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

ARM Overview

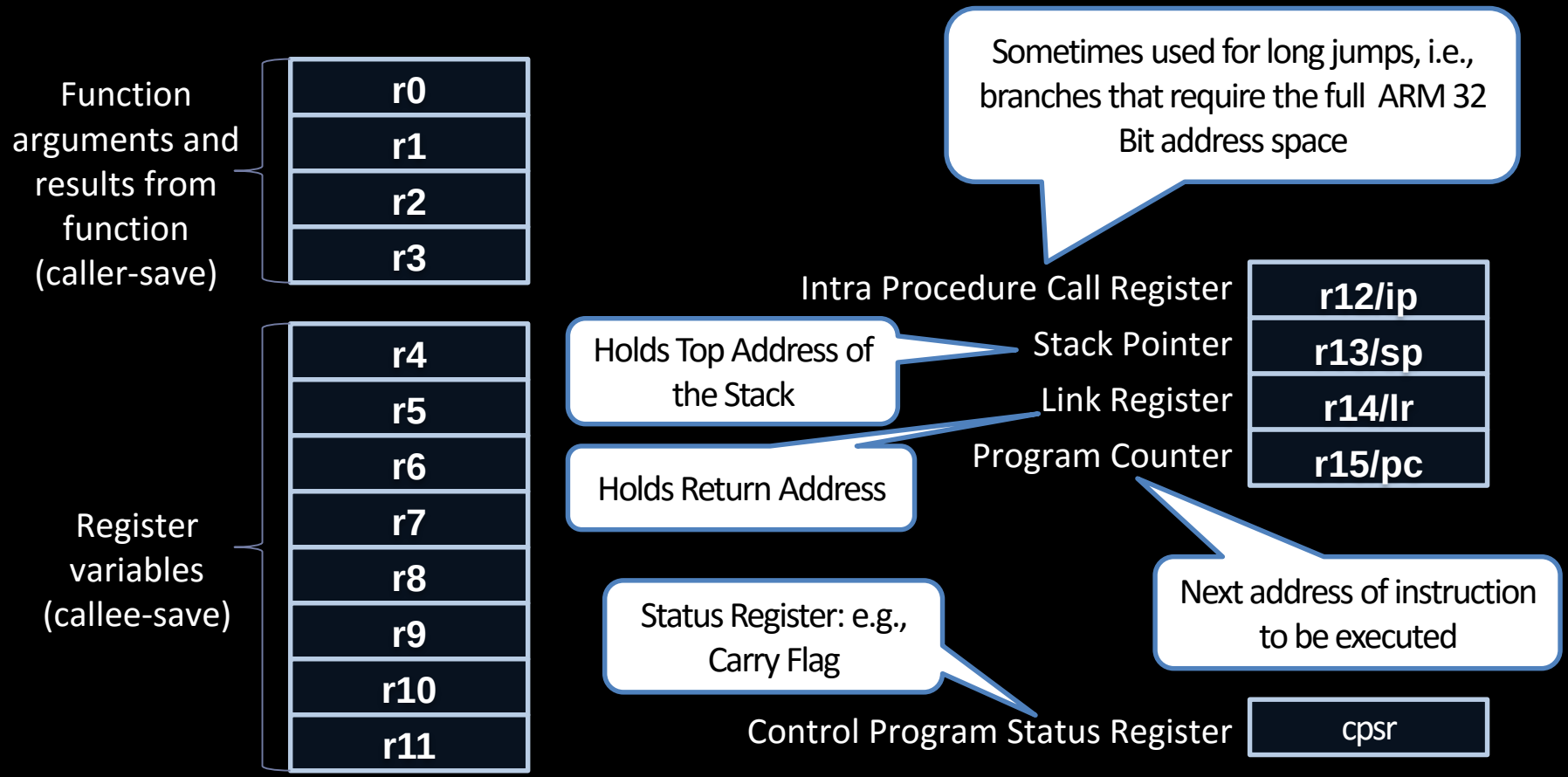
- ARM stands for Advanced RISC Machine
- Main application area: Mobile phones, smartphones (Apple iPhone, Google Android), music players, tablets, and some netbooks
- Advantage: Low power consumption
- Follows RISC design
  - Mostly single-cycle execution
  - Fixed instruction length
  - Dedicated load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

ARM Overview

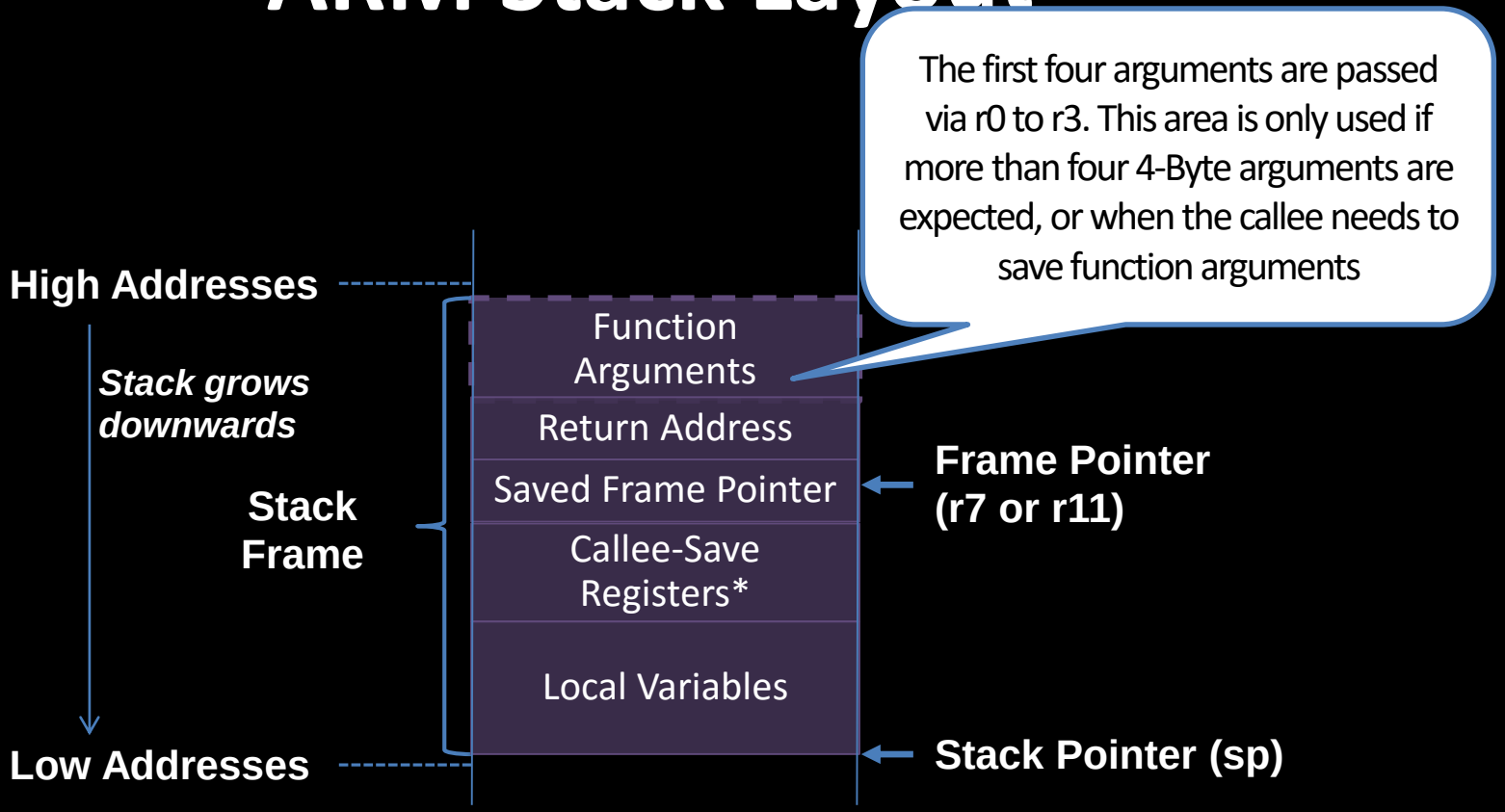
- Some features of ARM
  - Conditional Execution
  - Two Instruction Sets
    - ARM (32-Bit)
      - The traditional instruction set
    - THUMB (16-Bit)
      - Suitable for devices that provide limited memory space
    - The processor can exchange the instruction set on-the-fly
    - Both instruction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

ARM Registers

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



ARM Stack Layout



* 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

The Stack and Stack Frame Elements

- Stack is a last in, first out (LIFO) memory area where the Stack Pointer points to the last stored element on the stack
- The stack can be accessed by two basic operations
  1. PUSH elements onto the stack (SP is decremented)
  2. POP elements off the stack (SP is incremented)
- Stack is divided into individual stack frames
  - Each function call sets up a new stack frame on top of the stack
  1. Function arguments
     - Arguments provided by the caller of the function
  2. Callee-save Registers
     - Registers that a subroutine (callee) needs to reset before returning to the caller of the subroutine
  3. Return address
     - Upon function return control transfers to the code pointed to by the return address (i.e., control transfers back to the caller of the function)
  4. Saved Frame Pointer/Saved Base Pointer
     - Frame pointer/Base pointer of the calling function
     - Variables and arguments are accessed via an offset to the frame pointer/base pointer
     - Provided in register r11 (ARM code), r7 (THUMB code), or EBP (x86 code)
  5. Local variables
     - Variables that the called function uses internally

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

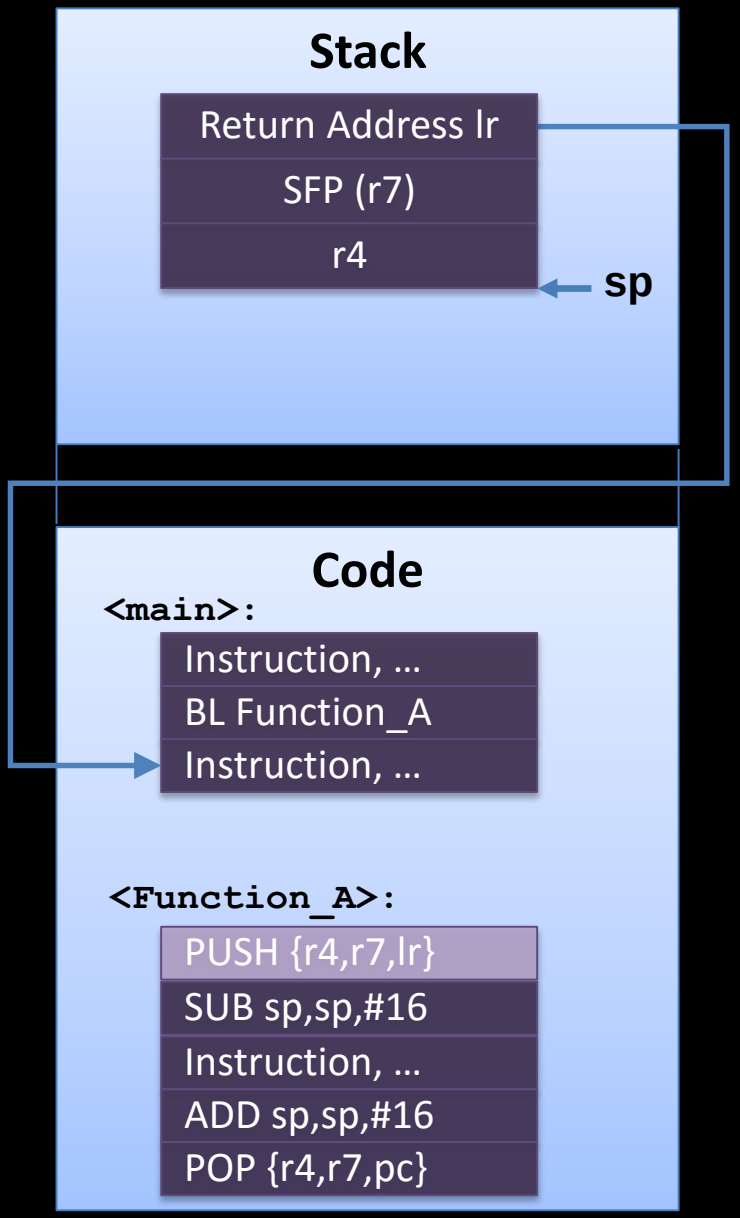
- Branches to the return address stored in the link register lr
- Register-based return for leaf functions

POP {pc}

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

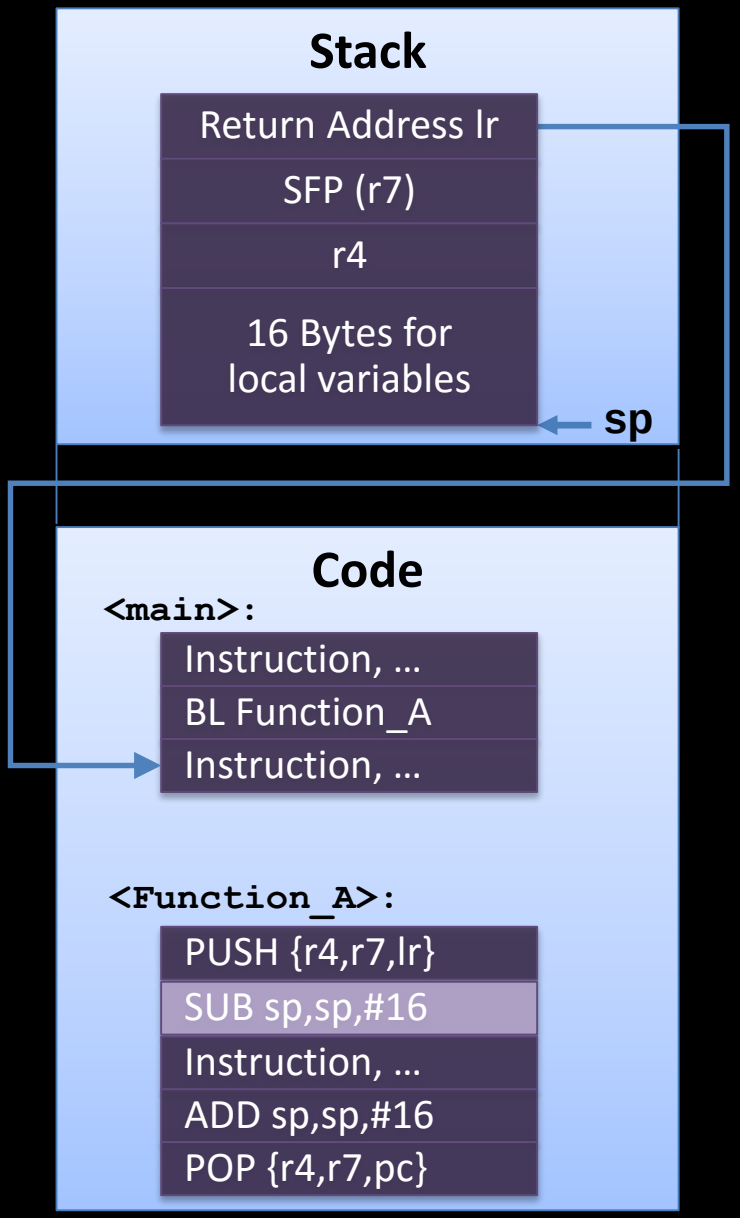
THUMB Example for Calling Convention

- 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



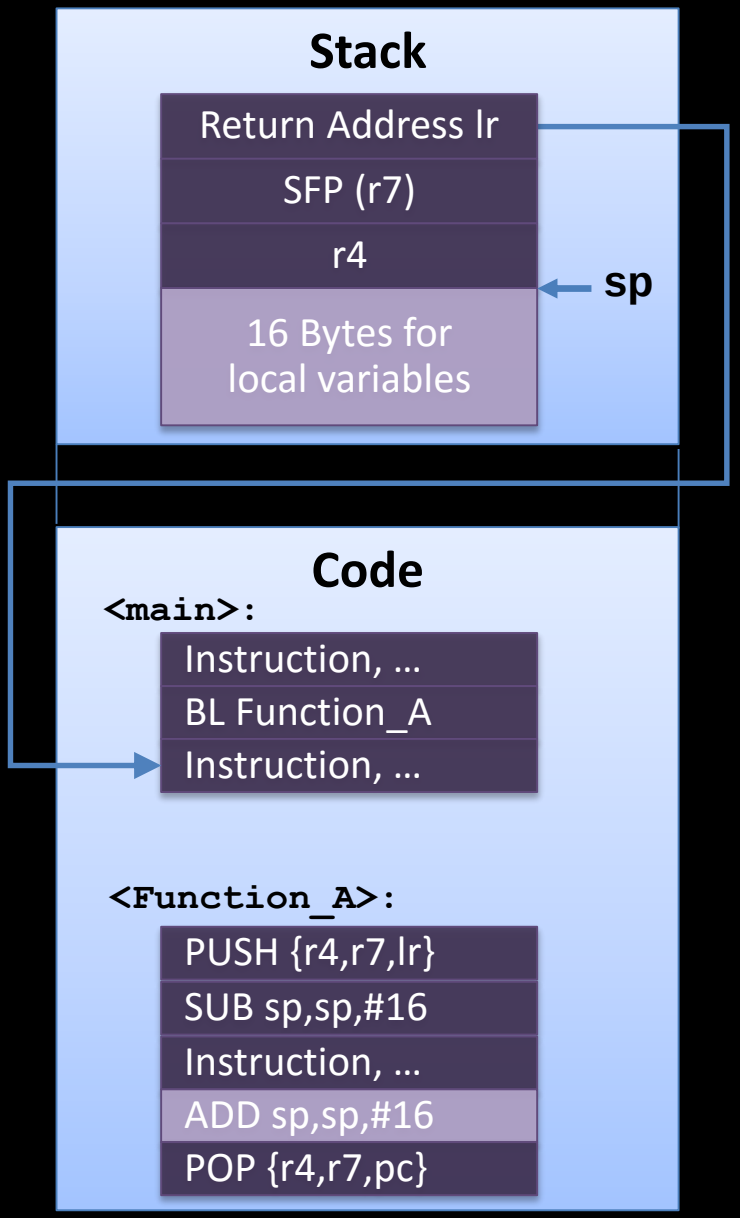
THUMB Example for Calling Convention

- 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



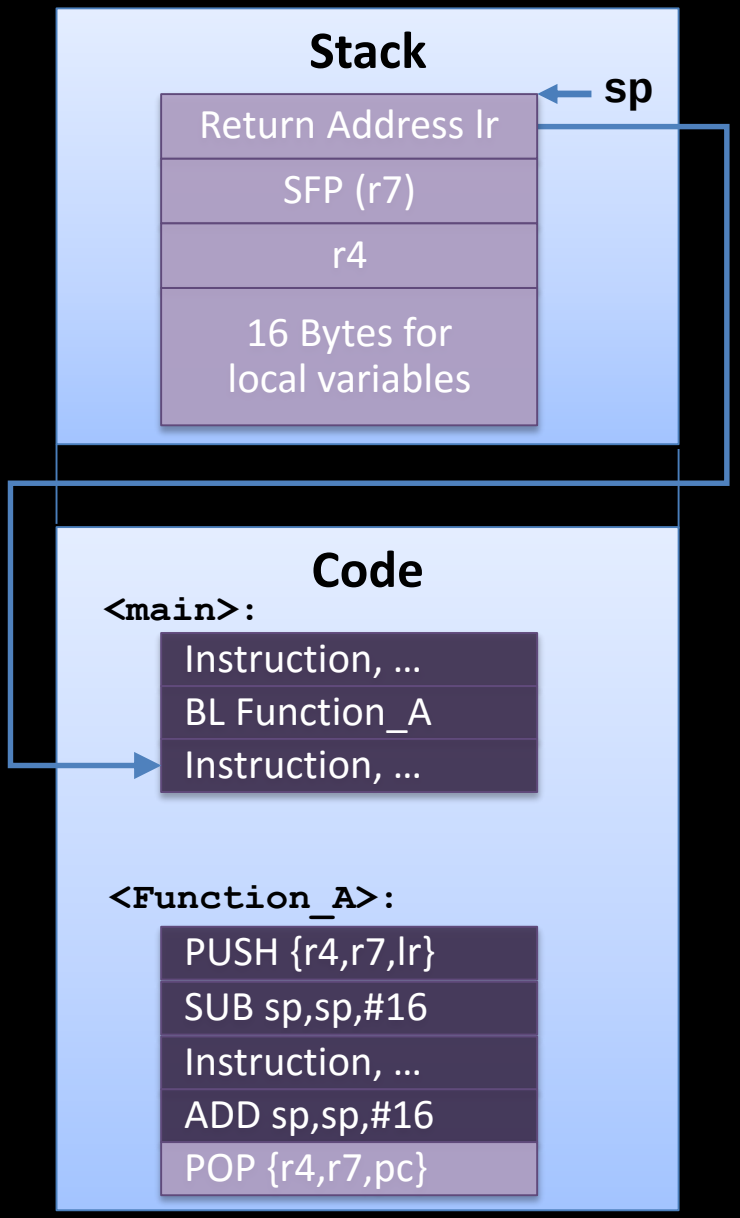
THUMB Example for Calling Convention

- 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



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



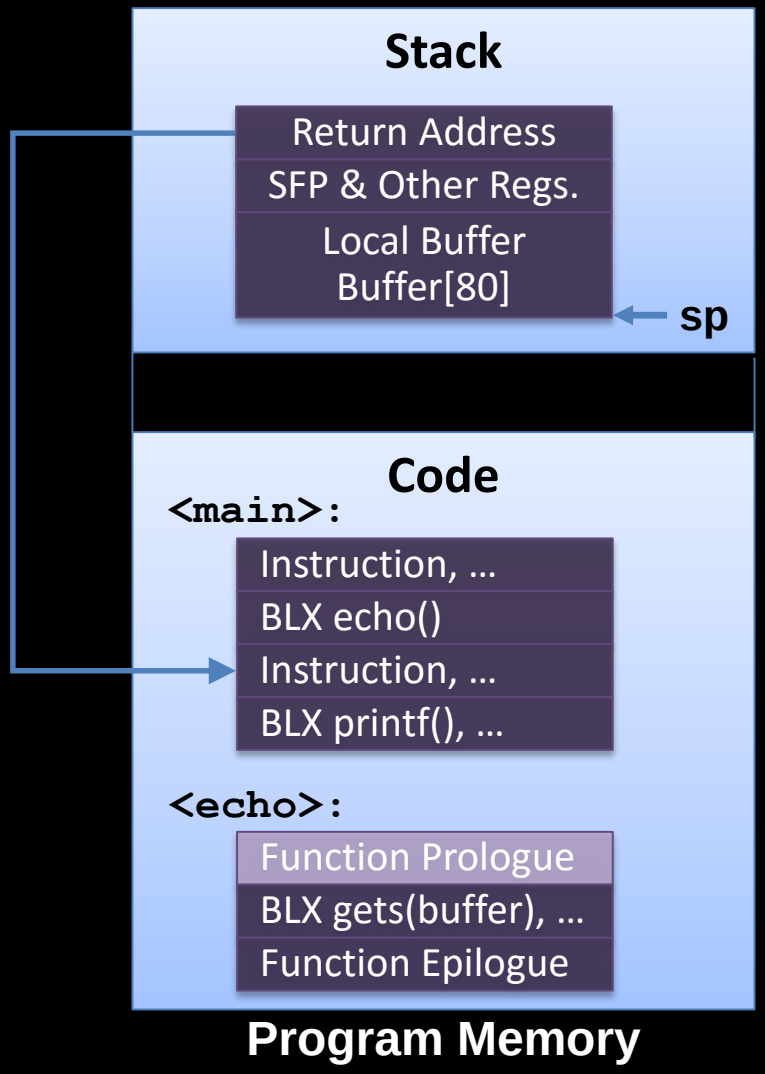
Let‘s go back to runtime attacks

Running Example

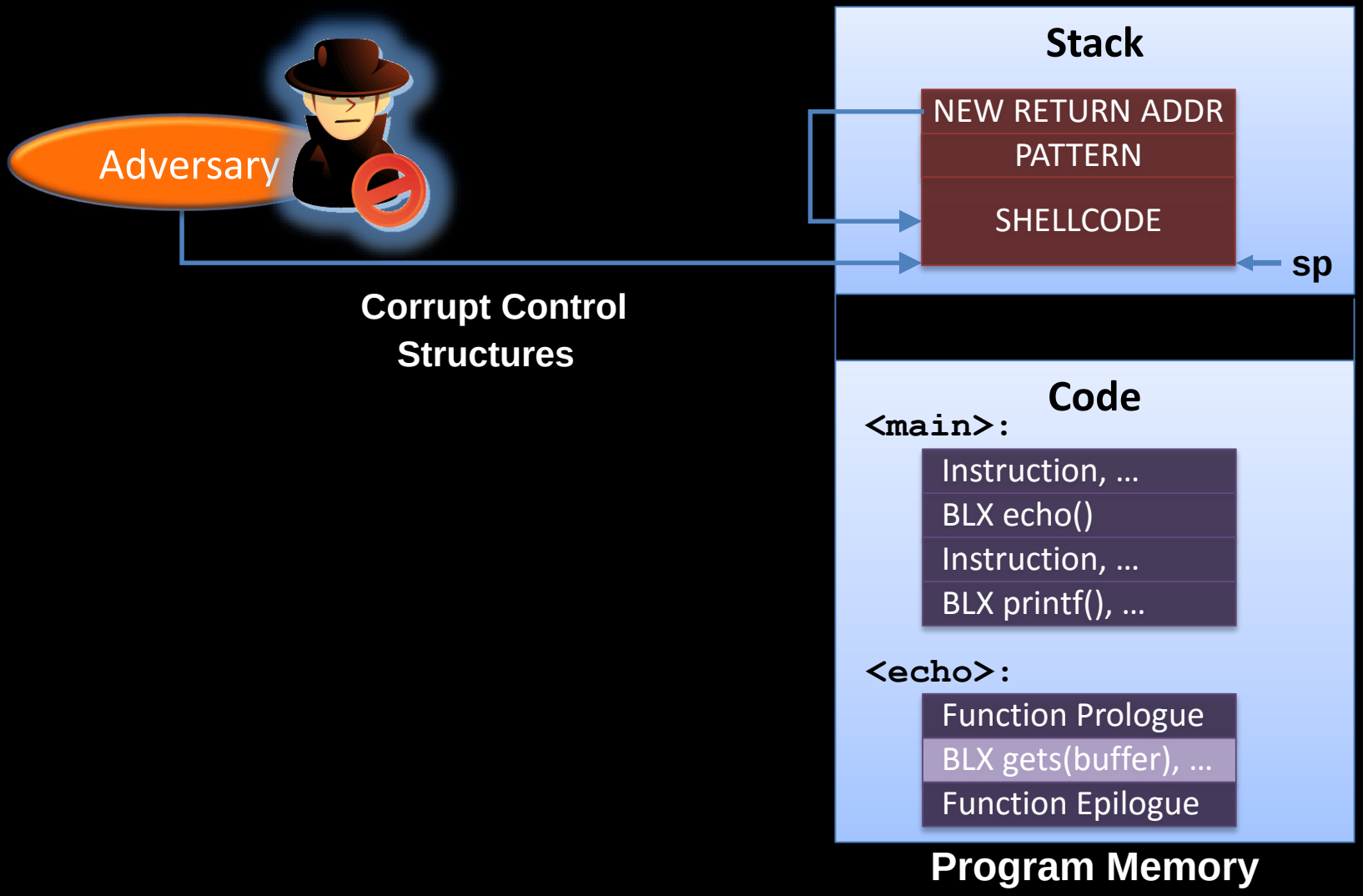
```
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main ()
{
    echo();
    printf("Done");
    return 0;
}
```

Launching a code injection attack against the vulnerable program

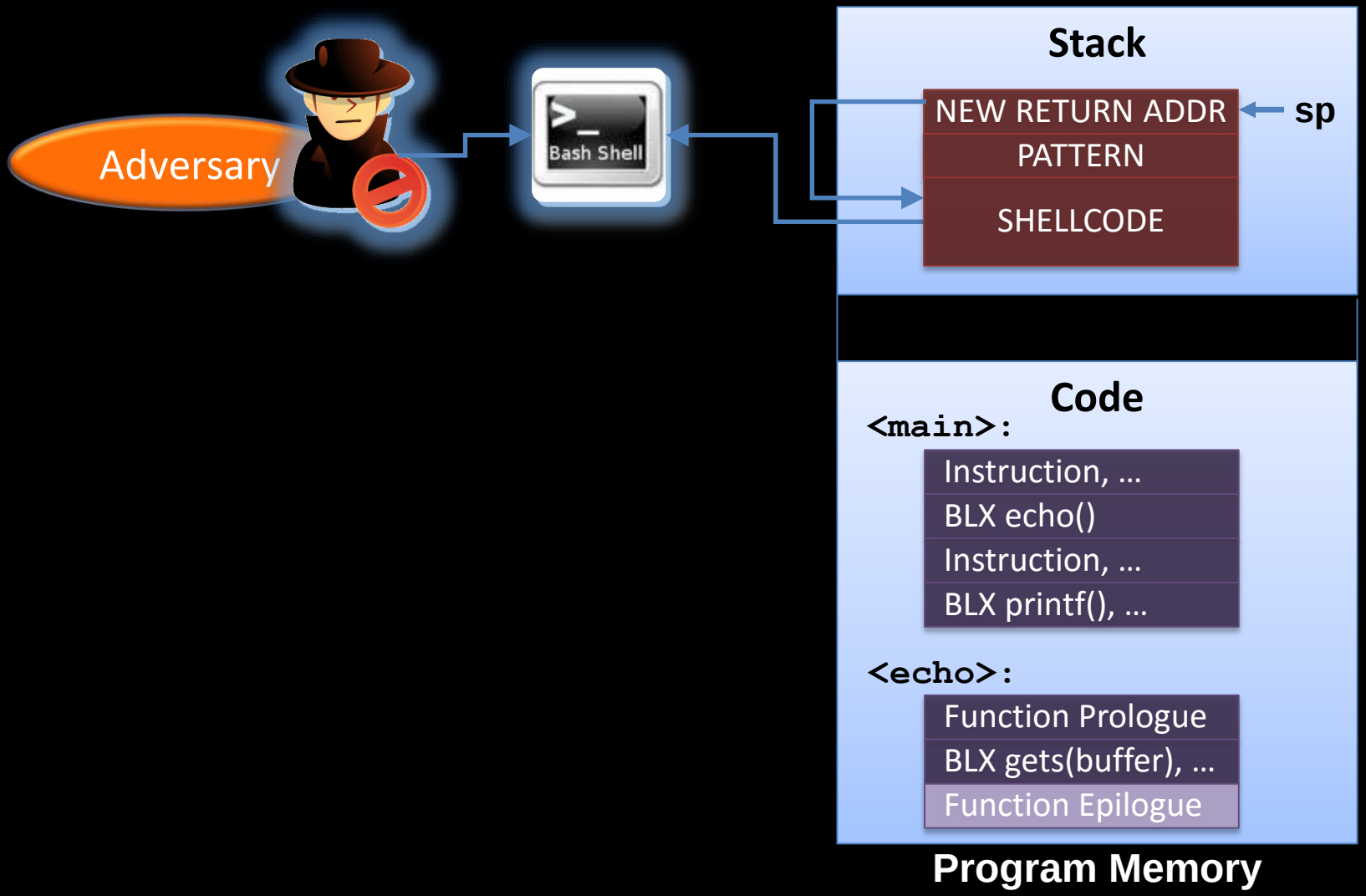
Code Injection Attack on ARM



Code Injection Attack on ARM



Code Injection Attack on ARM



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()

Implementation of return-into-libc attacks on ARM

Stack

Program Code

<main>:

Instruction, …

BLX echo()

Instruction, …

<echo>:

Function Prologue

BLX gets(buffer), …

Function Epilogue

Library Code

<system>:

Function Prologue

Instruction, …

Function Epilogue

<XYZ_function>:

Instruction, …

POP {R0}

POP {PC}

Inject environment variable

Environment Variables

$SHELL = "/bin/sh"

Program Memory

Stack

Program Code

`<main>:`

- Instruction, …
- BLX echo()
- Instruction, …

`<echo>:`

- Function Prologue
- BLX gets(buffer), …
- Function Epilogue

Library Code

`<system>:`

- Function Prologue
- Instruction, …
- Function Epilogue

`<XYZ_function>:`

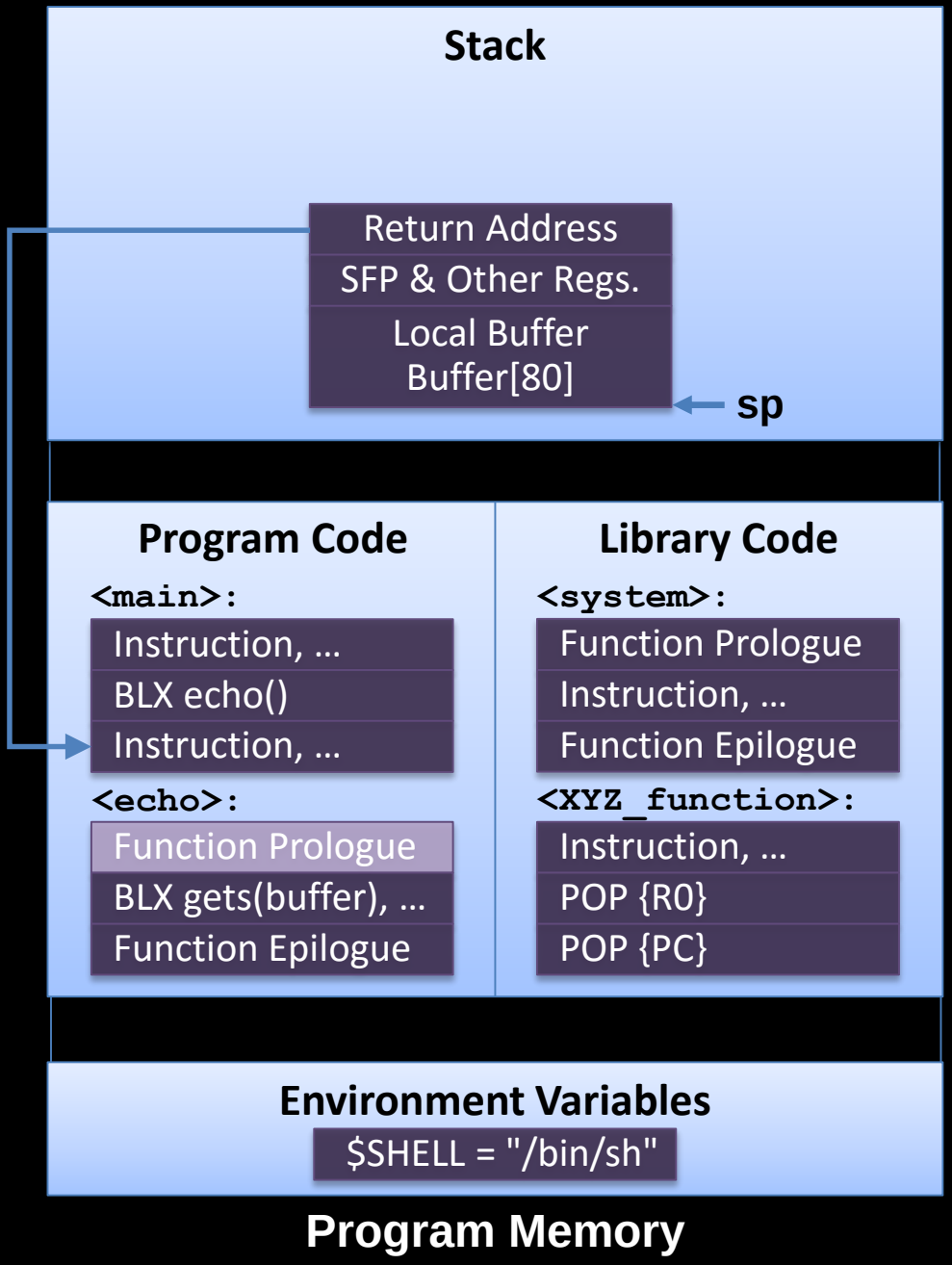
- Instruction, …
- POP {R0}
- POP {PC}

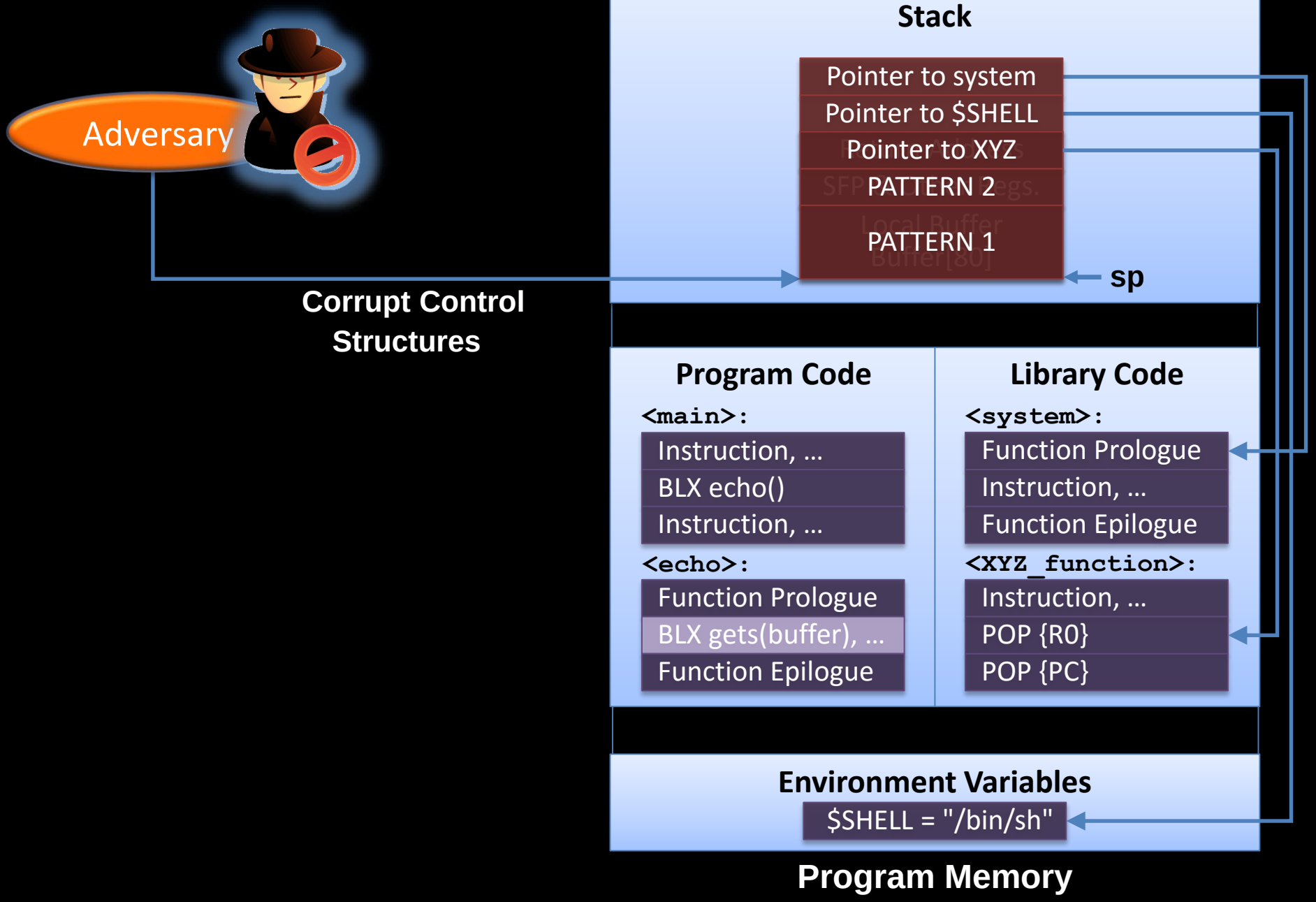
Environment Variables

$SHELL = "/bin/sh"

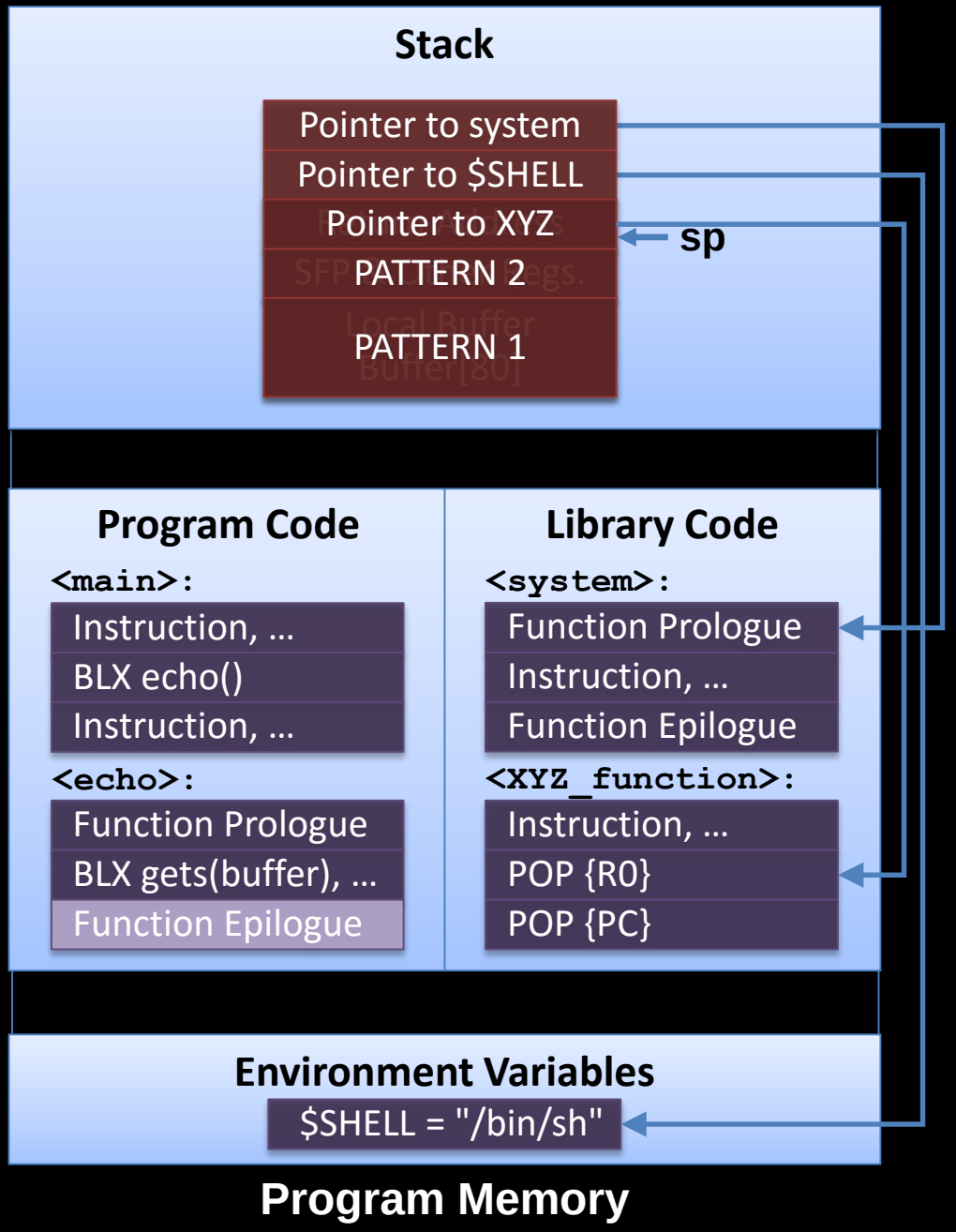
Program Memory


Adversary


Stack
Return Address
SFP & Other Regs.
Local Buffer
Buffer[80]
sp
Program Code
<main>:
Instruction, …
BLX echo()
Instruction, …
<echo>:
Function Prologue
BLX gets(buffer), …
Function Epilogue
Library Code
<system>:
Function Prologue
Instruction, …
Function Epilogue
<XYZ_function>:
Instruction, …
POP {R0}
POP {PC}
Environment Variables
$SHELL = "/bin/sh"
Program Memory


Adversary
Corrupt Control Structures
Stack
Pointer to system
Pointer to $SHELL
Pointer to XYZ
PATTERN 2
PATTERN 1
sp
Program Code
<main>:
Instruction, …
BLX echo()
Instruction, …
<echo>:
Function Prologue
BLX gets(buffer), …
Function Epilogue
Library Code
<system>:
Function Prologue
Instruction, …
Function Epilogue
<XYZ_function>:
Instruction, …
POP {R0}
POP {PC}
Environment Variables
$SHELL = "/bin/sh"
Program Memory

echo() now returns!



Execution redirected to XYZ_function

Stack

Pointer to system
Pointer to $SHELL
Pointer to XYZ
PATTERN 2
PATTERN 1

sp

Program Code

<main>:
Instruction, …
BLX echo()
Instruction, …

<echo>:
Function Prologue
BLX gets(buffer), …
Function Epilogue

Library Code

<system>:
Function Prologue
Instruction, …
Function Epilogue

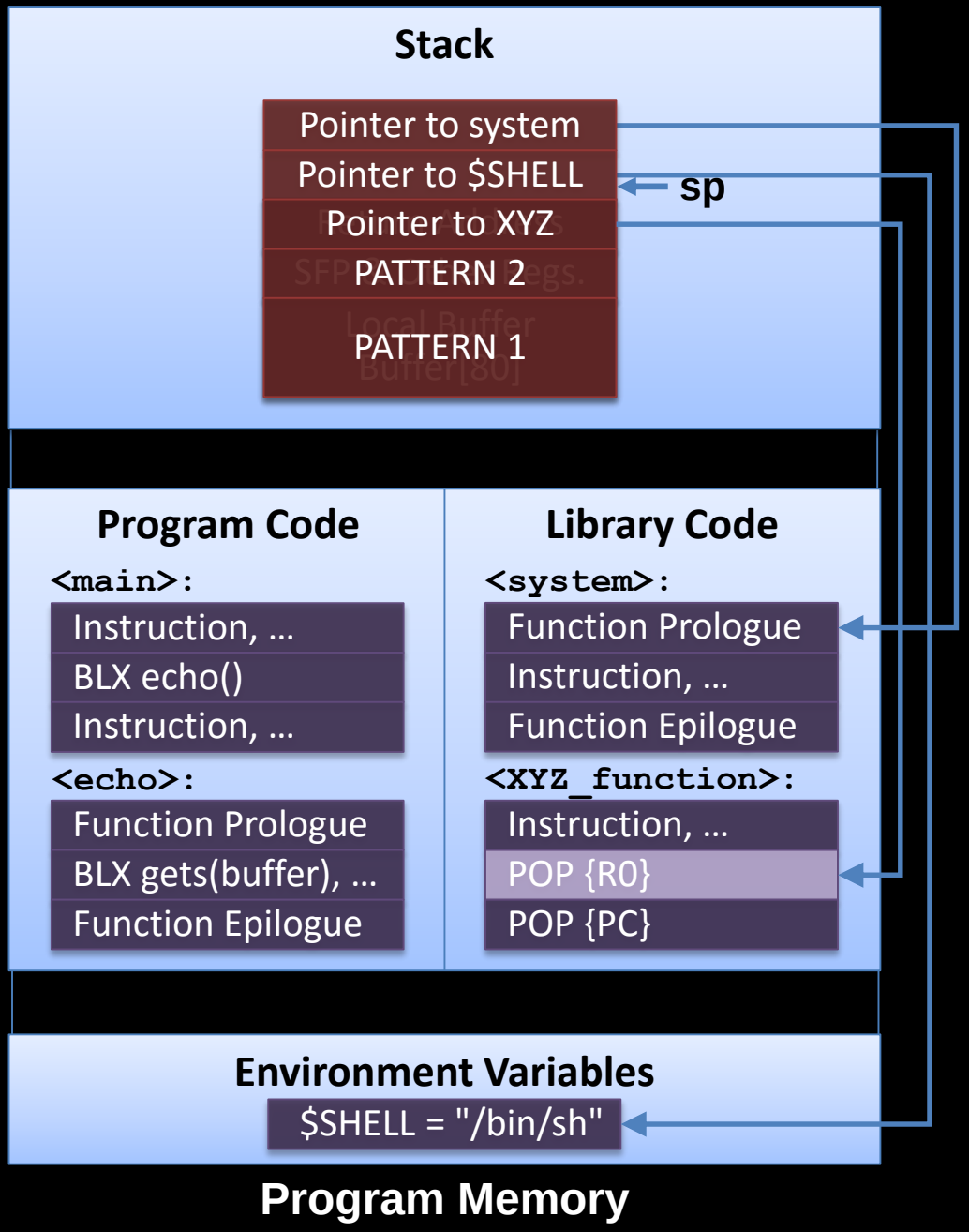
<XYZ_function>:
Instruction, …
POP {R0}
POP {PC}

Environment Variables

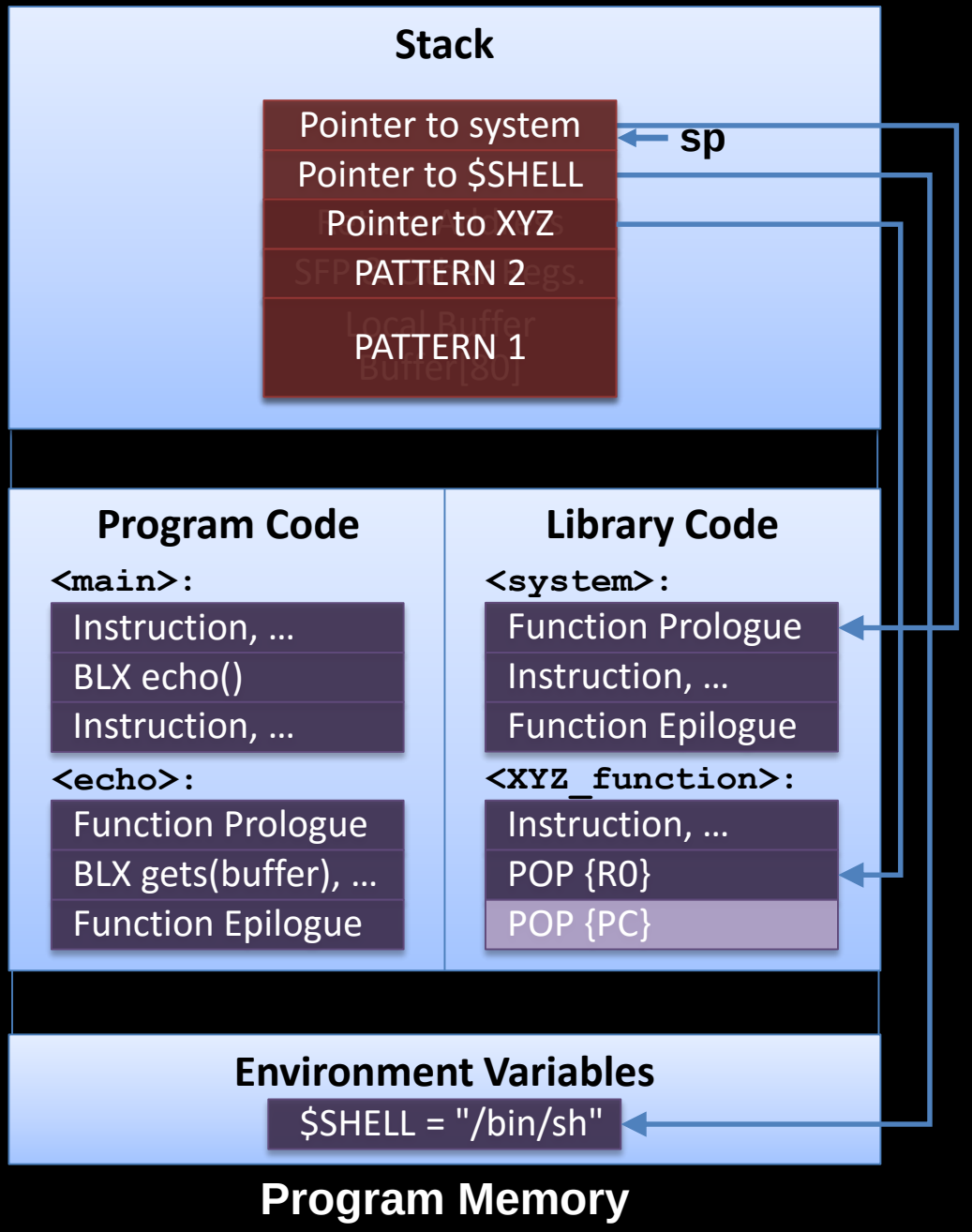
$SHELL = "/bin/sh"

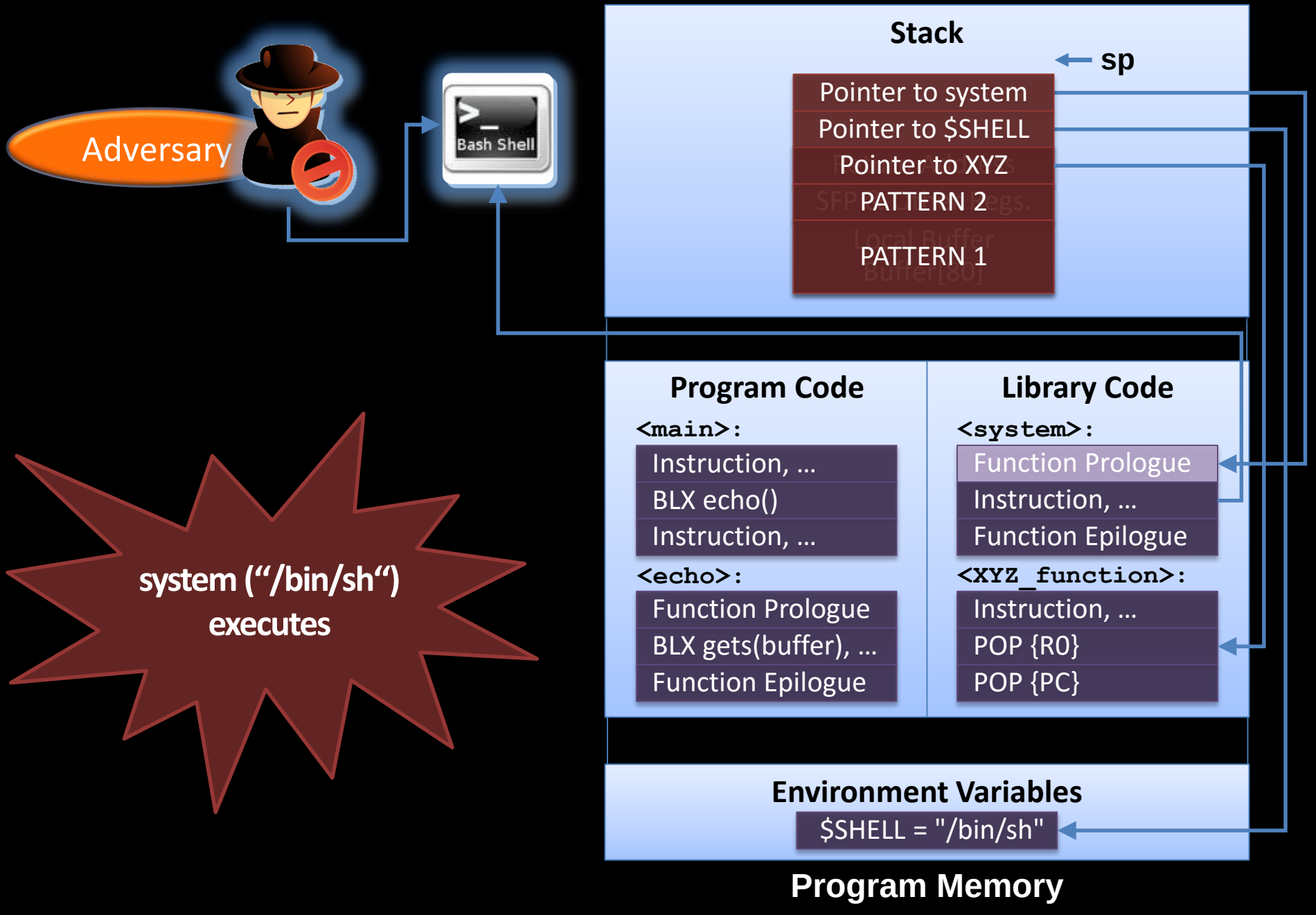
Program Memory

Load argument in R0, i.e., address to "/bin/sh" string is loaded into R0



Execution is now redirected to system



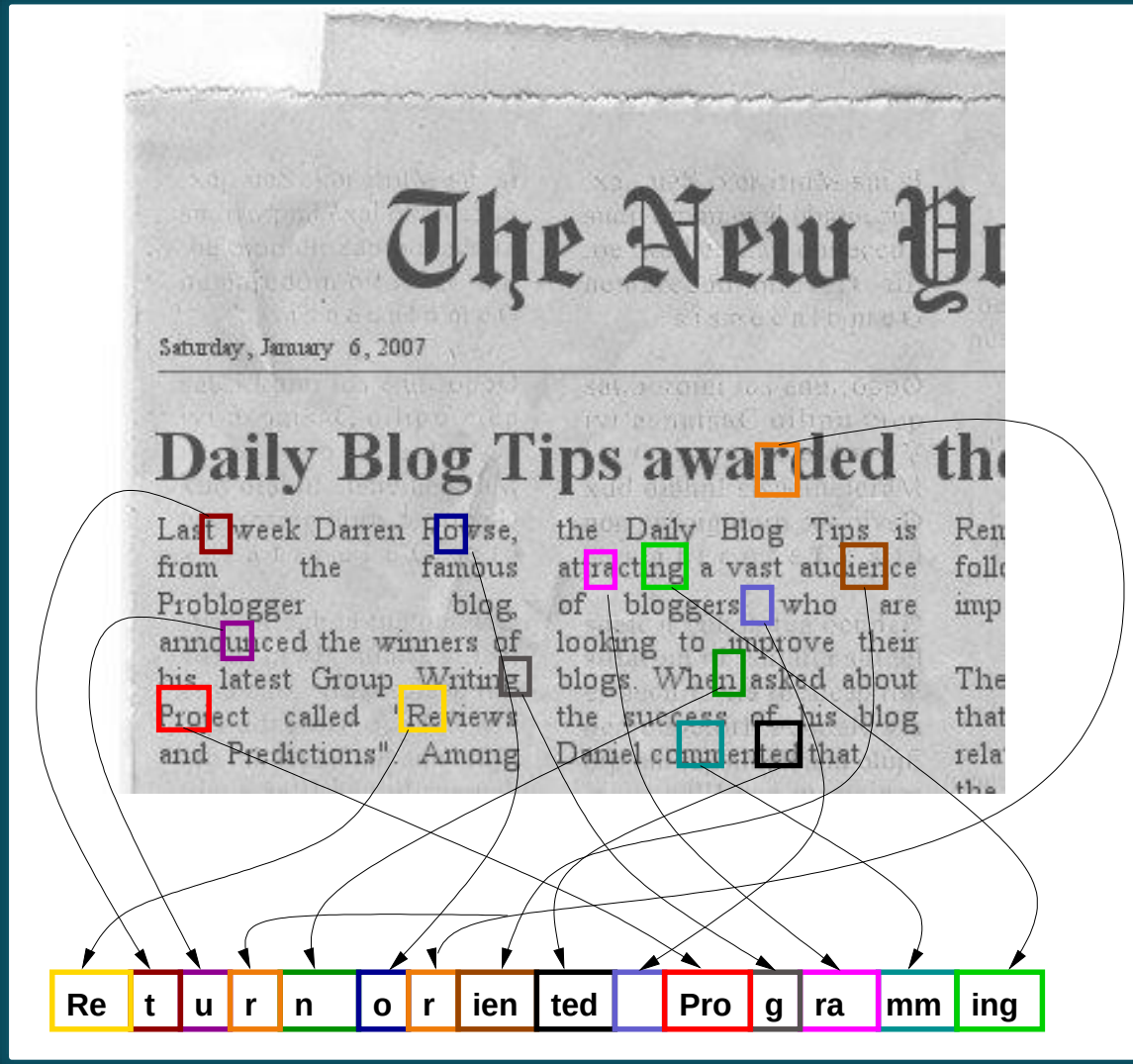

Adversary
Bash Shell
Stack
sp
Pointer to system
Pointer to $SHELL
Pointer to XYZ
PATTERN 2
PATTERN 1
Program Code
<main>:
Instruction, …
BLX echo()
Instruction, …
<echo>:
Function Prologue
BLX gets(buffer), …
Function Epilogue
Library Code
<system>:
Function Prologue
Instruction, …
Function Epilogue
<XYZ_function>:
Instruction, …
POP {R0}
POP {PC}
Environment Variables
$SHELL = "/bin/sh"
Program Memory
system ("/bin/sh") executes

Limitations

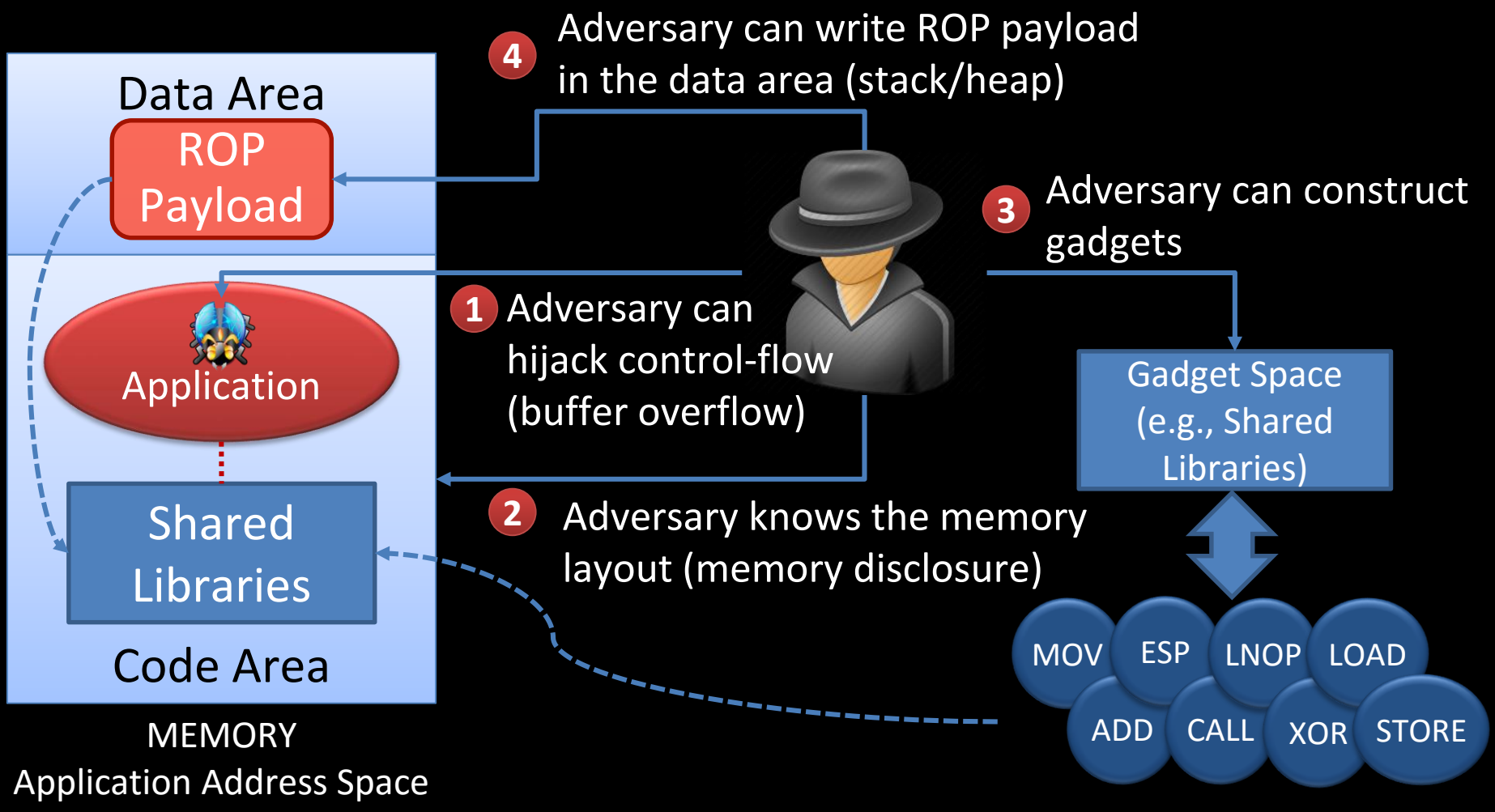
- No branching, i.e., no arbitrary code execution
- Critical functions can be eliminated or wrapped

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

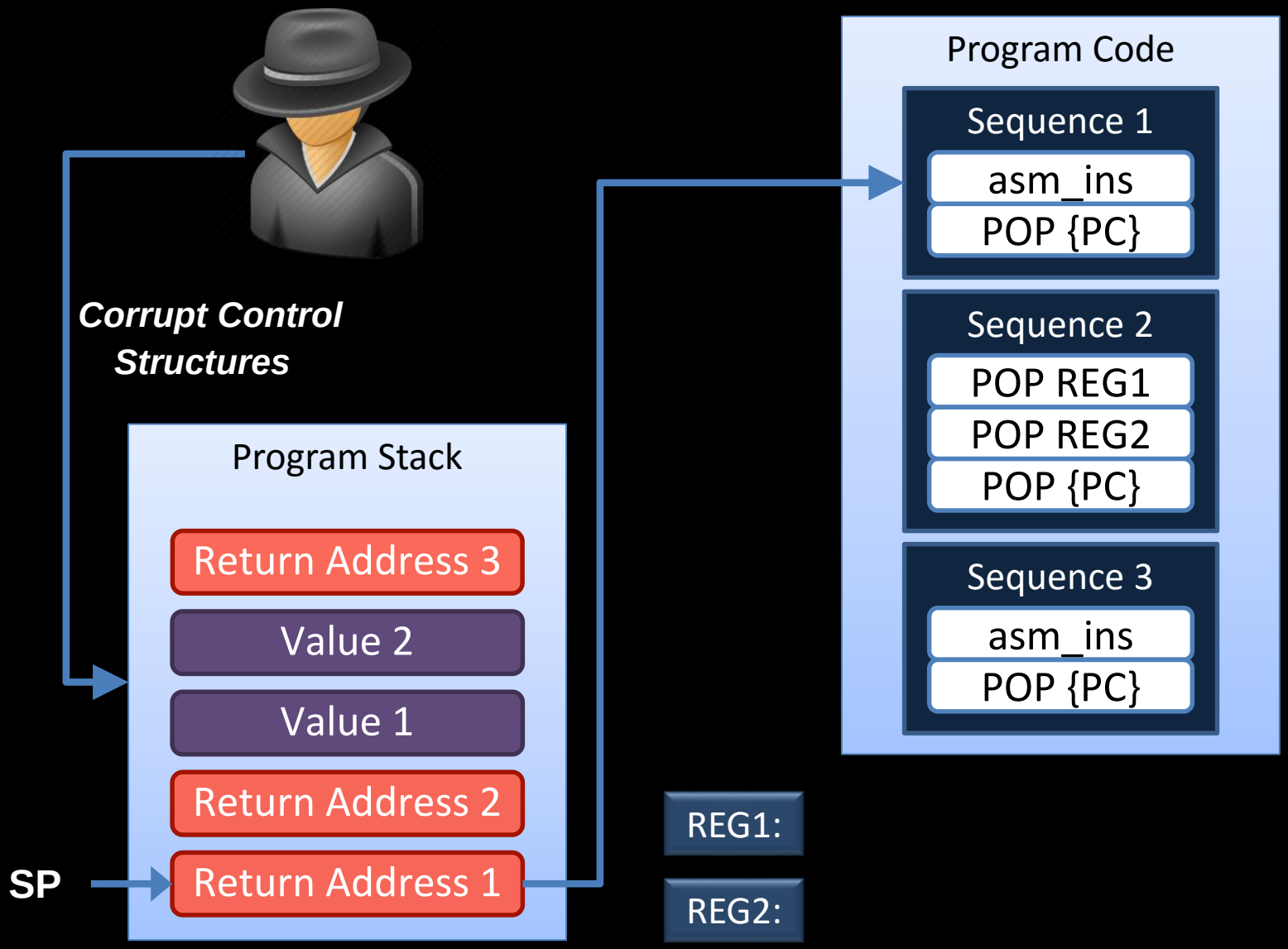
The Big Picture



ROP Adversary Model/Assumption



ROP Attack Technique: Overview



ROP Attack Technique: Overview

ROP Attack Technique: Overview

ROP Attack Technique: Overview

ROP Attack Technique: Overview

Program Code

Sequence 1

asm_ins

POP {PC}

Sequence 2

POP REG1

POP REG2

POP {PC}

Sequence 3

asm_ins

POP {PC}

...

Program Stack

SP

Return Address 3

Value 2

Value 1

Return Address 2

Return Address 1

REG1: Value 1

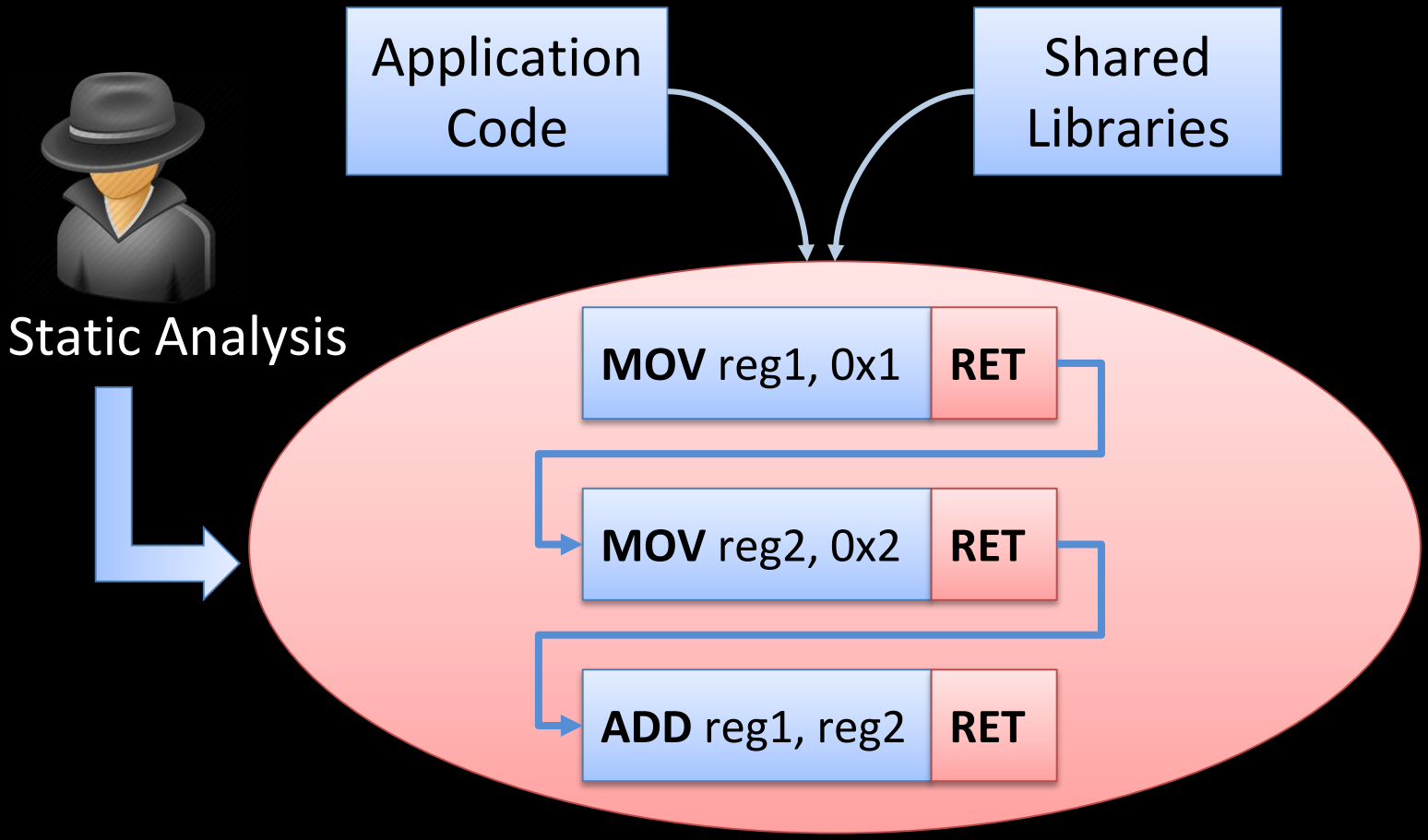
REG2: Value 2

Summary of Basic Idea

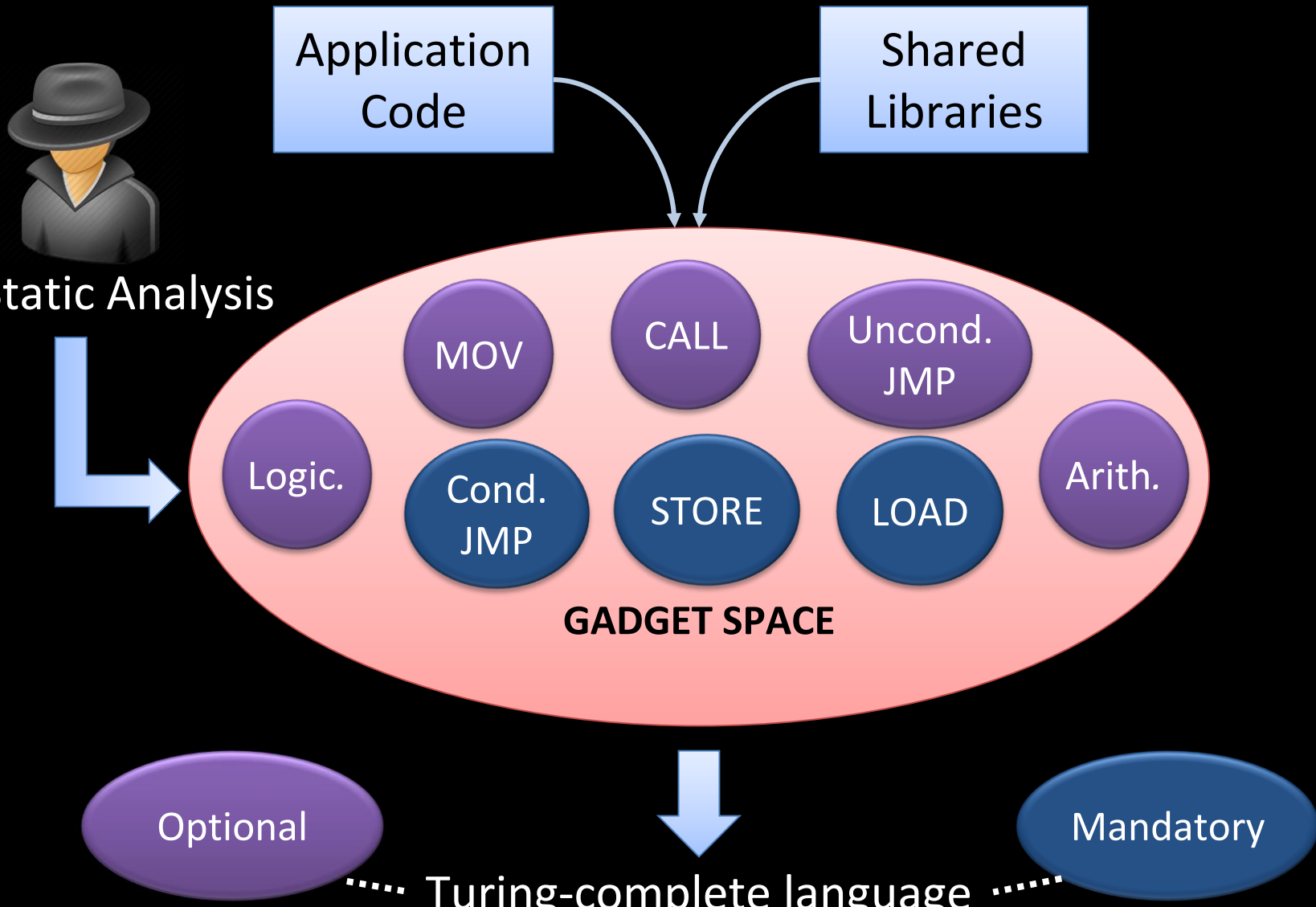
- Perform arbitrary computation with return-into-libc techniques
- Approach
  - Use small instruction sequences (e.g., of libc) instead of using whole functions
  - Instruction sequences range from 2 to 5 instructions
  - All sequences end with a return (POP{PC}) instruction
  - Instruction sequences 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



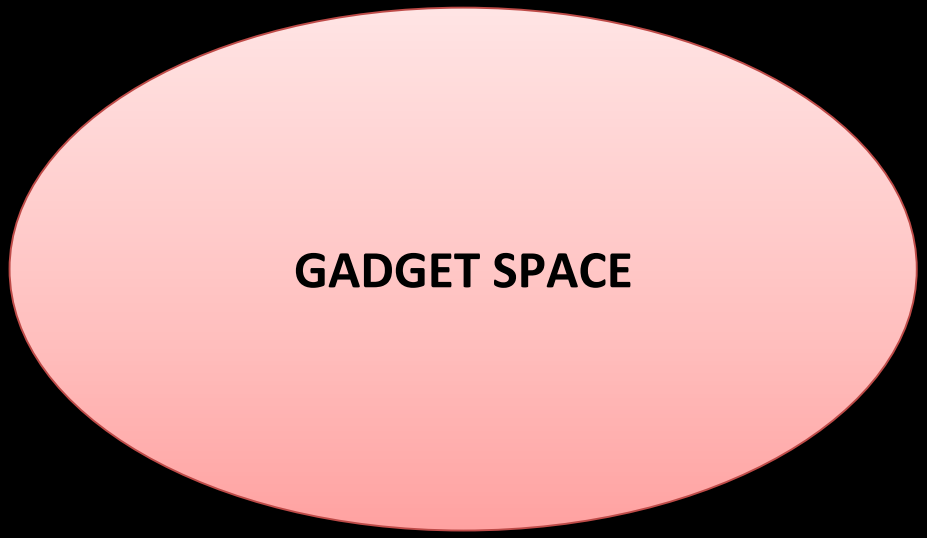
Code Base and Turing-Completeness

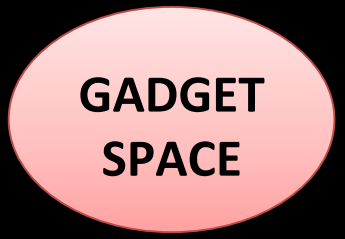


Gadget Space on Different Architectures

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

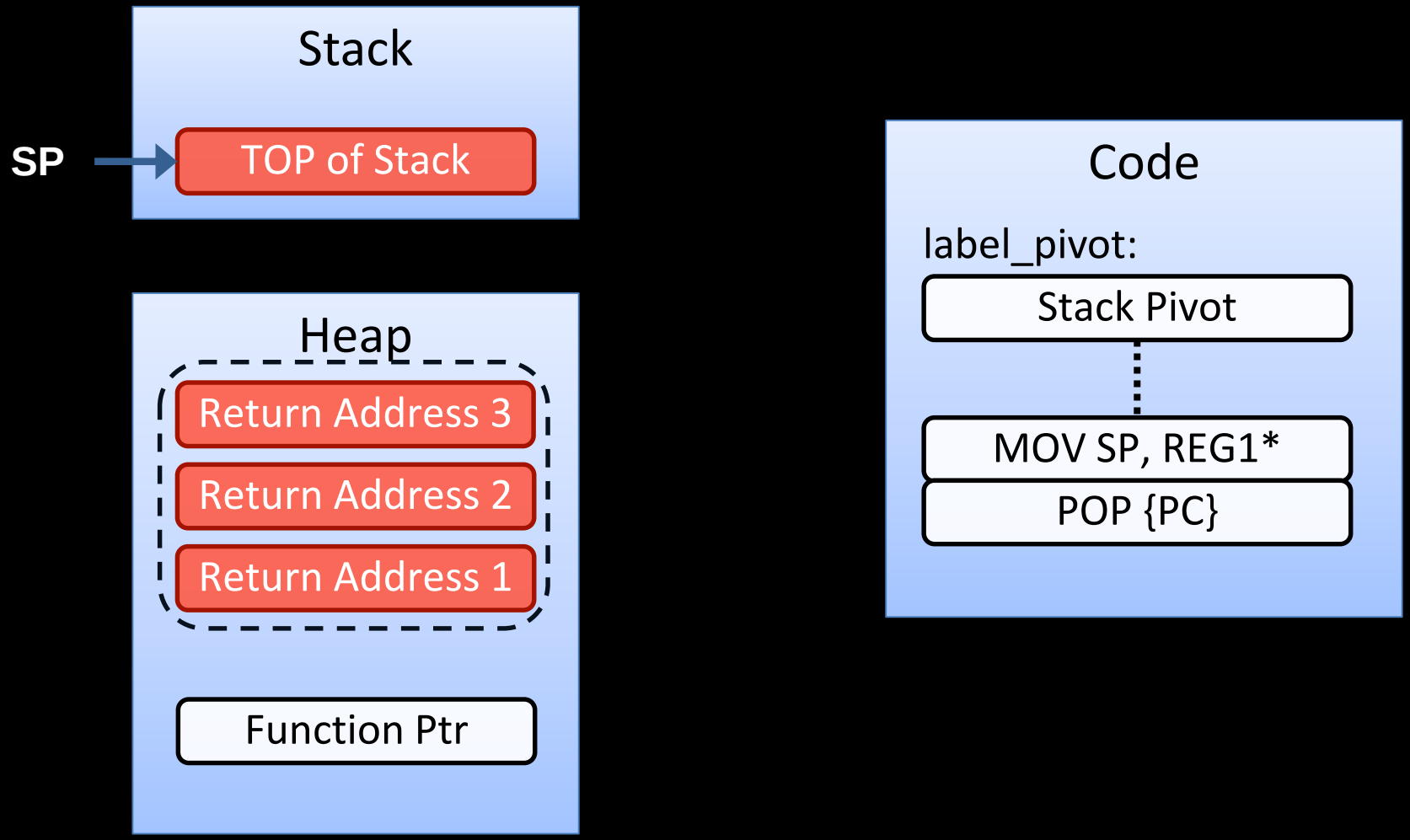
```
add %al,(%eax)
add %ch,%cl
ret
```

Stack Pivot

[Zovi, RSA Conference 2010]

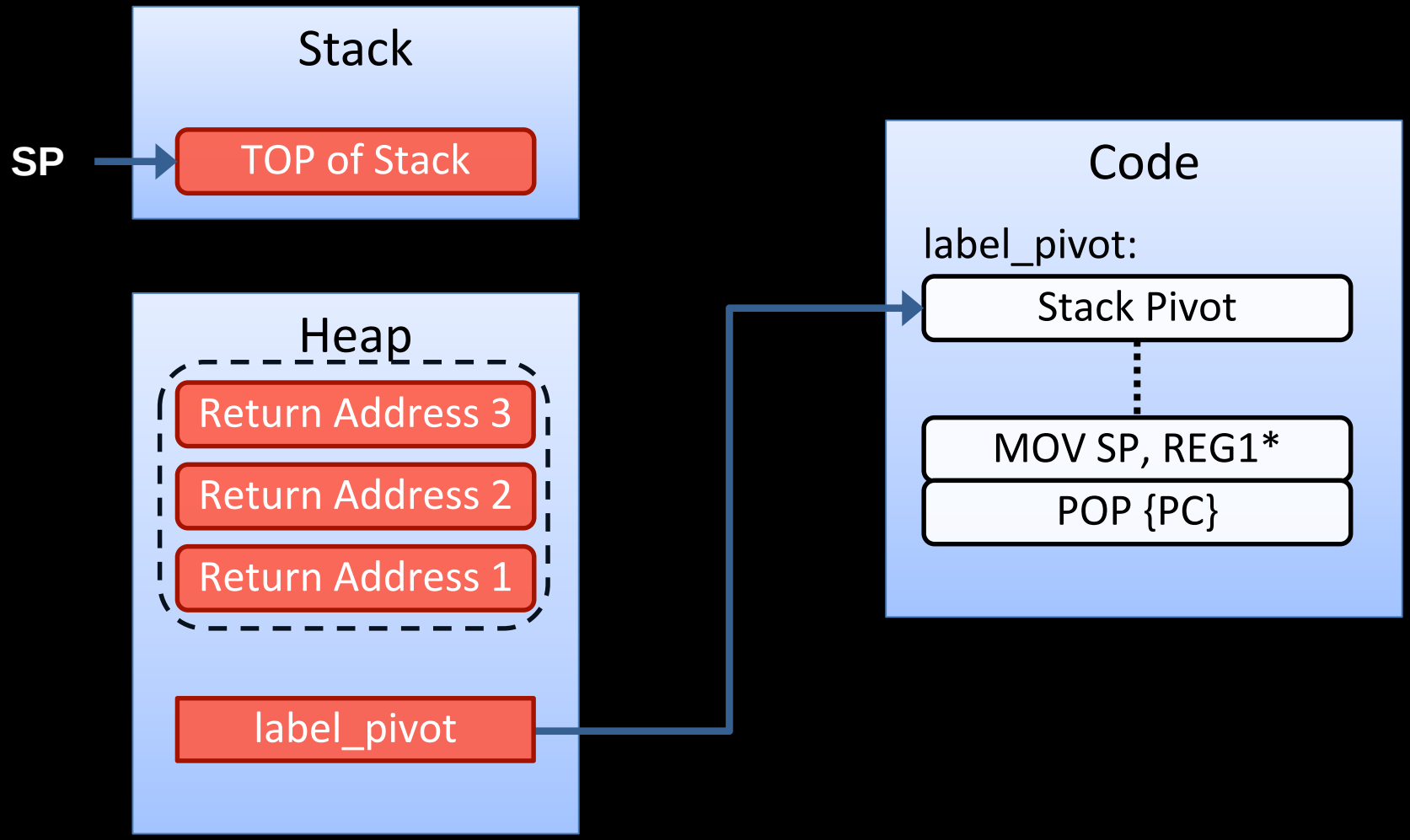
- Stack pointer plays an important role
  - It operates as an instruction pointer in ROP attacks
- Challenge
  - In order to launch a ROP exploit based on a heap overflow, we need to set the stack pointer to point to the heap
  - This is achieved by a stack pivot

Stack Pivot in Detail



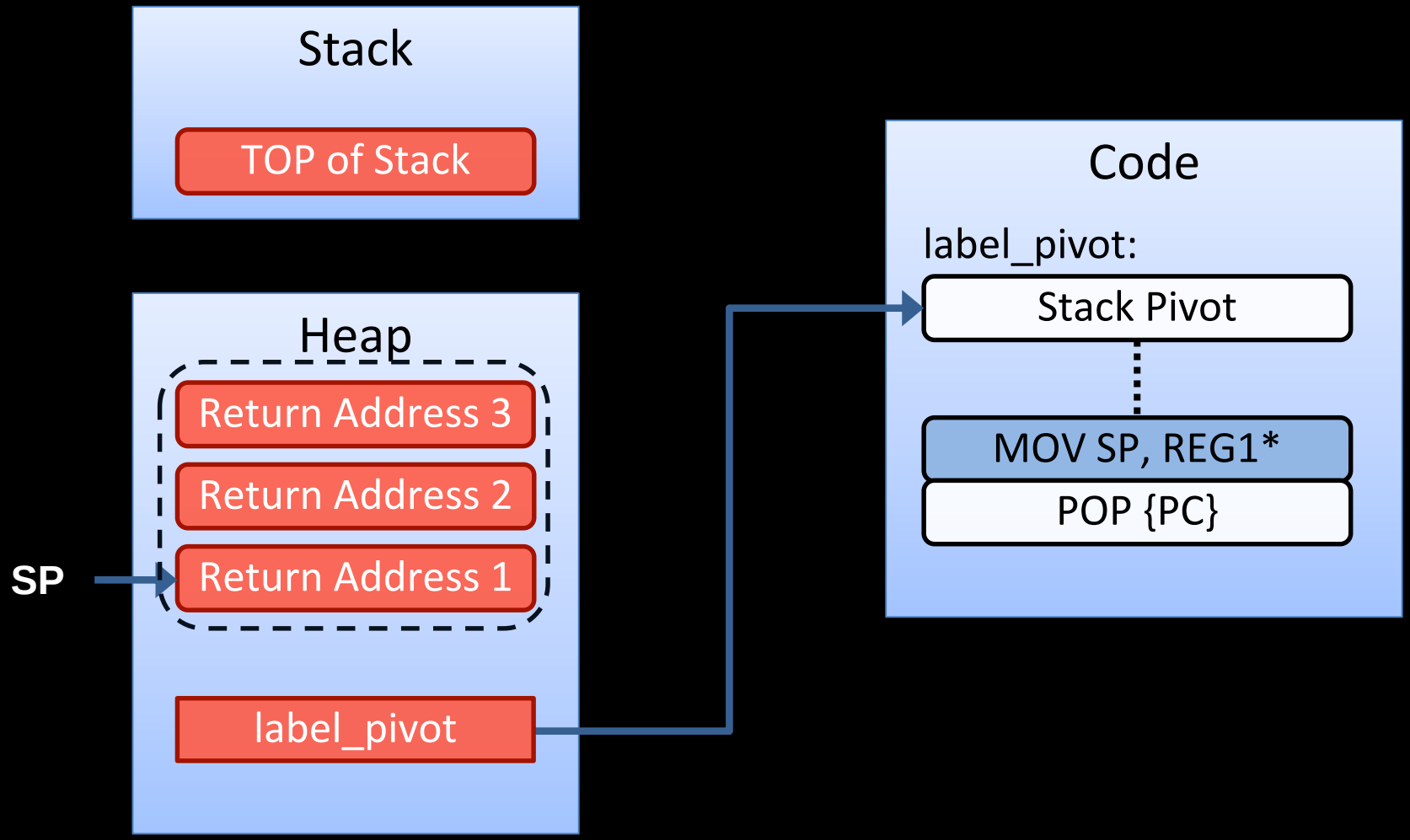
*REG1 is controlled by the adversary and holds beginning of ROP payload

Stack Pivot in Detail



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

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)*; **Blind ROP** *Bittau et al (IEEE S&P)*; **Out-Of-Control** *Göktas et al (IEEE S&P)*

2014 — **Stitching Gadgets** *Davi et al (USENIX)*; **Flushing Attacks** *Schuster et al (RAID)*; **ROP is Dangerous** *Carlini et al (USENIX)*