ROP with a 2nd Stack

- or -

This Exploit is a Recursive Fibonacci Sequence Generator

Nicholas Mosier, Class of 2020
Pete Johnson, Assistant Professor of Computer Science
Middlebury College

Middlebury

Overview

Goals:

- Compile a general-purpose, Turing-complete programming language to ROP shellcode.
- Create a "shellcode stack" for shellcode use.

Features

- Shellcode stack
- Subroutine calls
- Library calls
- 64-bit

Outline

- Intro to ROP
- Previous work
- ROPC, a ROP compiler
- ROPC-IR
- Two-stage exploits
- Demos

Work done on an x86_64 virtual machine running Arch Linux

Intro to x86_64

- x86_64 instruction set
    - CISC
- Registers
    - **`%rsp`** = stack pointer
    - **`%rbp`** = frame pointer
    - **`%rip`** = program counter
    - **`%rax`** - **`%rdx`**, **`%rsi`**, **`%rdi`** = “general-purpose” registers
- Intel syntax (`mov %rax, %rdx` means `%rax <- %rdx`)

Nomenclature

- **target program** — a vulnerable program whose execution the attacker is hijacking
- **target stack** — the call stack of the target program
- **shellcode** — the sequence of bytes fed to the target program that causes it to execute attacker-specified instructions
- **shellcode stack** — a stack that the shellcode can use for computation.
  NOTE: distinct from the target stack

Intro to Return-Oriented Programming (ROP)

Return-Oriented Programming

- Stack smashing attack that circumvents W^X / DEP
- Inject return addresses into **gadgets** onto stack
  - Each return address points to an instruction sequence, such as those in `libc`
  - **gadget** — short instruction sequence ending in a control transfer instruction, traditionally **`ret`**
- ROP shellcode chains these gadgets together to perform useful computation

Gadgets

```
0000000000010a5f0 <__memset_chk_ifunc>:
  [...]
  10a648:  85 c0                  test   eax,eax
  10a64a:  48 8d 15 6f 98 f9 ff   lea    rdx,[rip-0x66791]
  10a651:  48 8d 05 e8 97 f9 ff   lea    rax,[rip-0x66818]
  10a658:  48 0f 45 c2            cmovne rax,rdx
  10a65c:  c3                     ret
  10a65d:  0f 1f 00               nop    DWORD PTR [rax]
  10a660:  81 e2 00 00 02 00      and    edx,0x20000
  [...]

000000000004d7f0 <__push___start_context>:
   [...]
   4d83a:  f3 48 0f 1e cf        rdsspq rdi
   4d83f:  f3 41 0f 01 68 f8     rstorssp QWORD PTR [r8-0x8]
   4d845:  f3 0f 01 ea           saveprevssp
   4d849:  49 89 b9 a8 03 00 00  mov    QWORD PTR [r9+0x3a8],rdi
   4d850:  48 89 d4              mov    rsp,rdx
   4d853:  c3                    ret
   [...]
```

Gadgets

```
000000000010a5f0 <__memset_chk_ifunc>:
  [...]
  10a648:  85 c0                  test   eax,eax
  10a64a:  48 8d 15 6f 98 f9 ff   lea    rdx,[rip-0x66791]
  10a651:  48 8d 05 e8 97 f9 ff   lea    rax,[rip-0x66818]
  10a658:  48 0f 45 c2            cmovne rax,rdx
  10a65c:  c3                     ret
  10a65d:  0f 1f 00               nop    DWORD PTR [rax]
  10a660:  81 e2 00 00 02 00      and    edx,0x20000
  [...]

00000000004d7f0 <__push___start_context>:
  [...]
  4d83a:  f3 48 0f 1e cf        rdsspq rdi
  4d83f:  f3 41 0f 01 68 f8     rstorssp QWORD PTR [r8-0x8]
  4d845:  f3 0f 01 ea           saveprevssp
  4d849:  49 89 b9 a8 03 00 00 mov    QWORD PTR [r9+0x3a8],rdi
  4d850:  48 89 d4              mov    rsp,rdx
  4d853:  c3                    ret
  [...]
```

Previous Work

- Q: Exploit Hardening Made Easy (2011): https://edmcman.github.io/papers/usenix11.pdf
- Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming (2012): https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf
- pakt's ROP compiler (2013): https://github.com/pakt/ropc
- jeffball55's pyrop (2016): https://github.com/jeffball55/rop_compiler/tree/master/pyrop

Previous Work

	Turing complete	safe library calls	shellcode stack	shellcode calls	64-bit
Turing-complete set of gadgets	X				X
Q: Exploit Hardening Made Easy (2011)		X			
Microgadgets (2012)	X				
pakt's ROP compiler (2013)	X		X	X	
jeffball55's pyrop (2016)		X			X

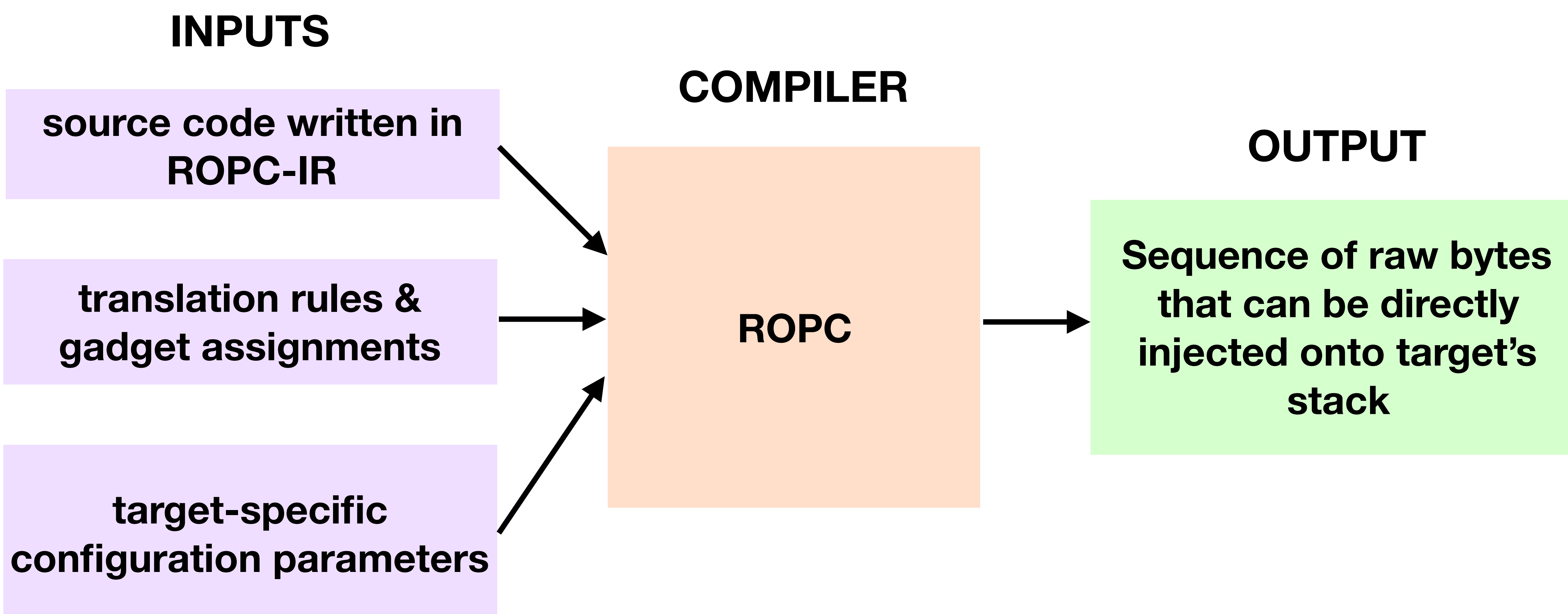
Previous Work

	Turing complete	safe library calls	shellcode stack	shellcode calls	64-bit
Turing-complete set of gadgets	X				X
Q: Exploit Hardening Made Easy (2011)		X			
Microgadgets (2012)	X				
pakt's ROP compiler (2013)	X		X	X	
jeffball55's pyrop (2016)		X			X
nmosier's ROPC (2019)	X	X	X	X	X

ROPC: an x86_64 ROP Compiler

Turing-Complete • Shellcode Stack

ROPC Overview



Features

- Turing-complete
- Shellcode stack
- Subroutine calls (into shellcode)
- Library calls (into `libc`)
- 64-bit

The Shellcode Stack

We will access the shellcode stack through a stack pointer, **SP**.

Suppose we don't hard-code the shellcode stack pointer, **SP**.

Then **SP** *must* be stored in a register.

But library calls destroy registers.

How to preserve **SP** without hard-coding any addresses?

The Shellcode Stack

<u>Solution</u>: use **`%rbp`**, the target frame pointer, since library functions preserve **`%rbp`**.

`printf:`

`push %rbp`	
`mov %rbp,%rsp`	
…	
`leave`	`mov %rsp,%rbp`
	`pop %rbp`
`ret`	

The Shellcode Stack

Challenge: What to use as the shellcode stack?

The Shellcode Stack

Challenge: What to use as the shellcode stack?

Solution: `malloc(3)`

- Dynamic address (not hard-coded)
- Arbitrarily-sized shellcode stack

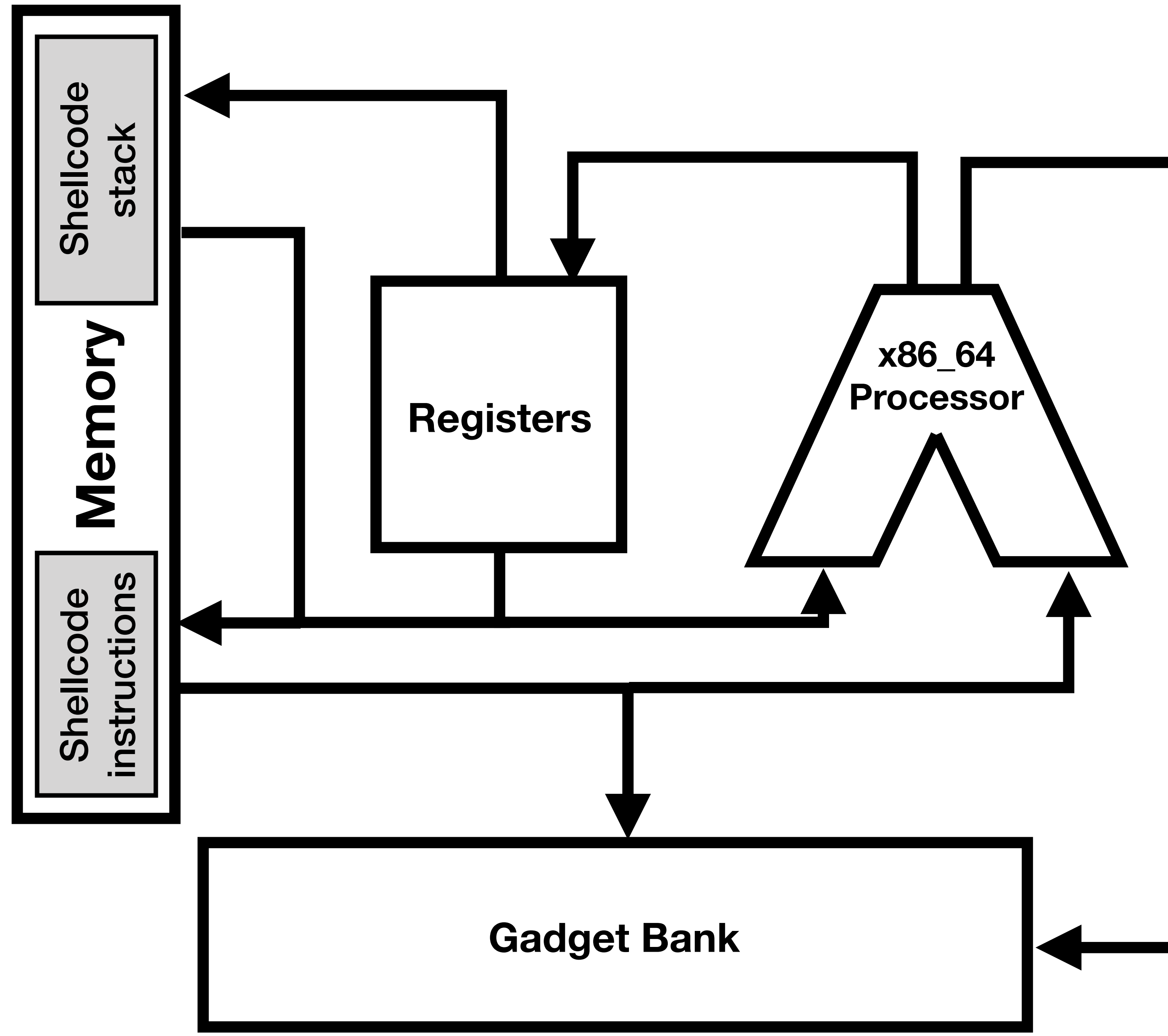
ROPC-IR

Intro to ROPC-IR

- **ROPC-IR**: the made-up shellcode assembly language accepted by ROPC
- **ROPC-IR program** consists of **ROPC-IR instructions**
- **ROPC-IR instruction** is a well-defined operation
- **ROPC-IR instruction space** <**=>** target stack
- Shellcode program counter **`PC <=> %rsp`**
- ROPC-IR instructions are *interfaces* — must be reimplemented using gadgets from each target program

ROPC-IR Architecture

- 64-bit word size
- 3 registers
  - **ACC**: accumulator (`%rax`)
  - **SP**: shellcode stack pointer (`%rbp`)
  - **PC:** shellcode program counter (`%rsp`)
- 2 flags (**ZF**, **CF**)



Instructions Overview

Directives	ret, dq, db, resq, org
Move	MOV
Stack	PUSH, POP, PEEK, GET, PUT, LEA, ALLOC, LEAVE
Memory	LD, STO
Arithmetic	ADD, ADDFROM, SUB, NEG, INC, DEC
Comparison	CMP
Branching	JMP, JEQ, JNE, JLT, JGT
Calls	CALL, RET, SYSCALL3, LIBCALL3

MOV *imm64*

- DESC: Move 64-bit constant *imm64* into **ACC**.
- EXAMPLE:

```
MOV imm64 :=
    ret &0xfd9f8
    dq imm64
```

MOV *imm64*

- DESC: Move 64-bit constant *imm64* into **ACC**.
- EXAMPLE:

```
MOV imm64 :=
    ret &0xfd9f8
    dq imm64
```

```
<semctl@@GLIBC_2.2.5>:
  [...]
  fd9f8:  58    pop rax
  fd9f9:  c3    ret
  [...]
```

NEG

- DESC: Negate the value in `ACC` (two's complement).
- EXAMPLE:

```
NEG :=
    ret &0x119227
    ret &0xfd9f8
    dq 0
    ret &0xa39e3
```

NEG

- <u>DESC</u>: Negate the value in `ACC` (two's complement).

- <u>EXAMPLE</u>:

```
NEG :=
    ret &0x119227   ] _MOV rdx,rax
    ret &0xfd9f8    ]
                    ] _MOV rax,0
    dq 0            ]
    ret &0xa39e3    ] _SUB rax,rdx
```

NEG

- DESC: Negate the value in `ACC` (two's complement).
- EXAMPLE:

```
NEG :=
    _MOV rdx,rax
    _MOV rax,0
    _SUB rax,rdx
```

PUSH

- <u>DESC</u>: Push value in ACC onto shellcode stack.
  `(SP <- SP - 8, [SP] <- ACC)`

- <u>EXAMPLE</u>:

```
PUSH :=
        _MOV rcx,rax ; preserve ACC
        _MOV rax,rbp ; get SP
        _ADD rax,-8  ; dec SP; destroys rdi
        _MOV rdx,rax
        _MOV [rdx],rcx ; move ACC into SP
        _MOV rbp,rax
        _MOV rax,rcx ; restore ACC
```

`JMP` *`imm64`*

- <u>DESC</u>: Direct jump to address *imm64*.
  (**PC** <- *imm64*)

- <u>EXAMPLE</u>:

```
JMP imm64 :=
    _MOV rsp,imm64
```

JEQ *imm64*

- DESC: Direct jump if equal, i.e. zero flag (ZF) is set.

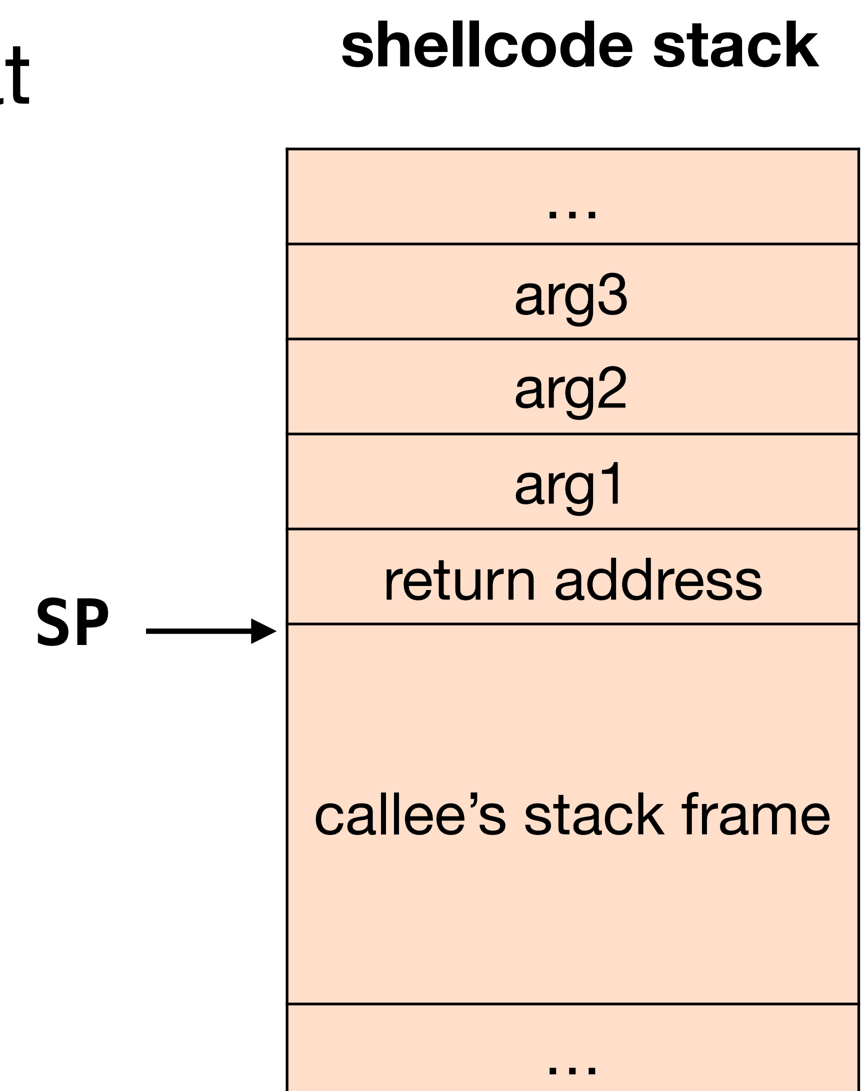
```
if (ZF) {
  PC <- imm64
}
```

- EXAMPLE:

```
JEQ imm64 :=
    _MOV   r9,rax          ; preserve ACC
    _MOV   rax, $ + 8*9
    _MOV   rdx,imm64
    _CMOVE rax,rdx
    _MOV   rdx,rax
    _MOV   rax,r9          ; restore ACC
    _MOV   rsp,rdx         ; indirect jump
```

CALL *imm64*

- DESC: Call shellcode subroutine at address *imm64*.
- Caller pushes arguments onto shellcode stack (before CALLing)
- EXAMPLE:




```
CALL imm64 :=
     PUSH $+240 ; 240 is size of
     JMP imm64  ; PUSH instruction
```

`LIBCALL` *`imm64`*

Observation: Previous approach for subroutine calls inadequate because library functions don't "know" to access stack through `%rbp`, not `%rsp`

Challenge: How to sandbox stack usage of library function, ensuring it won't overwrite nearby shellcode instructions?

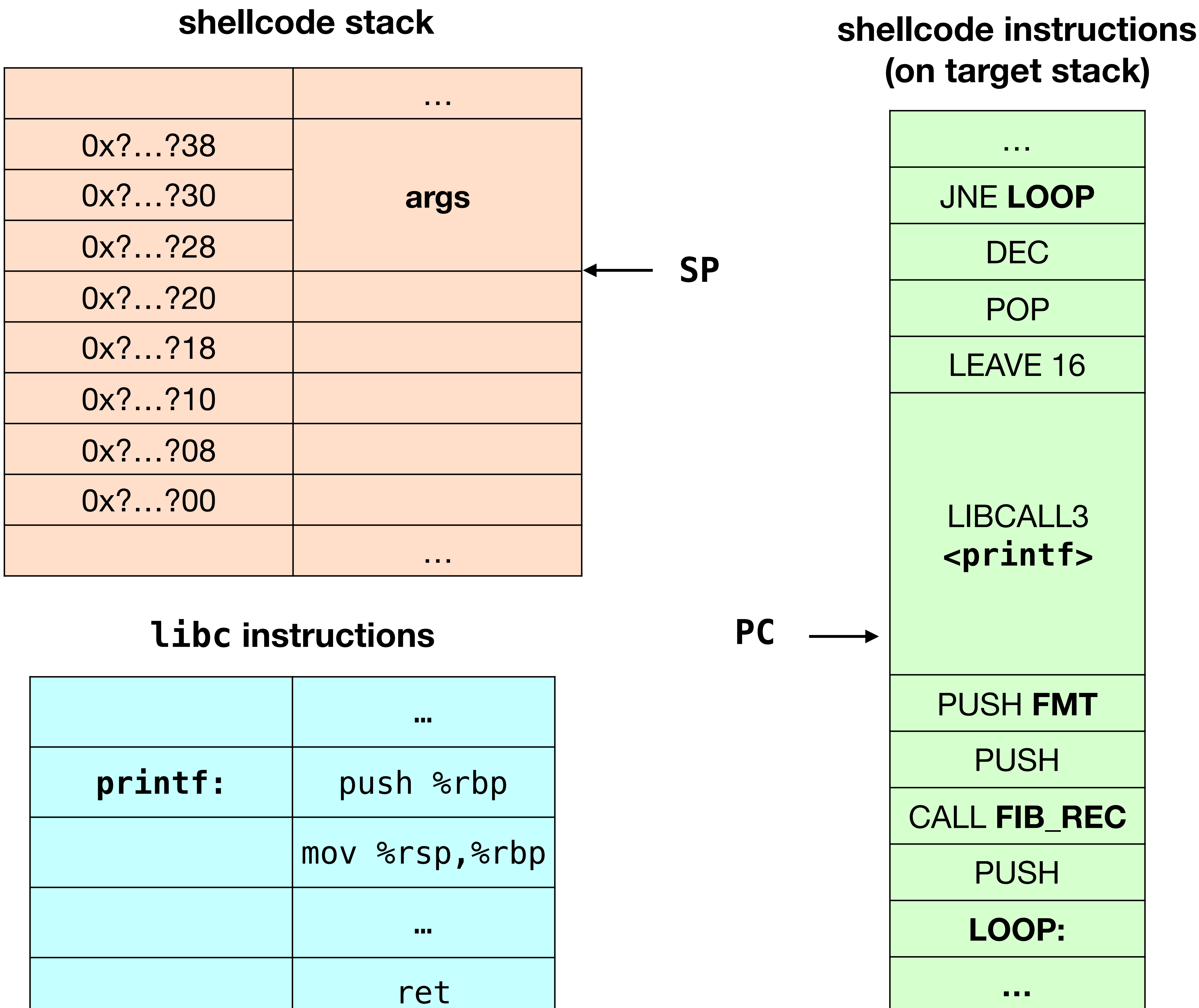
`LIBCALL` *`imm64`*

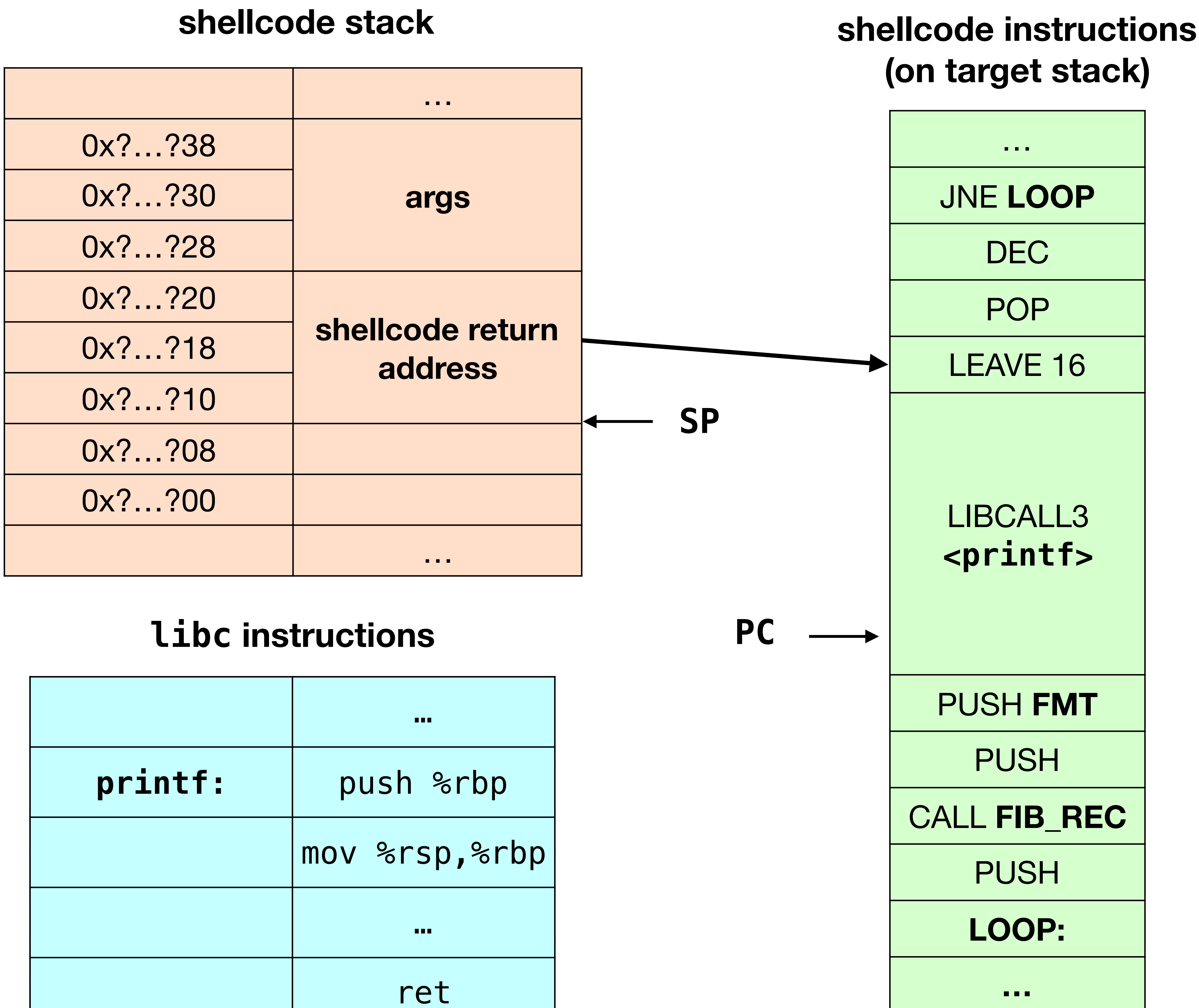
Idea: Use the shellcode stack! That is, point **`%rsp`** to the top of the shellcode stack, **`SP`**

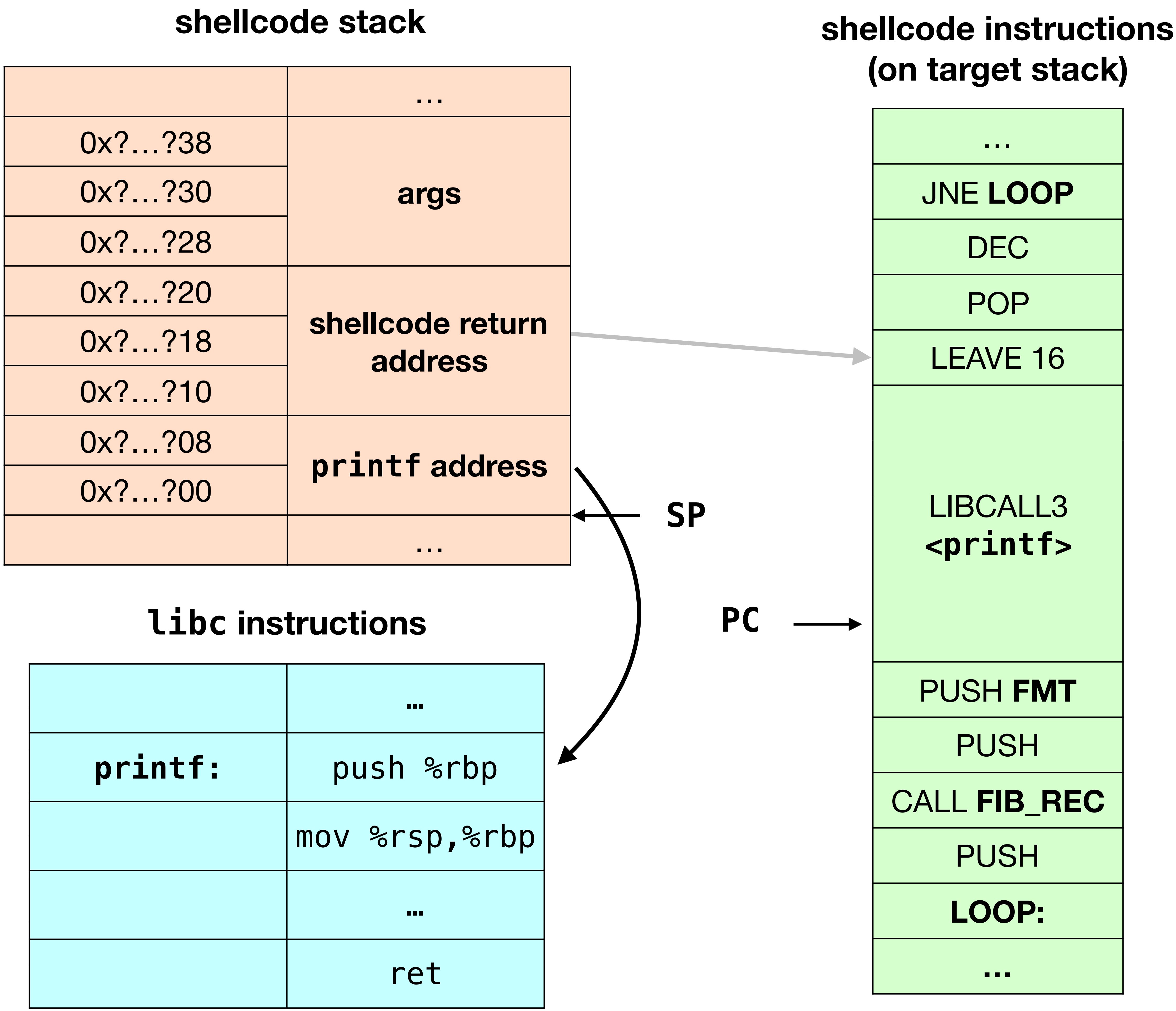
Challenge: But **`%rsp`** is the shellcode **`PC`**, which must be preserved across libcalls. How to preserve **`PC`**?

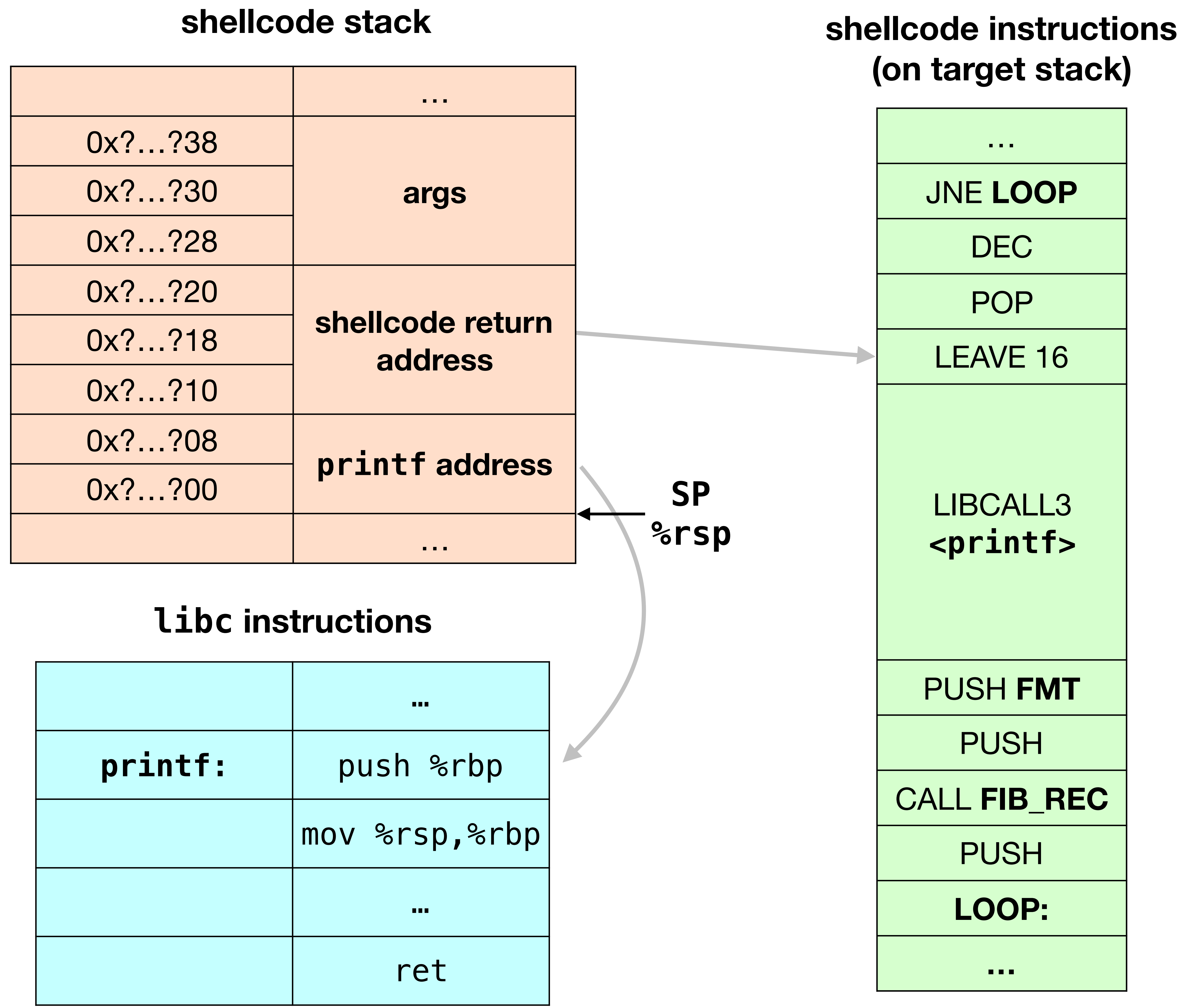
Solution: Store the shellcode return address on the shellcode stack, and restore it using a “`pop %rsp`” gadget.

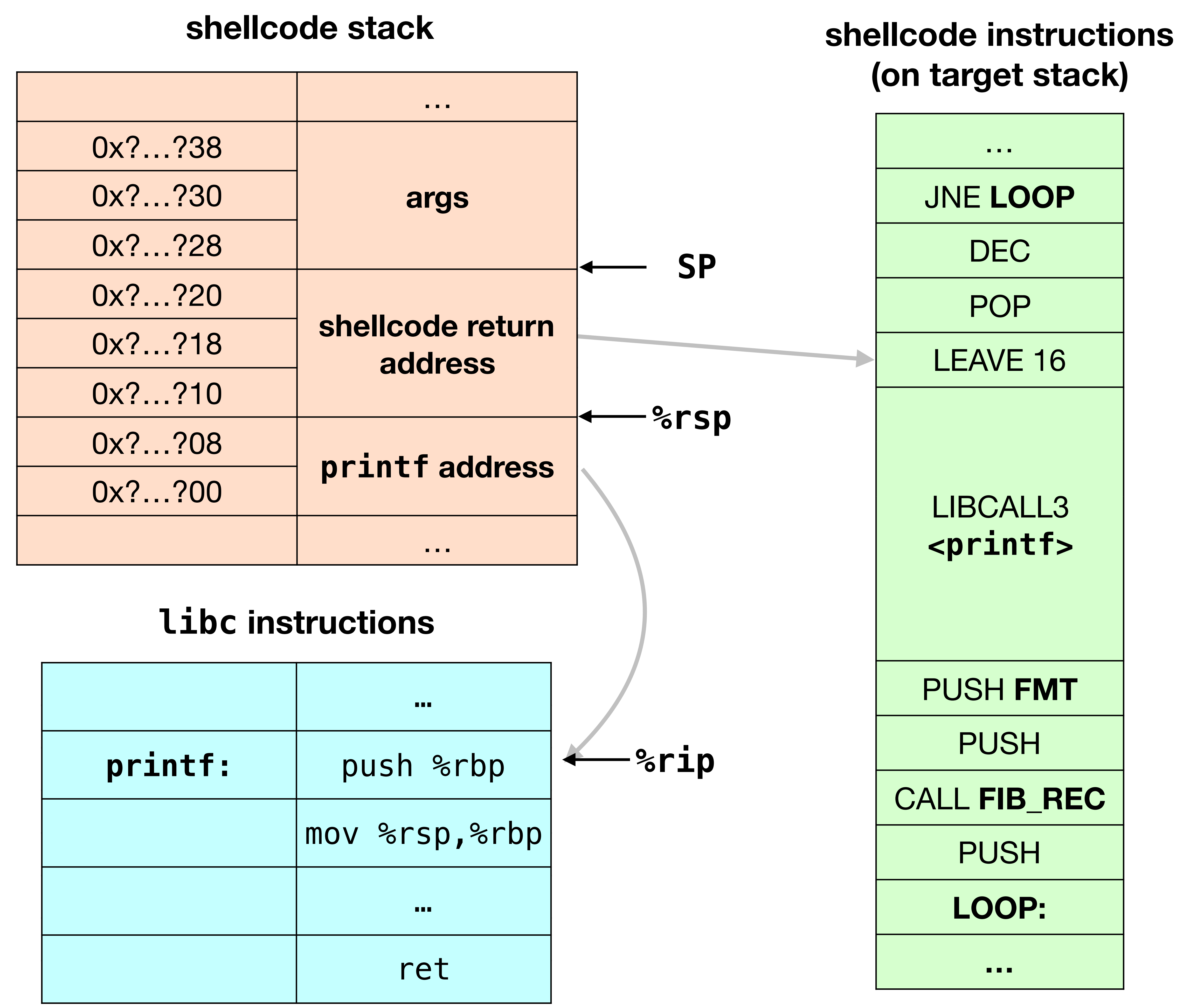
Library Call Diagram (simplified)


shellcode stack
...
0x?...?38
0x?...?30
args
0x?...?28
SP
0x?...?20
0x?...?18
0x?...?10
0x?...?08
0x?...?00
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PC
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

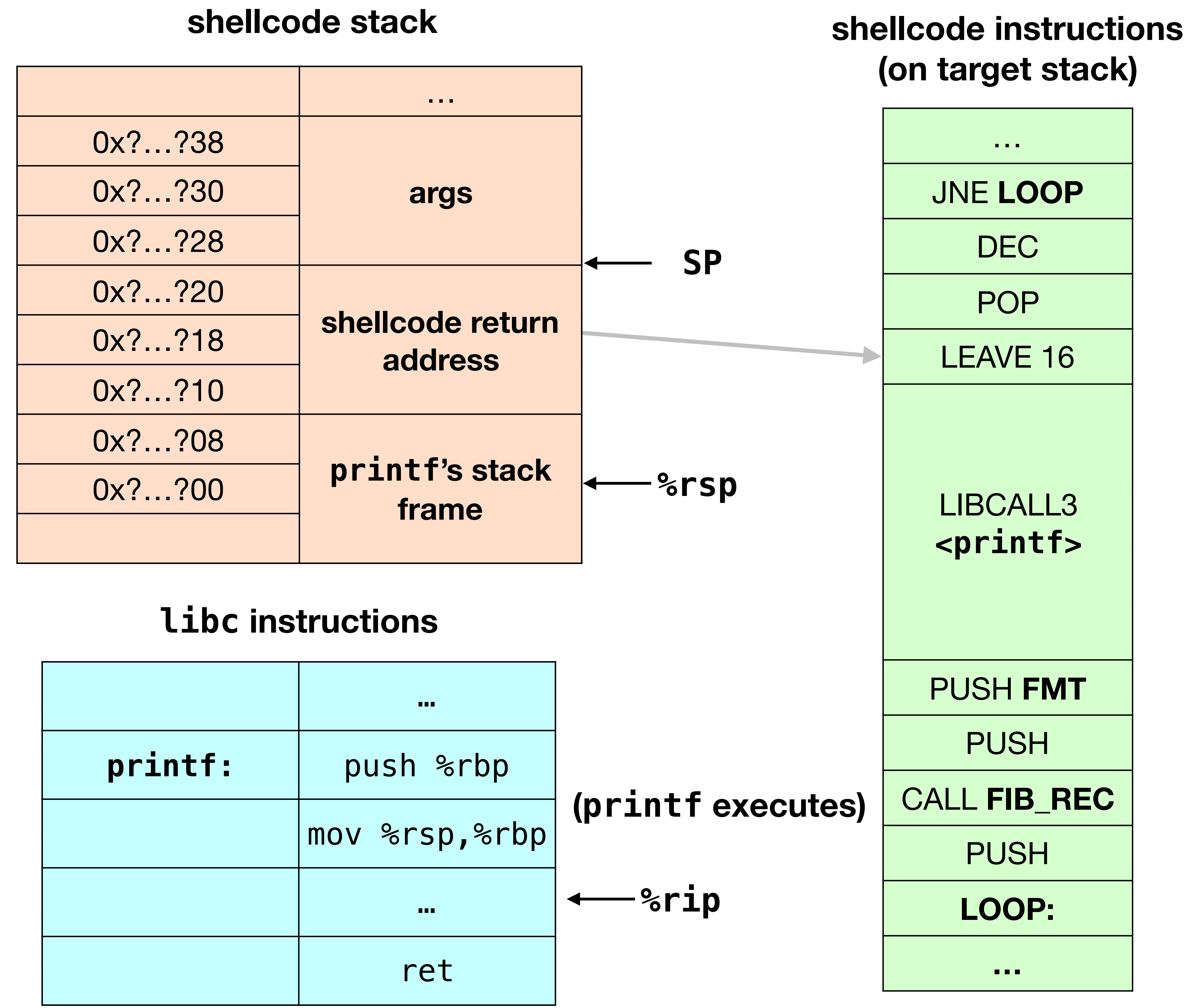

shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
0x?...?20
0x?...?18
0x?...?10
shellcode return address
0x?...?08
0x?...?00
...
SP
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PC
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret


shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
0x?...?20
0x?...?18
0x?...?10
shellcode return address
0x?...?08
0x?...?00
printf address
...
SP
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
PC
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

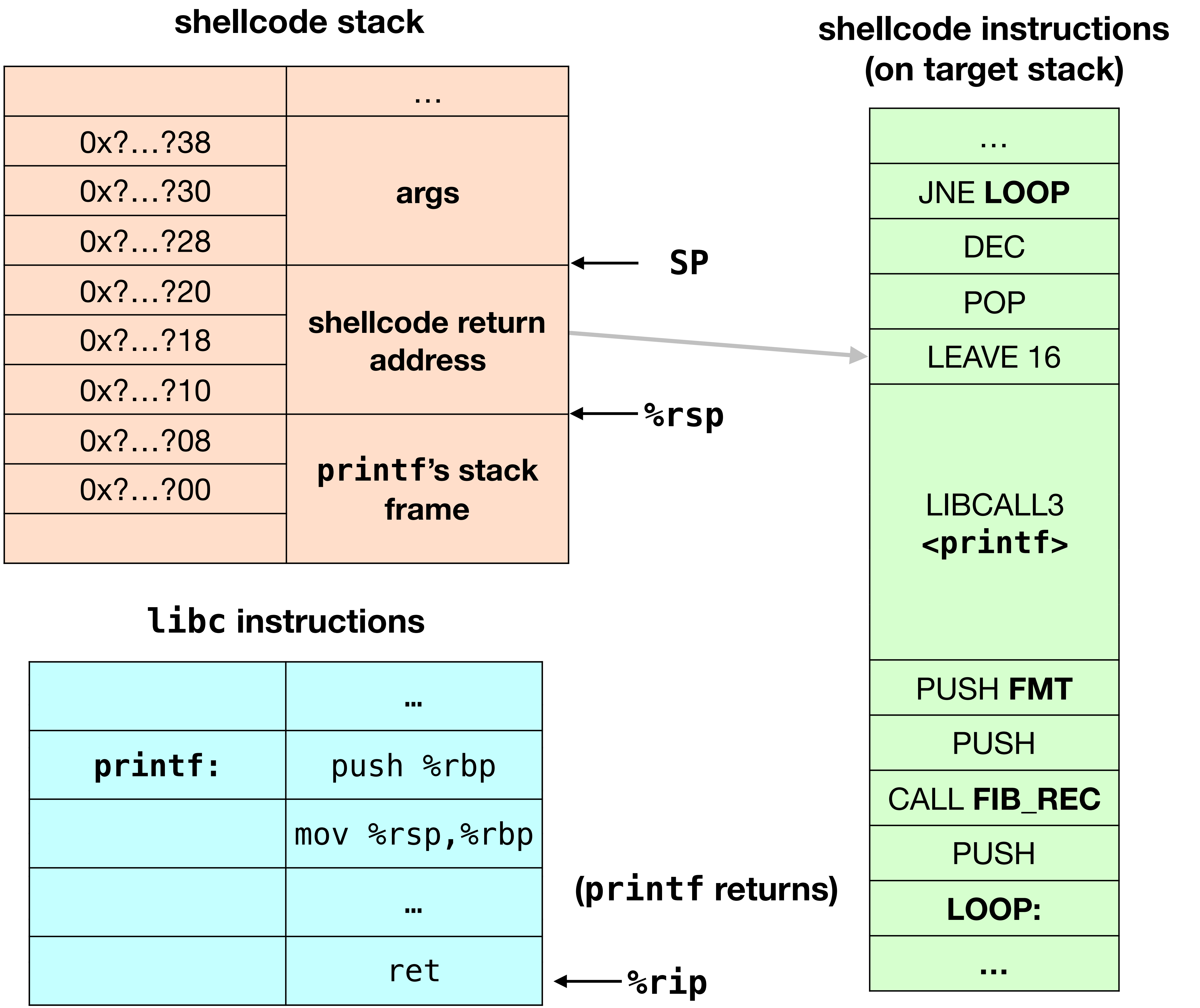

shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
0x?...?20
0x?...?18
0x?...?10
shellcode return address
0x?...?08
0x?...?00
printf address
...
SP
%rsp
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...



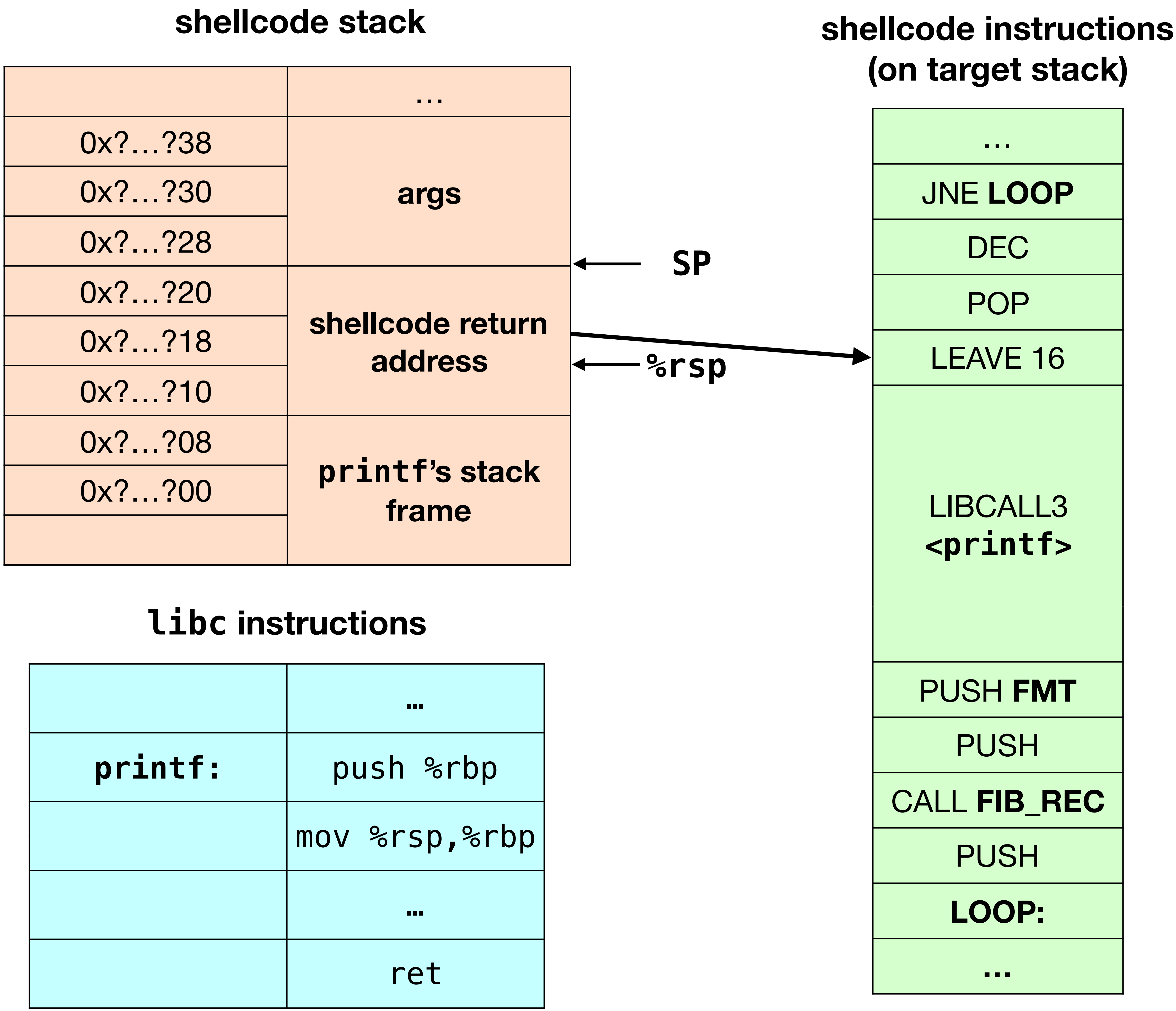
shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
SP
0x?...?20
0x?...?18
0x?...?10
shellcode return address
%rsp
0x?...?08
0x?...?00
printf address
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

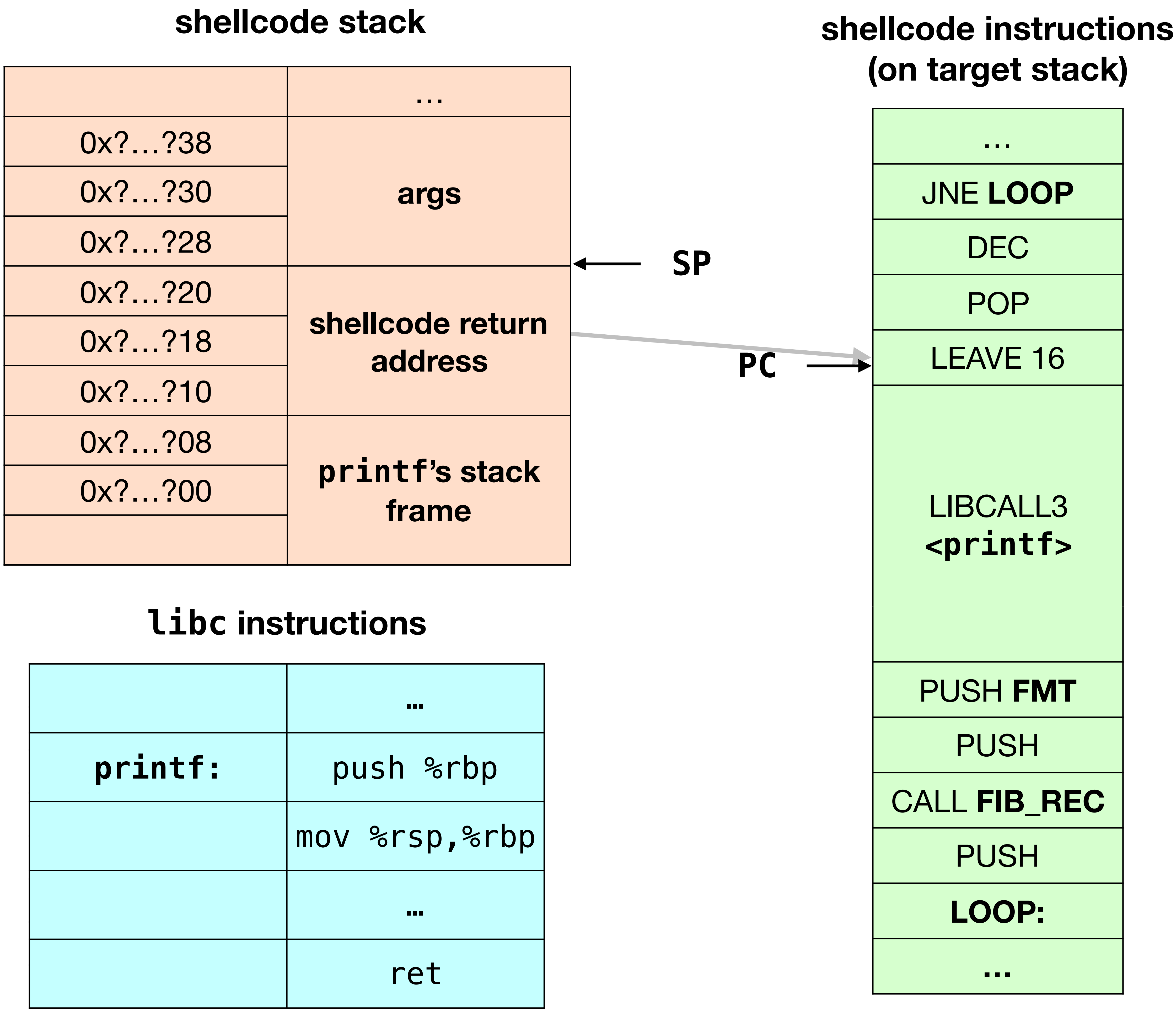


shellcode stack
...
0x?…?38
0x?…?30
0x?…?28
args
SP
0x?…?20
0x?…?18
0x?…?10
shellcode return address
0x?…?08
0x?…?00
printf's stack frame
%rsp
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
(printf executes)
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

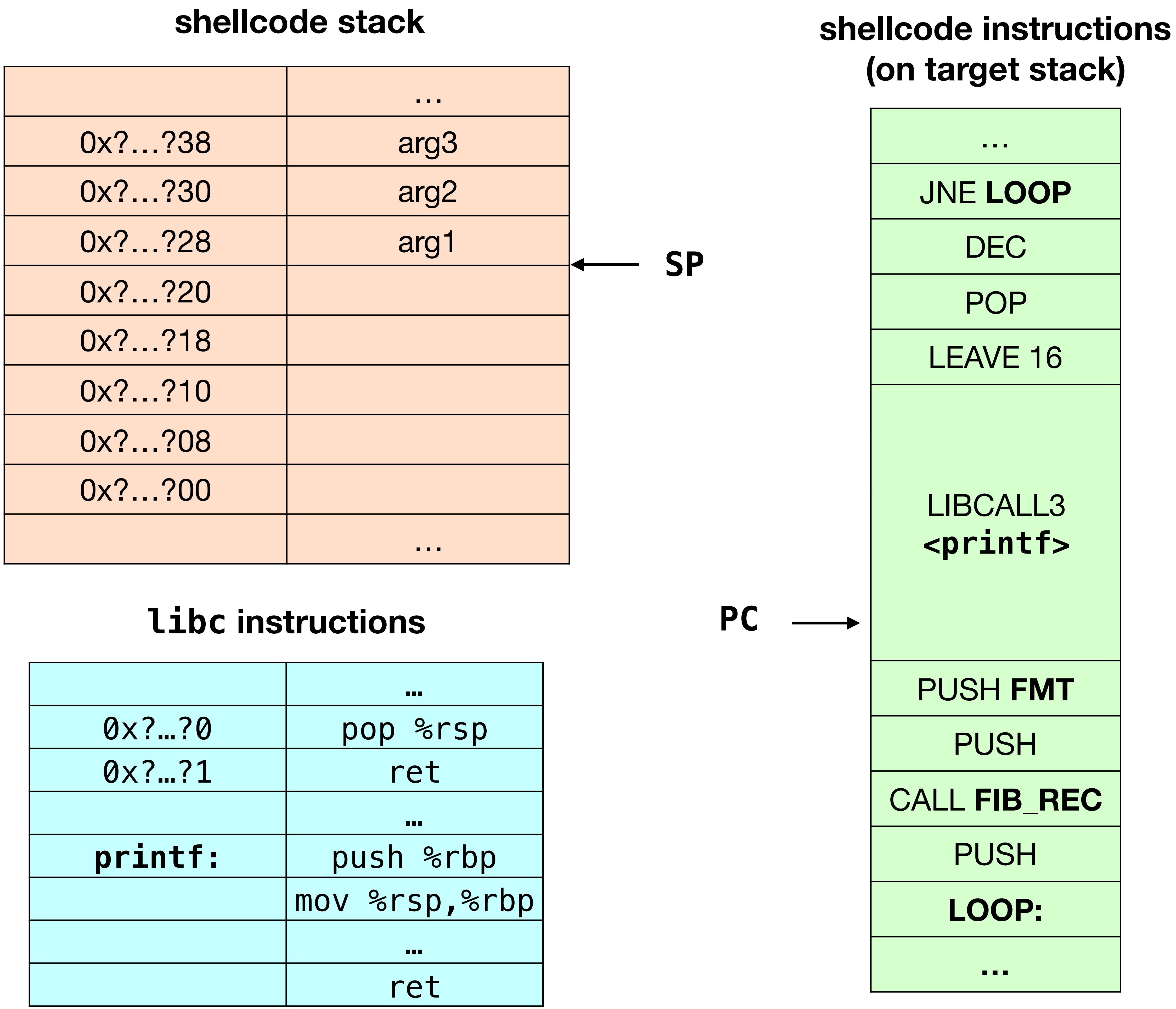


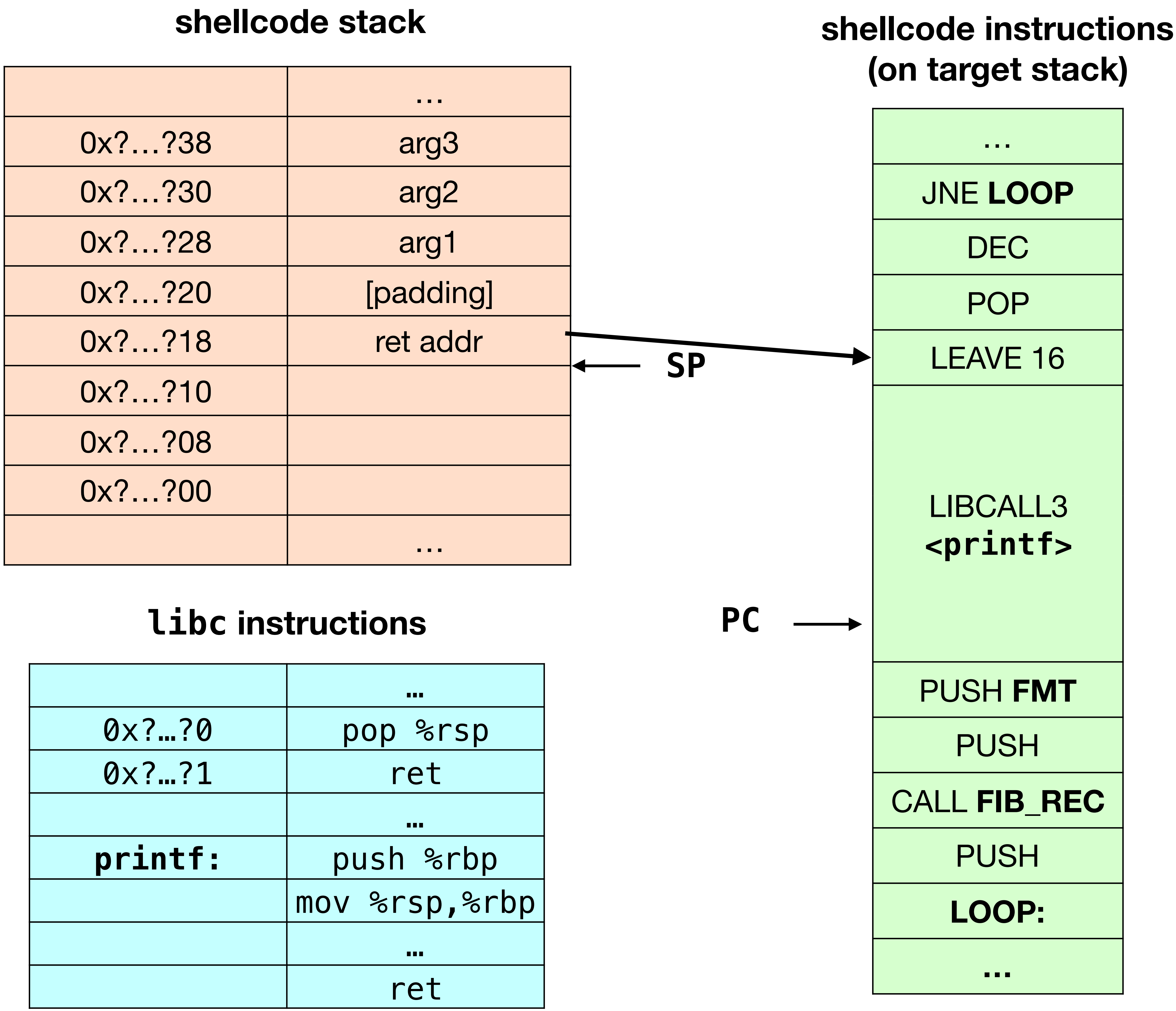
shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
SP
0x?...?20
0x?...?18
0x?...?10
shellcode return address
%rsp
0x?...?08
0x?...?00
printf's stack frame
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
(printf returns)
%rip


shellcode stack
...
0x?…?38
0x?…?30
0x?…?28
args
0x?…?20
0x?…?18
0x?…?10
shellcode return address
0x?…?08
0x?…?00
printf's stack frame
SP
%rsp
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret

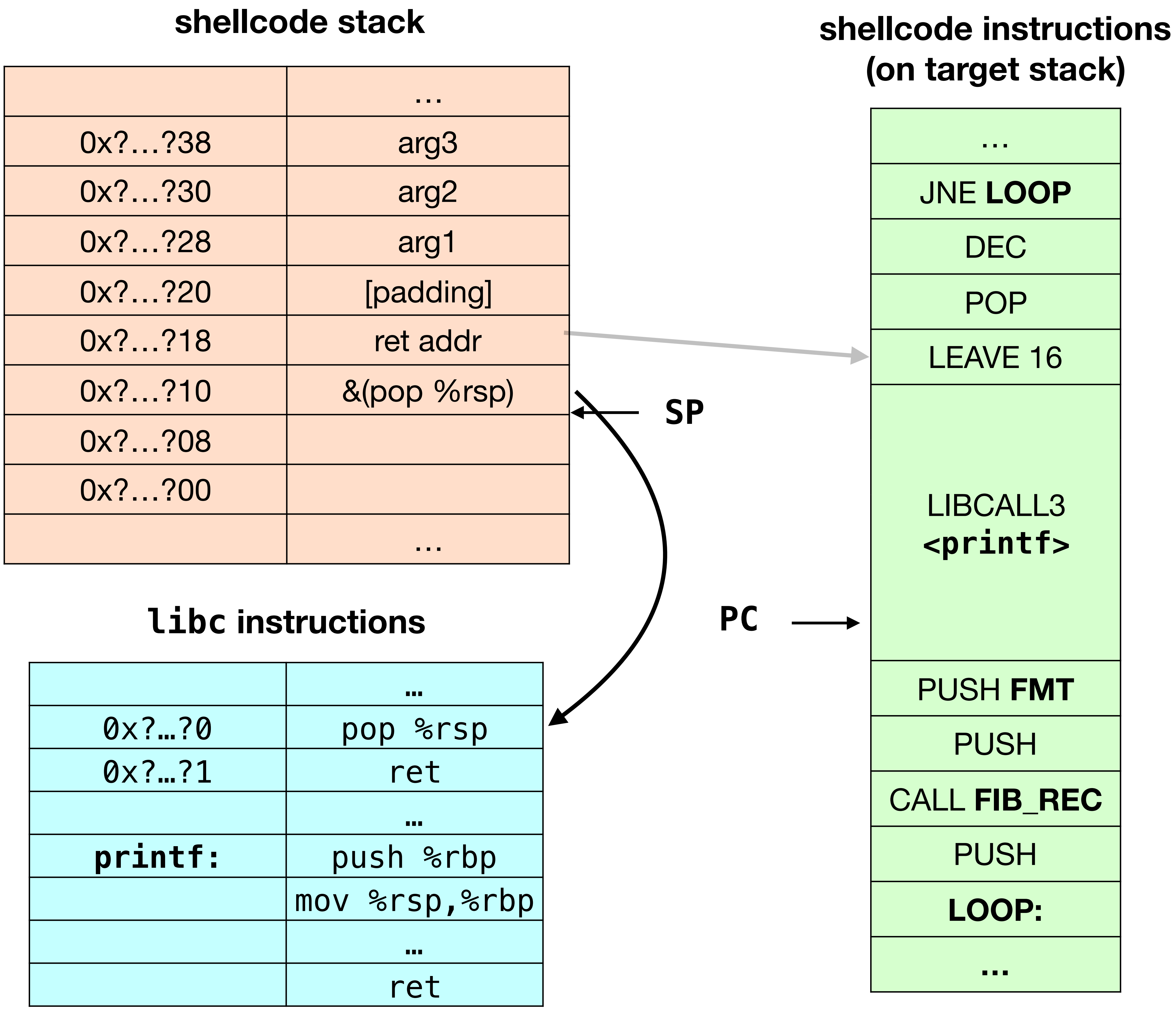

shellcode stack
...
0x?...?38
0x?...?30
0x?...?28
args
0x?...?20
0x?...?18
0x?...?10
shellcode return address
0x?...?08
0x?...?00
printf's stack frame
SP
PC
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
printf:
push %rbp
mov %rsp,%rbp
...
ret

Library Call Diagram
(in depth)

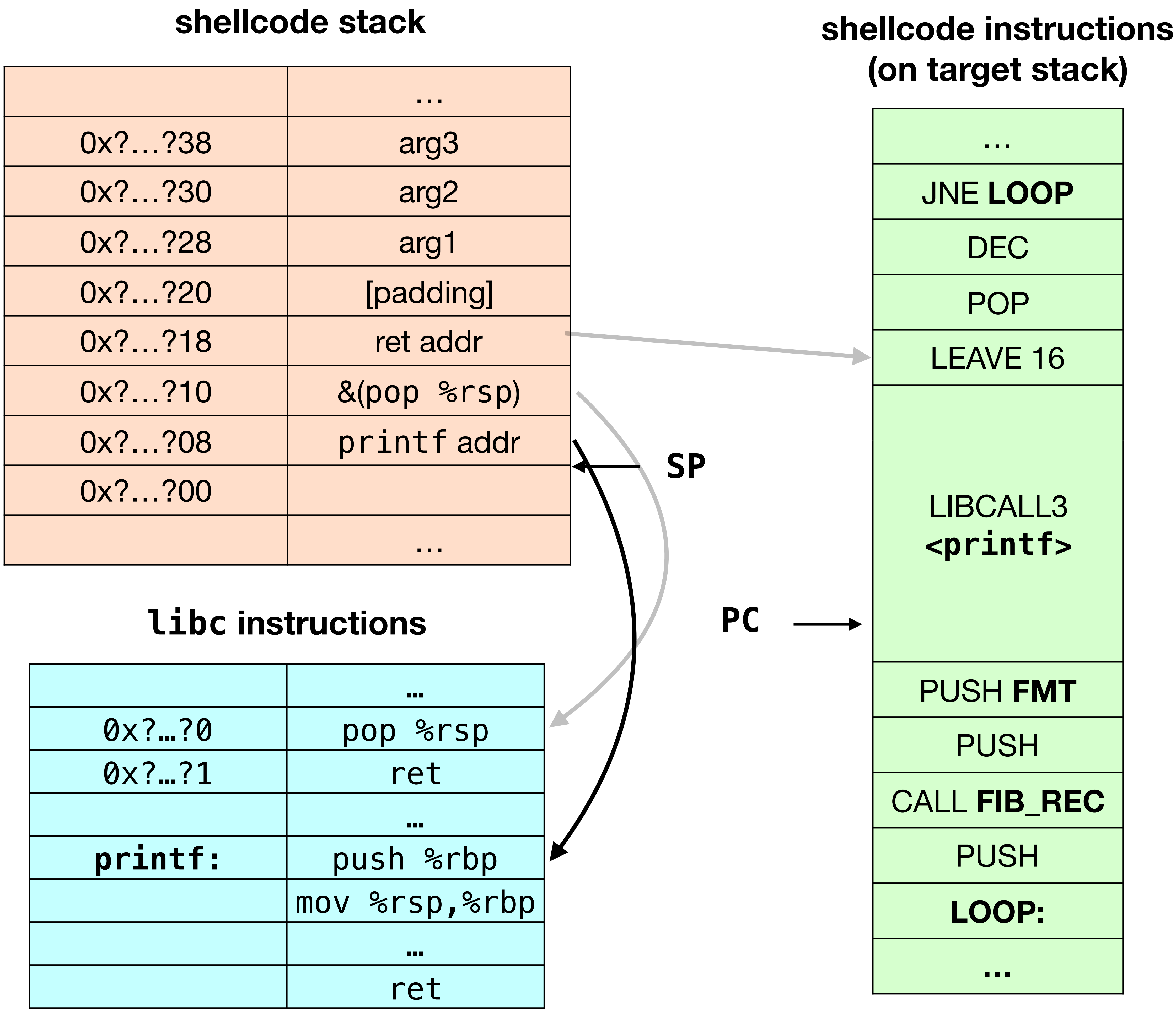

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
SP
0x?...?20
0x?...?18
0x?...?10
0x?...?08
0x?...?00
...
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PC
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

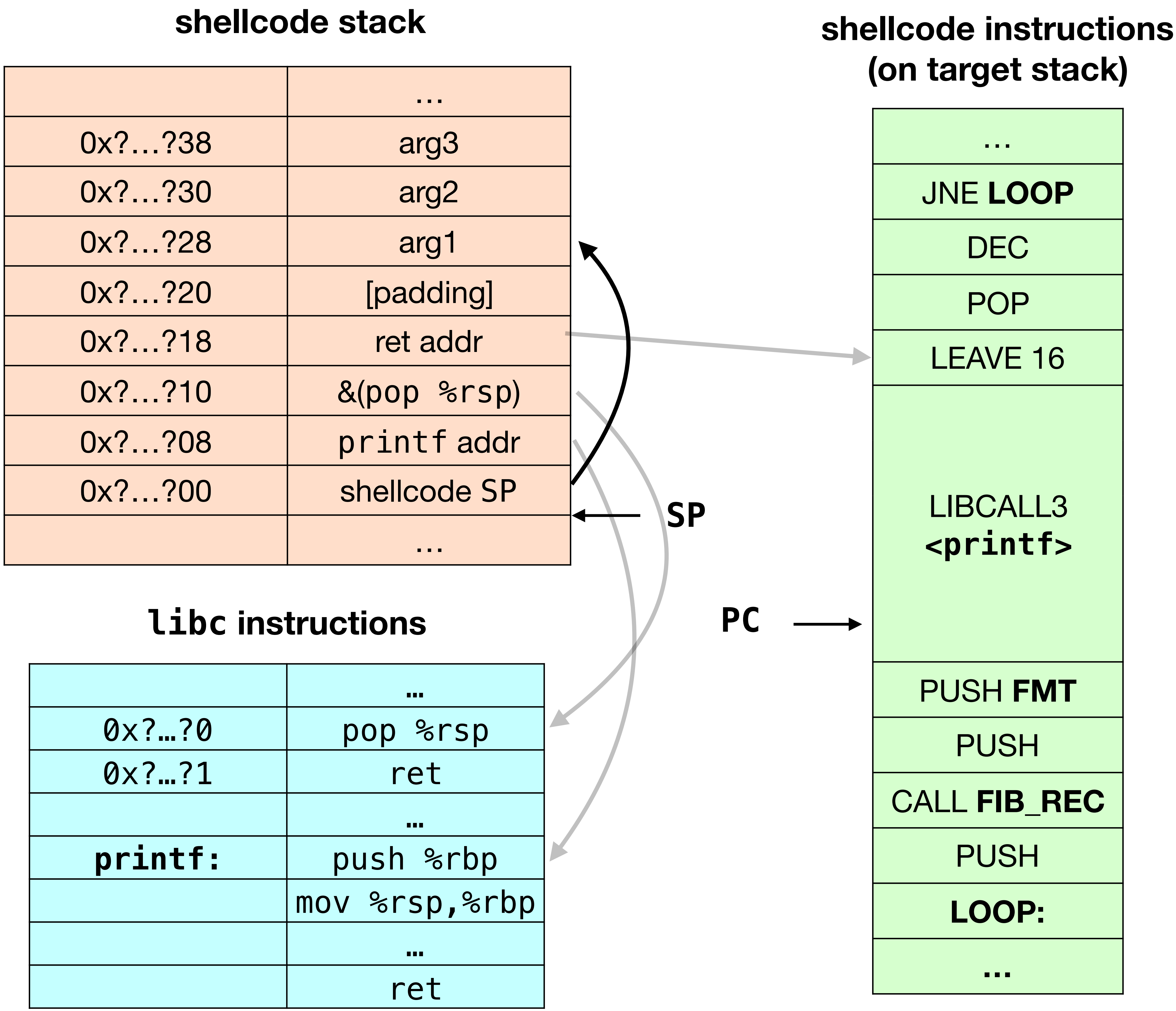


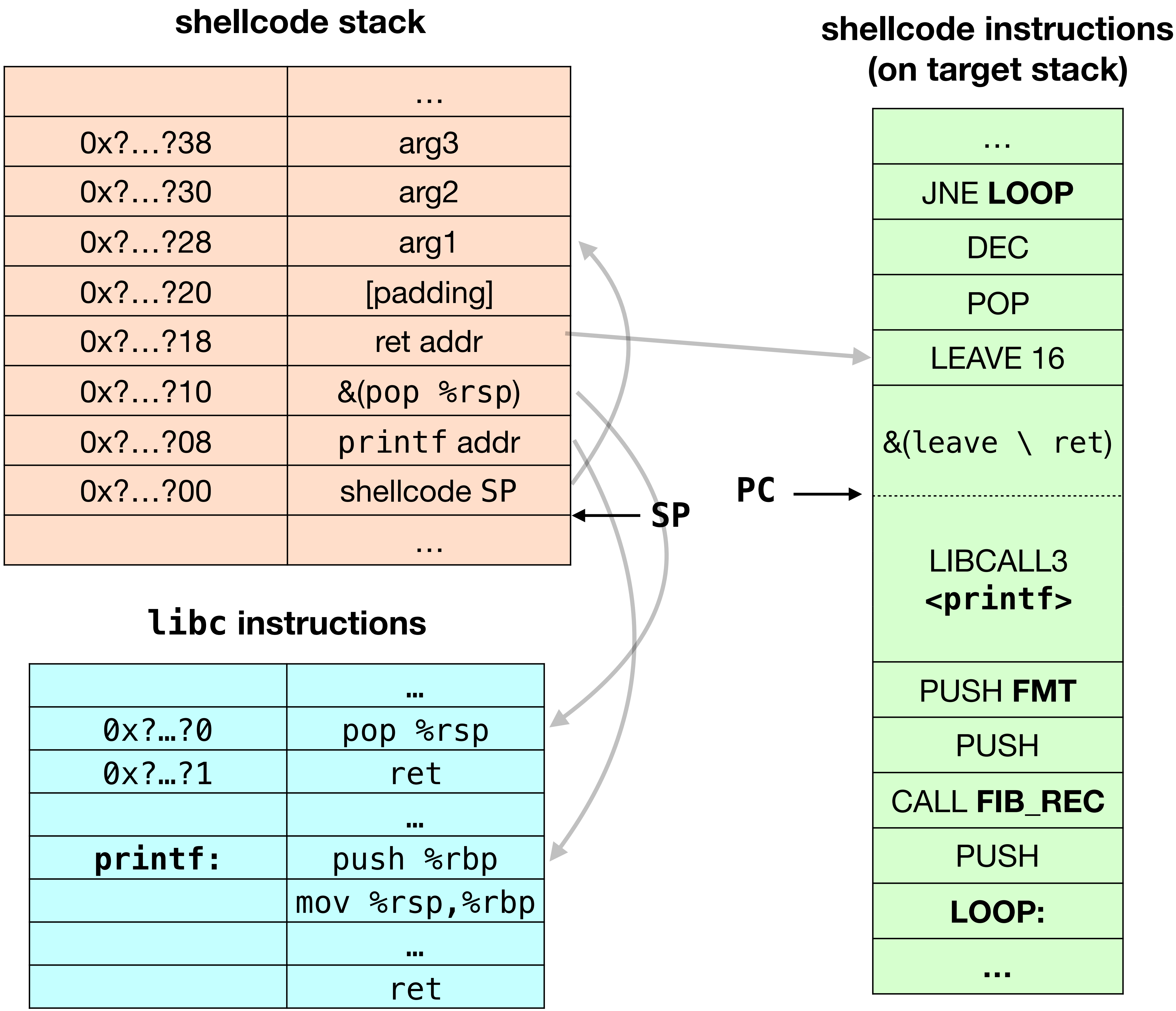
shellcode stack
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
0x?…?08
0x?…?00
SP
shellcode instructions
(on target stack)
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PC
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
libc instructions
0x?…?0
pop %rsp
0x?…?1
ret
printf:
push %rbp
mov %rsp,%rbp
ret



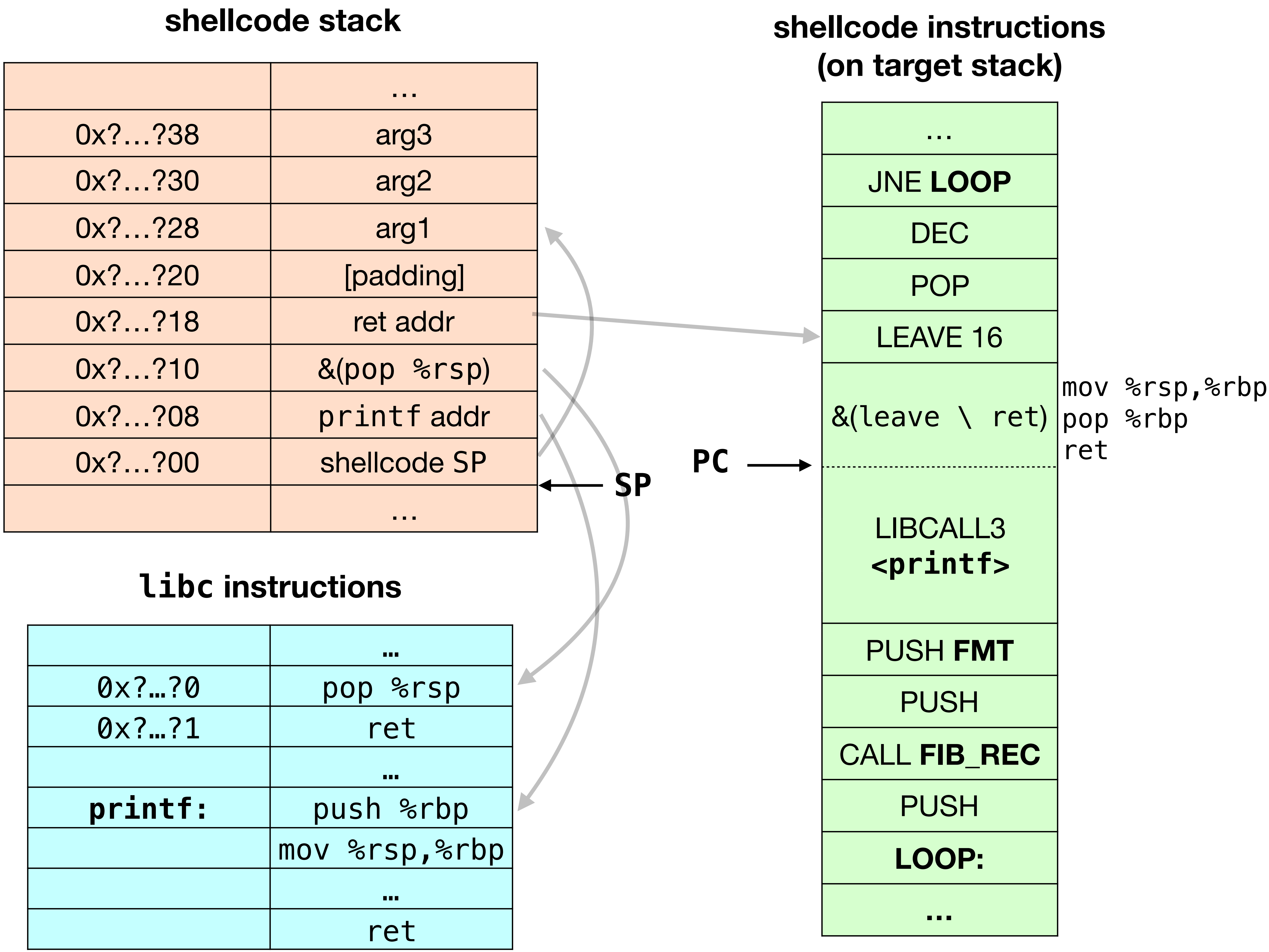
shellcode stack
...
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
0x?…?00
...
SP
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PC
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
…
printf:
push %rbp
mov %rsp,%rbp
…
ret

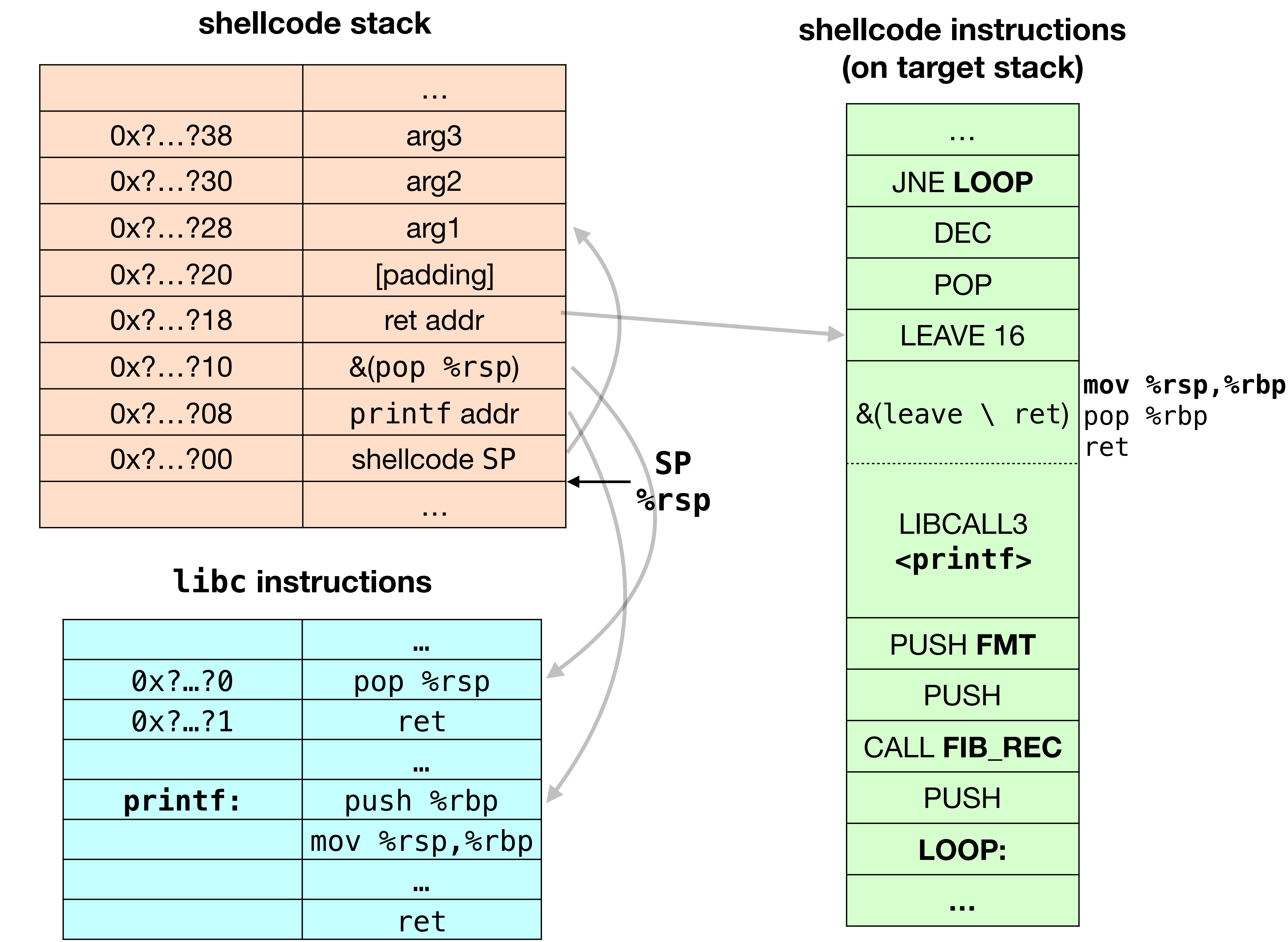

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
...
SP
PC
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

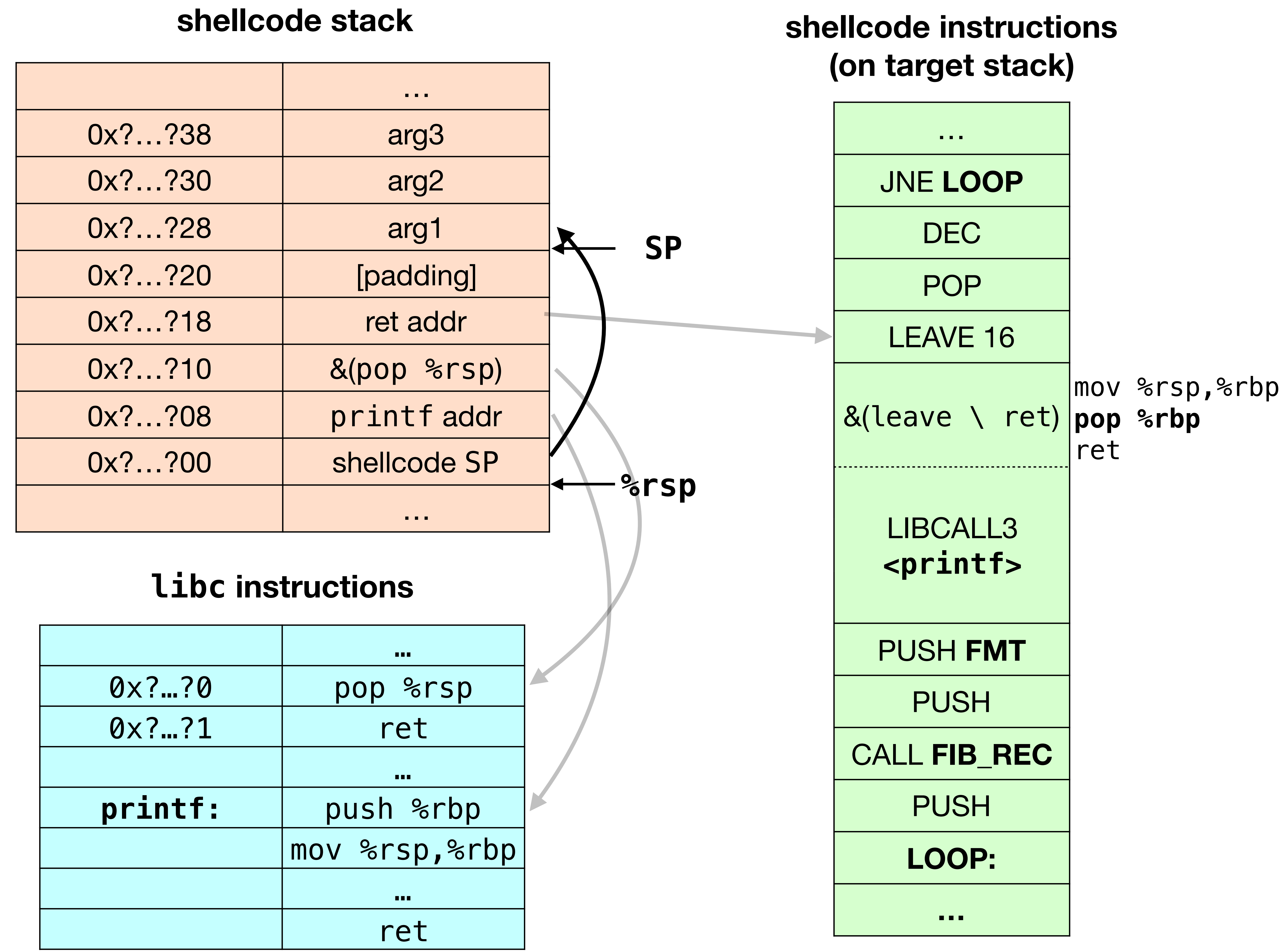

shellcode stack
...
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
printf addr
0x?…?00
shellcode SP
...
SP
PC
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
…
printf:
push %rbp
mov %rsp,%rbp
…
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

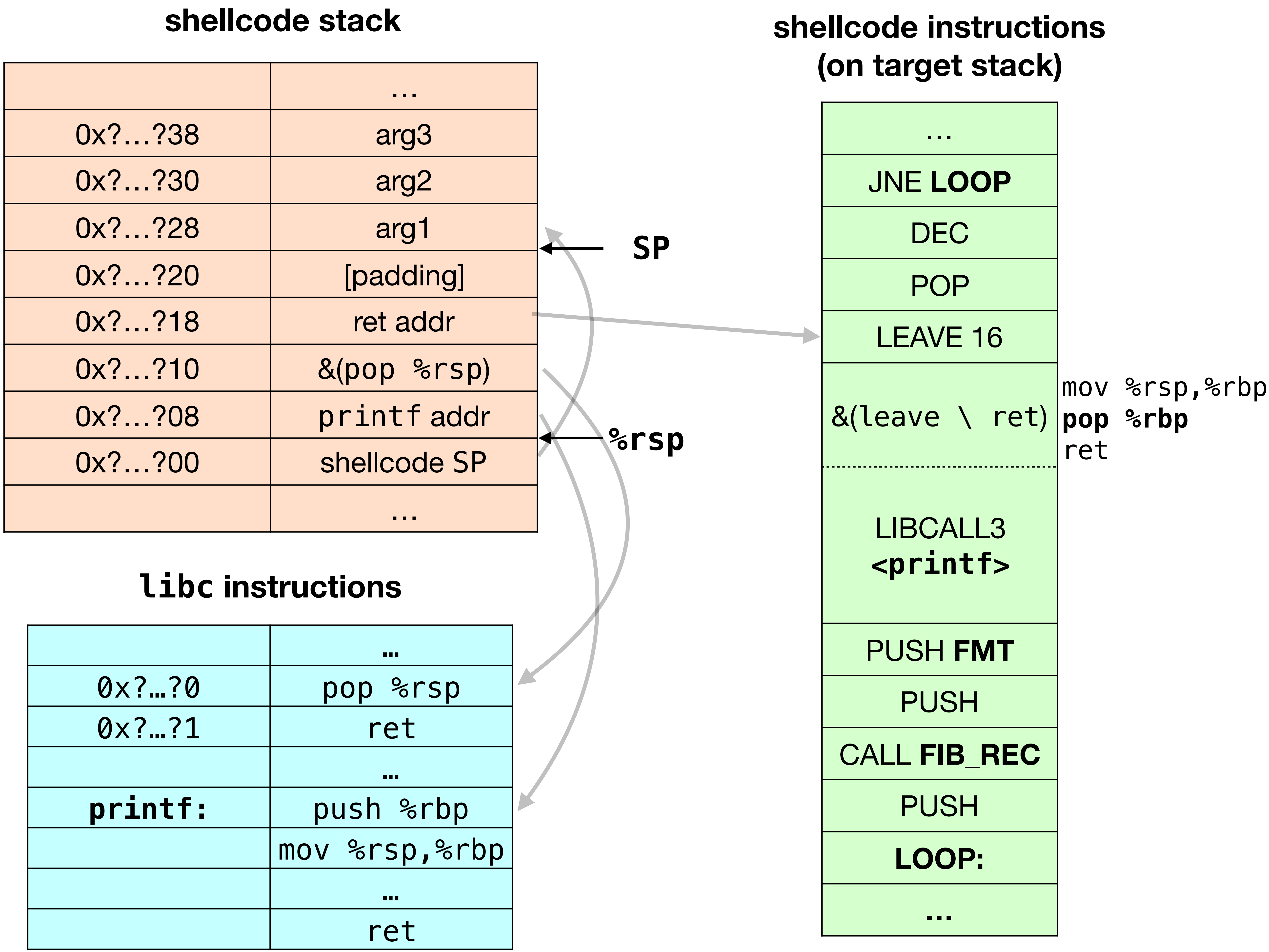


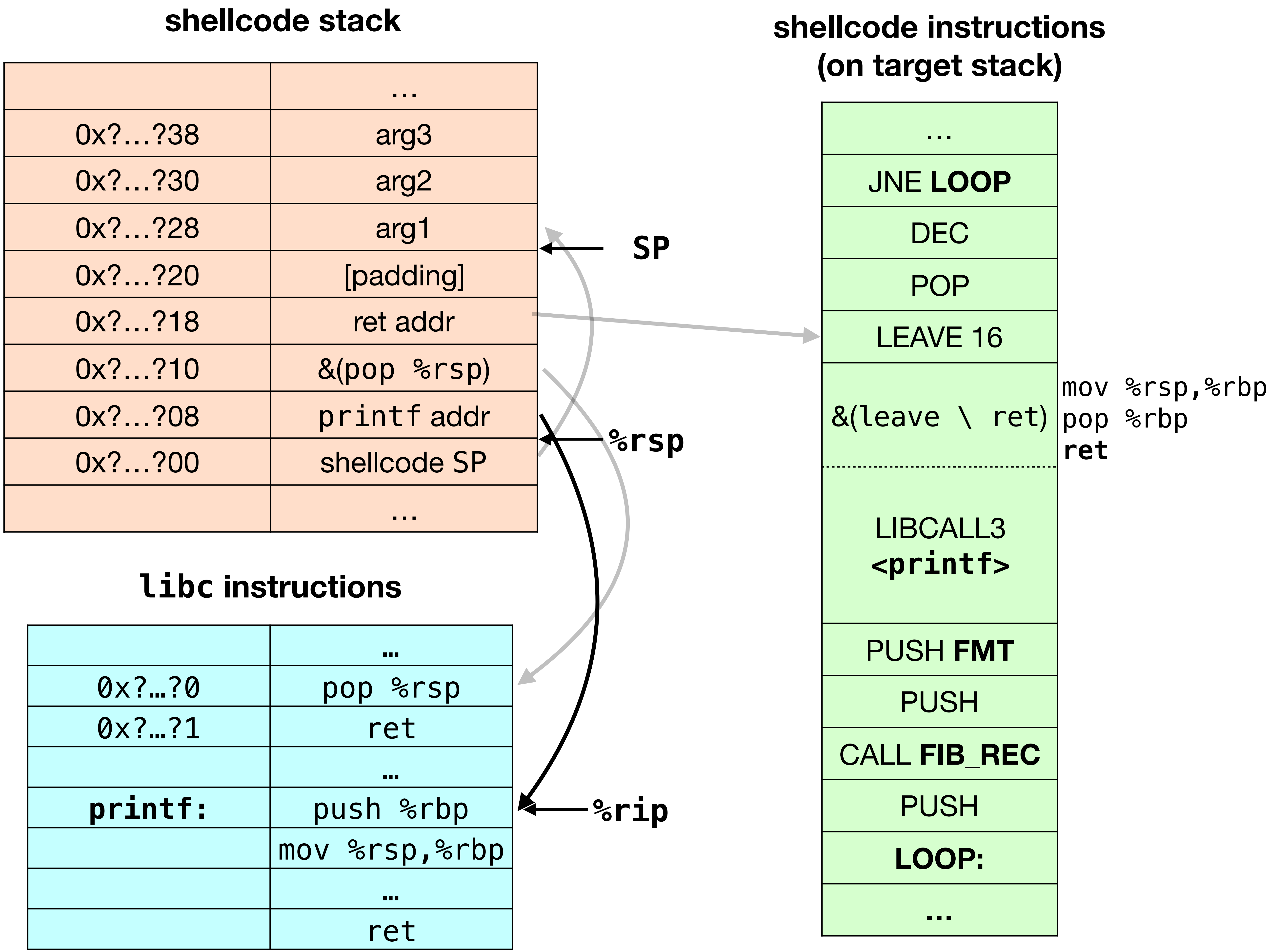
shellcode stack
...
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
printf addr
0x?…?00
shellcode SP
...
SP
PC
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
…
printf:
push %rbp
mov %rsp,%rbp
…
ret

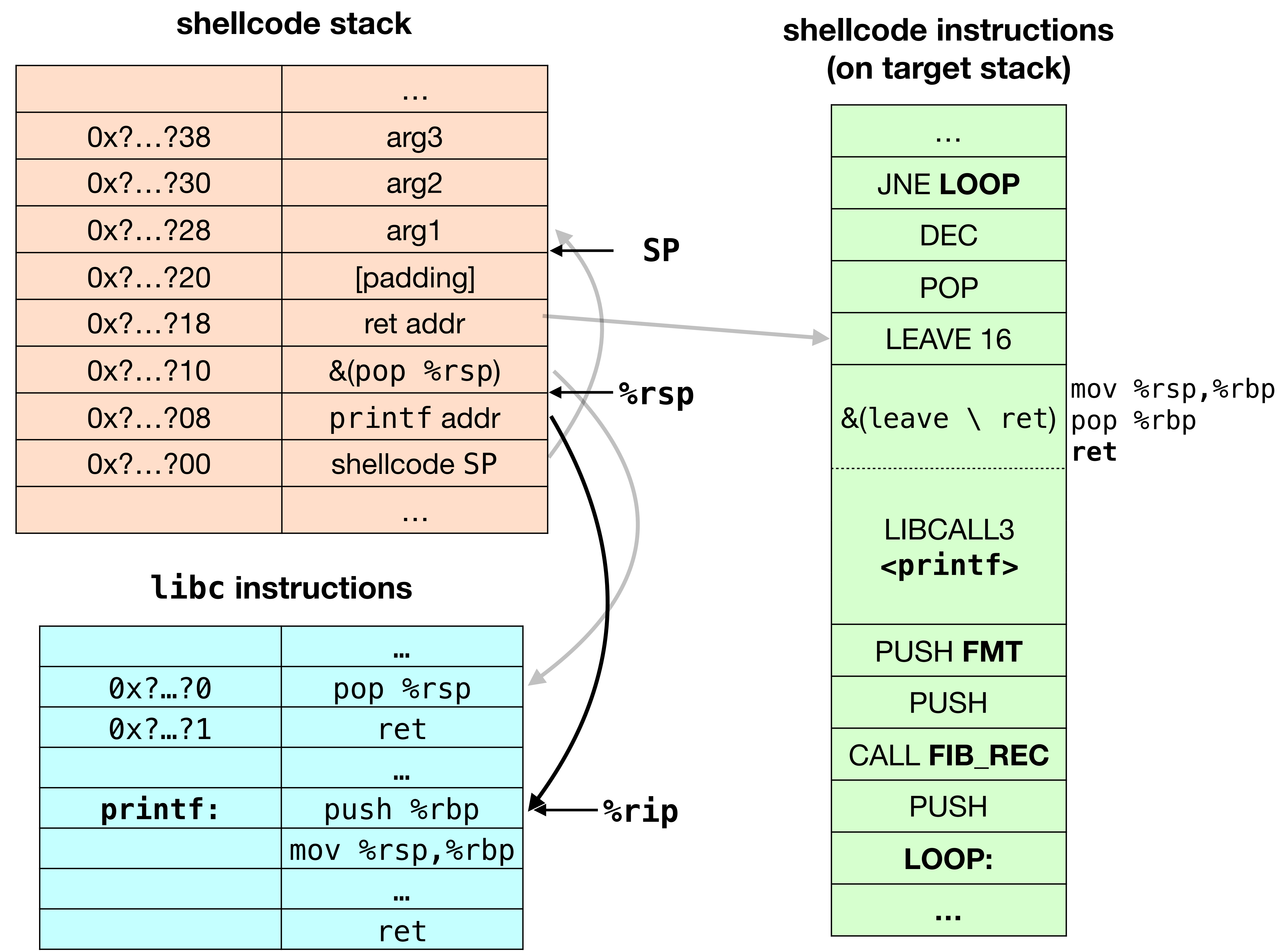

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
PC
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

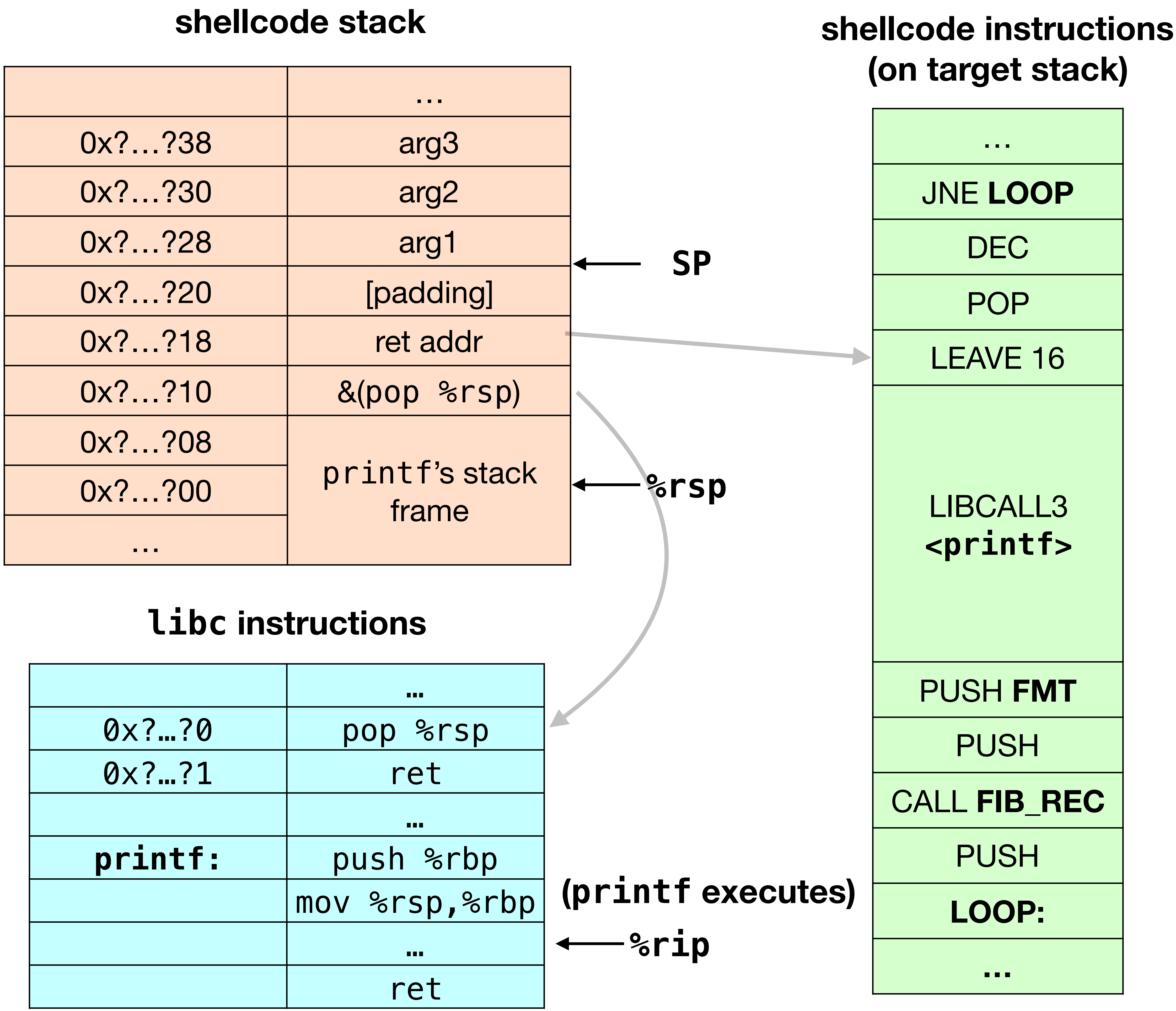

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
%rsp
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...


shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
%rsp
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

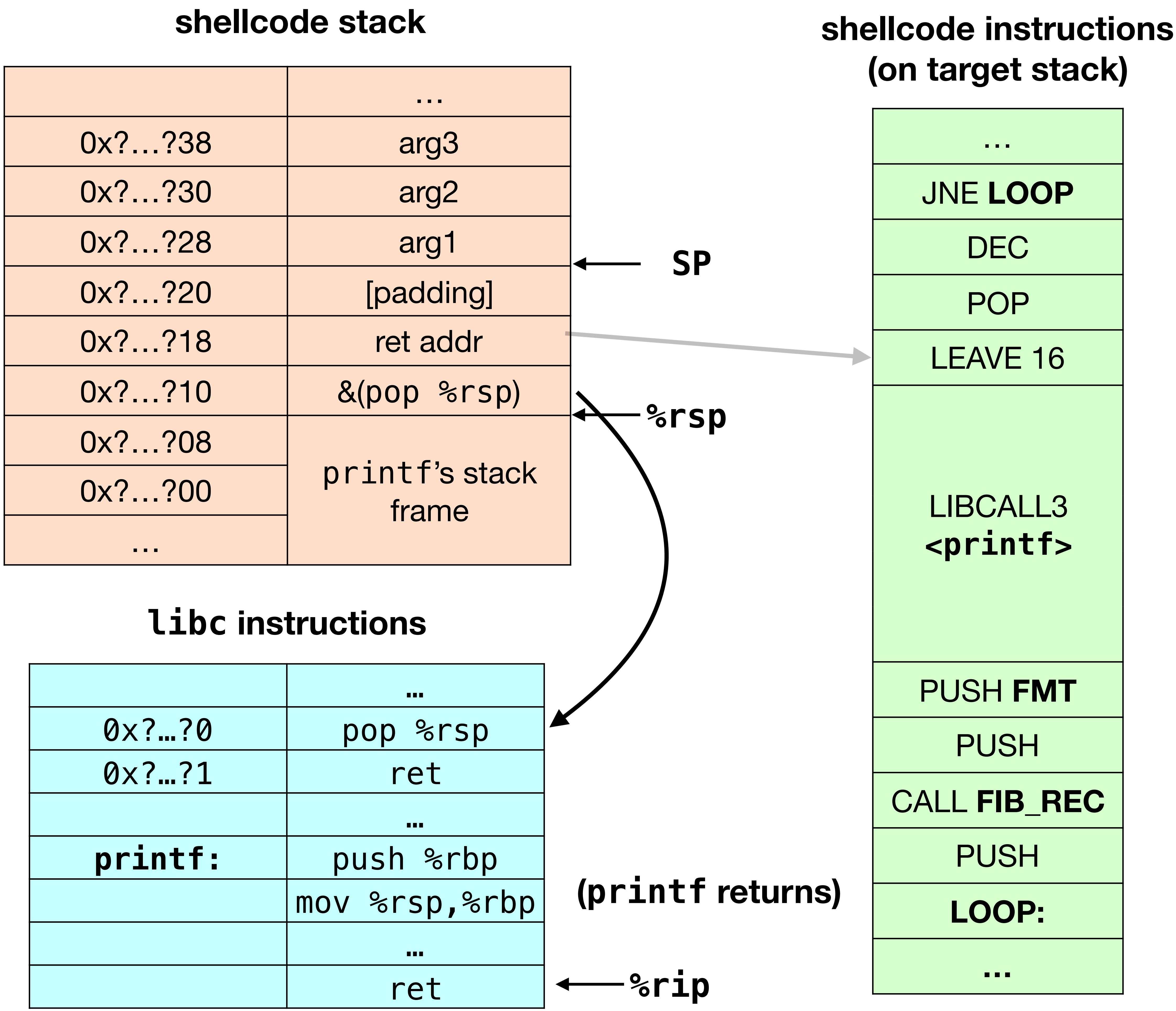

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
%rsp
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret

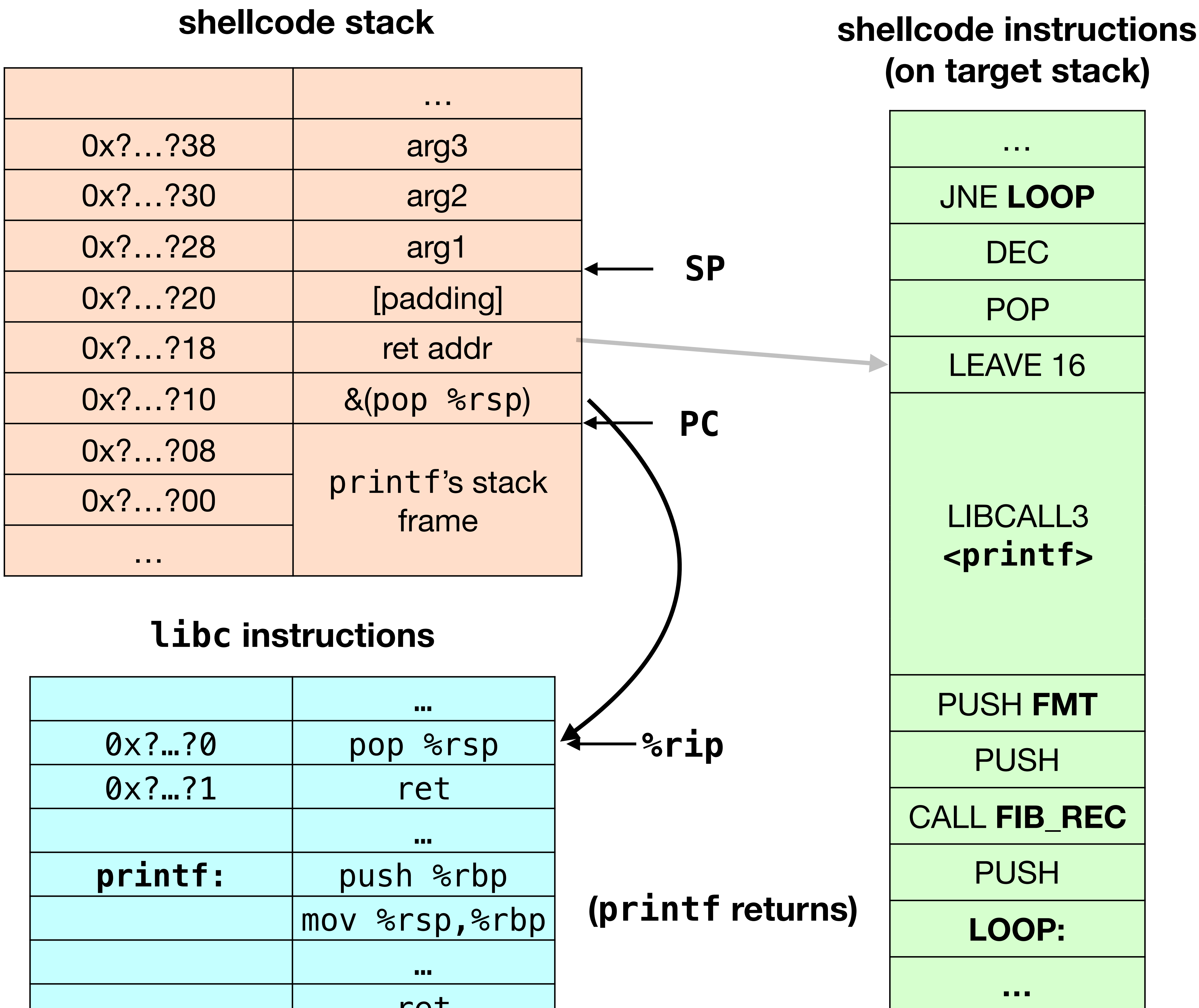

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
%rsp
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...


shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
printf addr
0x?...?00
shellcode SP
...
SP
%rsp
%rip
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
shellcode instructions
(on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
&(leave \ ret)
mov %rsp,%rbp
pop %rbp
ret
LIBCALL3
<printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...

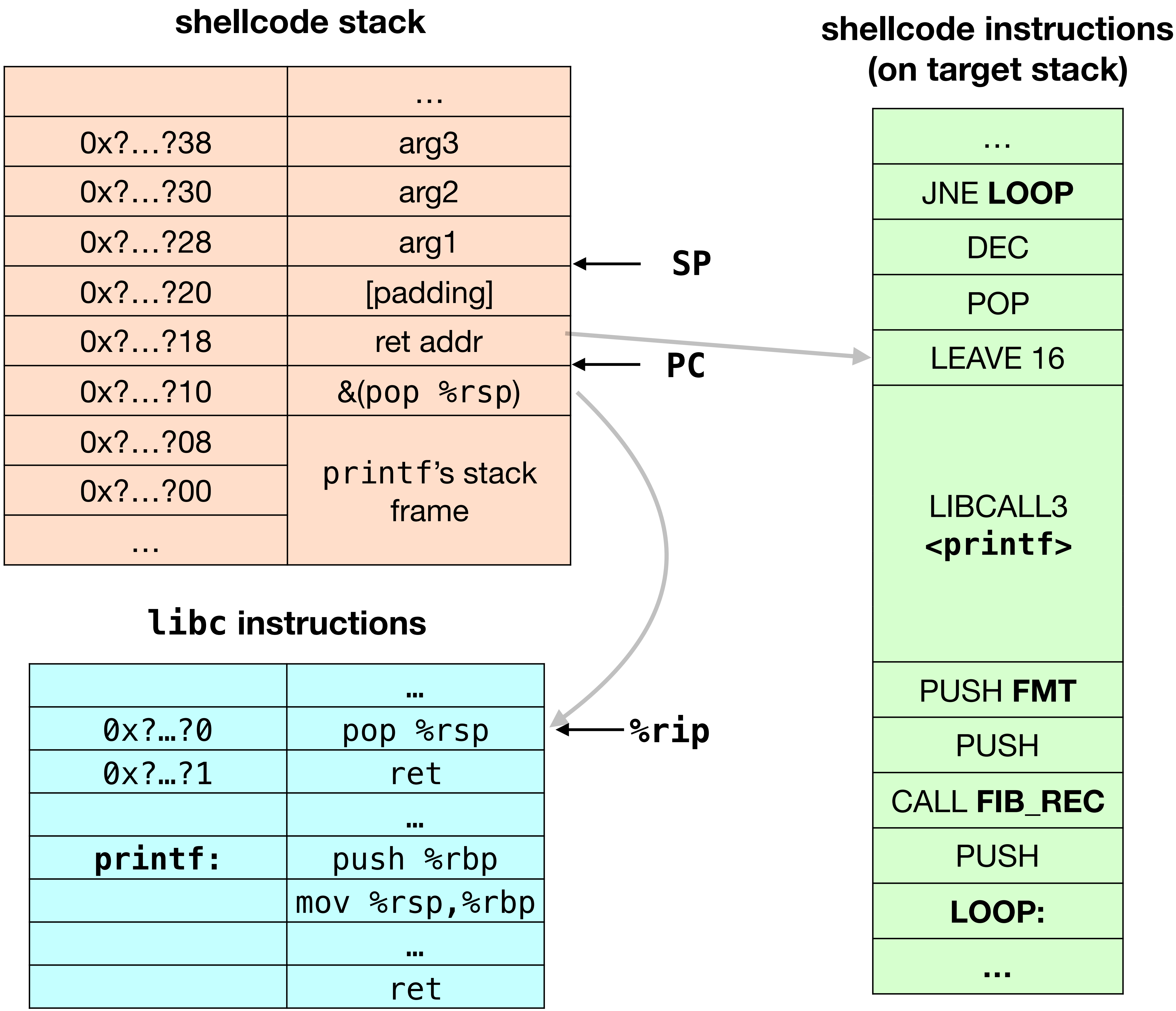


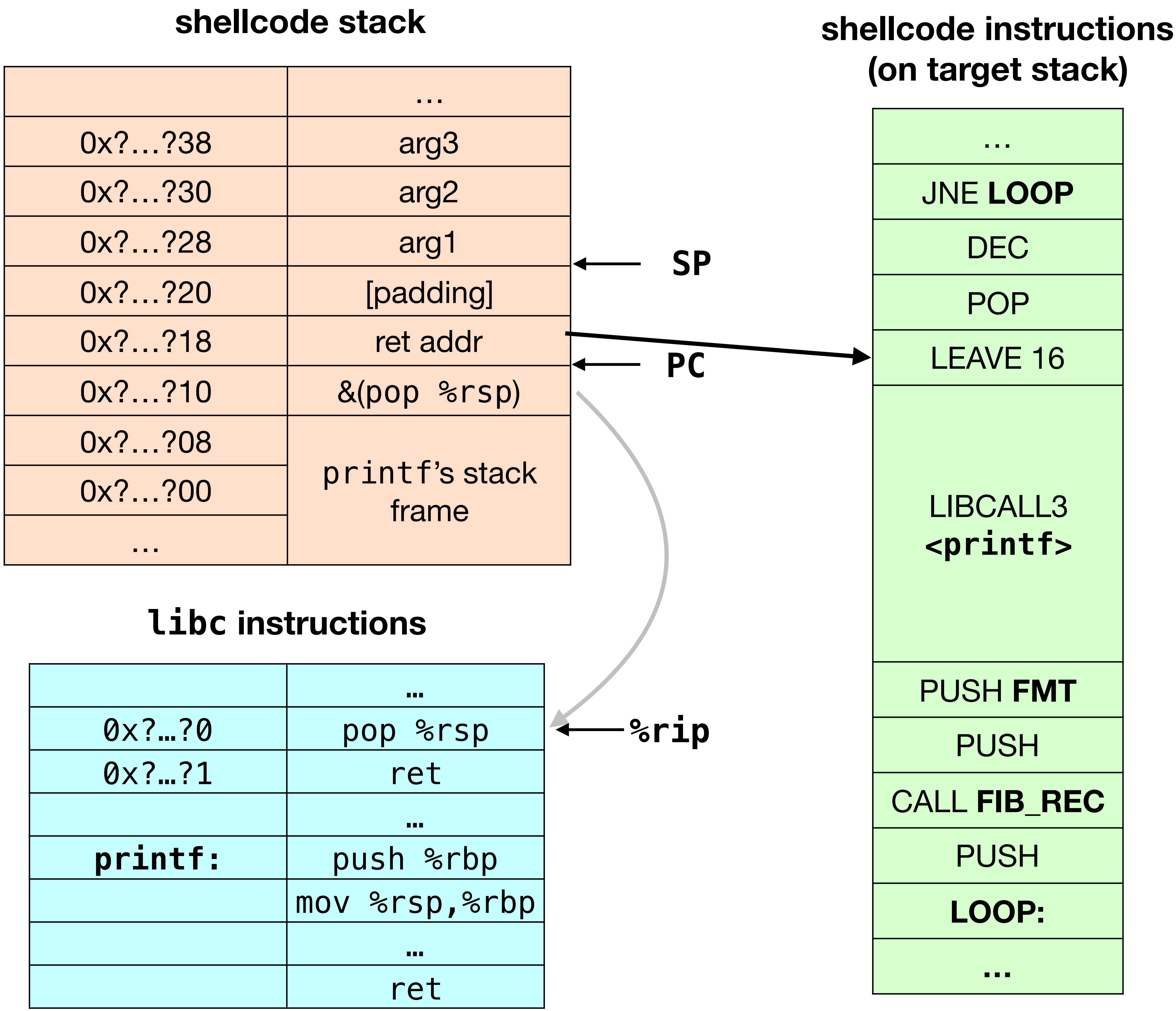
shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
SP
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
0x?...?00
printf's stack frame
%rsp
...
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
(printf executes)
%rip

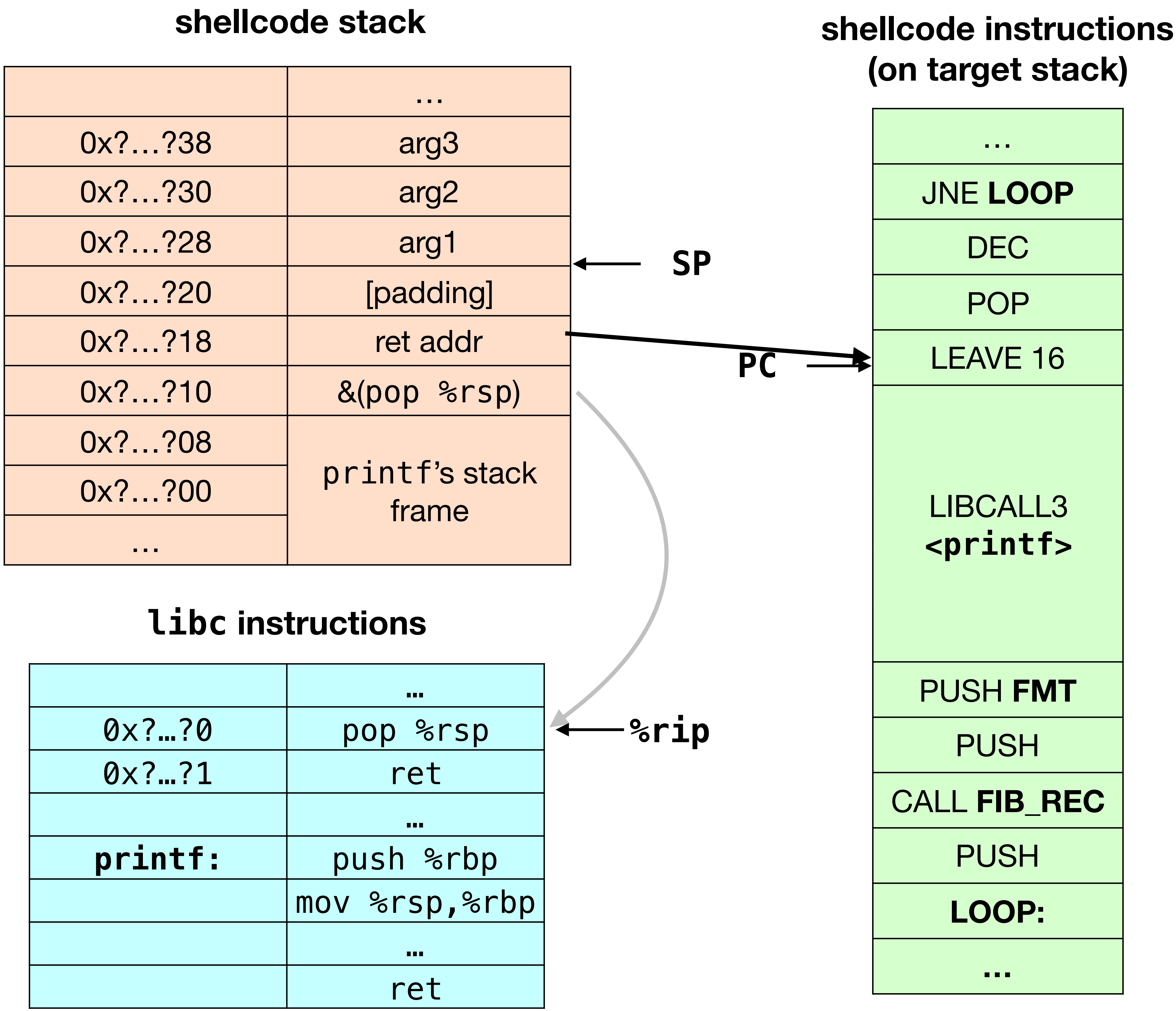

shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
SP
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
%rsp
0x?...?08
0x?...?00
printf's stack frame
...
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
(printf returns)

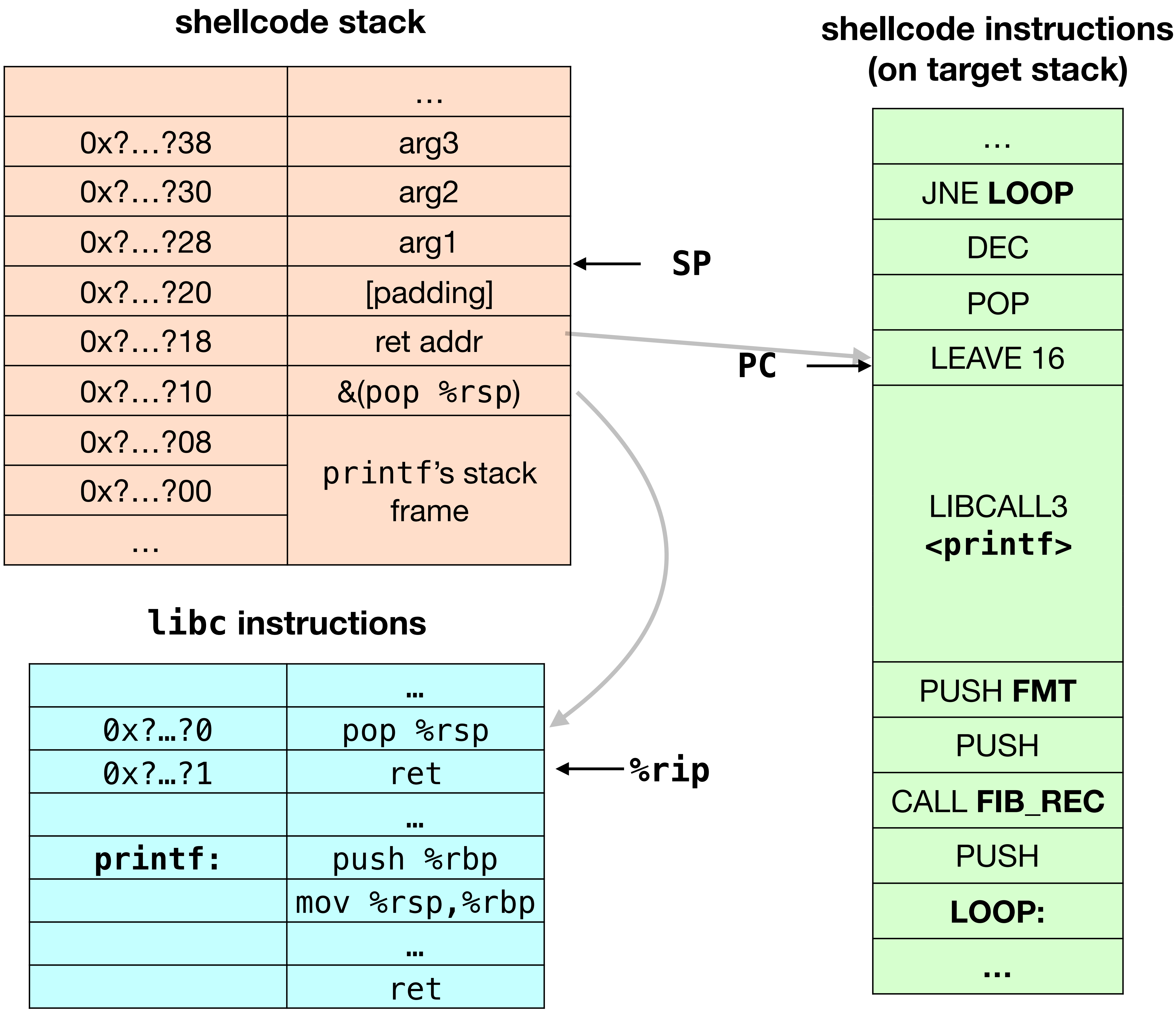


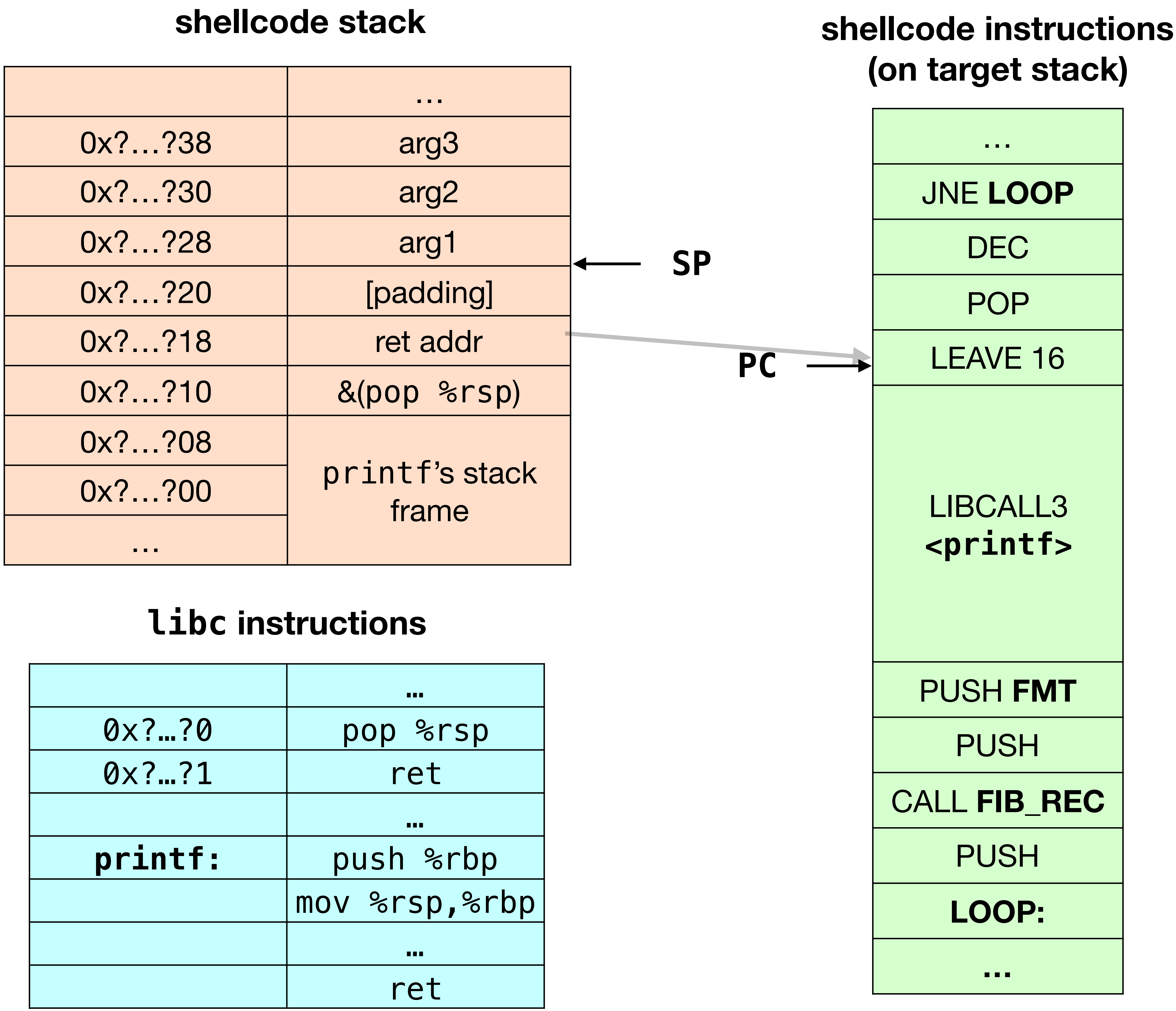
shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
0x?...?00
printf's stack frame
...
SP
PC
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
(printf returns)


shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
0x?...?00
printf's stack frame
...
SP
PC
libc instructions
...
0x?...?0
pop %rsp
0x?...?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret
%rip
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...


shellcode stack
…
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
0x?…?00
printf's stack frame
…
SP
PC
shellcode instructions (on target stack)
…
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
…
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
…
printf:
push %rbp
mov %rsp,%rbp
…
ret
%rip


shellcode stack
...
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
SP
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
0x?…?00
printf's stack frame
...
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
PC
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
…
printf:
push %rbp
mov %rsp,%rbp
…
ret
%rip


shellcode stack
…
0x?…?38
arg3
0x?…?30
arg2
0x?…?28
arg1
SP
0x?…?20
[padding]
0x?…?18
ret addr
0x?…?10
&(pop %rsp)
0x?…?08
0x?…?00
printf's stack frame
…
shellcode instructions (on target stack)
…
JNE LOOP
DEC
POP
PC
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
…
libc instructions
…
0x?…?0
pop %rsp
0x?…?1
ret
%rip
…
printf:
push %rbp
mov %rsp,%rbp
…
ret


shellcode stack
...
0x?...?38
arg3
0x?...?30
arg2
0x?...?28
arg1
0x?...?20
[padding]
0x?...?18
ret addr
0x?...?10
&(pop %rsp)
0x?...?08
0x?...?00
...
printf's stack frame
SP
PC
shellcode instructions (on target stack)
...
JNE LOOP
DEC
POP
LEAVE 16
LIBCALL3 <printf>
PUSH FMT
PUSH
CALL FIB_REC
PUSH
LOOP:
...
libc instructions
...
0x?…?0
pop %rsp
0x?…?1
ret
...
printf:
push %rbp
mov %rsp,%rbp
...
ret

Size Problem

- **Problem**: Only short exploits can fit entirely on the target's stack
- Possible solutions
  - Exec the shell (or a malicious program, etc.)
    - Defeats the purpose of Turing-complete ROP
  - ???

Two-Stage Exploit

- **Stage 1: the Exploit** (on target stack)
  - Exploits stack buffer overflow
  - Execute gadget sequence that maps input into memory (at a fixed address)
- **Stage 2: the Payload** (at fixed address)
  - Contains the main shellcode program
  - No limit to the size of this

Future Work

- Position-independent ROP code
  - Add relative jump and call instructions to ROPC-IR
- Reduce the code size of library calls
  - Common code can be factored out

Demos

Exploit 1: Fibonacci Sequence Generator

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Exploit 2:
Shell Prompt

Exploit 3:
This Meta-Exploit is a Binomial Coefficient Generator

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Thanks to
Travis Goodspeed

LinkedIn: in/nmosier

GitHub: nmosier

Email: nmosier@middlebury.edu

https://github.com/nmosier/rop-tools