ECE 6504

Embedded and Controls Systems Security: Stack Smashing III

Reading:

T: *The Geometry of Innocent Flesh on the Bone*

N: *Code Injection Attacks on Harvard-Architecture Devices* and *Can DREs Provide Long-Lasting Security?*

VirginiaTech
Invent the Future

“If at first you don't succeed, destroy all evidence that you tried.”

–Steven Wright

preventing code injection attacks:
lessons from history

older exploit code:

```
execve("/bin/sh", NULL, NULL);
```

kernel executes program → a ‘system’ call

```
#include <stdlib.h>

int main() {
  execve("/bin/sh", NULL, NULL);
}
```

```
#include <stdlib.h>

int main() {
  execve("/bin/sh", NULL, NULL);
}
```

```
0x00000000004003d4 <main+0>:    push   %rbp
0x00000000004003d5 <main+1>:    mov    %rsp,%rbp
0x00000000004003d8 <main+4>:    mov    $0x0,%rdx
0x00000000004003dd <main+9>:    mov    $0x0,%rsi
0x00000000004003e2 <main+14>:   mov    $0x46c610,%rdi
0x00000000004003e7 <main+19>:   callq  0x40ad30 <execve>
0x00000000004003ec <main+24>:   leaveq
0x00000000004003ed <main+25>:   retq
```

```
#include <stdlib.h>

int main() {
  execve("/bin/sh", NULL, NULL);
}
```

arguments put in registers

```
0x00000000004003d4 <main+0>:    push   %rbp
0x00000000004003d5 <main+1>:    mov    %rsp,%rbp
0x00000000004003d8 <main+4>:    mov    $0x0,%rdx
0x00000000004003dd <main+9>:    mov    $0x0,%rsi
0x00000000004003e2 <main+14>:   mov    $0x46c610,%rdi
0x00000000004003e7 <main+19>:   callq  0x40ad30 <execve>
0x00000000004003ec <main+24>:   leaveq
0x00000000004003ed <main+25>:   retq
```

```
#include <stdlib.h>

int main() {
  execve("/bin/sh", NULL, NULL);
}
```

```
0x00000000004003d4 <main+0>:    push   %rbp
0x00000000004003d5 <main+1>:    mov    %rsp,%rbp
0x00000000004003d8 <main+4>:    mov    $0x0,%rdx
0x00000000004003dd <main+9>:    mov    $0x0,%rsi
0x00000000004003e2 <main+14>:   mov    $0x46c610,%rdi
0x00000000004003e7 <main+19>:   callq  0x40ad30 <execve>
0x00000000004003ec <main+24>:   leaveq
0x00000000004003ed <main+25>:   retq
```

arguments put in registers

call function

system calls (`execve`):

pass arguments via registers

(most OSs; e.g. Linux and Windows)

system calls (**`execve`**):

pass arguments via registers

(most OSs; e.g. Linux and Windows)

function calls (**`system`**):

pass arguments via stack

system calls (**`execve`**):

pass arguments via registers

(most OSs; e.g. Linux and Windows)

function calls (**`system`**):

pass arguments via stack

Q: assuming W ^ X, why must shellcode use **`system`** instead of **`execve`**

Q: assuming W ^ X, why must shellcode use **`system`** instead of **`execve`**

A: because W ^ X won't allow execution of stack

Q: assuming W ^ X, why must shellcode use **`system`** instead of **`execve`**

A: because W ^ X won't allow execution of stack

no way to get arguments into registers

Q: assuming W ^ X, why must shellcode use **`system`** instead of **`execve`**

A: because W ^ X won't allow execution of stack

no way to get arguments into registers

which **`execve`** requires

attacker: fine just use `system`

it's in `libc`

attacker: fine just use **`system`**

it's in `libc`

attacker: fine just use **`system`**

```
system("/bin/sh")
```

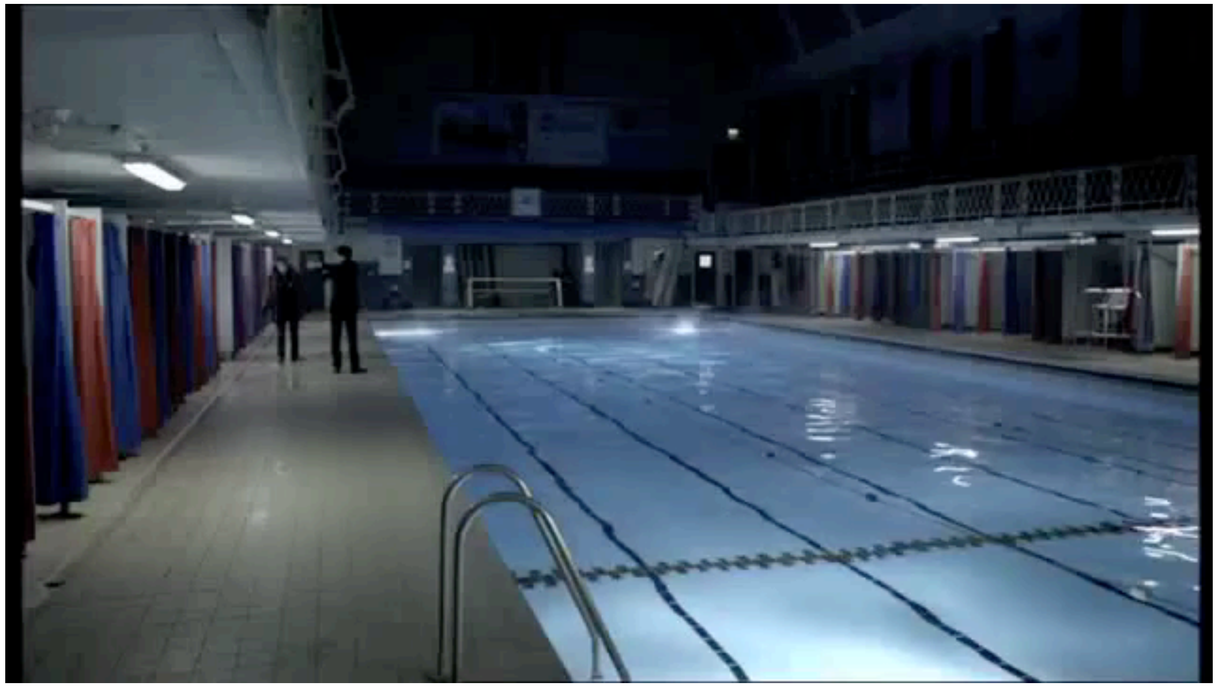
attacker:

replace ‘diploma’ with **`execve`**

attacker:

replace ‘diploma’ with **`execve`**

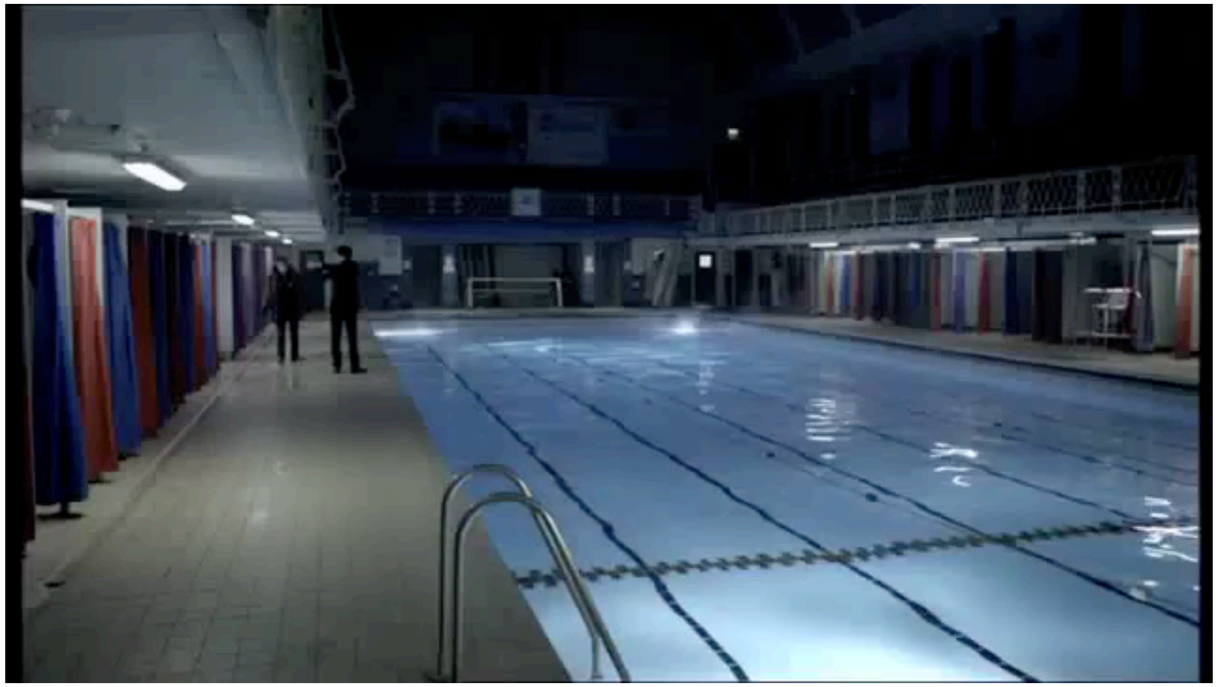
attacker: fine just use `system`



defender: let's get rid of that

excise `system` when not needed

attacker: fine just use `system`



defender: let's get rid of that

excise `system` when not needed

defences:

1. excise `system` when not needed
2. require all function calls to use registers for passing

defences:

1. excise `system` when not needed
2. require all function calls to use registers for passing

i.e. operate like system calls

defences:

1. excise `system` when not needed
2. require all function calls to use registers for passing

i.e. operate like system calls

need extra registers: x86-64

attacker response?:

alas, no

attacker response?:

alas, no

attacker:

useful programs make system calls

can’t remove them

attacker:

useful programs make system calls

can’t remove them

attacker:

useful programs make system calls

goal: figure out how to use them

can’t remove them

attacker:

useful programs make system calls

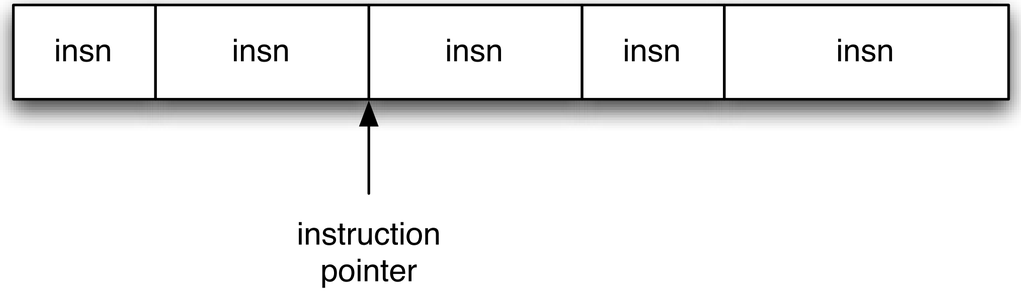
goal: figure out how to use them

return-**o**riented **p**rogramming

return-**o**riented **p**rogramming...

return-**o**riented **p**rogramming...

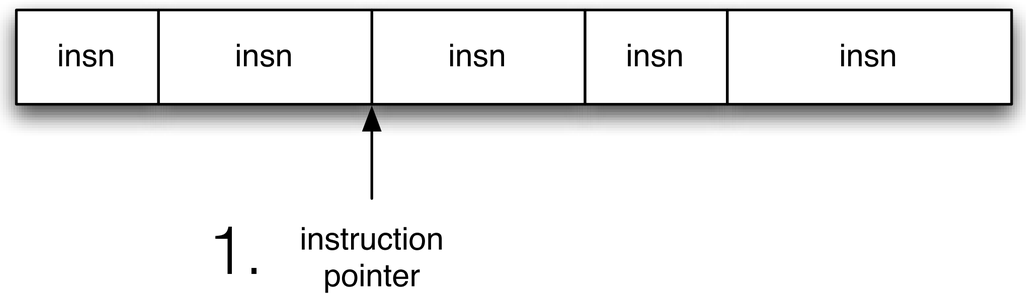
ordinary programming:



procedure:

1. IP points at instruction
2. instruction executed
3. IP incremented by length of instruction

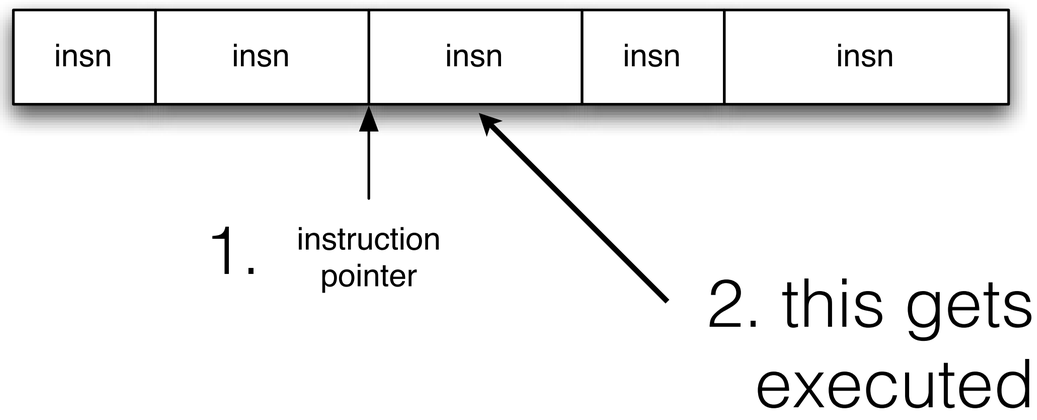
ordinary programming:



procedure:

1. IP points at instruction
2. instruction executed
3. IP incremented by length of instruction

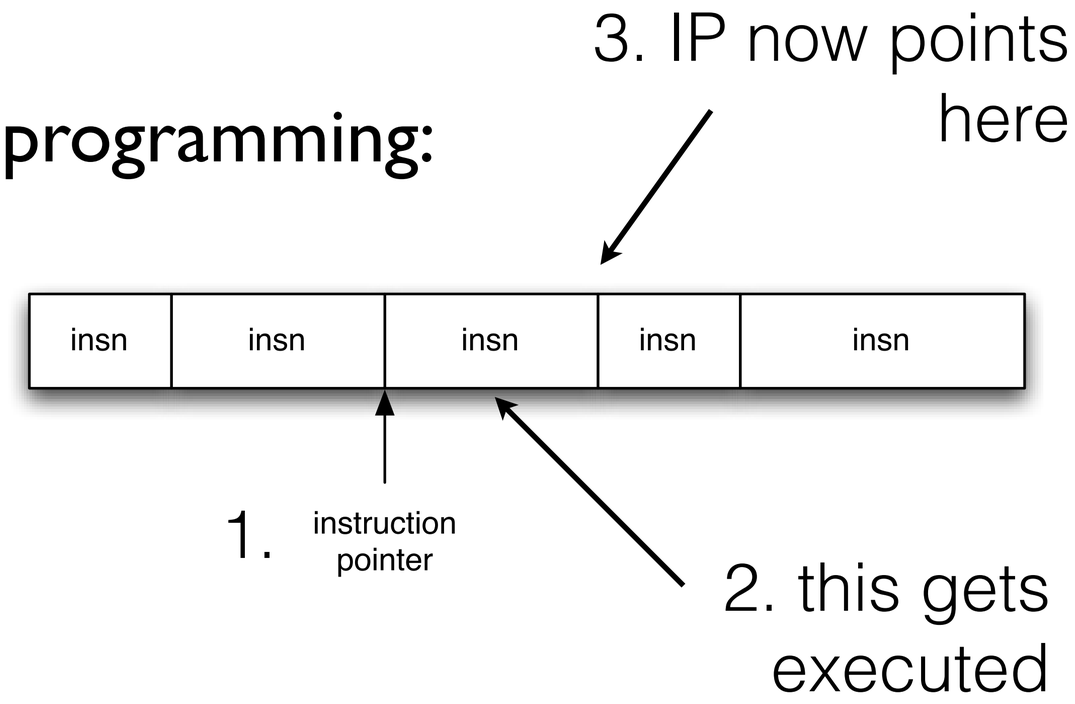
ordinary programming:



procedure:

1. IP points at instruction
2. instruction executed
3. IP incremented by length of instruction

ordinary programming:



procedure:

1. IP points at instruction
2. instruction executed
3. IP incremented by length of instruction

return-oriented programming (ROP)

setup stack to point to instructions that end in return

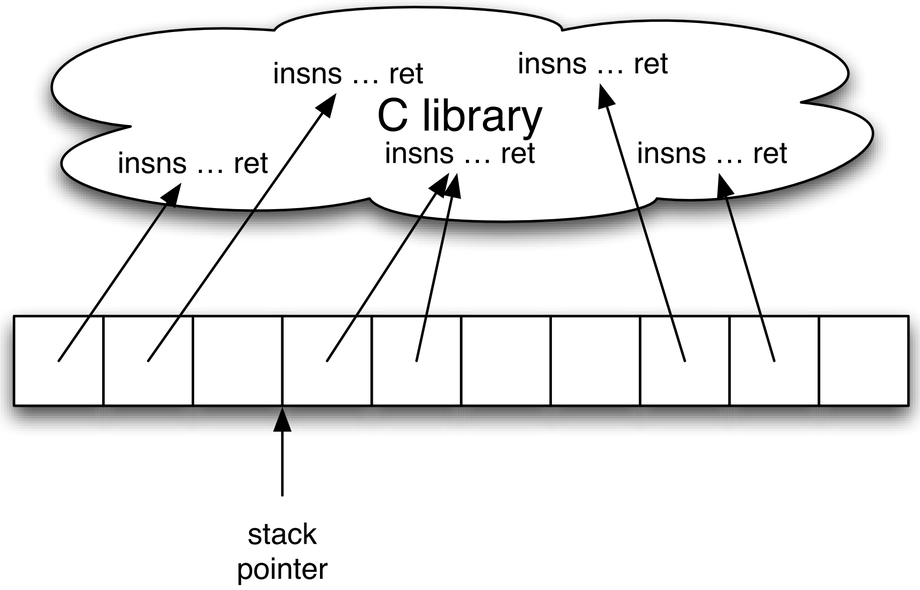
return-oriented programming (ROP)

setup stack to point to instructions that end in return

SP replaces IP
(we ‘execute’ on the stack)

return-oriented programming (ROP)

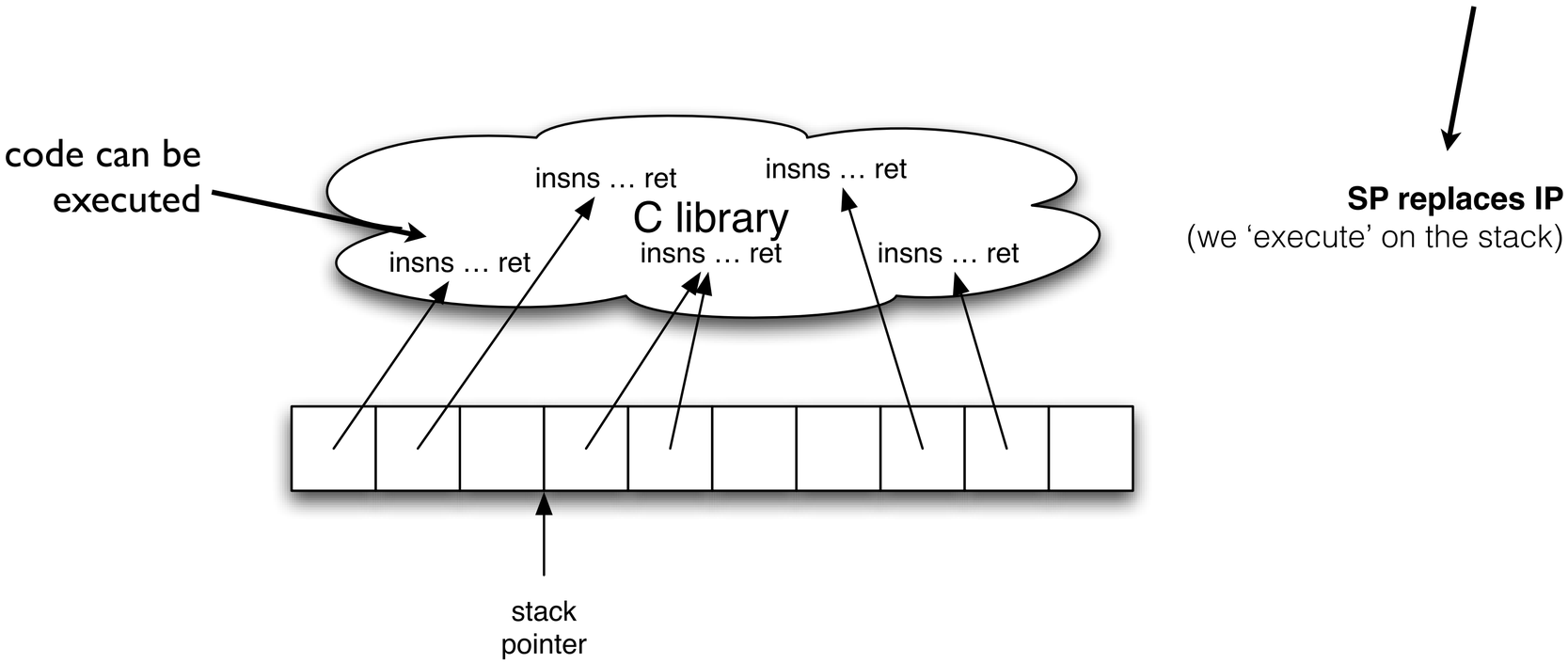
setup stack to point to instructions that end in return



SP replaces IP
(we 'execute' on the stack)

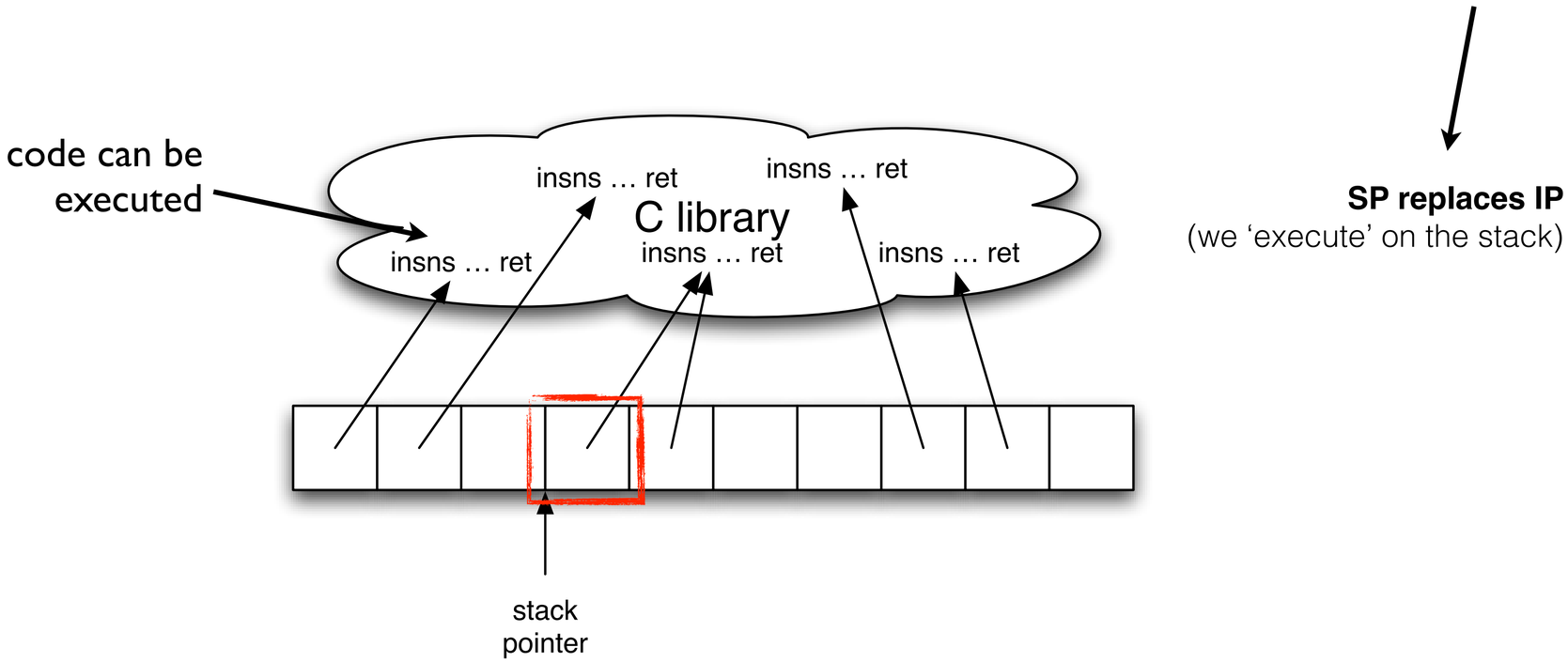
return-oriented programming (ROP)

setup stack to point to instructions that end in return



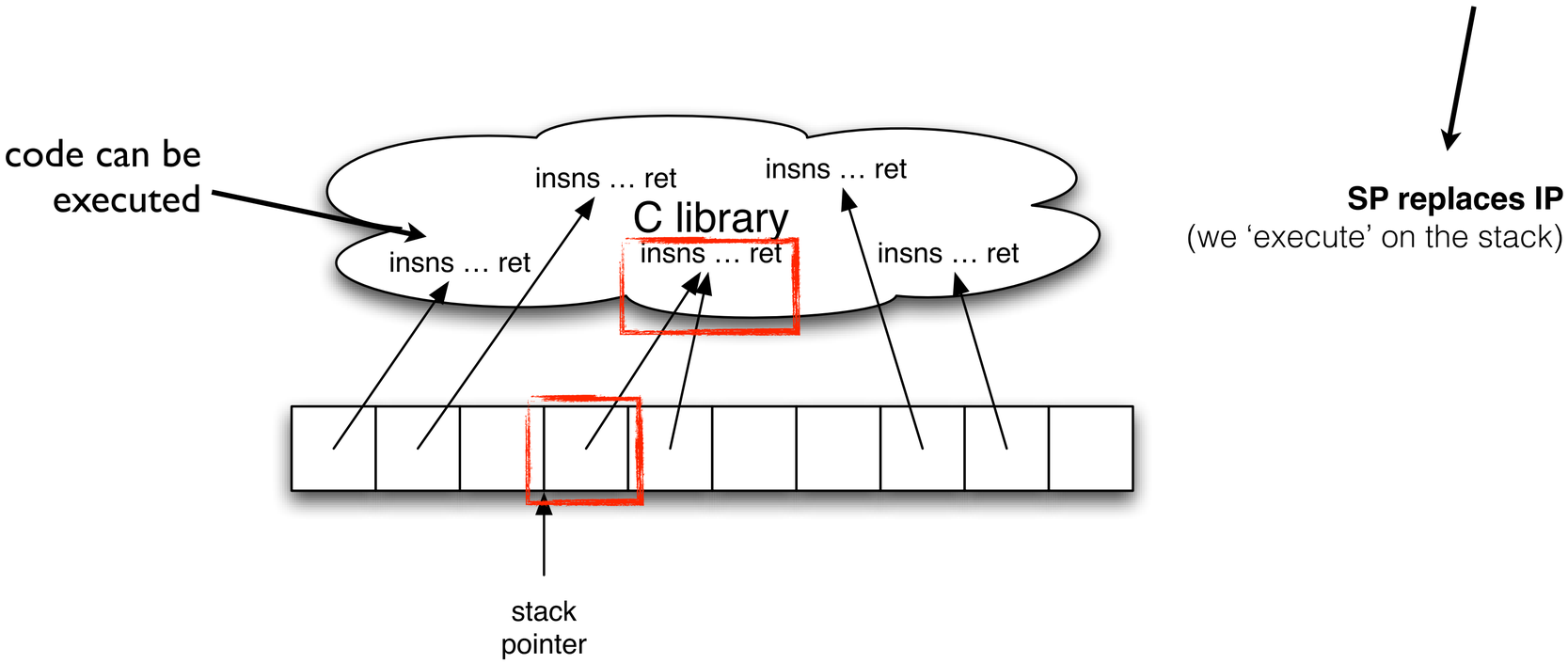
return-oriented programming (ROP)

setup stack to point to instructions that end in return



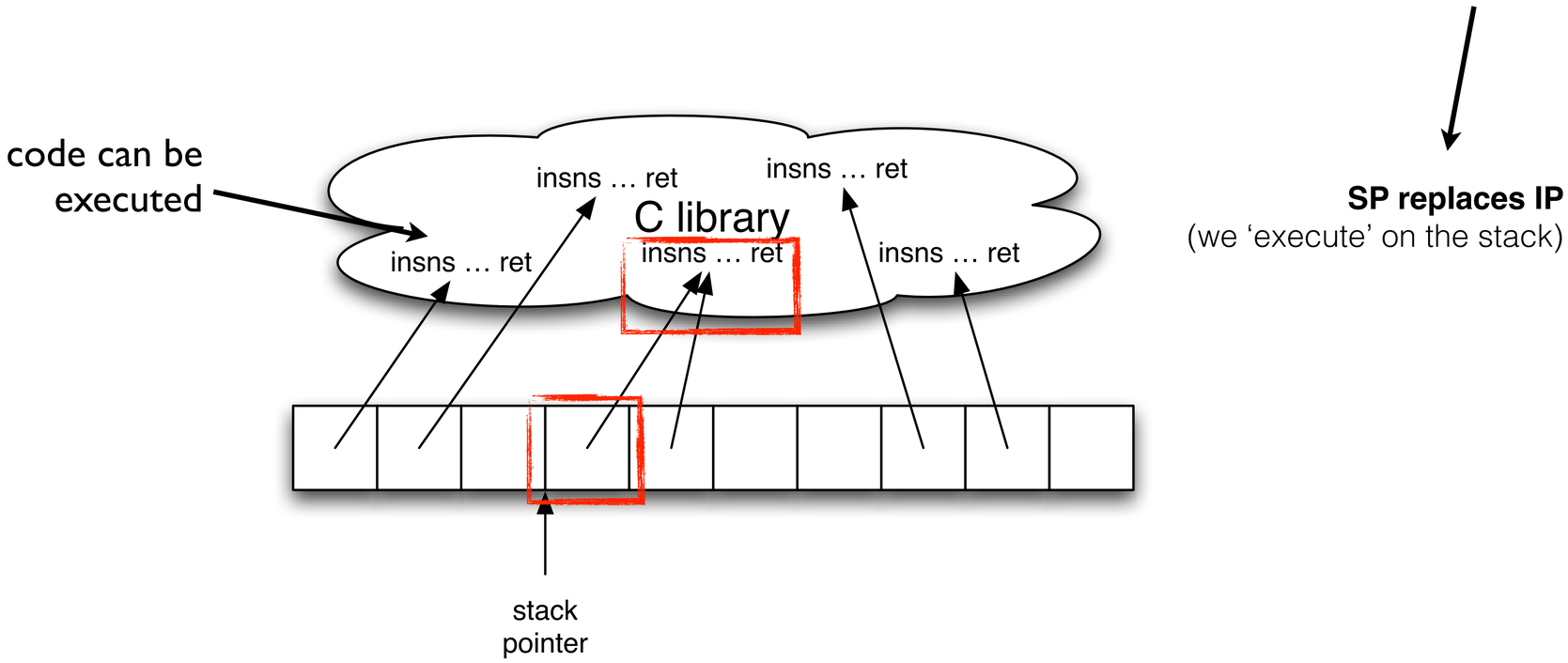
return-oriented programming (ROP)

setup stack to point to instructions that end in return



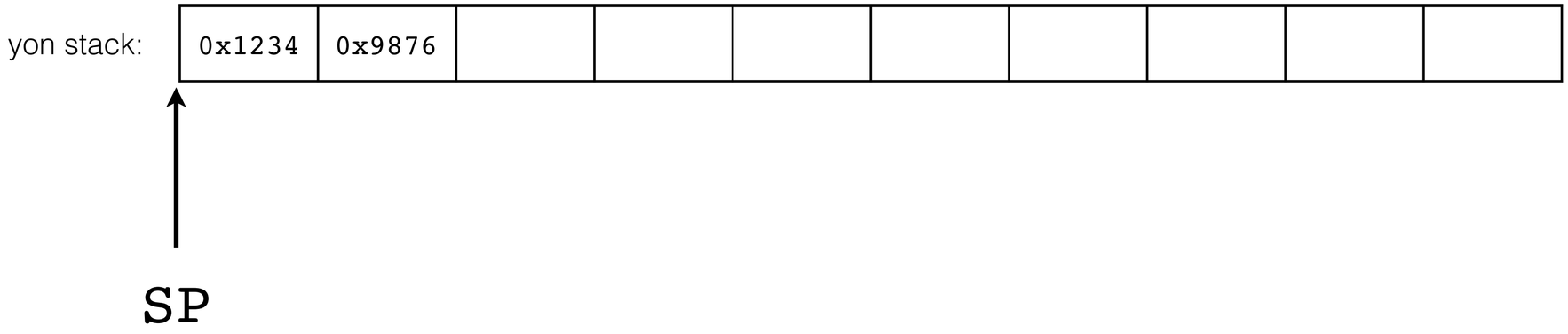
return-oriented programming (ROP)

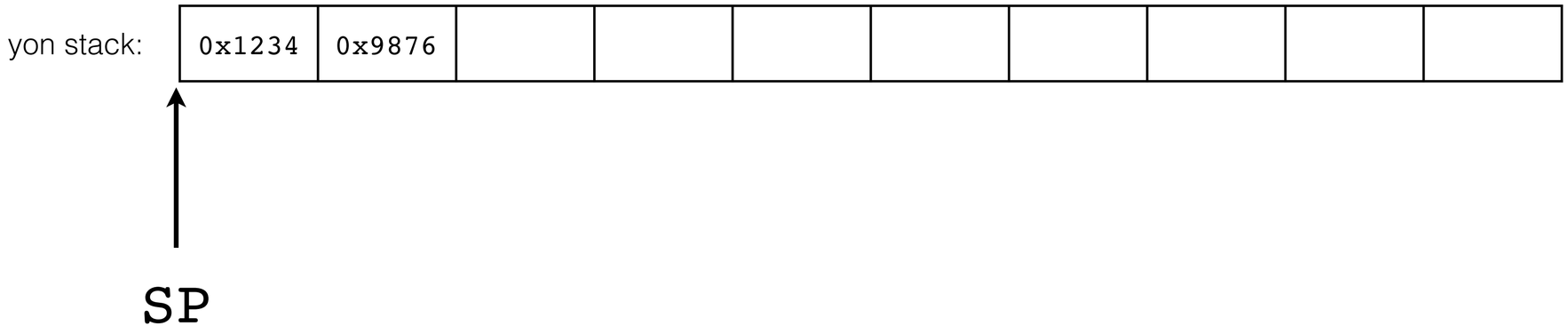
setup stack to point to instructions that end in return



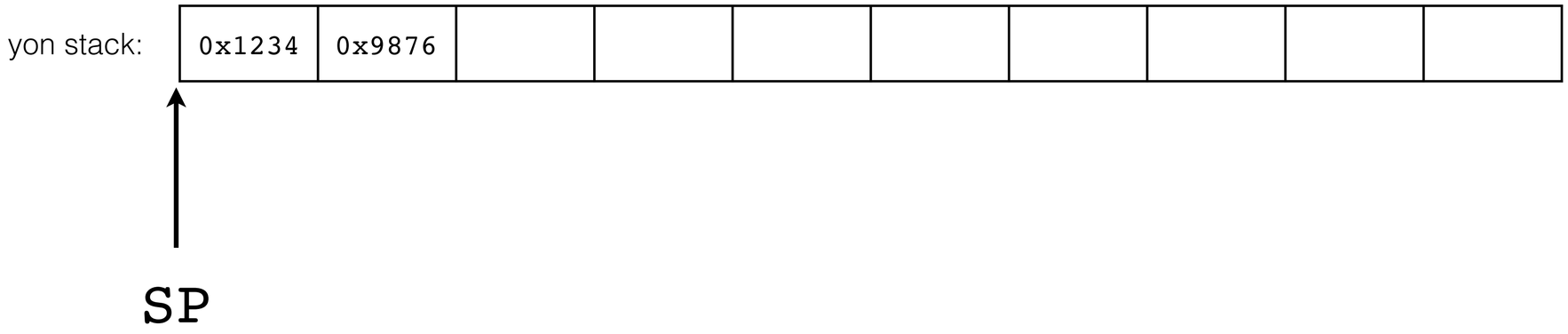
procedure:

1. setup stack via buffer overflow
2. issue return
3. pop addr from stack
4. IP executes instruction at addr





1. exploited function issues `ret`

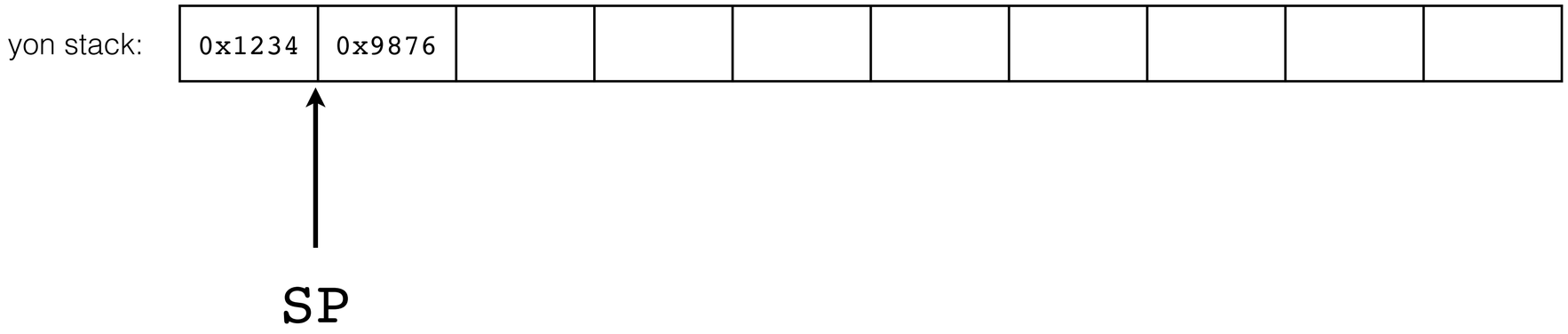


1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

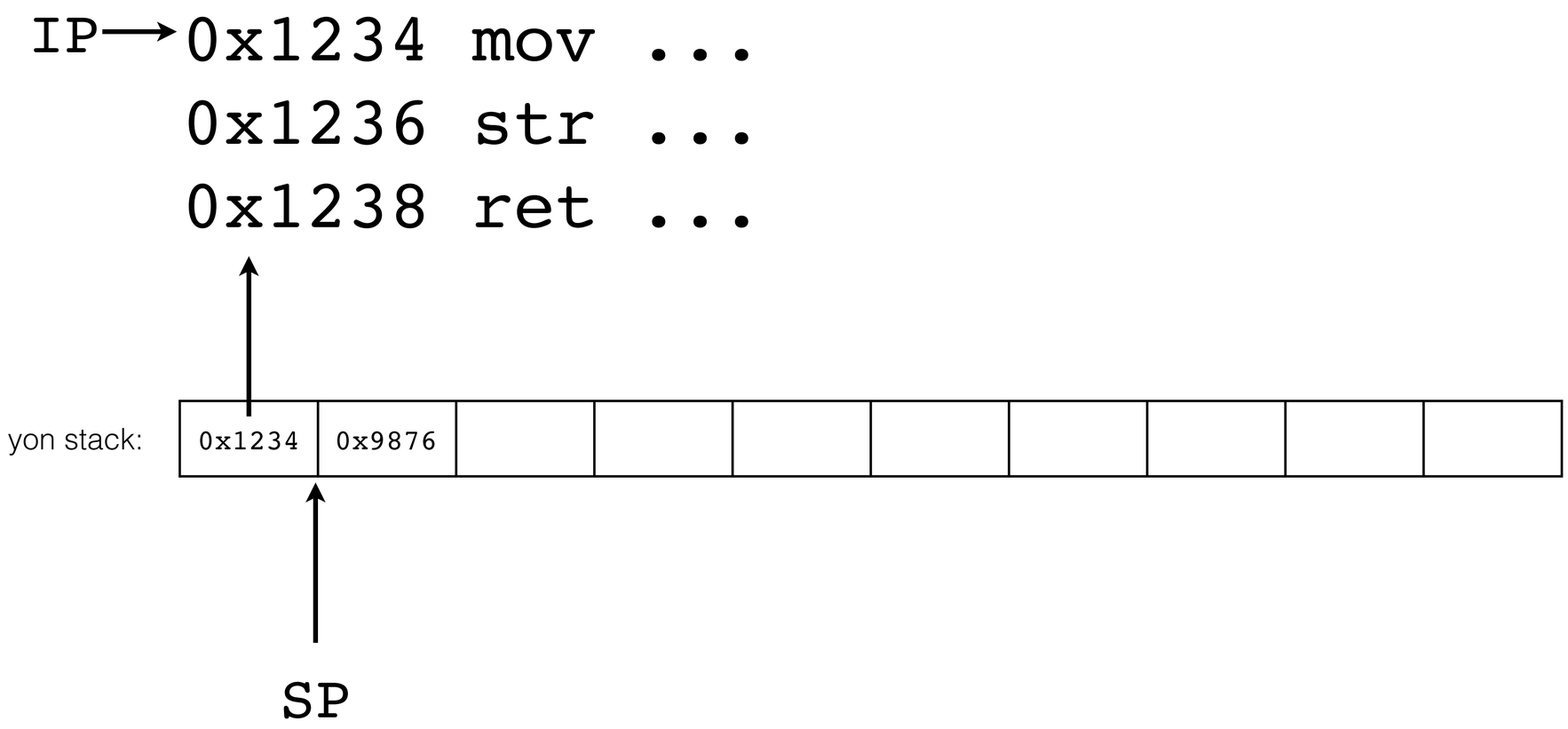
yon stack:

`0x1234`	`0x9876`								

1. exploited function issues **`ret`**
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes



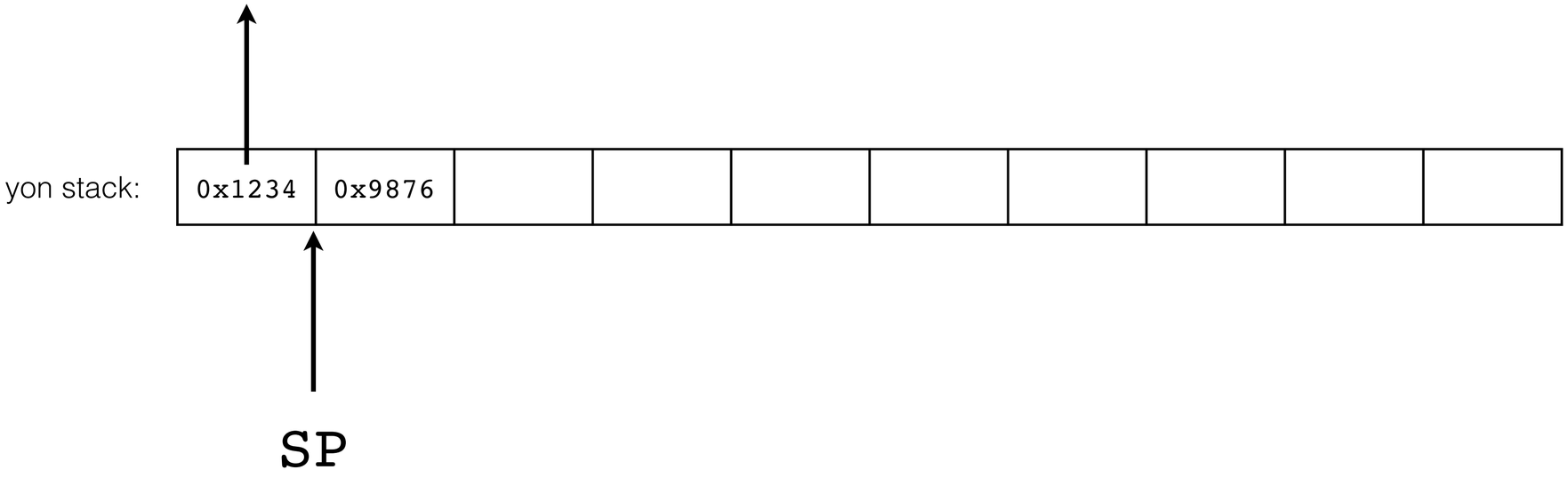
1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes



1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

```
IP→0x1234 mov ...
   0x1236 str ...
   0x1238 ret ...
```

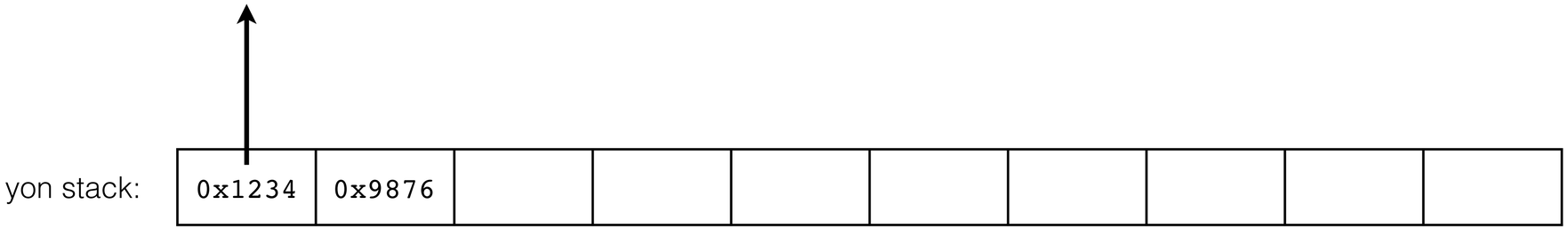
3. a. execute instructions
   b. issue `ret`
   c. follow step 2



1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

```
IP→0x1234 mov ...
   0x1236 str ...
   0x1238 ret ...
```

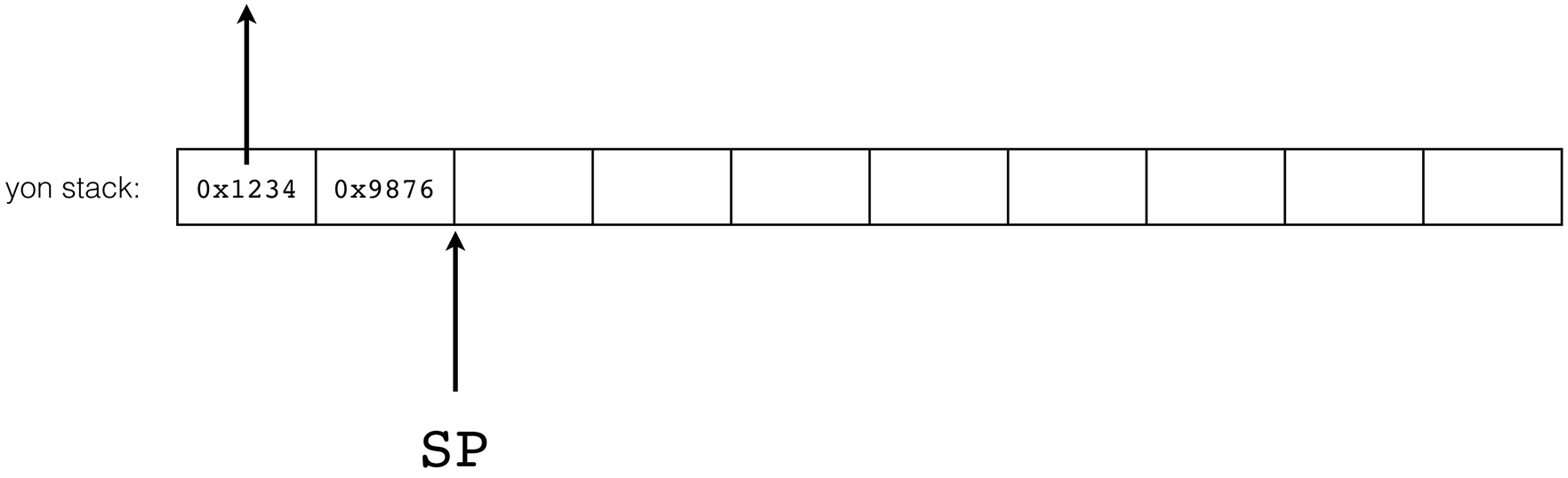
3. a. execute instructions
   b. issue `ret`
   c. follow step 2



1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

```
IP→0x1234 mov ...
   0x1236 str ...
   0x1238 ret ...
```

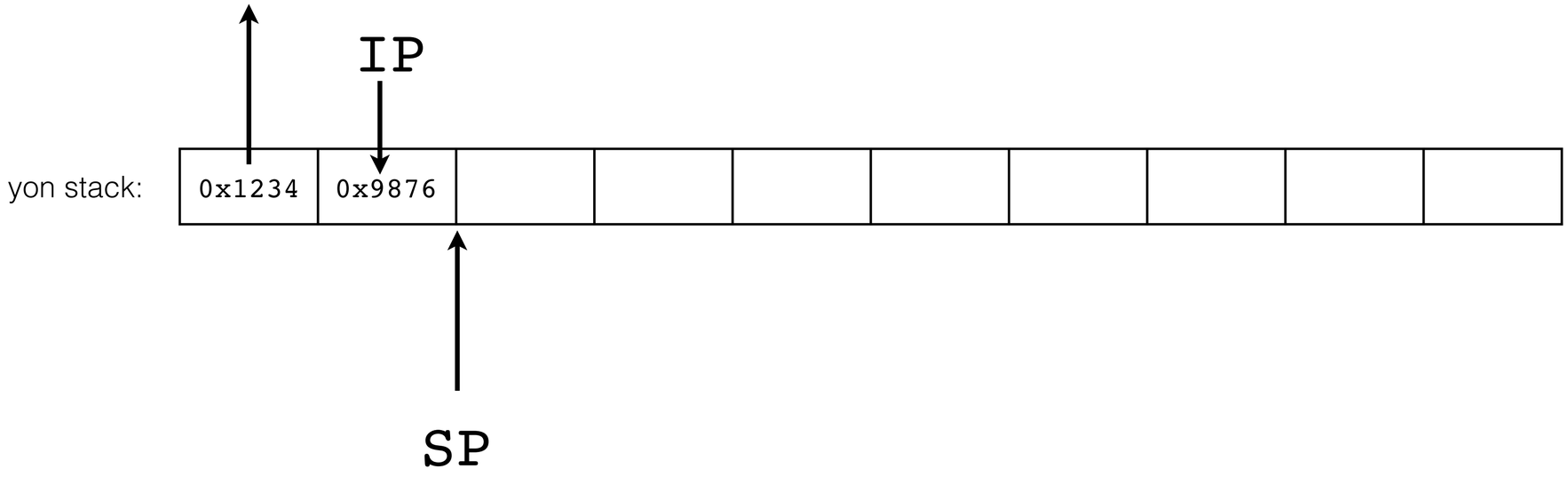
3. a. execute instructions
   b. issue `ret`
   c. follow step 2



1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

```
IP→0x1234 mov ...
   0x1236 str ...
   0x1238 ret ...
```

3. a. execute instructions
   b. issue `ret`
   c. follow step 2



1. exploited function issues `ret`
2. a. two bytes popped off stack
   b. SP incremented by two
   c. jump to address of bytes

why ROP works:

1. instruction set
2. lots of code in shared libraries

x86 instructions

x86 instructions

unaligned and variable length

x86 instructions

unaligned and variable length

can jump to point within instruction

x86 instructions

unaligned and variable length

can jump to point within instruction

IP here

```
f7 c7 07 00 00 00   →   test $0x00000007, %edi
0f 95 45 c3             setnzb -61(%ebp)
```

x86 instructions

unaligned and variable length

can jump to point within instruction

IP here

```
f7 c7 07 00 00 00      test $0x00000007, %edi
0f 95 45 c3            setnzb -61(%ebp)
```

get these instructions

IP here
(one byte over)

```
c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```

x86 instructions

unaligned and variable length

can jump to point within instruction

IP here

```
f7 c7 07 00 00 00      test $0x00000007, %edi
0f 95 45 c3            setnzb -61(%ebp)
```

get these instructions

IP here
(one byte over)

```
c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```

the cool thing: can still work on sensible instruction sets (RISC)

`libc`: lots of instructions

`libc`: lots of instructions

`ret`

search for sequences that end in `0xc3`

`libc`: lots of instructions

`ret`

search for sequences that end in `0xc3`

string together on stack to do useful things

defender is thinking...

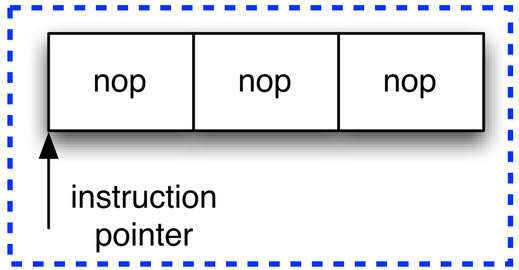
...this could be bad

defender is thinking...

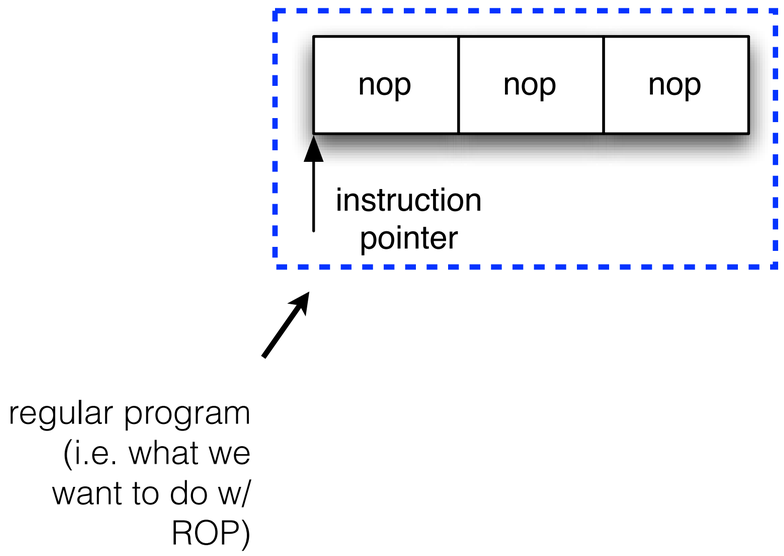
...this could be bad

ROP NOP:

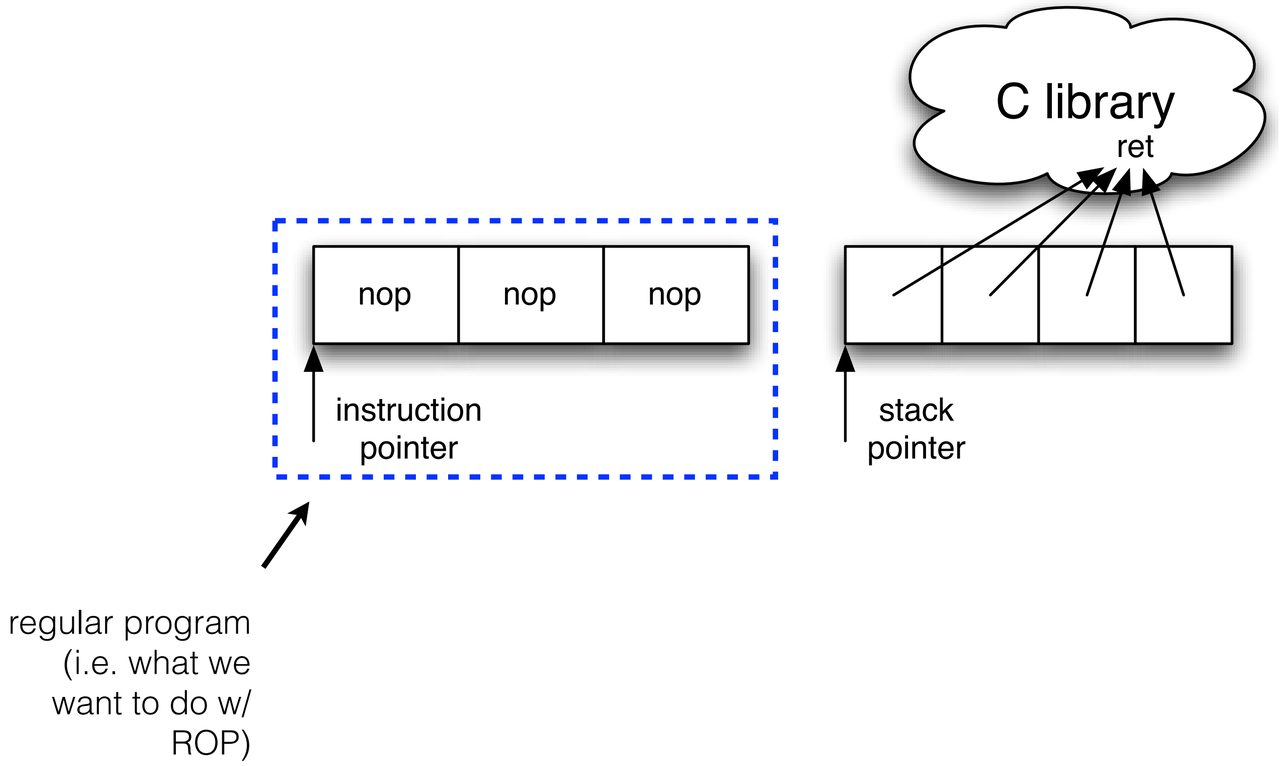
ROP NOP:



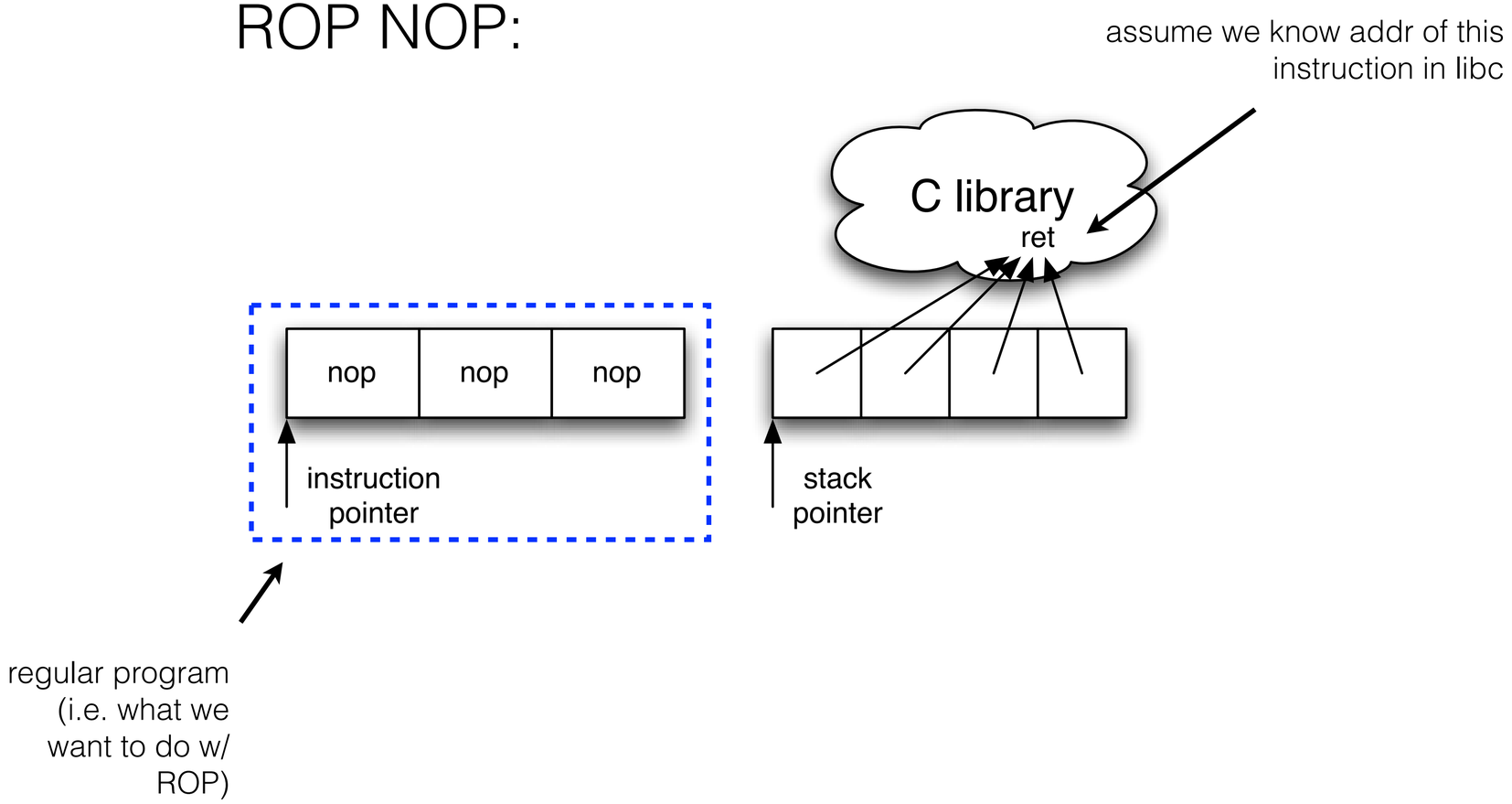
ROP NOP:



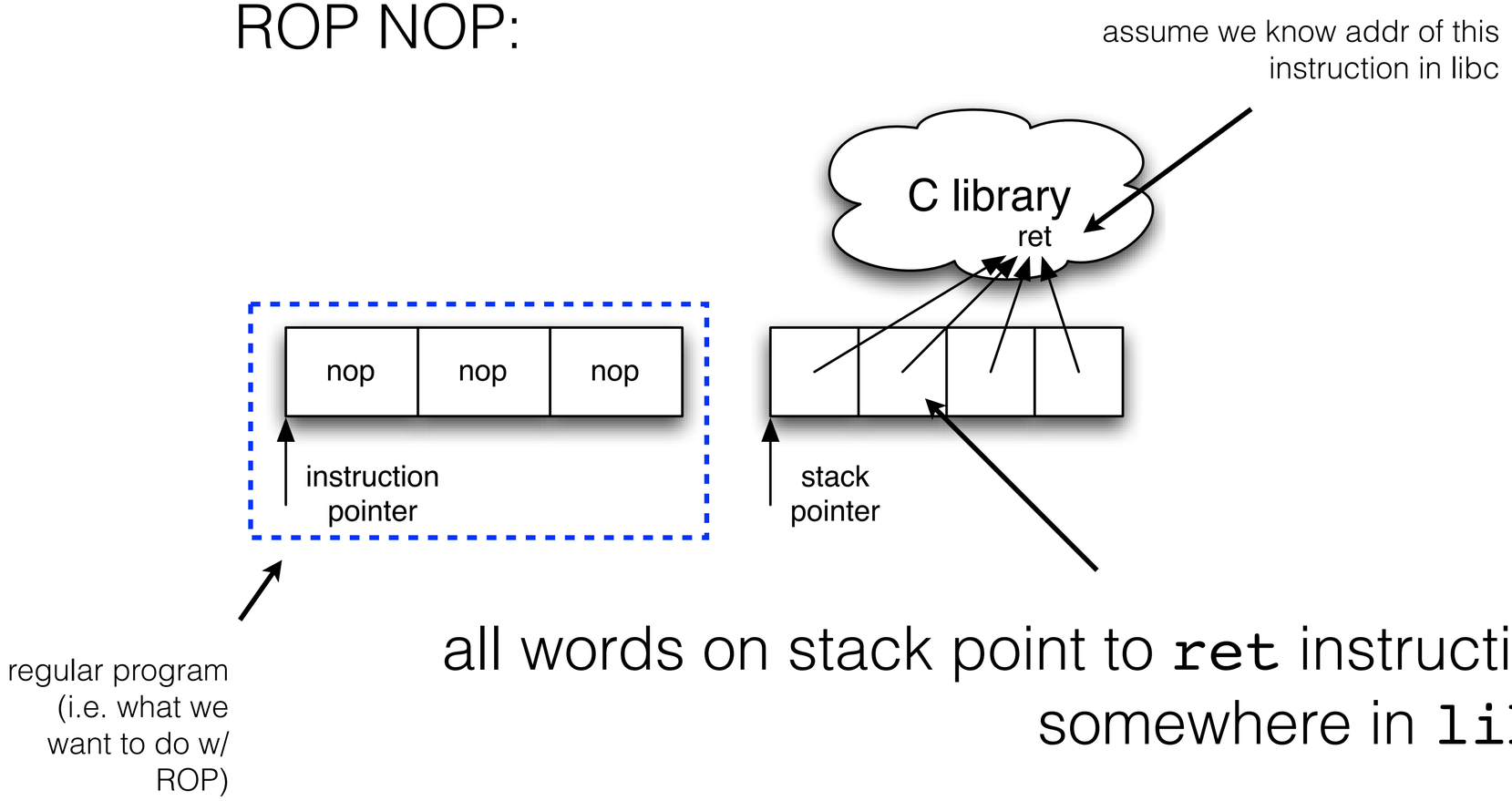
ROP NOP:



ROP NOP:

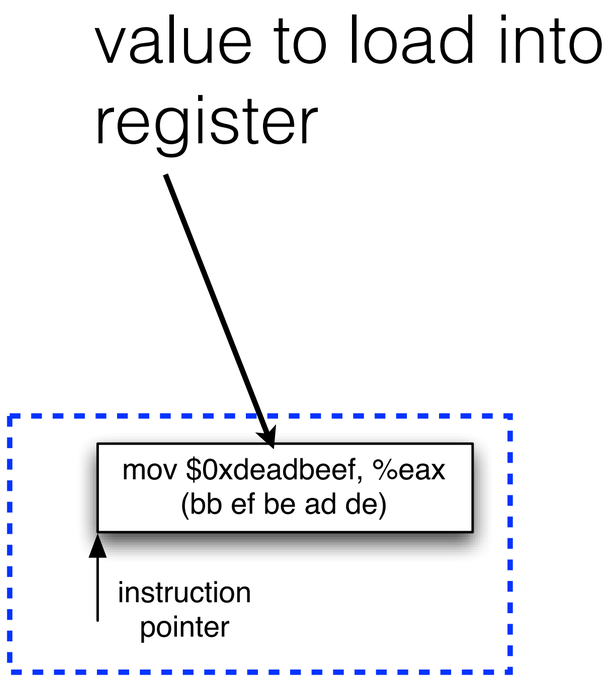


ROP NOP:

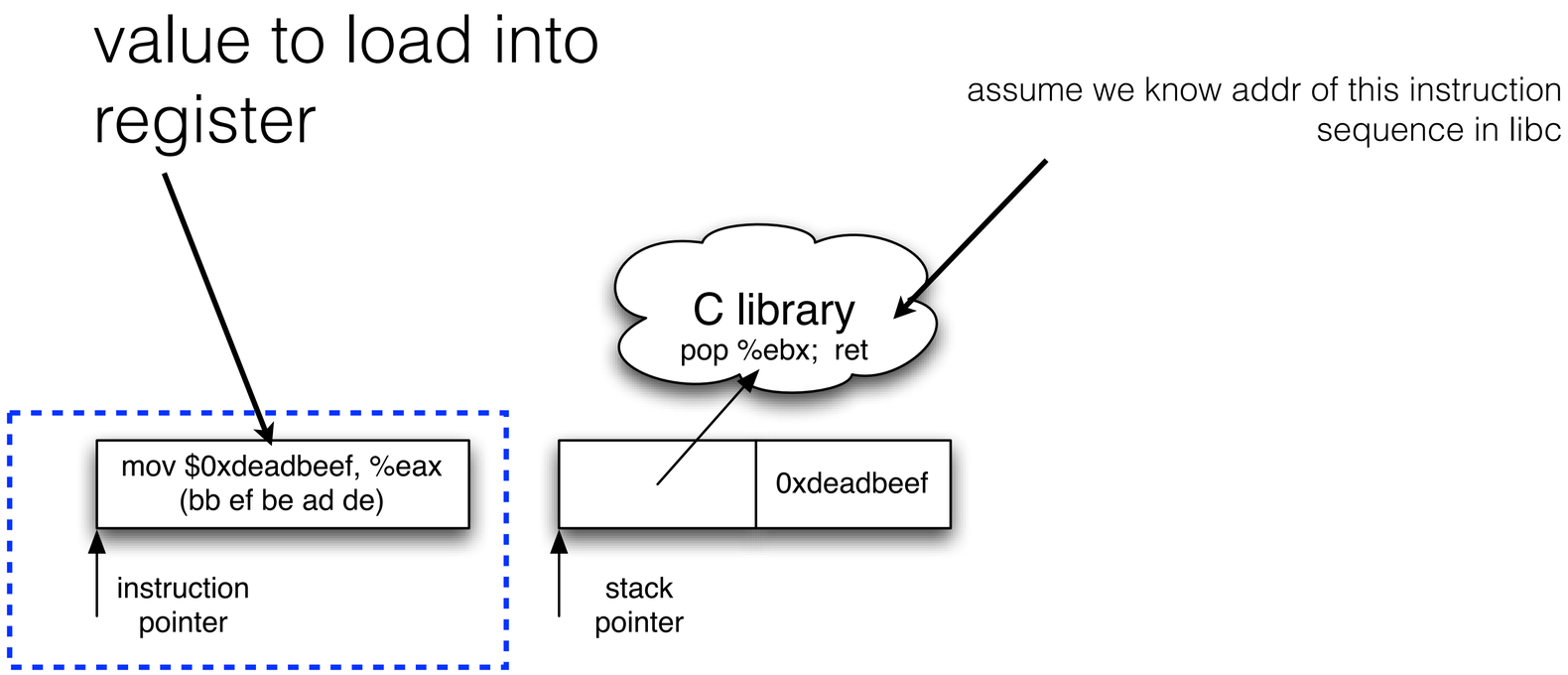


all words on stack point to `ret` instruction somewhere in `libc`

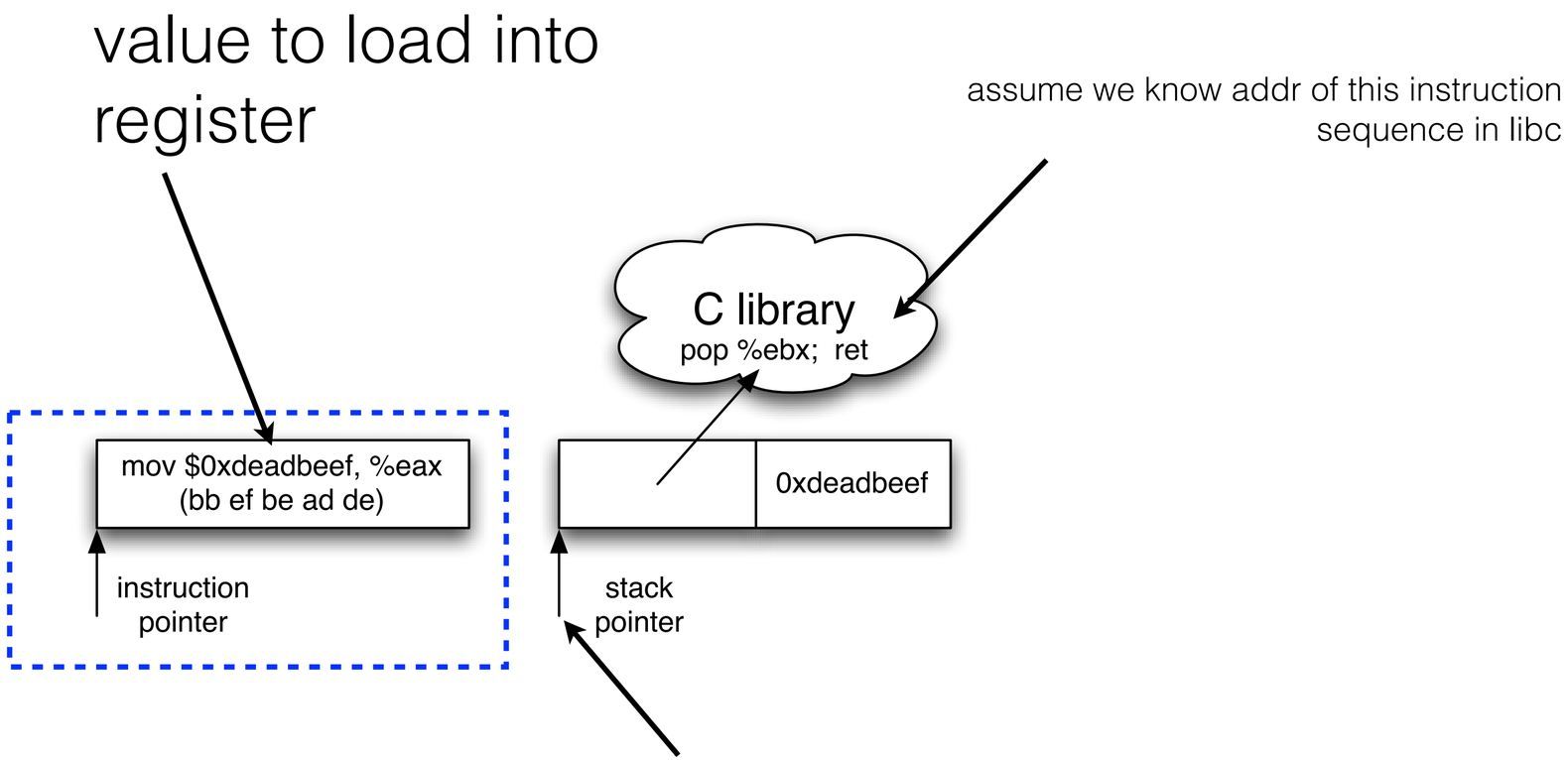
ROP immediate load



ROP immediate load

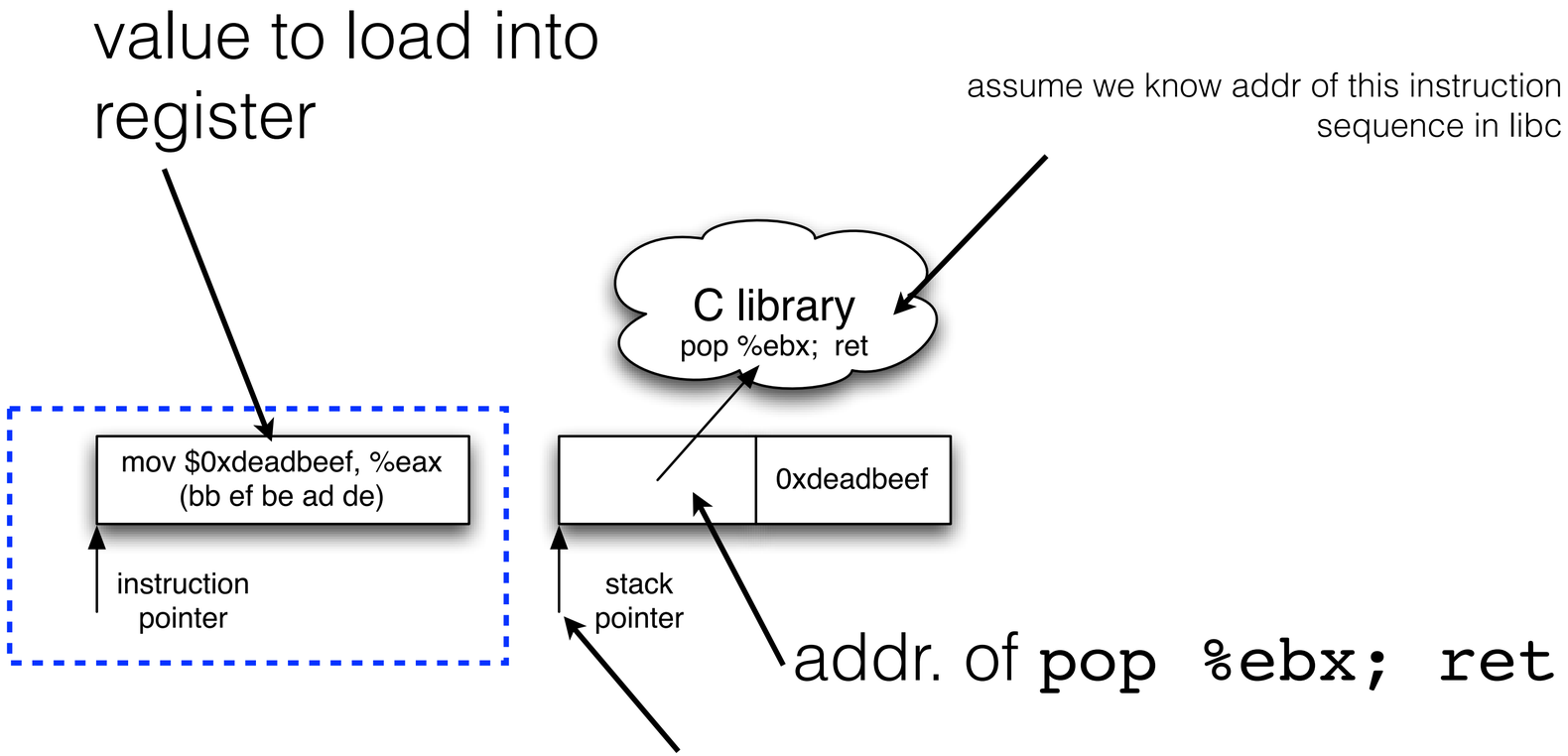


ROP immediate load



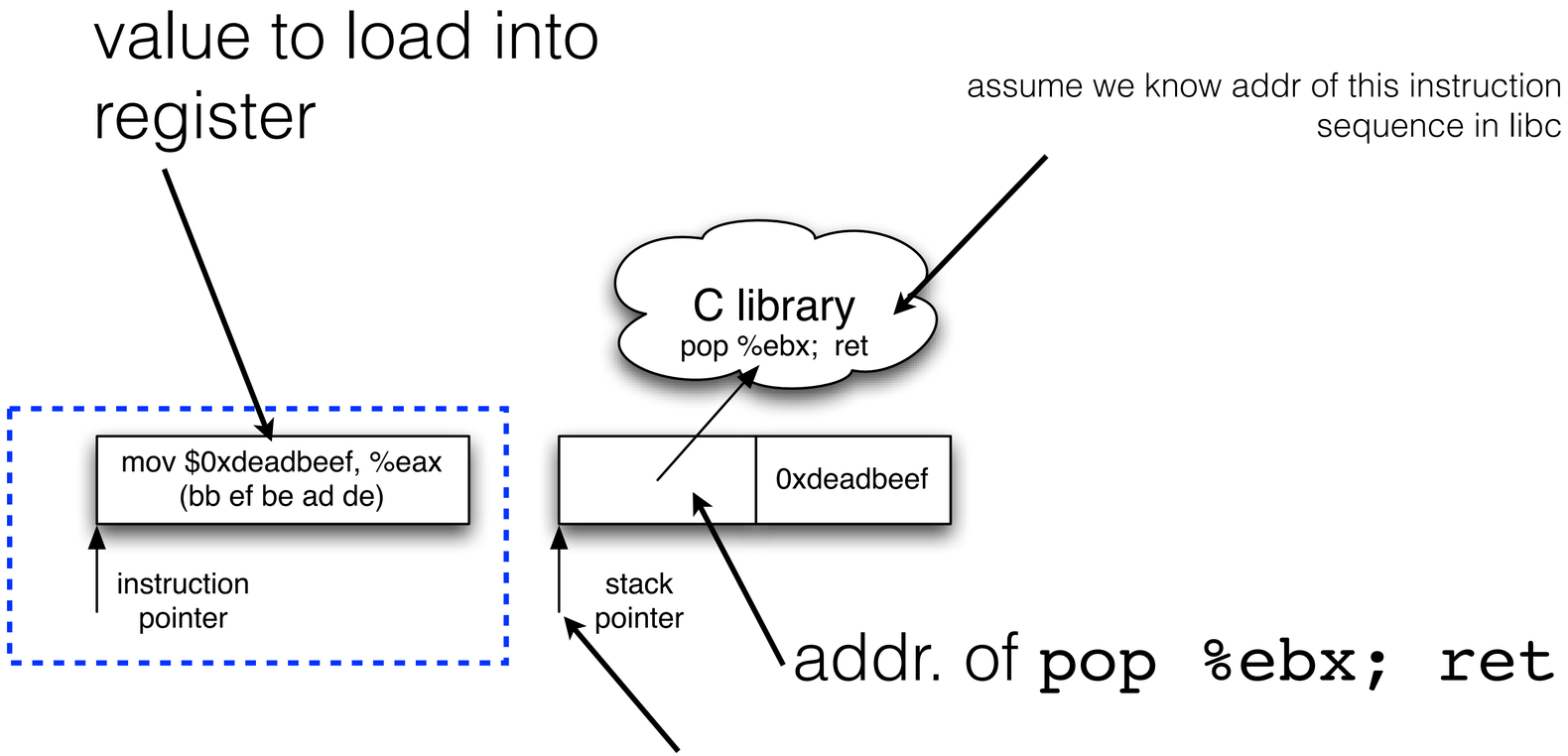
1. `ret` issued with SP here

ROP immediate load



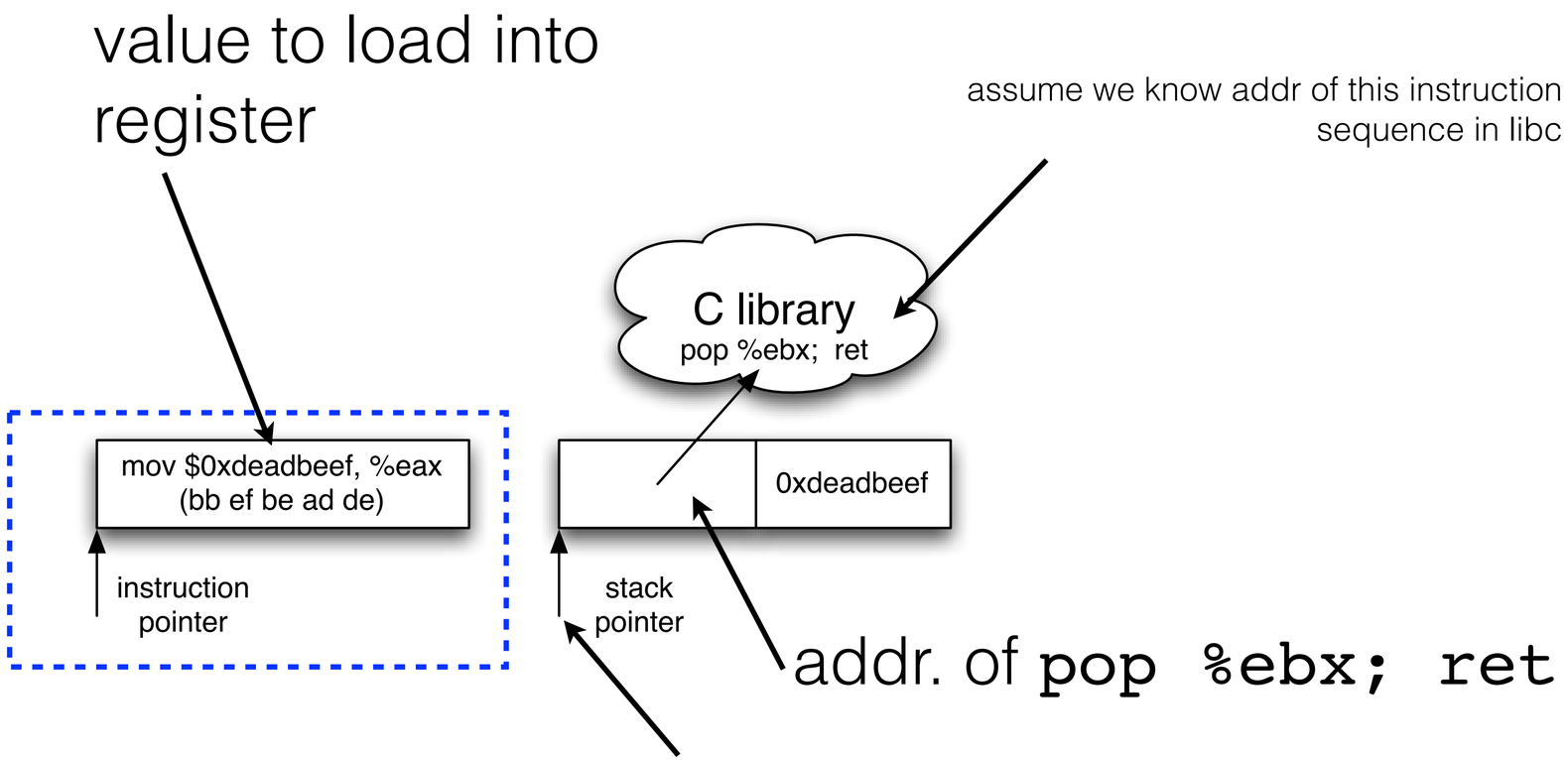
1. `ret` issued with SP here
2. IP pointed at addr of `pop`...

ROP immediate load



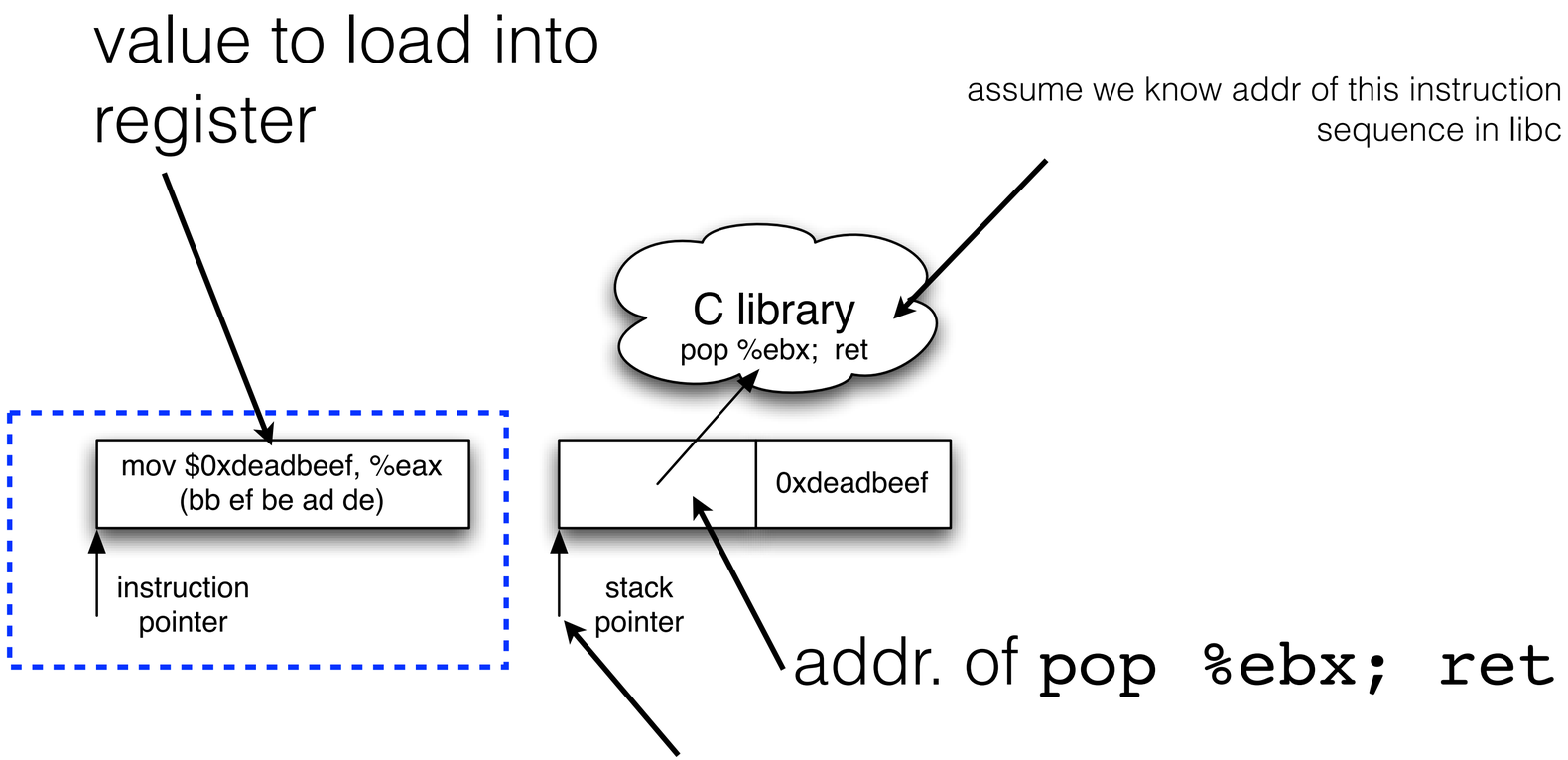
1. `ret` issued with SP here
2. IP pointed at addr of `pop`...
3. SP moved right one word

ROP immediate load



1. **`ret`** issued with SP here
2. IP pointed at addr of **`pop`**...
3. SP moved right one word
4. **`pop`** puts 0xdeadbeef into **`ebx`**

ROP immediate load



1. **ret** issued with SP here
2. IP pointed at addr of **pop**...
3. SP moved right one word
4. **pop** puts 0xdeadbeef into **ebx**
5. SP moved right one word

ROP word load

dot represents constructed stack

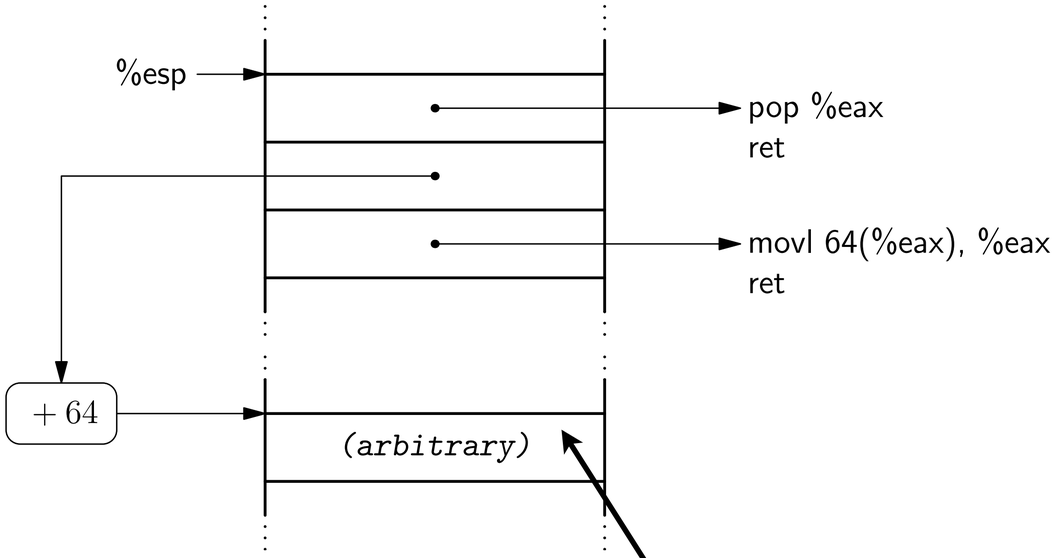
ROP word load

word into `%eax`

dot represents constructed stack

ROP word load

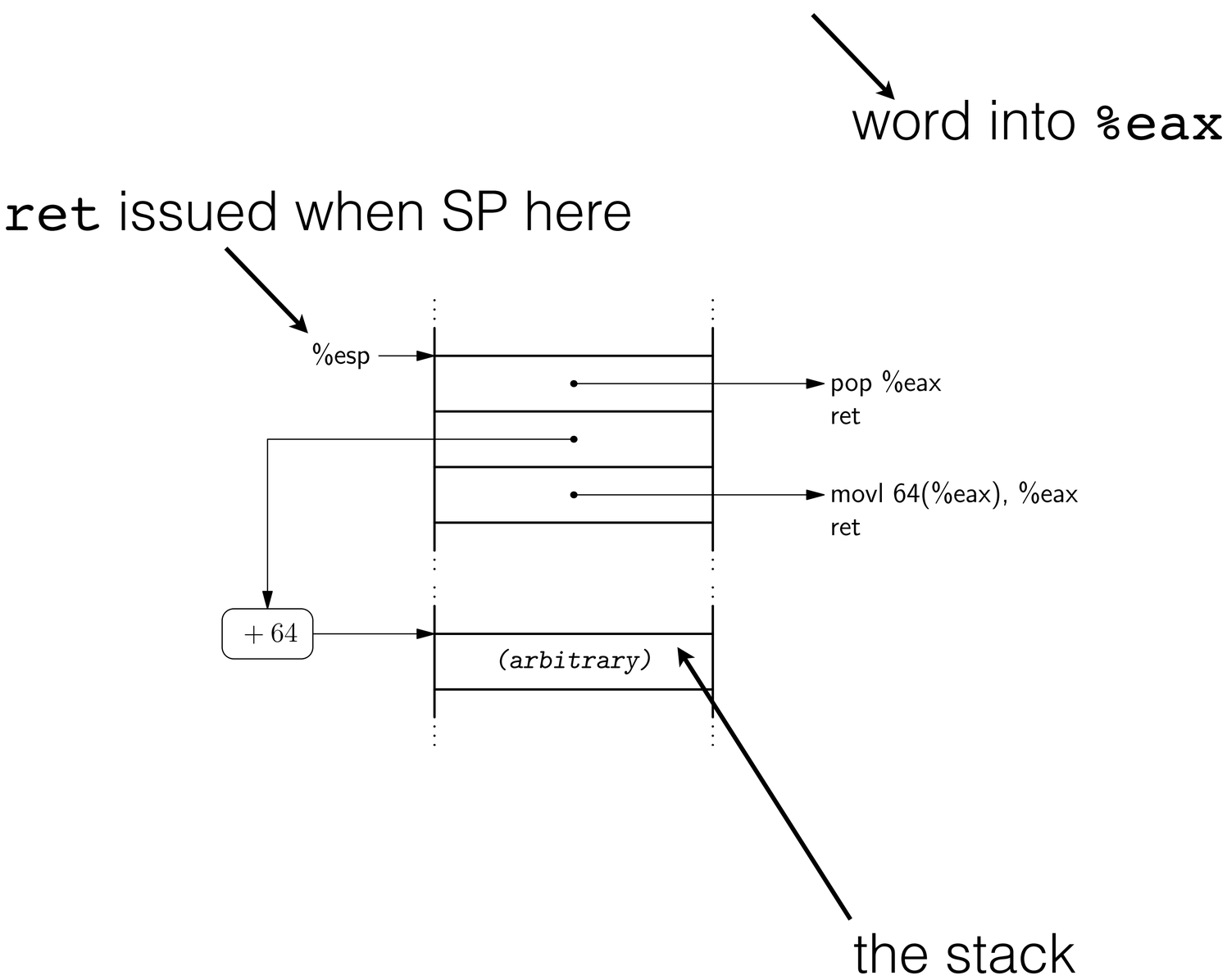
word into `%eax`



the stack

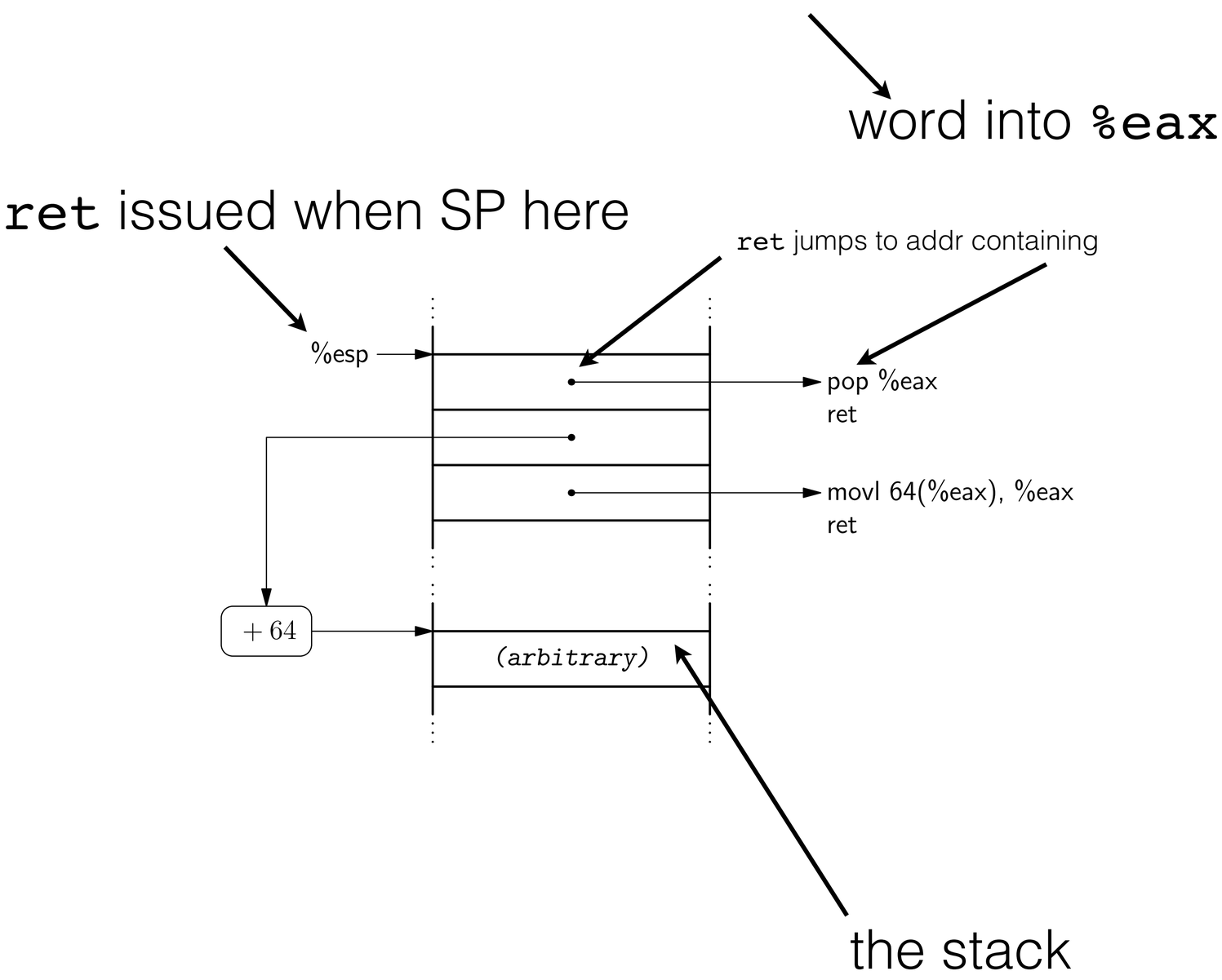
dot represents constructed stack

ROP word load



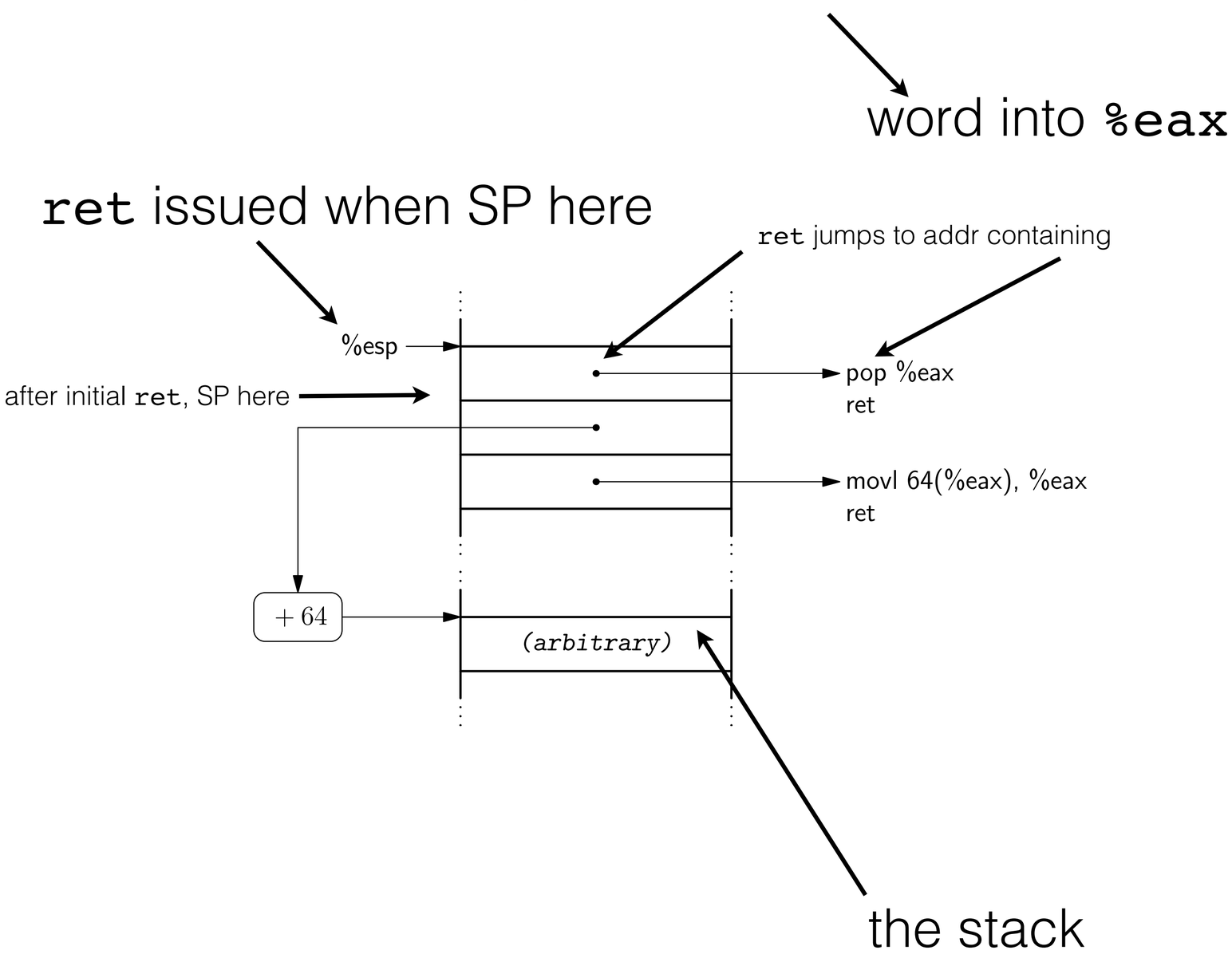
dot represents constructed stack

ROP word load



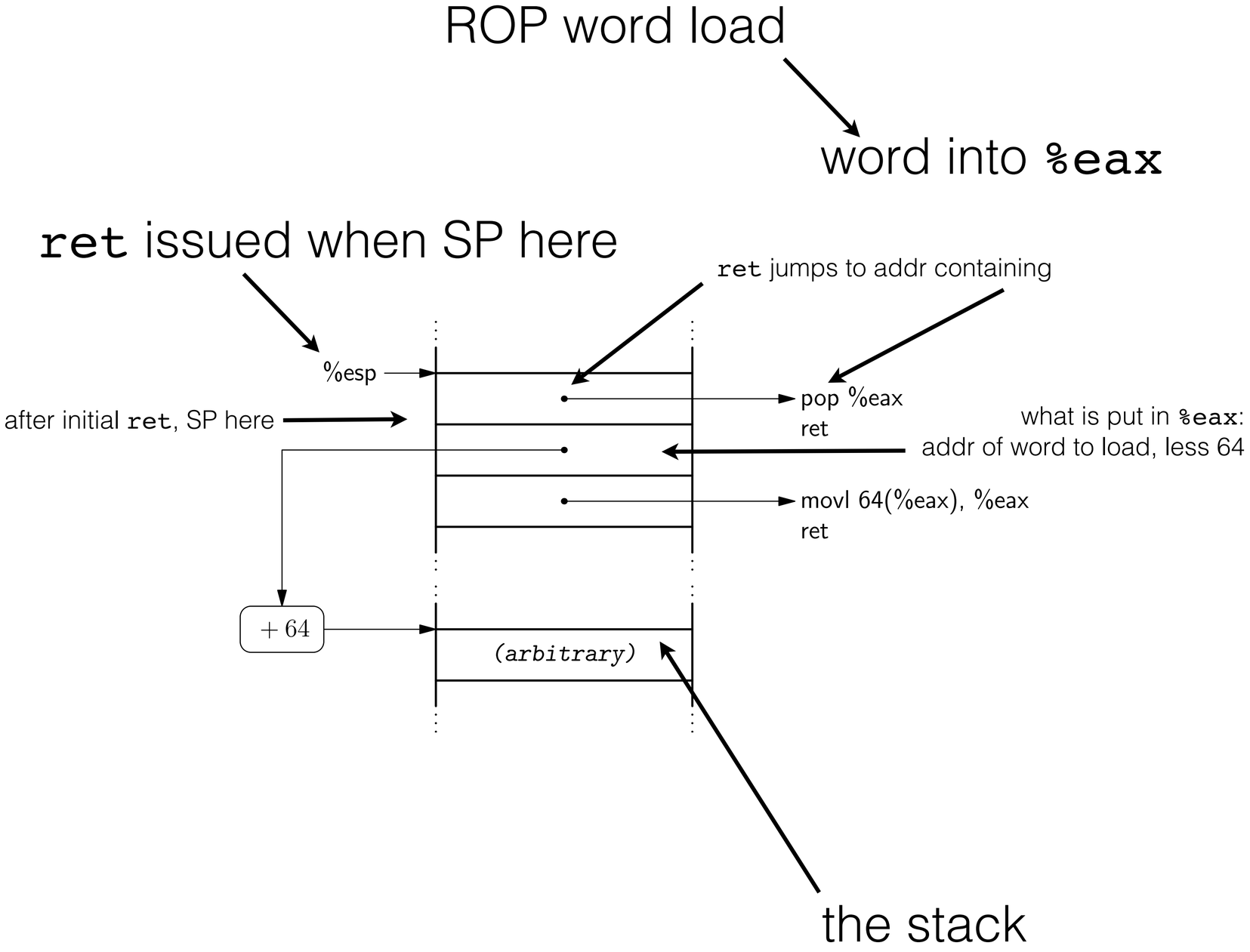
dot represents constructed stack

ROP word load



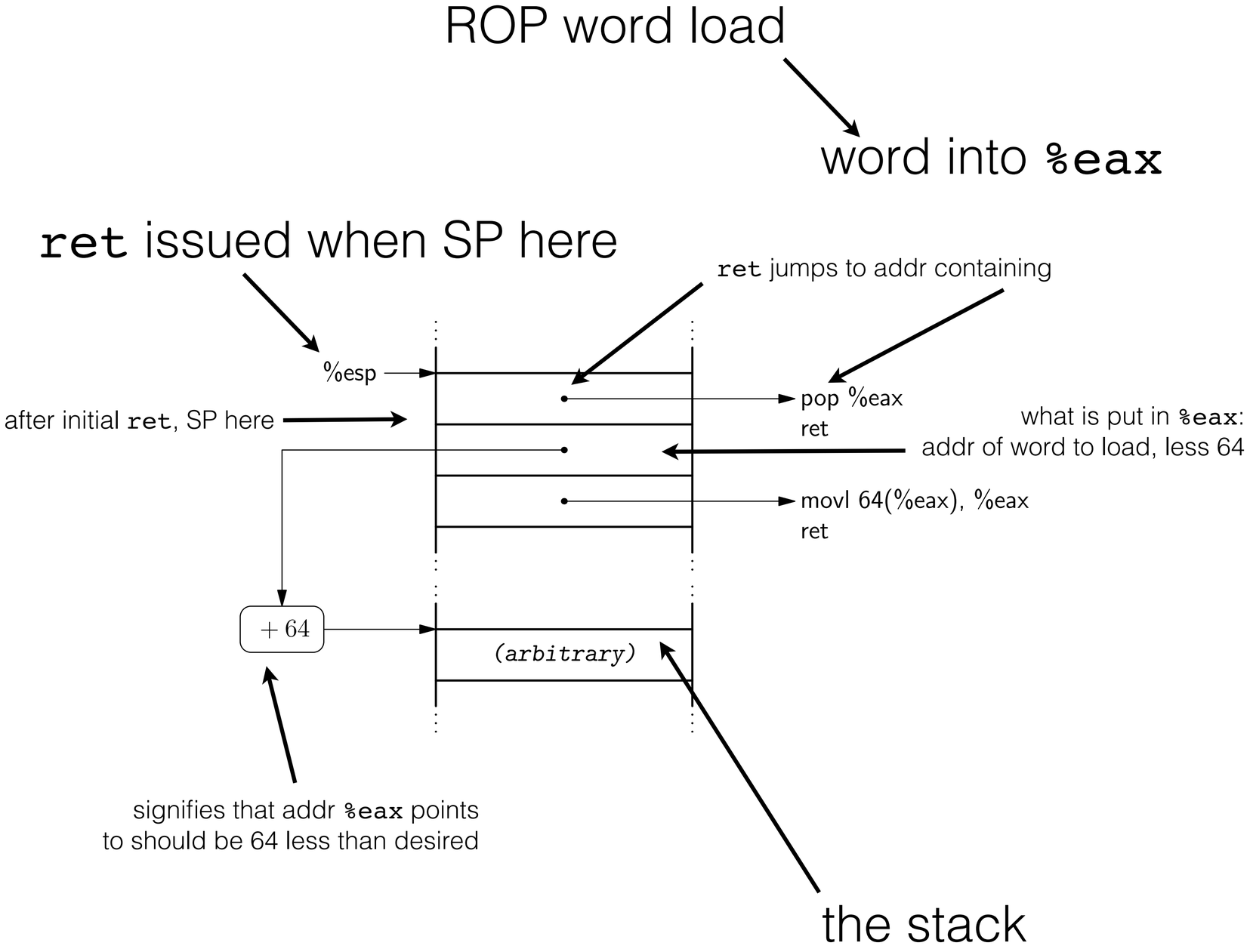
dot represents constructed stack

ROP word load



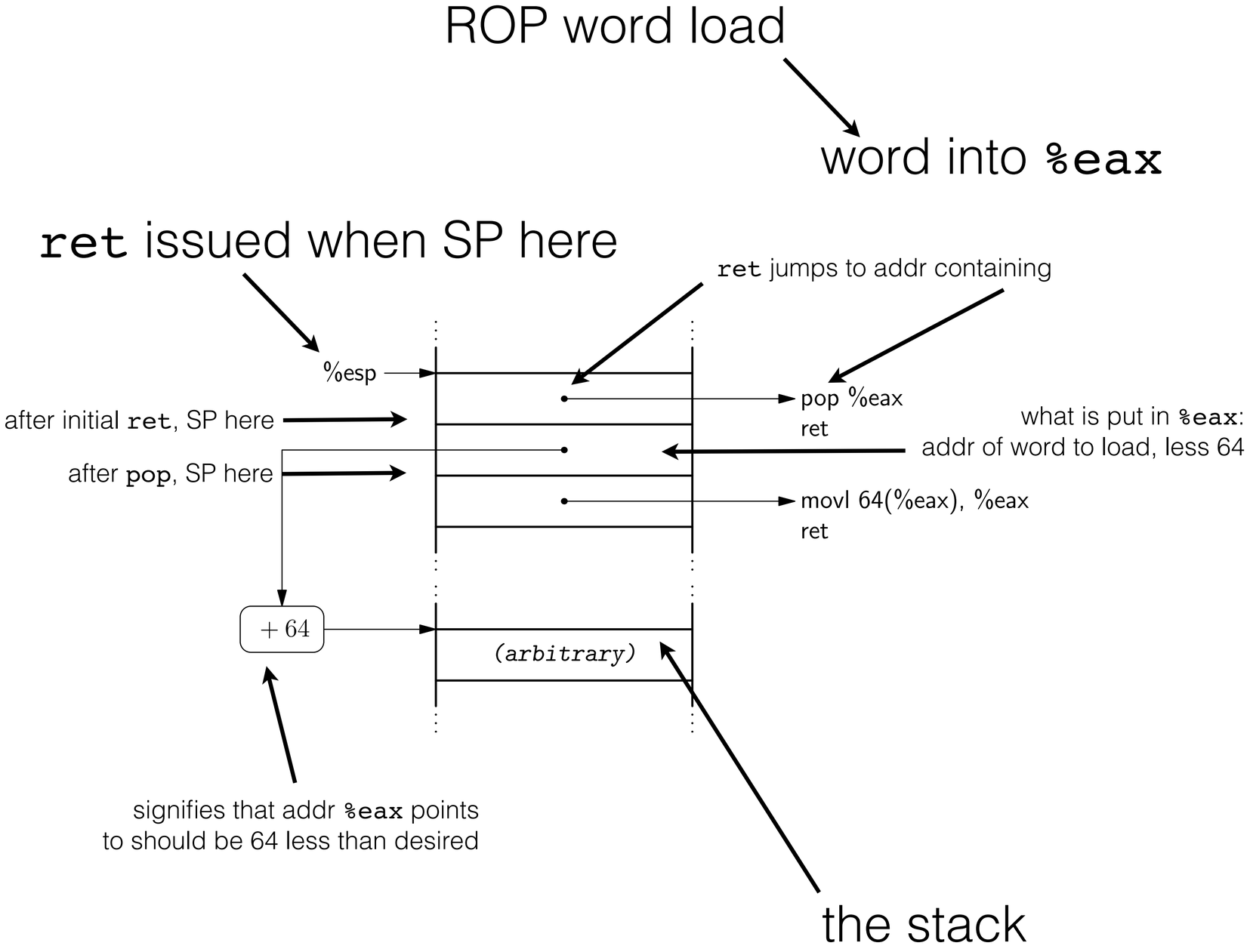
dot represents constructed stack

ROP word load



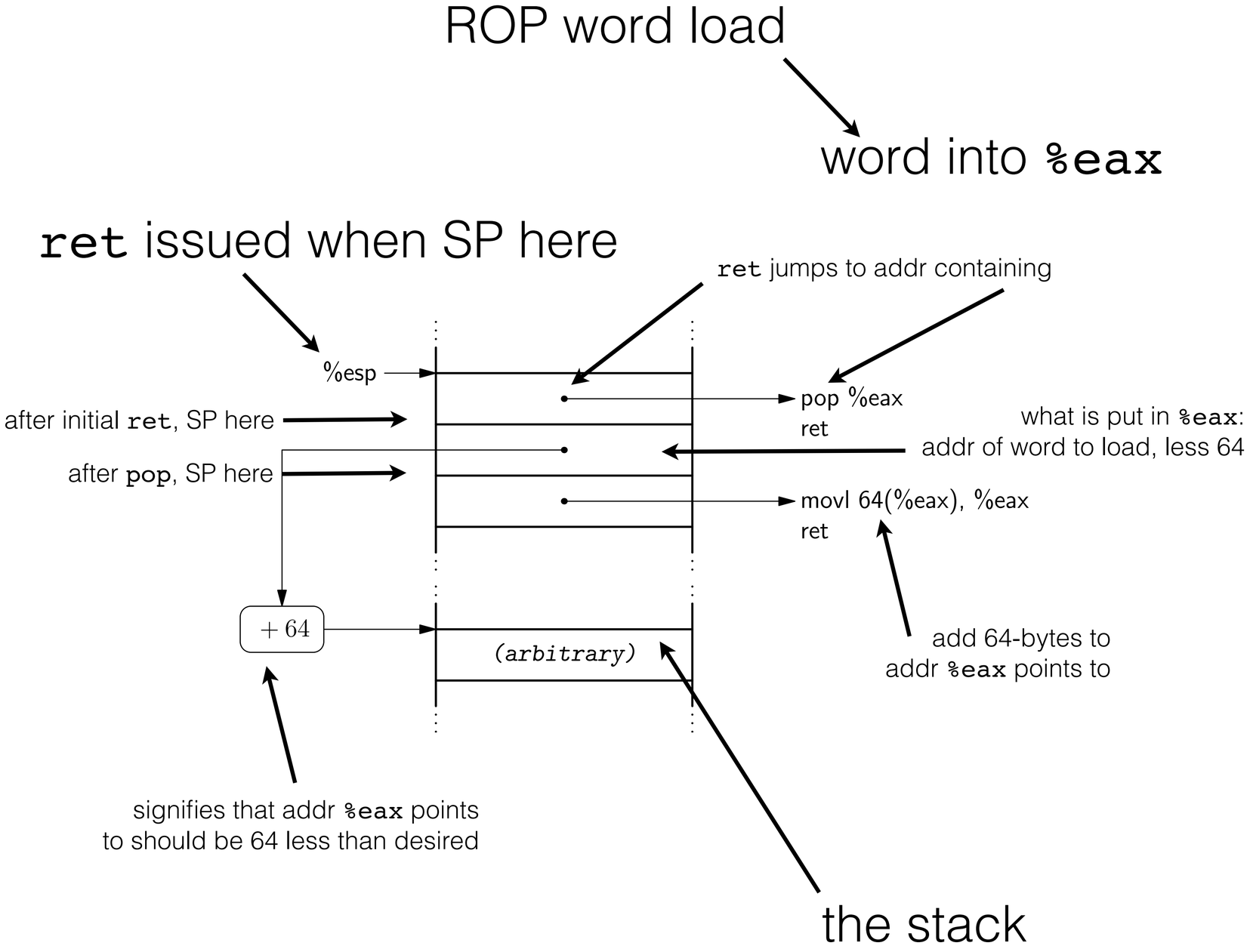
dot represents constructed stack

ROP word load



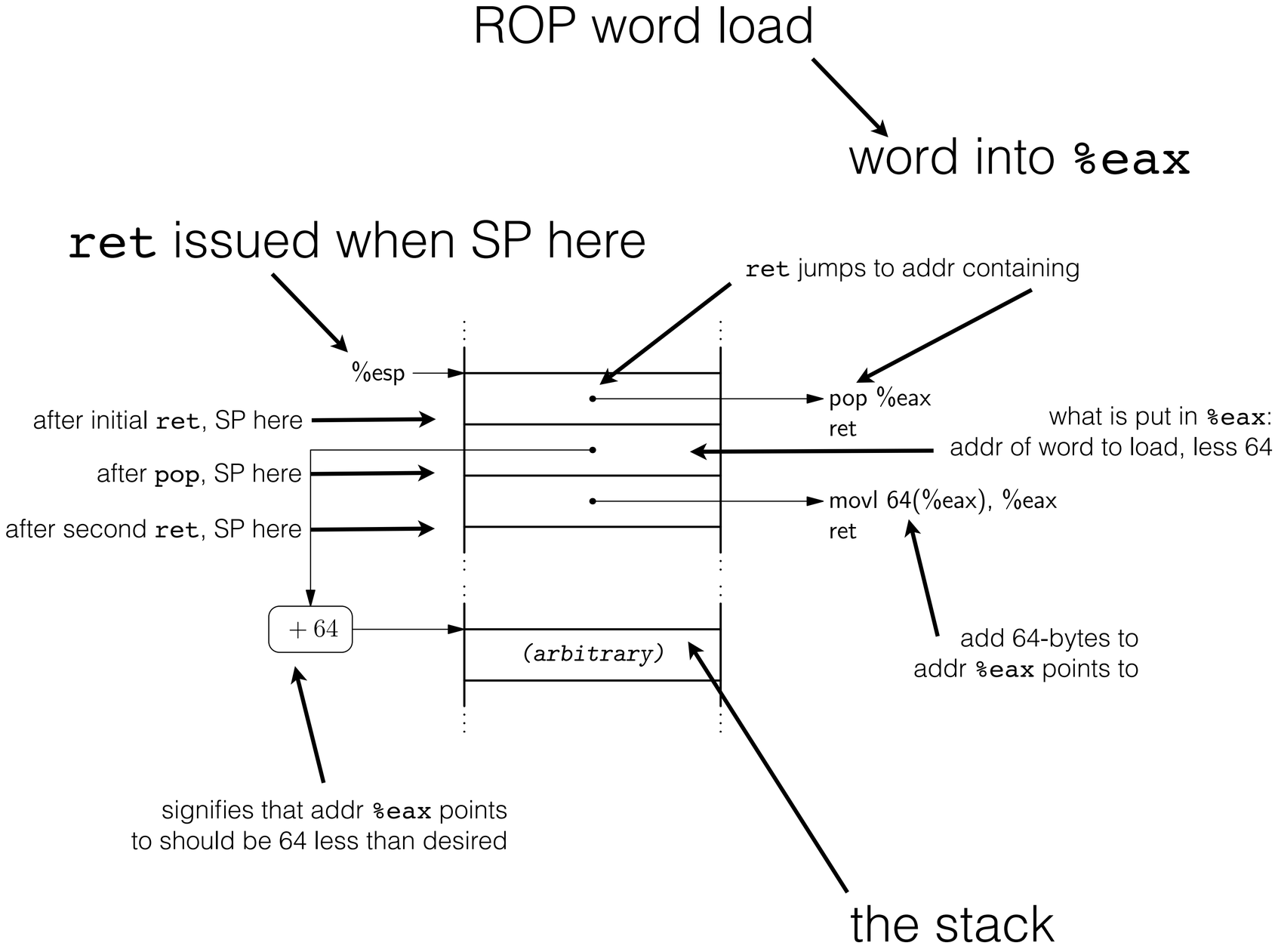
dot represents constructed stack

ROP word load



dot represents constructed stack

ROP word load



dot represents constructed stack

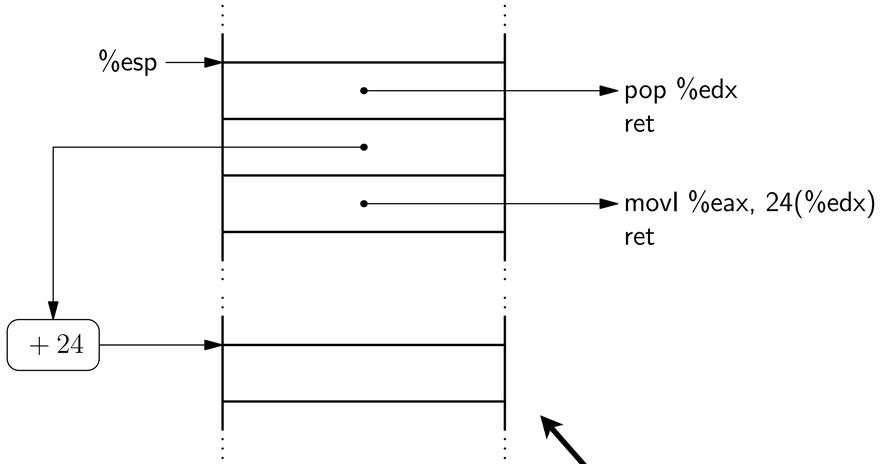
ROP word store

ROP word store

word in `%eax` to memory

ROP word store

word in `%eax` to memory

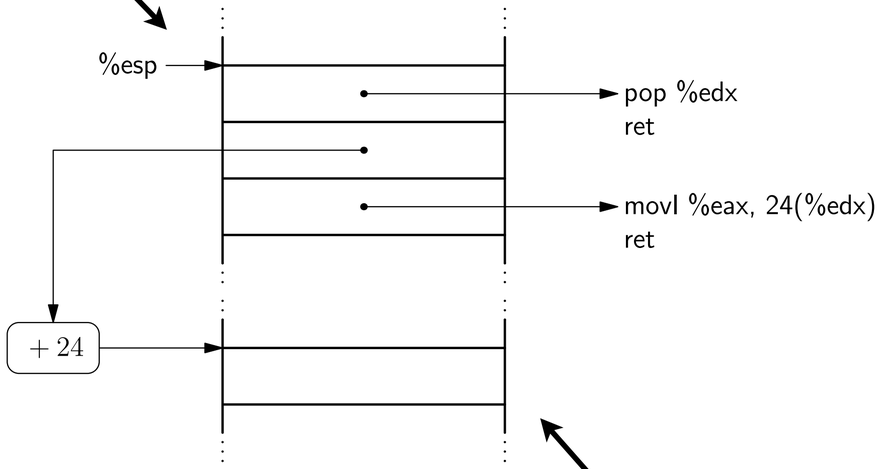


the stack

ROP word store

word in `%eax` to memory

`ret` issued when SP here



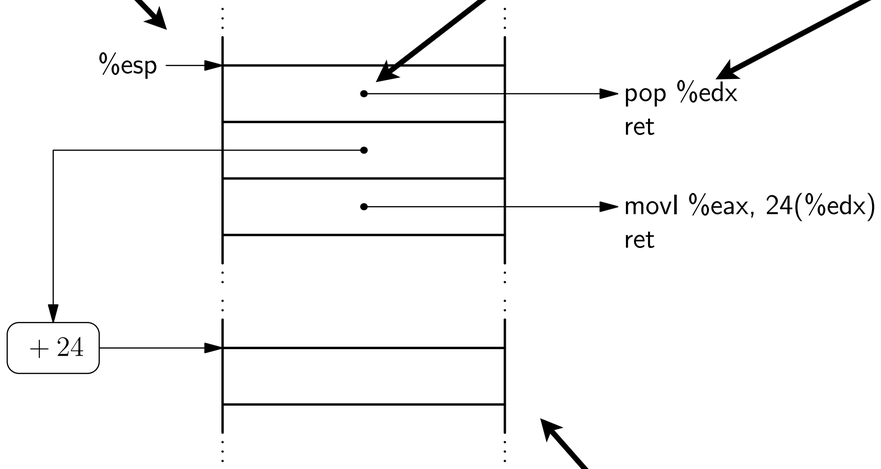
the stack

ROP word store

word in `%eax` to memory

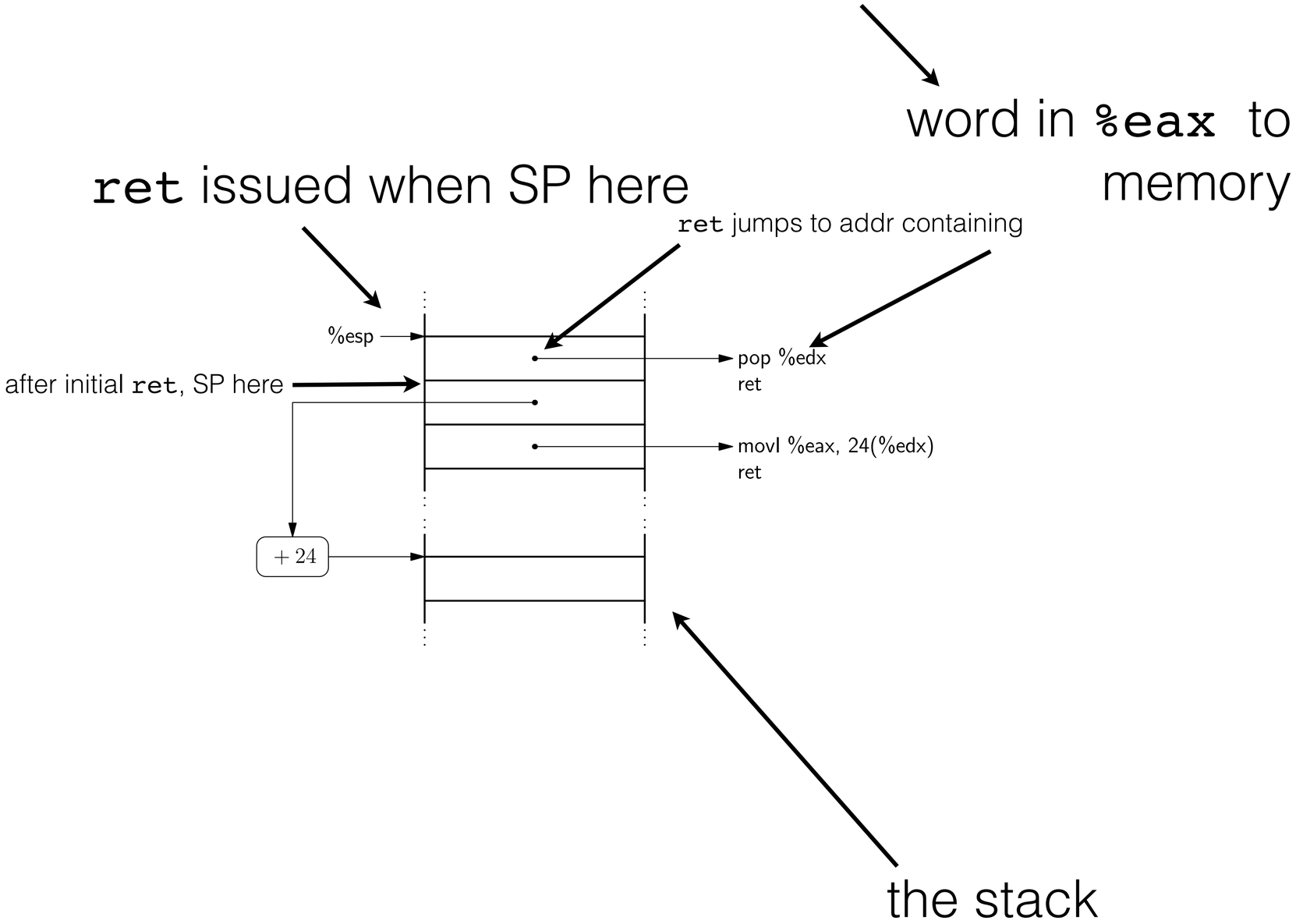
`ret` issued when SP here

`ret` jumps to addr containing

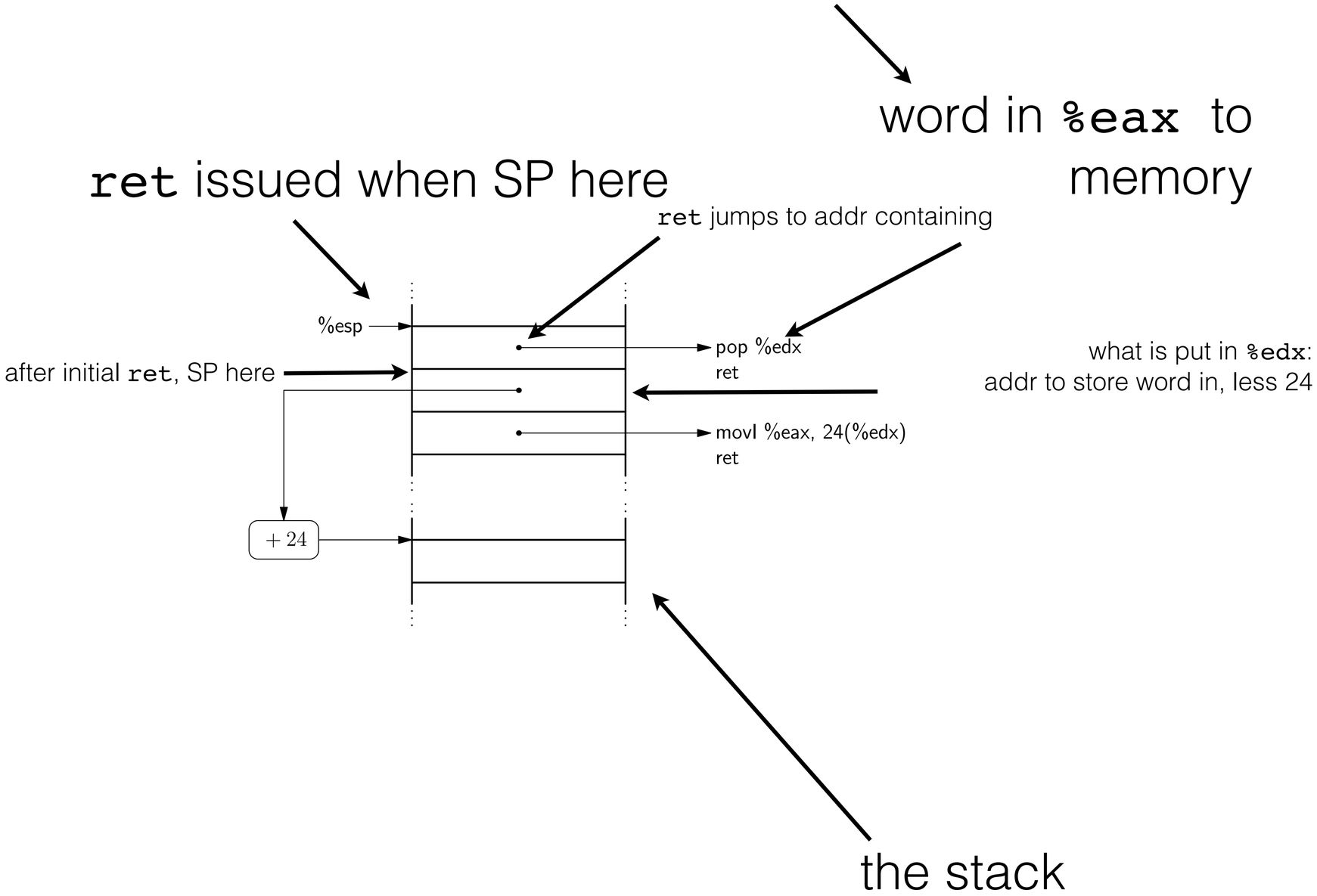


the stack

ROP word store



ROP word store

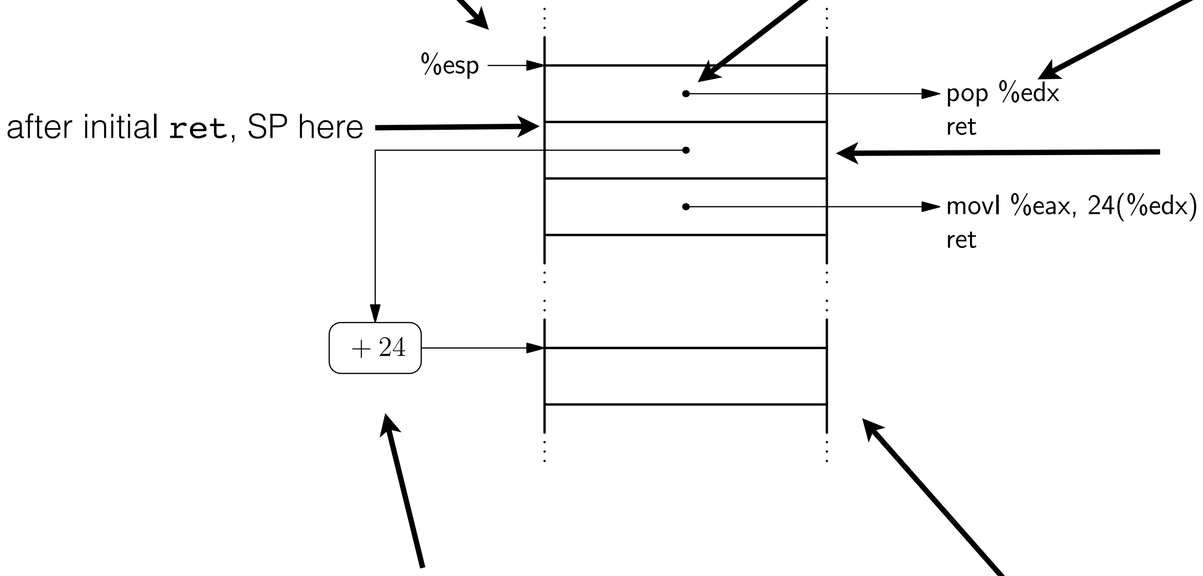


ROP word store

word in `%eax` to memory

`ret` issued when SP here

`ret` jumps to addr containing



what is put in `%edx`:
addr to store word in, less 24

signifies that addr `%edx` points to should be 24 less than desired

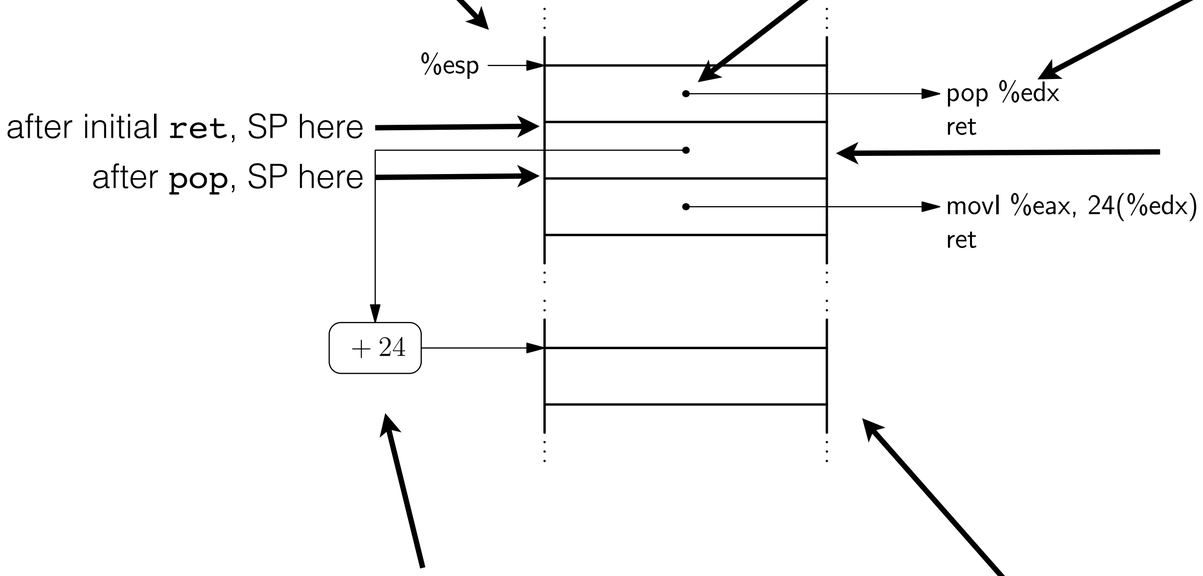
the stack

ROP word store

word in `%eax` to memory

`ret` issued when SP here

`ret` jumps to addr containing



what is put in `%edx`:
addr to store word in, less 24

signifies that addr `%edx` points to should be 24 less than desired

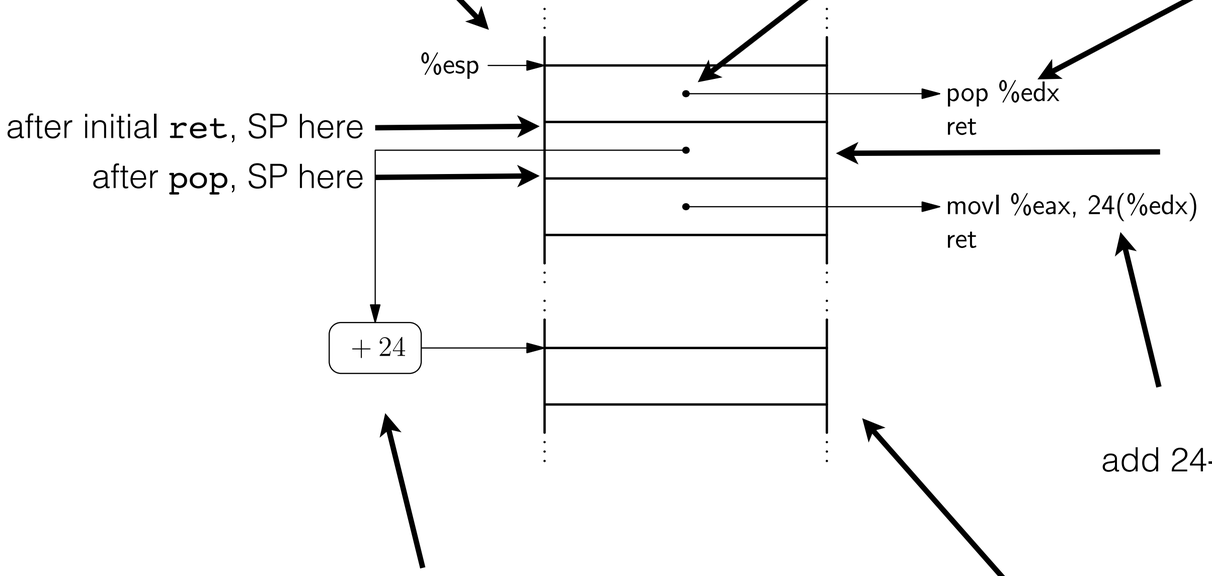
the stack

ROP word store

word in `%eax` to memory

`ret` issued when SP here

`ret` jumps to addr containing



what is put in `%edx`:
addr to store word in, less 24

add 24-bytes to addr `%edx` points to and store `%edx` there

signifies that addr `%edx` points to should be 24 less than desired

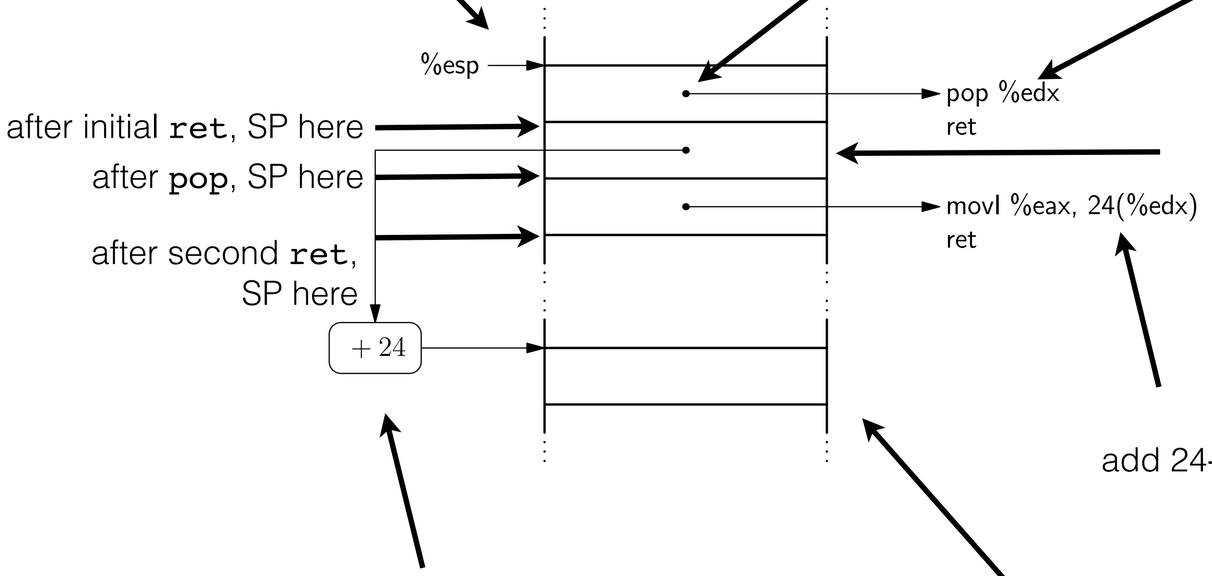
the stack

ROP word store

word in `%eax` to memory

`ret` issued when SP here

`ret` jumps to addr containing



what is put in `%edx`:
addr to store word in, less 24

add 24-bytes to addr `%edx` points to and store `%edx` there

signifies that addr `%edx` points to should be 24 less than desired

the stack

the good news:

can construct sequence of instructions using libc
that are Turing complete

what Turing complete means to an attacker:

arbitrary computation

what Turing complete means to an attacker:

arbitrary computation