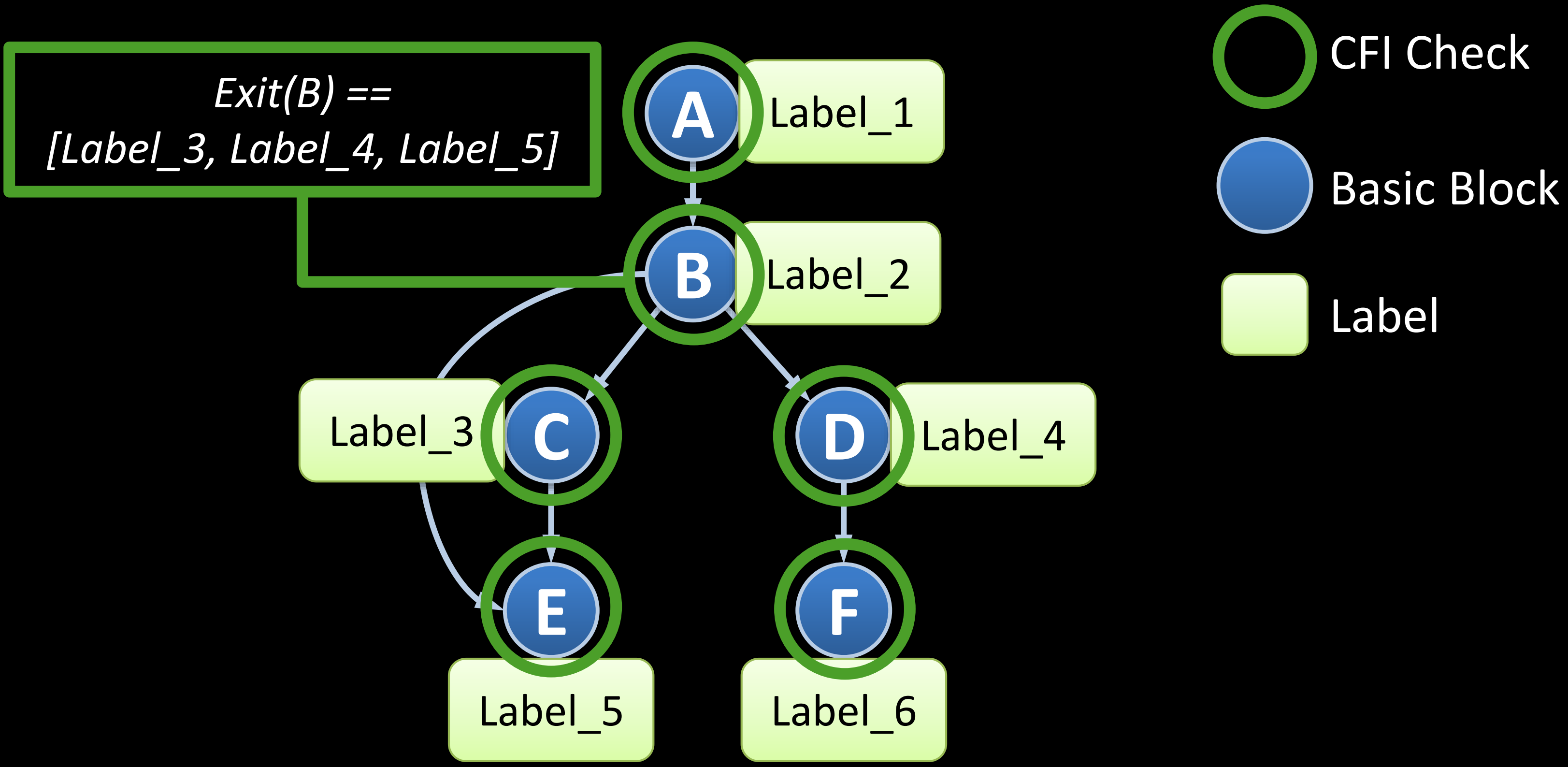
- **Purpose**: switch tables, dispatch to library functions
- **CFI Relevance**: Target address taken from either processor register or memory

Indirect Calls

- **Purpose**: call through function pointer, virtual table calls
- **CFI Relevance**: Target address taken from either processor register or memory

Label Granularity: Trade-Offs (1/2)

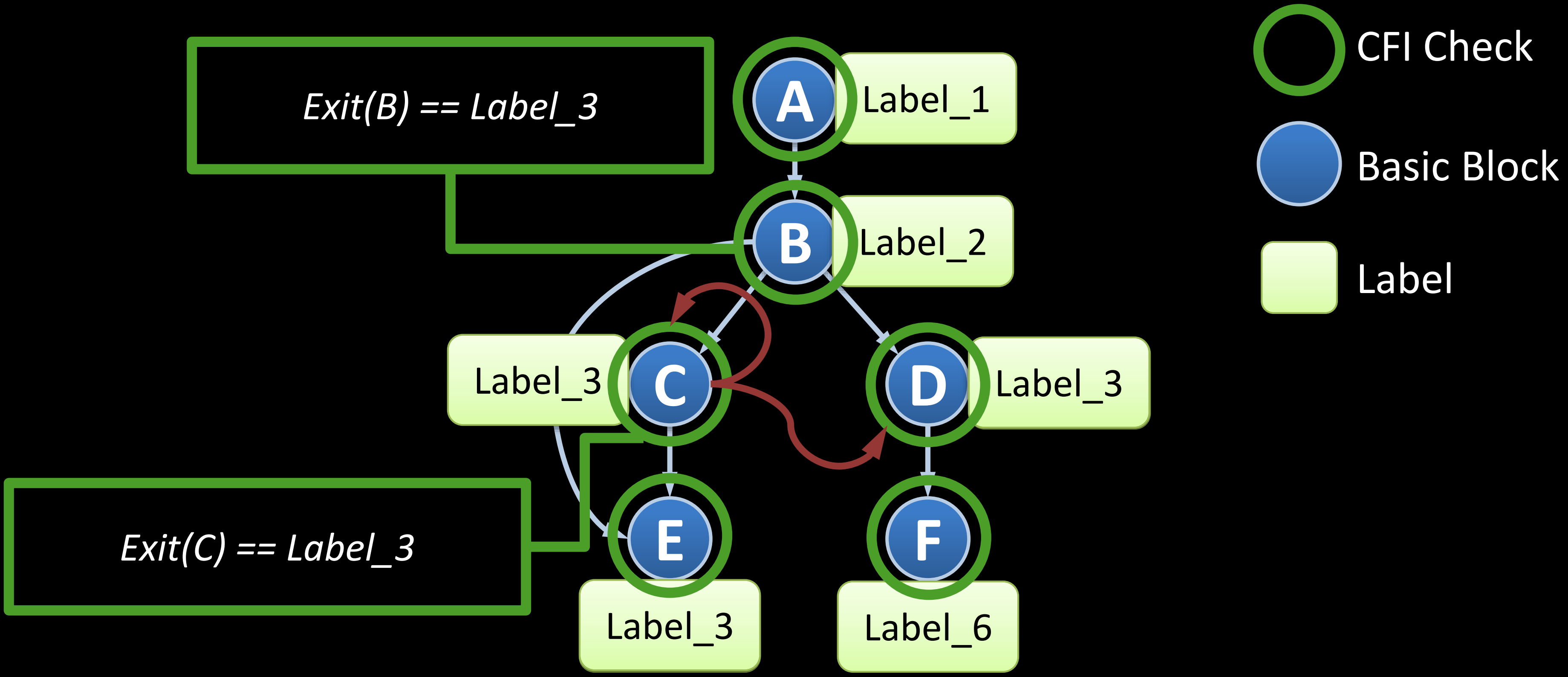
- Many CFI checks are required if unique labels are assigned per node



Exit(B) ==
[Label_3, Label_4, Label_5]

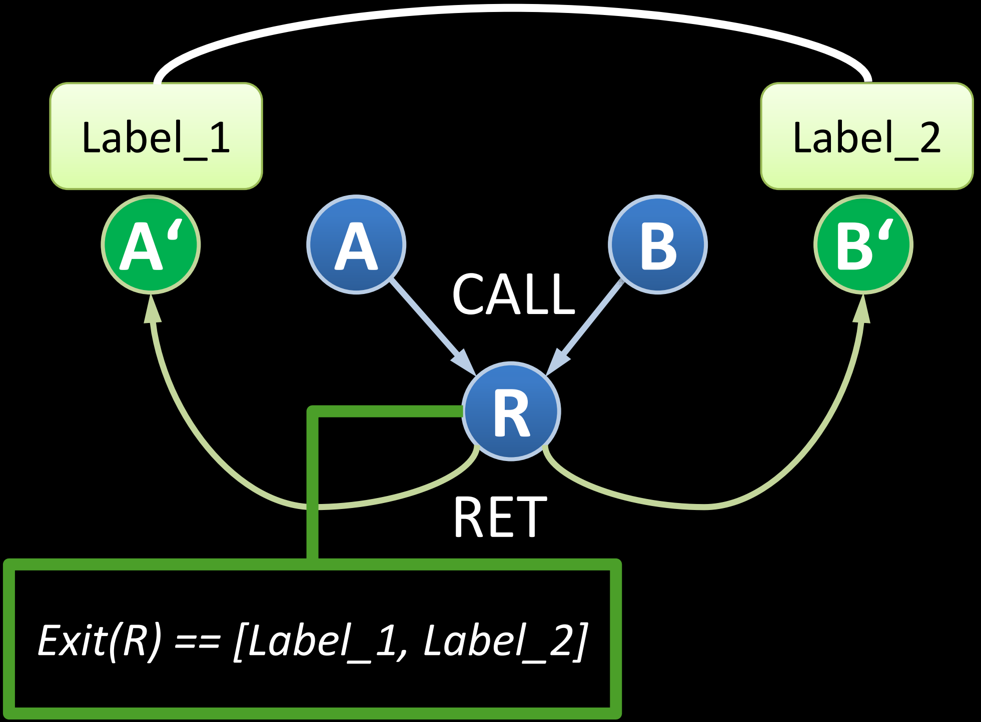
Label Granularity: Trade-Offs (2/2)

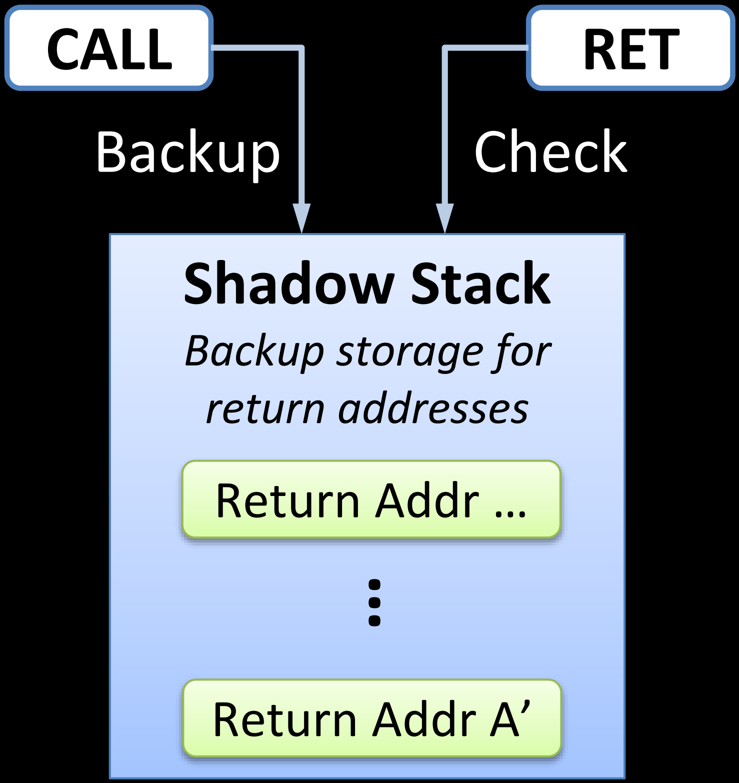
- Optimization step: Merge labels to allow single CFI check
- However, this allows for unintended control-flow paths



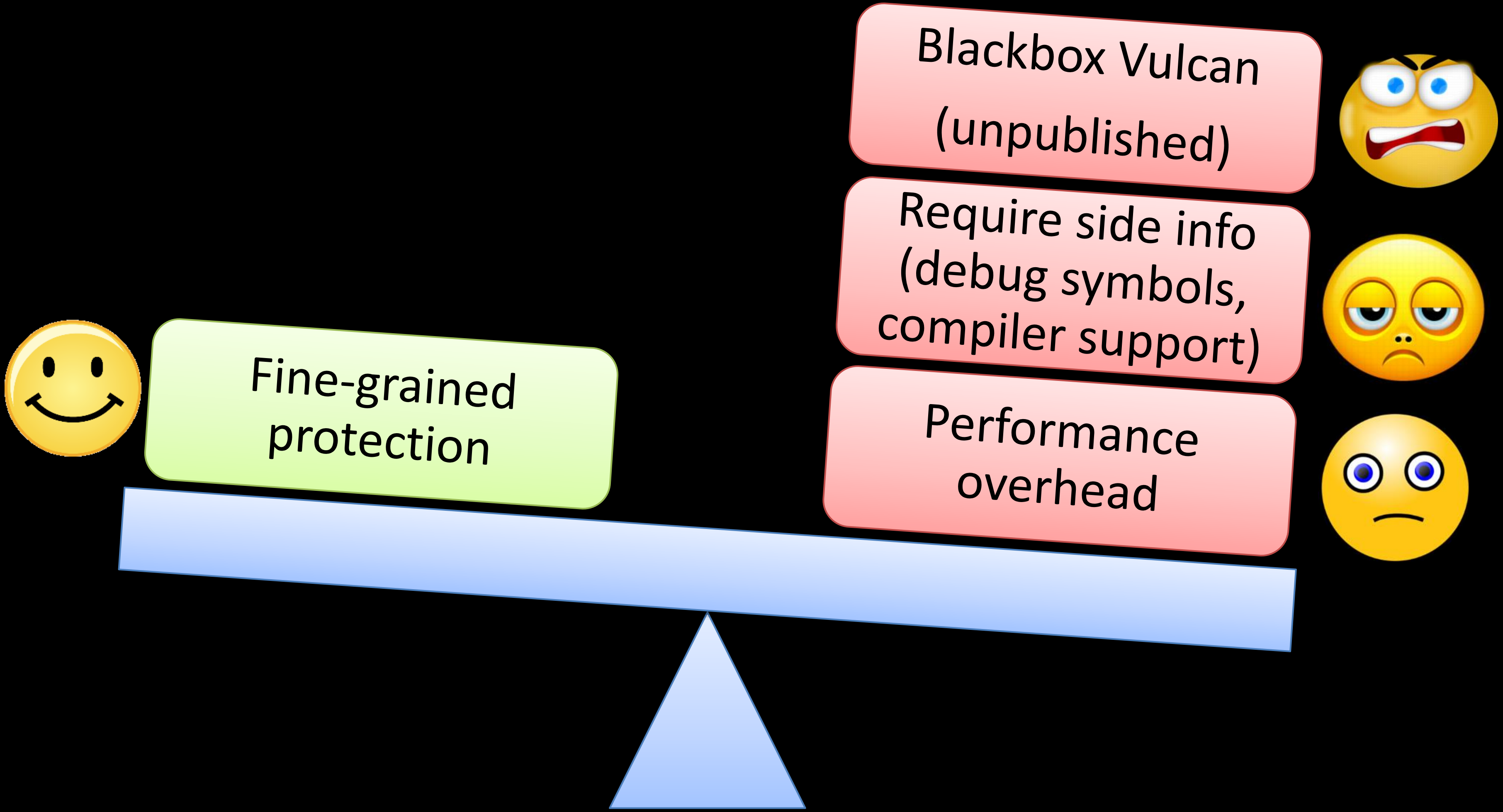
Label Problem for Returns

- Static CFI label checking leads to coarse-grained protection for returns
- Shadow stack allows for fine-grained return address protection but incurs higher overhead





Original CFI: Benefits and Limitations



Fine-grained protection

Blackbox Vulcan (unpublished)

Require side info (debug symbols, compiler support)

Performance overhead

Hot Research Topic:
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

CCFIR
[IEEE S&P’13]

ROPecker
[NDSS’14]

CFI for COTS Binaries
[USENIX Sec’13]

kBouncer
[USENIX Sec’13]

ROPGuard
[Microsoft EMET]

EMET
http://technet.microsoft.com/en-us/security/jj653751

Open Question:

Practical and secure mitigation of code reuse attacks

Turing-completeness of return-oriented programming

Negative Result:

All current (published) coarse-grained CFI solutions can be bypassed

Big Picture

Systematic Security Analysis of Coarse-Grained CFI

- CFI Policies
- Frequency of CFI Checks
- Deriving a CFI policy that combines all schemes

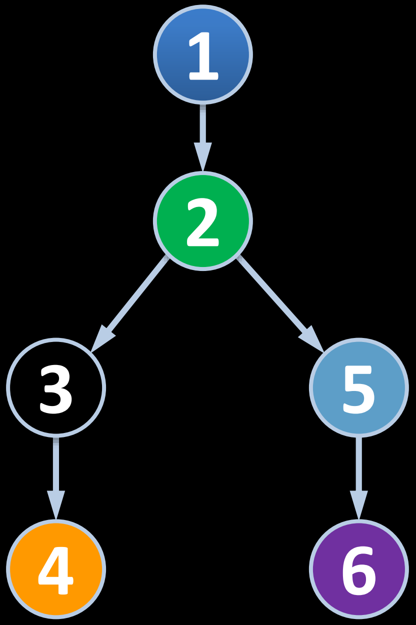
Gadget Analysis

- Turing-complete gadget set
- Gadgets to bypass heuristics

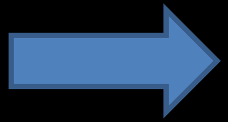
Exploit Development

1. Systematic Security Analysis of Coarse-Grained CFI

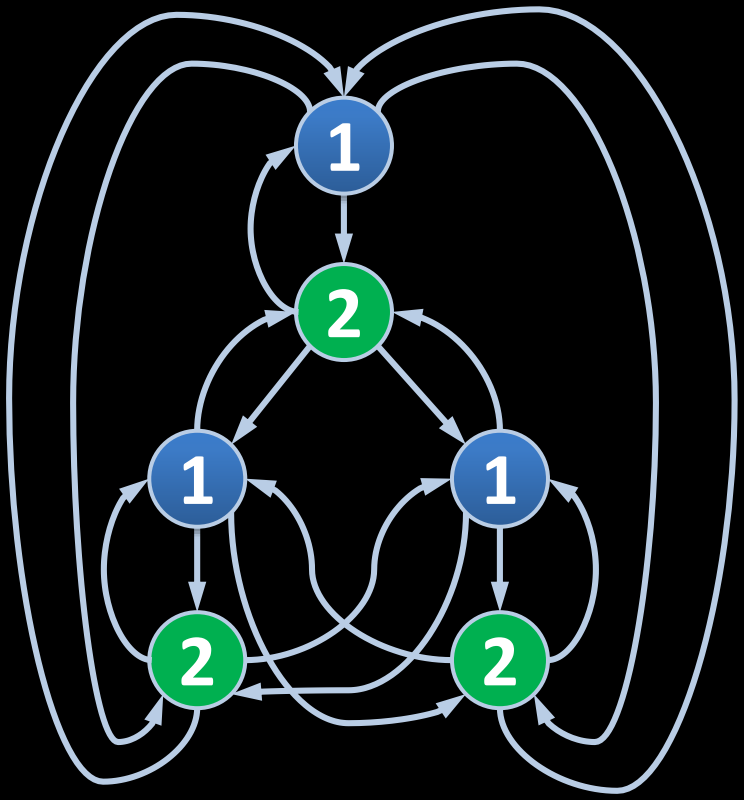
Coarse-grained CFI leads to CFG imprecision



Reducing number of labels

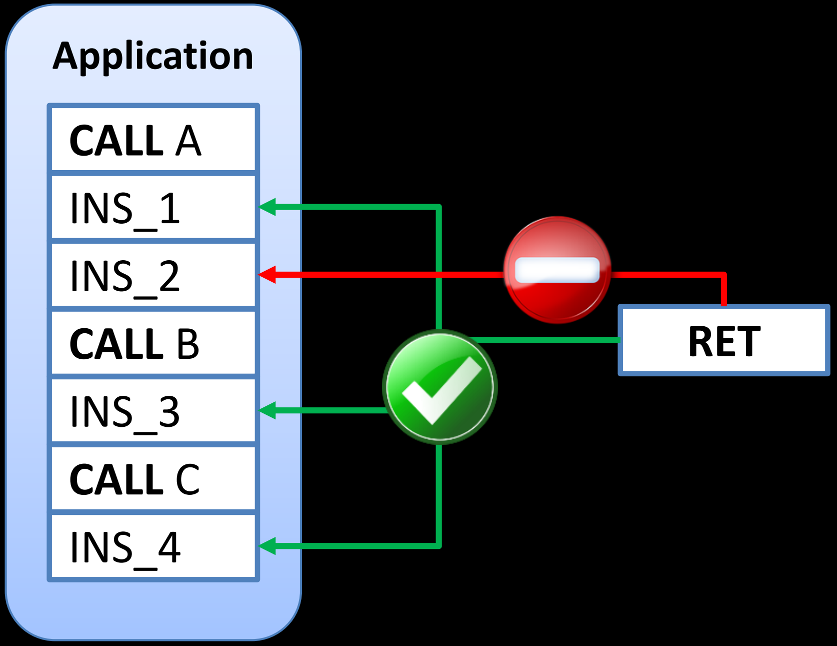


Allowed paths: 1→2 and 2→1

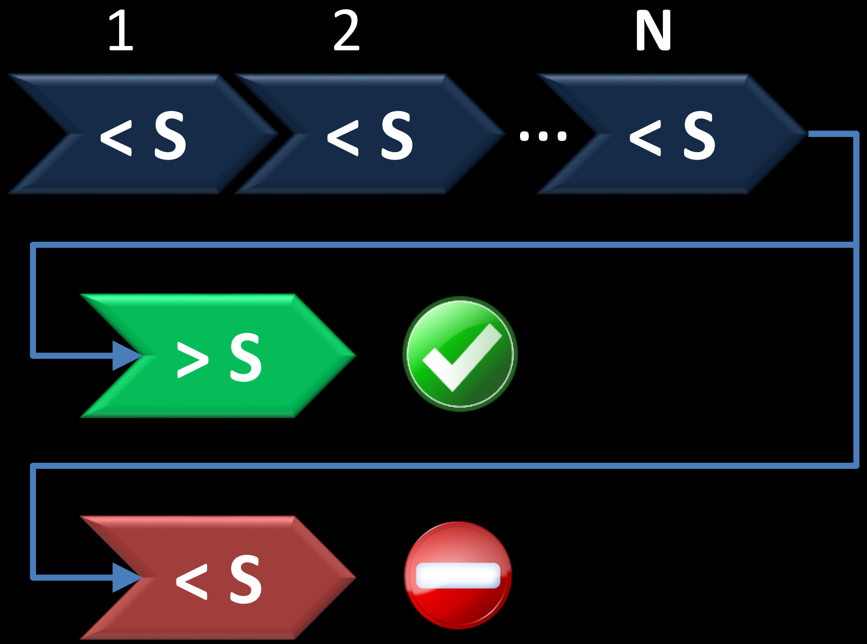


Main Coarse-Grained CFI Policies

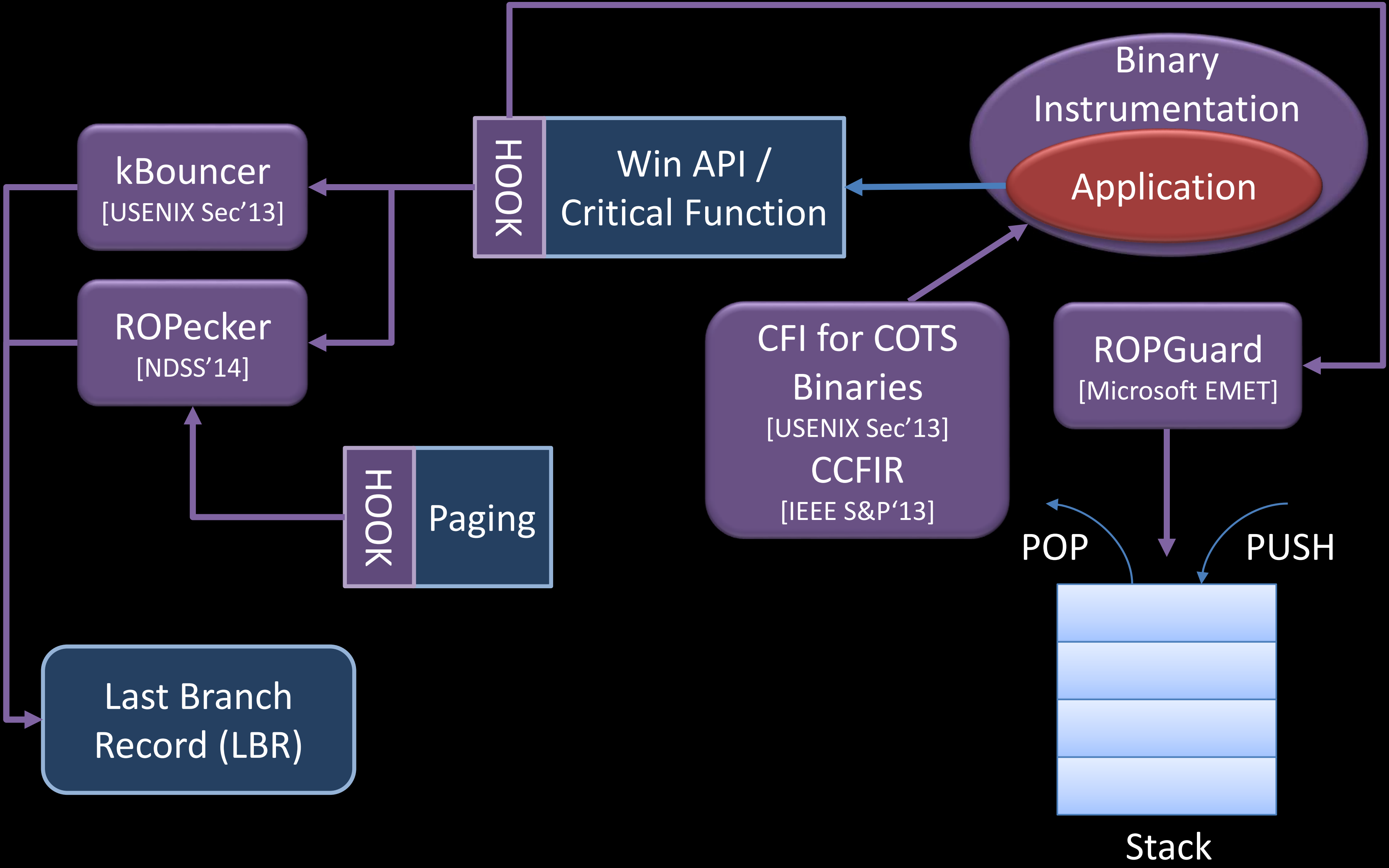
- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction
  - No shadow stack required



- **CFI Policy 2: Behavioral-Based Heuristics**
  - Prohibit a chain of **N** short sequences each consisting of less than **S** instructions



Coarse-Grained CFI Proposals



Deriving a Combined CFI Policy

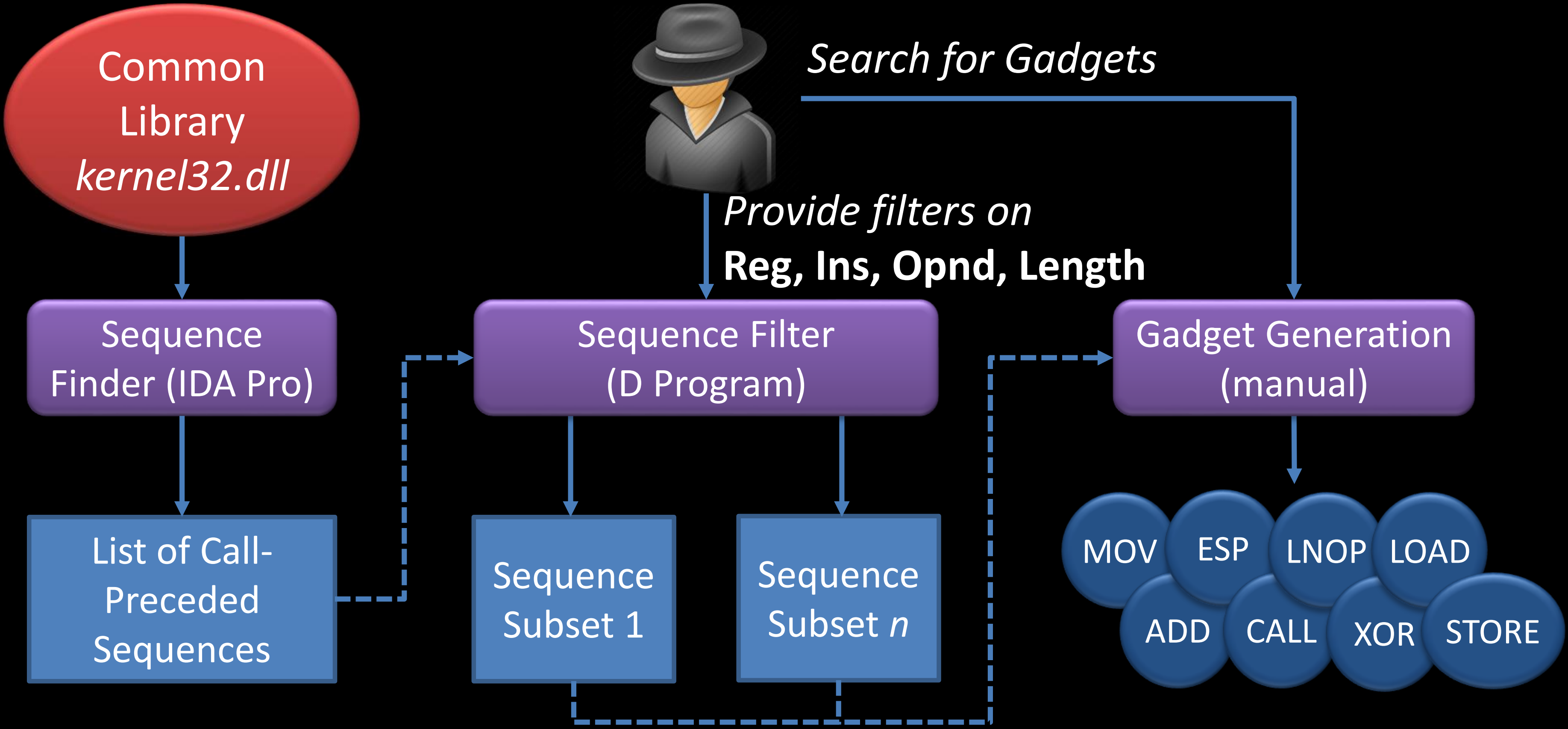
CFI Policy	**kBouncer** [USENIX Sec. 2013]	**ROPecker** [NDSS 2014]	**ROPGuard** [Microsoft EMET]	**CFI for COTS Binaries** [USENIX Sec. 2013]	***Combined CFI Policy***
CFI Policy 1 *Call-Preceded Sequences*	✓	⊖	✓	✓	✓
CFI Policy 2 *Behavioral-Based Heuristics*	✓	✓	⊖	⊖	✓
Time of CFI Check	WinAPI	2 Page Sliding Window/ Critical Functions	WinAPI/ Critical Functions	Indirect Branch	**Any Time**

⊖ No Restriction ✓ CFI Policy

Here only the core policies shown. However, we consider all other deployed policies in our analysis.

2. Gadget Analysis

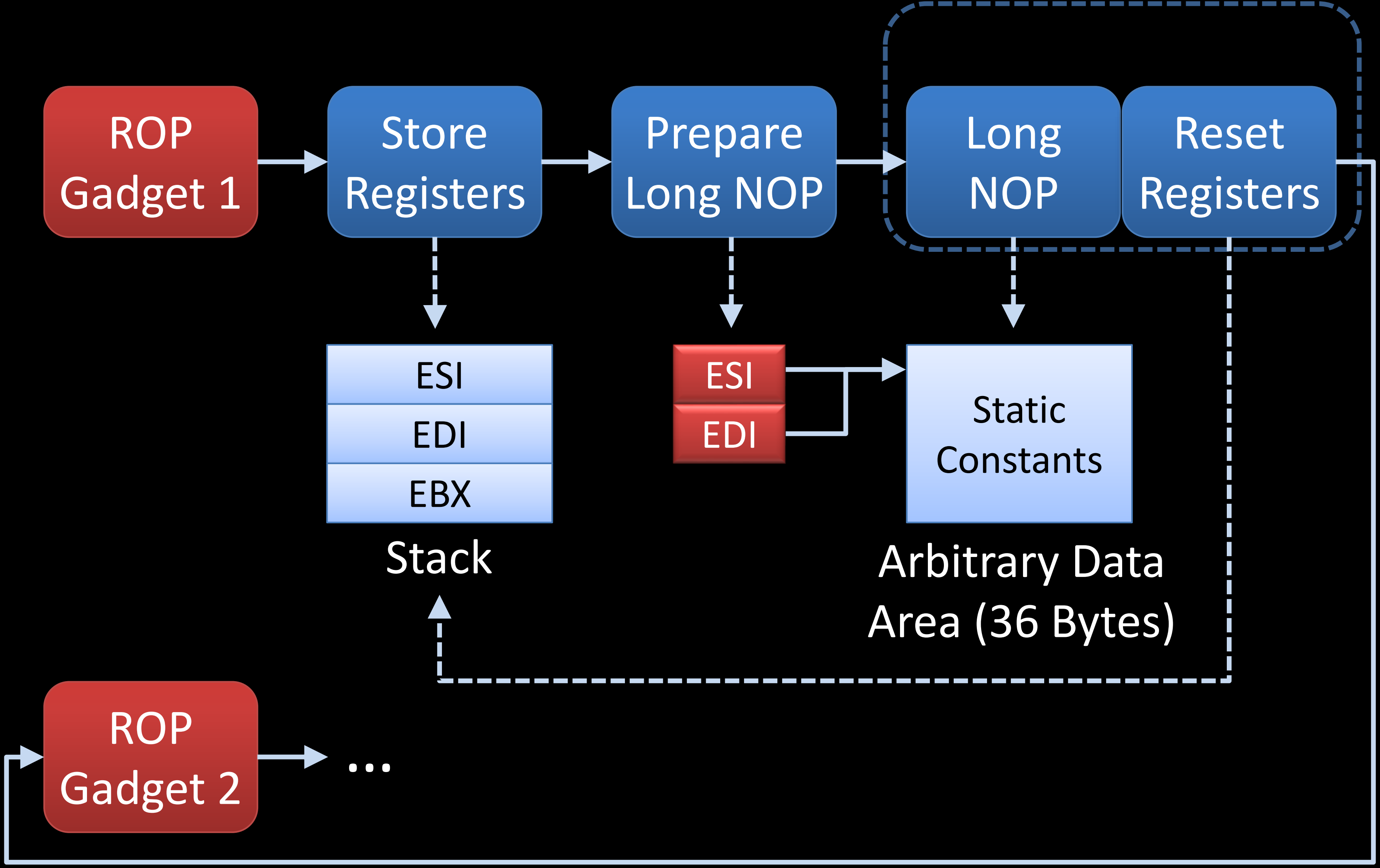
Methodology


Common Library *kernel32.dll*
Sequence Finder (IDA Pro)
List of Call-Preceded Sequences
Search for Gadgets
Provide filters on
Reg, Ins, Opnd, Length
Sequence Filter (D Program)
Sequence Subset 1
Sequence Subset *n*
Gadget Generation (manual)
MOV
ESP
LNOP
LOAD
ADD
CALL
XOR
STORE

(Excerpt of) Turing-Complete Gadget Set in CFI-Protected *kernel32.dll*

Gadget Type	CALL-Preceded Sequence *ending in a RET instruction*
LOAD Register	**EBP** := pop ebp **ESI** := pop esi; pop ebp **EDI** := pop edi; leave **ECX** := pop ecx; leave **EBX** := pop edi; pop esi; pop ebx; pop ebp **EAX** := mov eax,edi; pop edi; leave **EDX** := mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave
LOAD/STORE Memory	**LD(EAX)** := mov eax,[ebp+8]; pop ebp **ST(EAX)** := mov [esi],eax; xor eax,eax; pop esi; pop ebp **ST(ESI)** := mov [ebp-20h],esi **ST(EDI)** := mov [ebp-20h],edi
Arithmetic/ Logical	**ADD/SUB** := sub eax,esi; pop esi; pop ebp **XOR** := xor eax,edi; pop edi; pop esi; pop ebp
Branches	**unconditional branch 1** := leave **unconditional branch 2** := add esp,0Ch; pop ebp **conditional LD(EAX)** := neg eax; sbb eax,eax; and eax,[ebp-4]; leave

Long-NOP Gadget



3. Exploit Development

Adobe Reader 9.1
CVE-2010-0188

MPlayer Lite r33064 m3u
Buffer Overflow Exploit

Original exploits detected by coarse-grained CFI

Our instrumented exploits bypass coarse-grained CFI

Coarse-Grained CFI: Lessons Learned

1. **Too many call sites available**
   → Restrict returns to their actual caller (shadow stack)
2. **Heuristics are ad-hoc and ineffective**
   → Adjusted sequence length leads to high false positive
3. **Too many indirect jump and call targets**
   - Resolving indirect jumps and calls is non-trivial

   → Compromise: Compiler support

CURRENT RESEARCH

Stack Attacks

CURRENT RESEARCH
What’s next?

Hardware-Assisted CFI

HAFIX: Hardware-Assisted Flow Integrity Extension

DAC 2014 and DAC 2015

Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi (TU Darmstadt)

Patrick Koeberl (Intel Labs)

Orlando Arias, Yier Jin, Dean Sullivan (University of Central Florida)

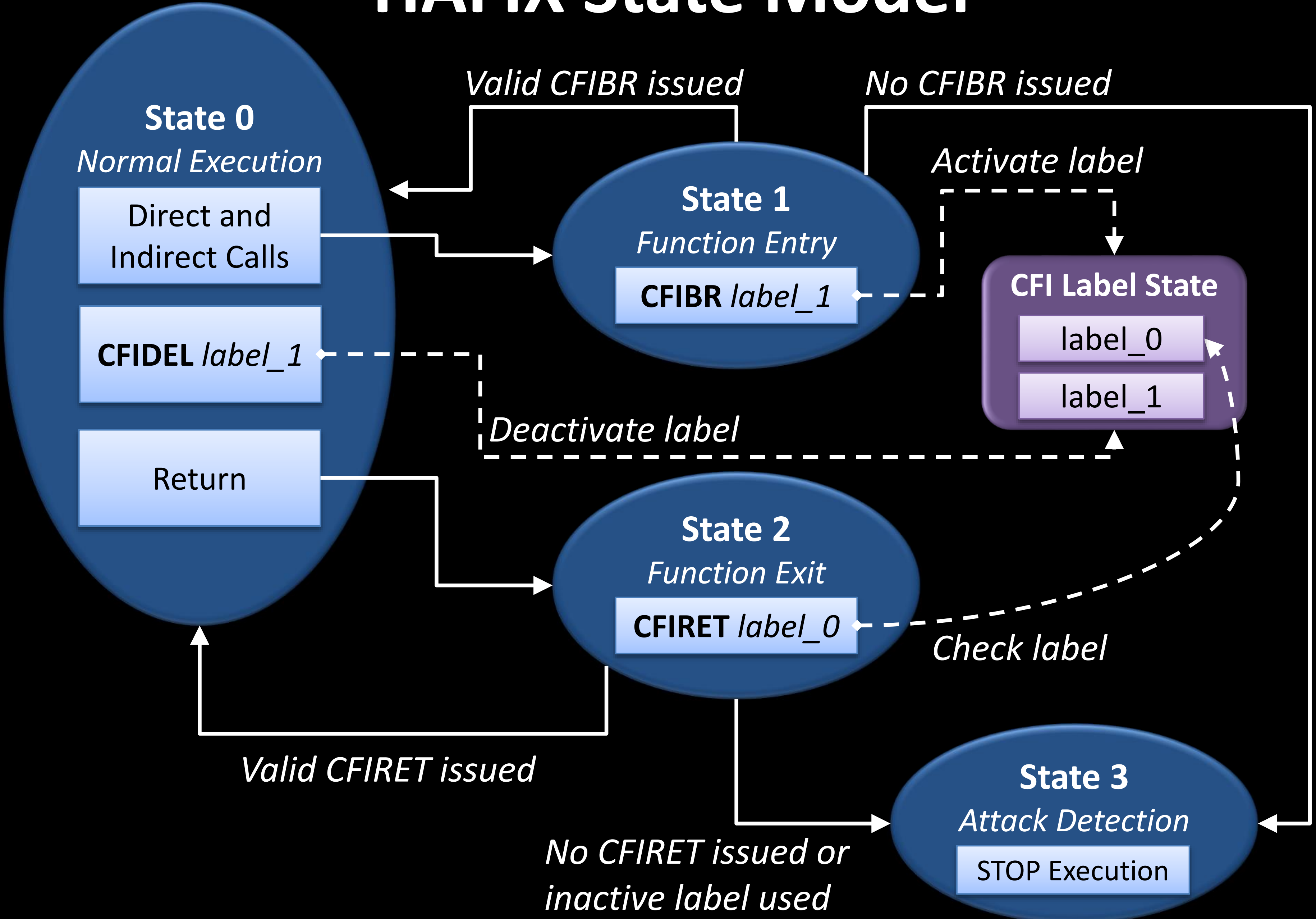
Why Leveraging Hardware for CFI ?

- Efficiency
    - Dedicated CFI instructions
- Security
    - On-chip memory for CFI data
    - CFI Context
        - No unintended sequences
        - Dynamic code protection

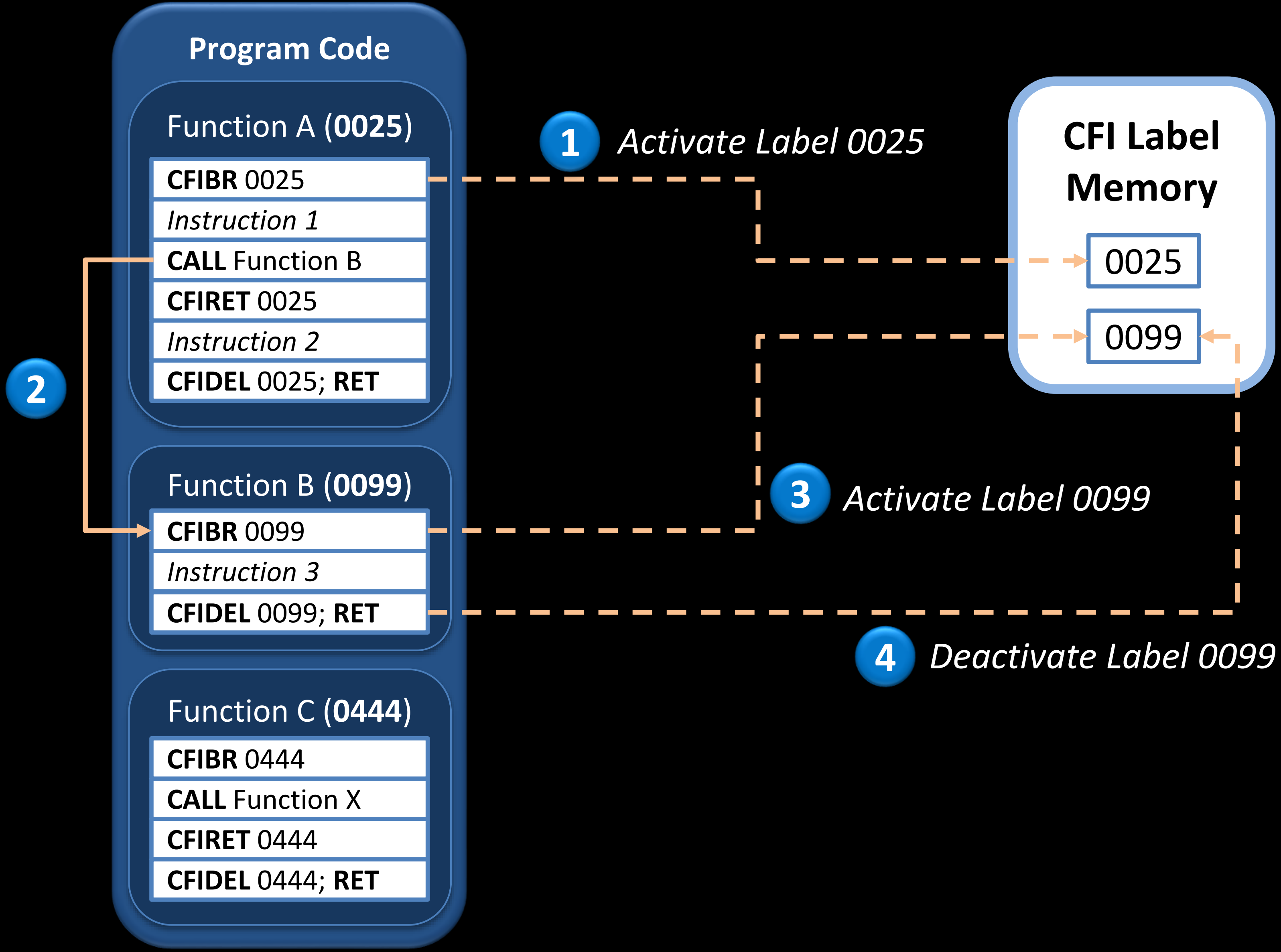
Our Objectives

Backward-Edge and Forward-Edge CFI	Stateful, Fine-granular
No burden on developer	No code annotations/changes
Security	Hardware protection On-chip memory for CFI Data No unintended sequences
High performance	< 3% overhead
Enabling technology	All applications can use CFI features Support of multitasking
Compatibility to legacy code	CFI and non-CFI code on same platform

HAFIX State Model



Instrumented Code Example



Program Code

Function A (**0025**)

CFIBR 0025

Instruction 1

CALL Function B

CFIRET 0025

Instruction 2

CFIDEL 0025; **RET**

Function B (**0099**)

CFIBR 0099

Instruction 3

CFIDEL 0099; **RET**

Function C (**0444**)

CFIBR 0444

CALL Function X

CFIRET 0444

CFIDEL 0444; **RET**

CFI Label Memory

0025

0099

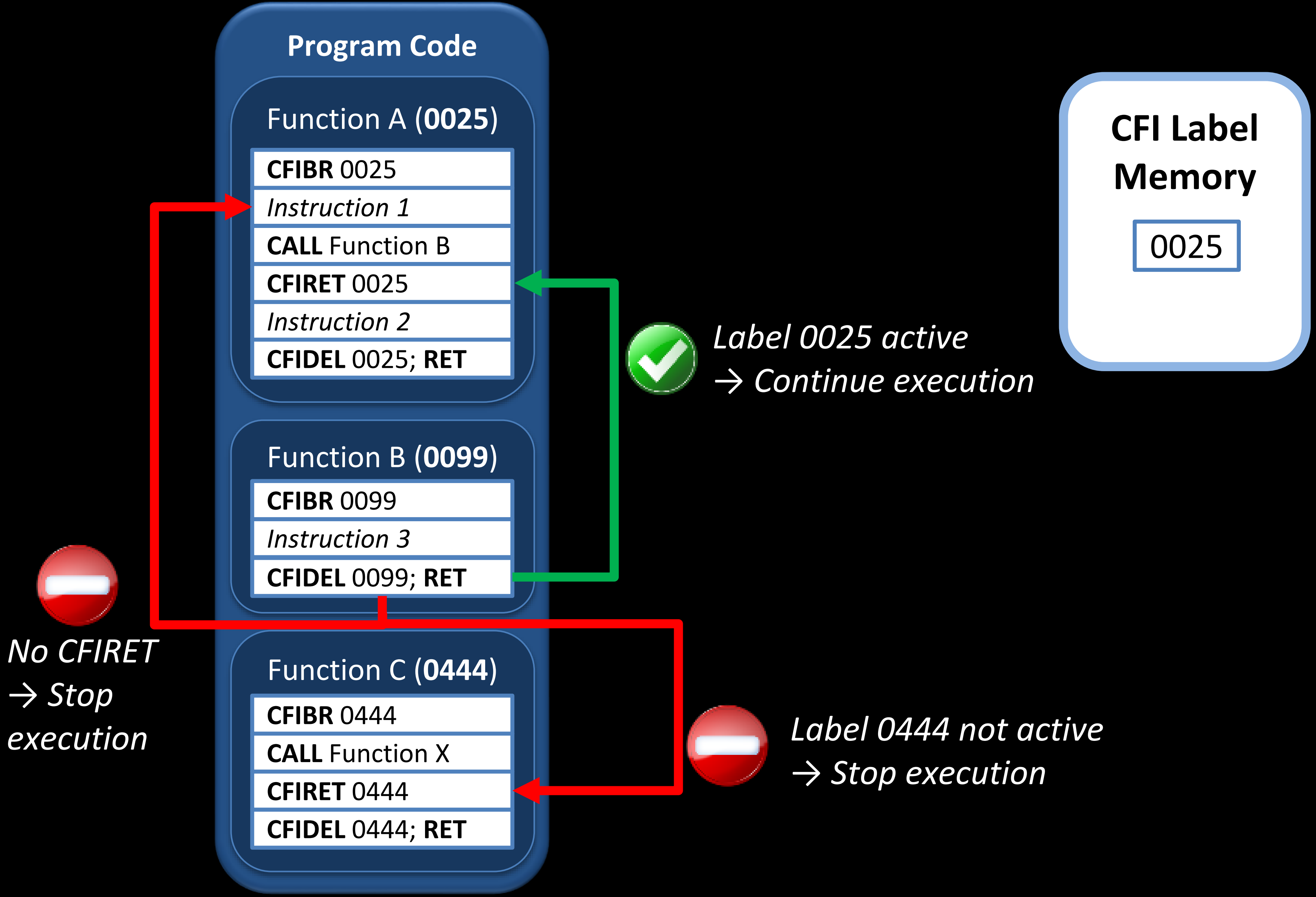
1 *Activate Label 0025*

2

3 *Activate Label 0099*

4 *Deactivate Label 0099*

Instrumented Code Example



Program Code

Function A (**0025**)

CFIBR 0025
Instruction 1
CALL Function B
CFIRET 0025
Instruction 2
CFIDEL 0025; **RET**

Function B (**0099**)

CFIBR 0099
Instruction 3
CFIDEL 0099; **RET**

Function C (**0444**)

CFIBR 0444
CALL Function X
CFIRET 0444
CFIDEL 0444; **RET**

CFI Label Memory

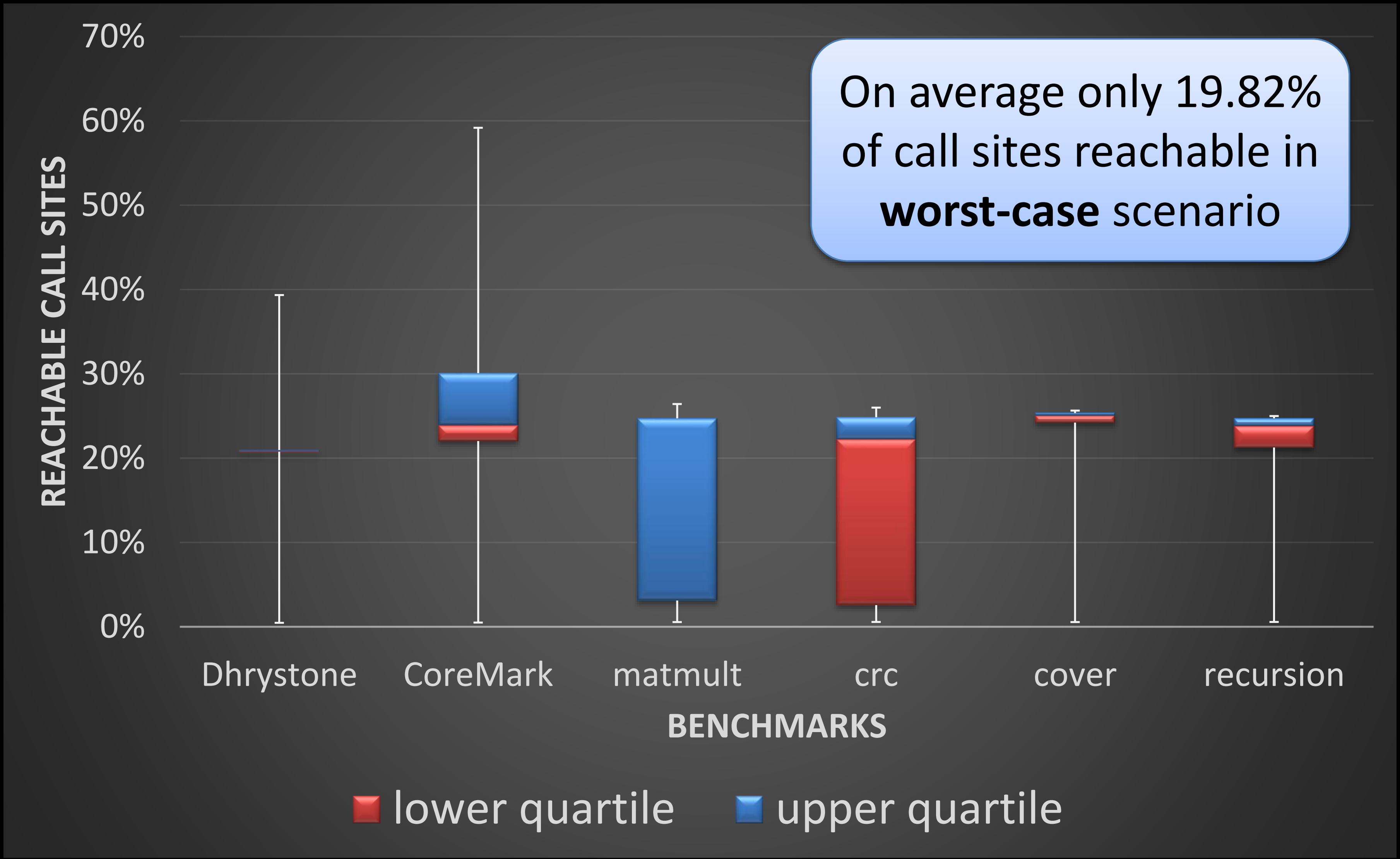
0025

Label 0025 active
→ Continue execution

No CFIRET
→ Stop execution

Label 0444 not active
→ Stop execution

Gadget Space compared to Coarse-Grained CFI for *Static* Binaries



On average only 19.82% of call sites reachable in **worst-case** scenario

Conclusion

- Code-reuse attacks are prevalent
  - Google and Microsoft take these attacks seriously
  - Many real-world exploits
  - Existing solutions can be bypassed
- Good News
  - Many innovative defense techniques have been proposed
- Promising new directions
  - Memory safety based on code-pointer integrity [Kuznetsov et al., OSDI 2014]