ECE 6504

Embedded and Controls Systems Security: Stack Smashing IV

Reading:

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

N: *kBouncer, ROPGuard, ROPecker,* and *Evaluating the Effectiveness of Current Anti-ROP Defenses*

“I intend to live forever, or die trying.”

–Groucho Marx

return-oriented programming (ROP)

setup stack to point to instructions that end in return

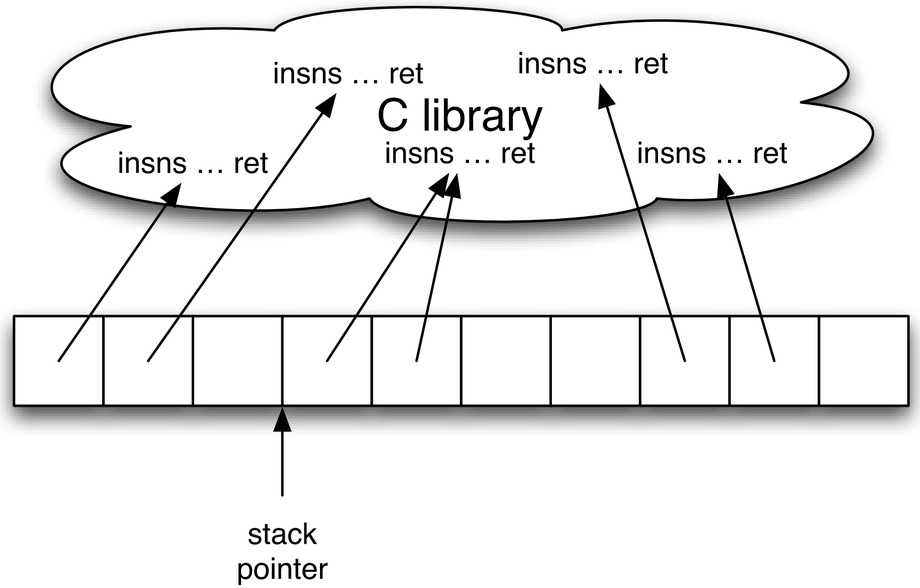
return-oriented programming (ROP)

setup stack to point to instructions that end in return

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

return-oriented programming (ROP)

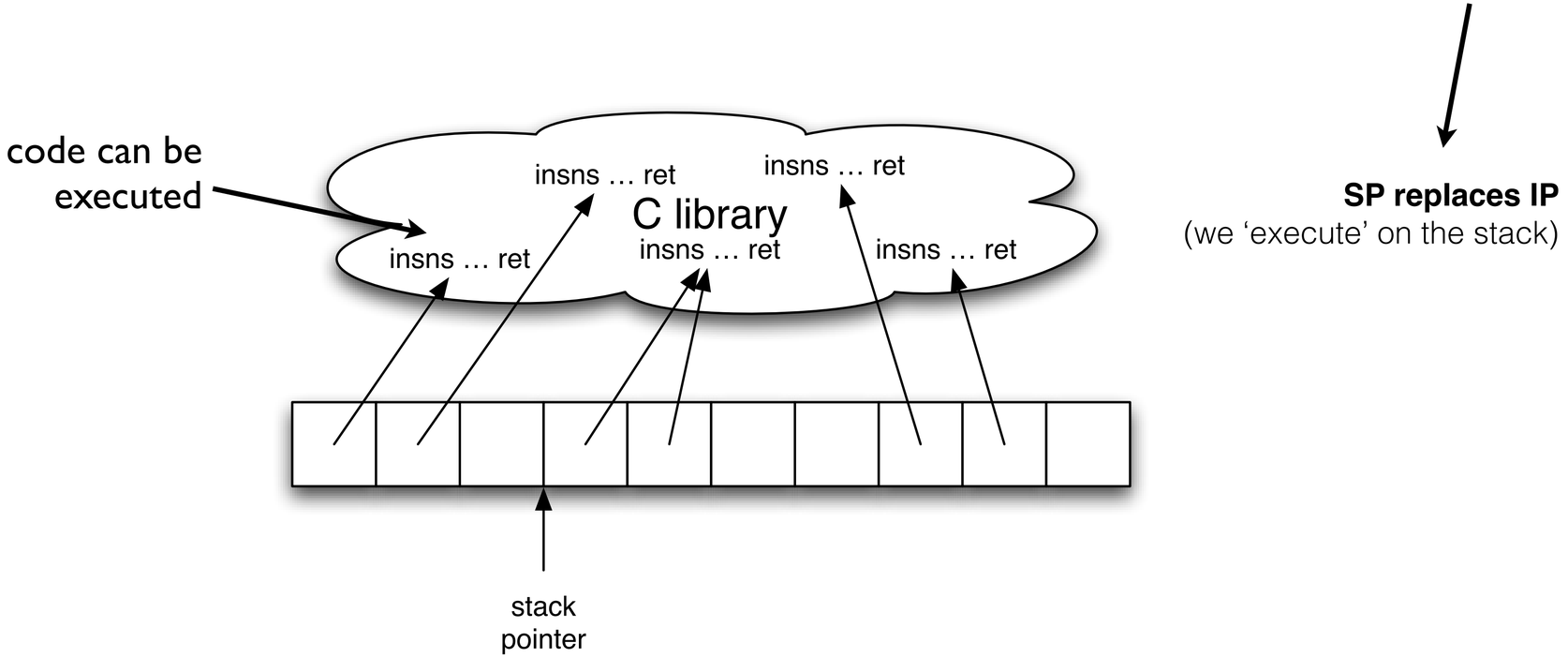
setup stack to point to instructions that end in return



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

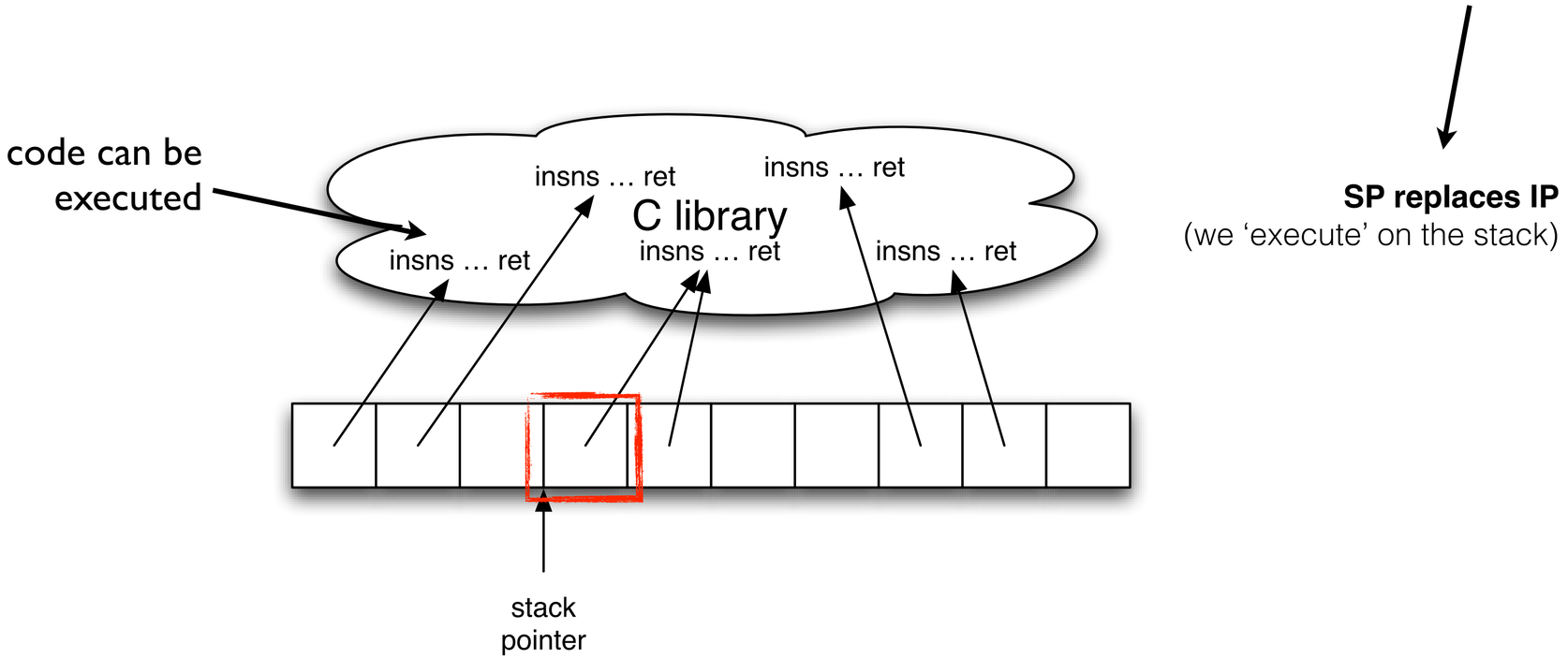
return-oriented programming (ROP)

setup stack to point to instructions that end in return



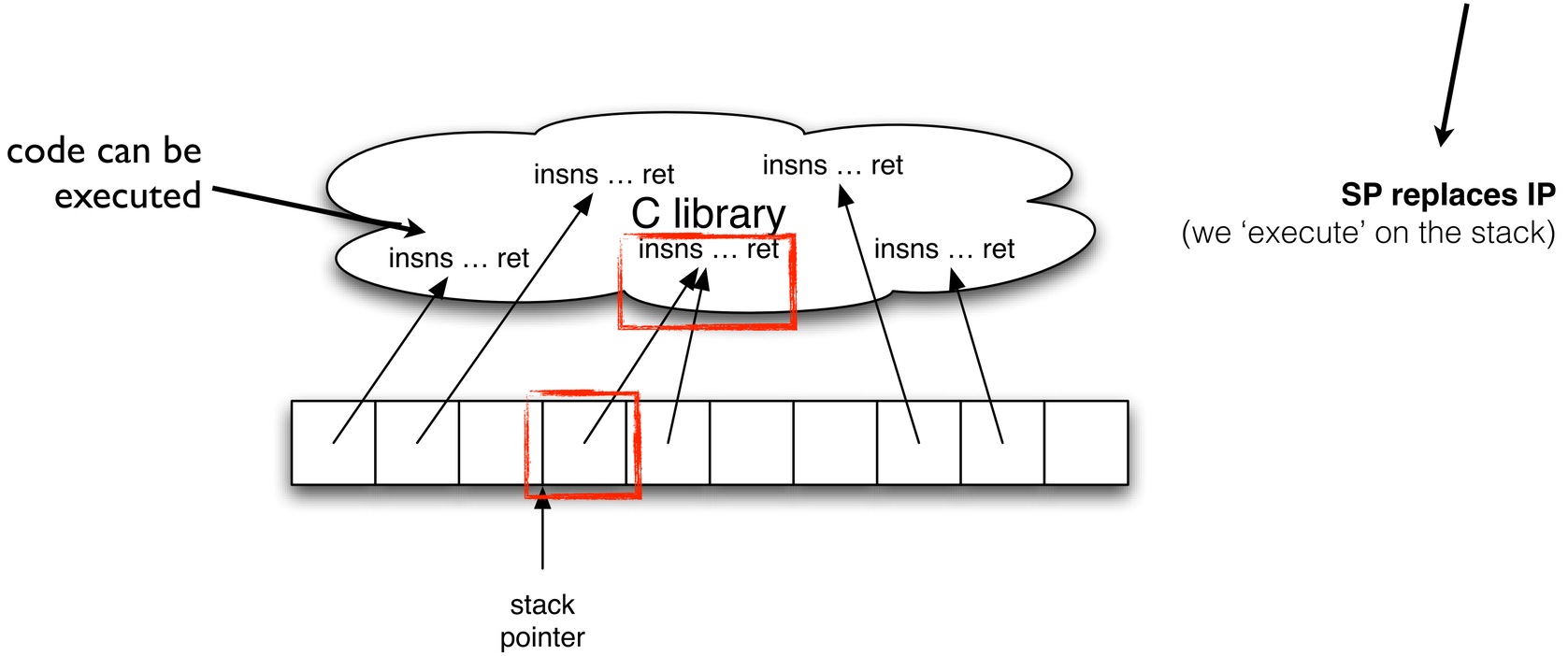
return-oriented programming (ROP)

setup stack to point to instructions that end in return



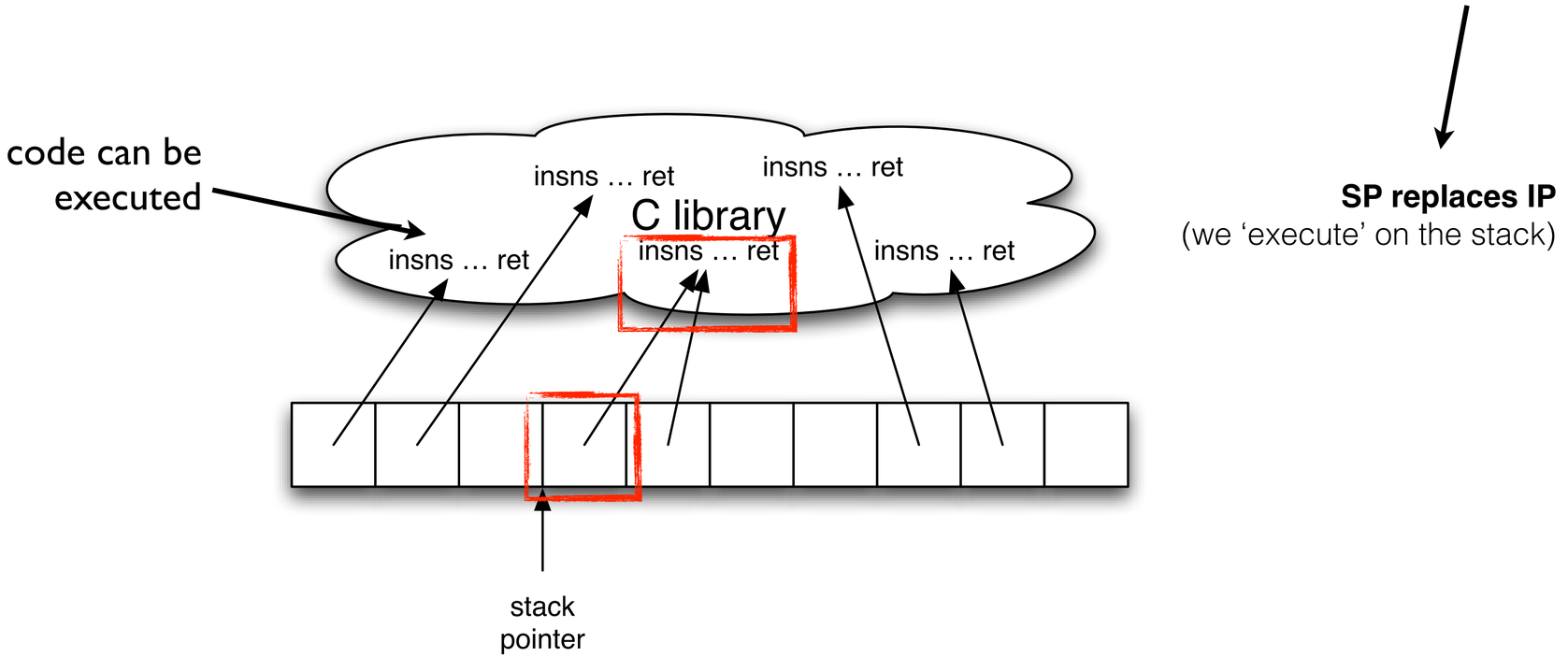
return-oriented programming (ROP)

setup stack to point to instructions that end in return



return-oriented programming (ROP)

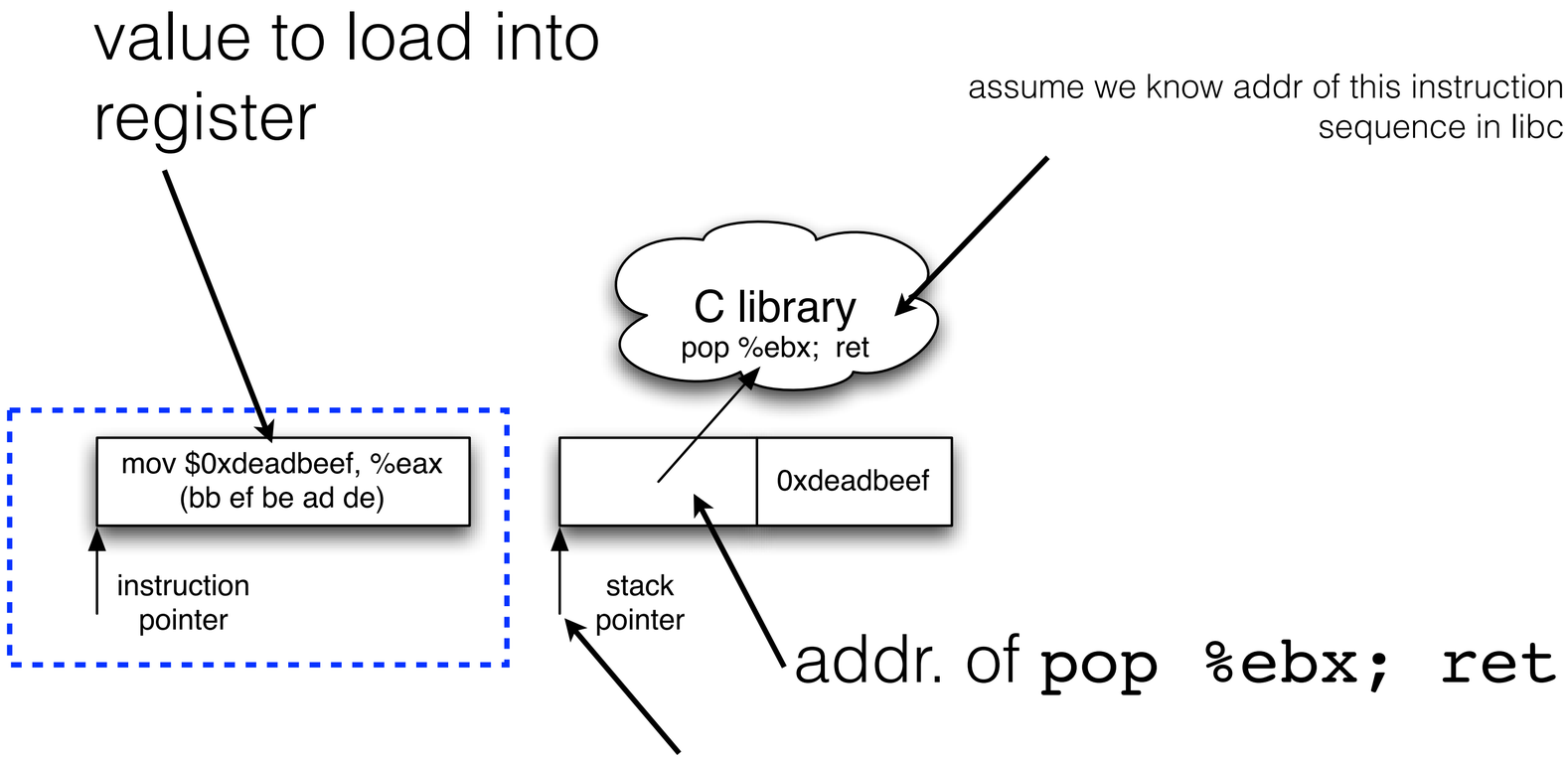
setup stack to point to instructions that end in return



procedure:

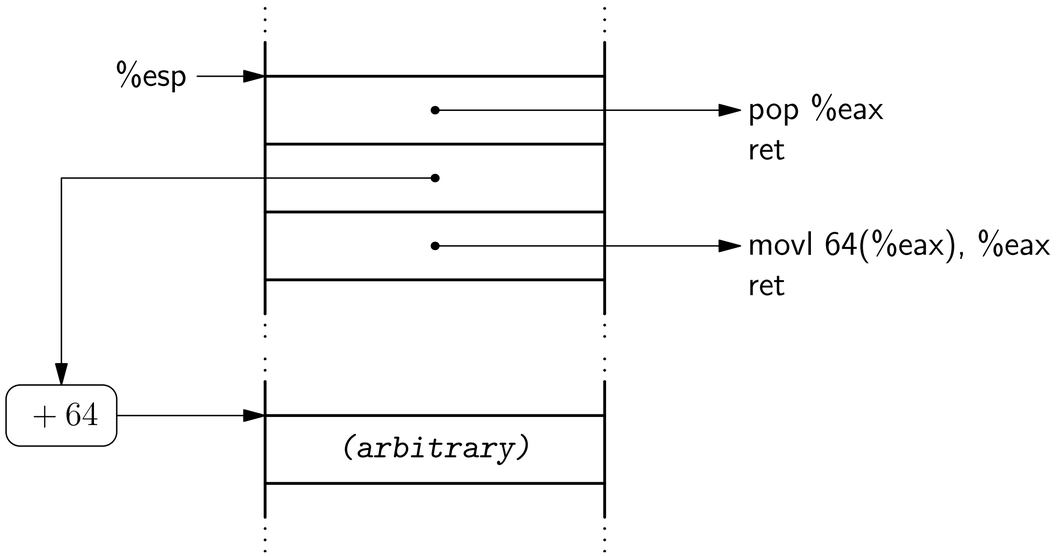
1. setup stack via buffer overflow
2. issue return
3. pop addr from stack
4. IP executes instruction at addr

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

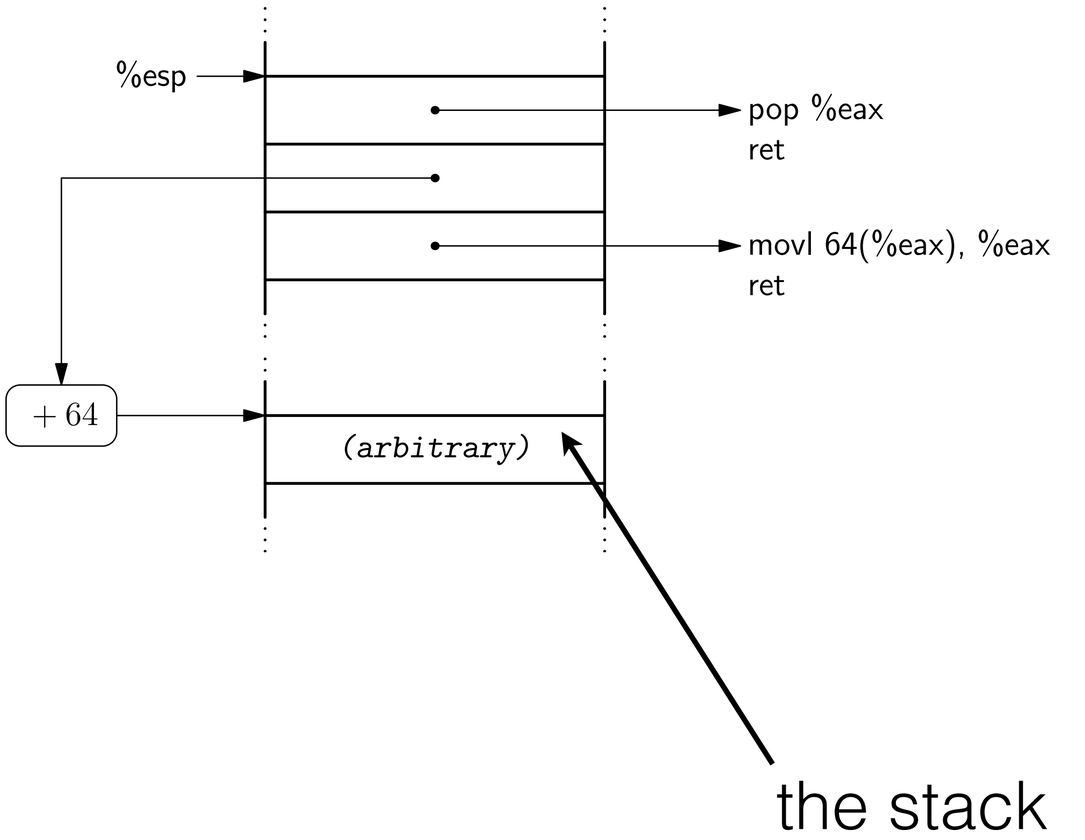
ROP word load



dot represents constructed stack

ROP word load

word into `%eax`

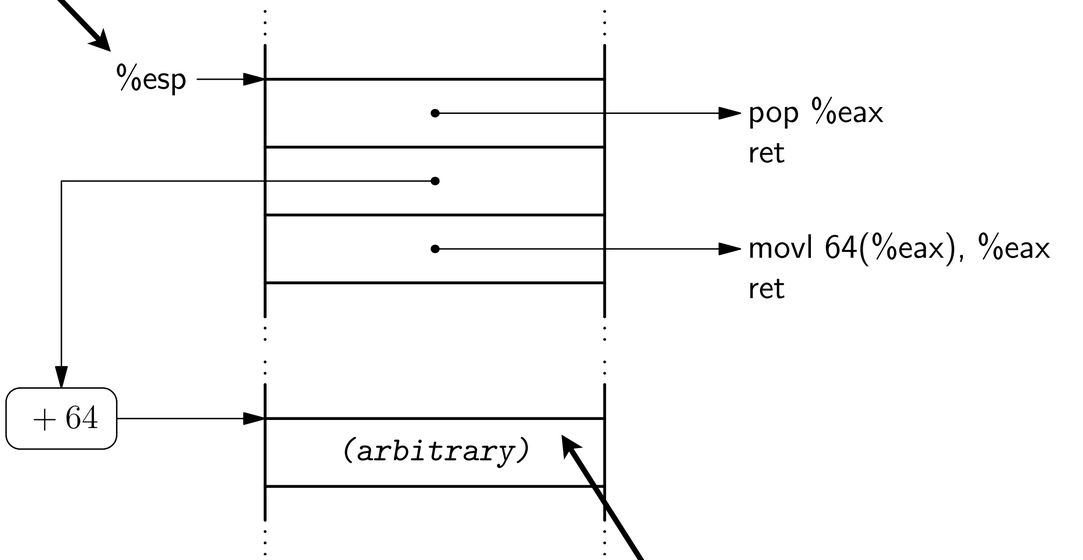


dot represents constructed stack

ROP word load

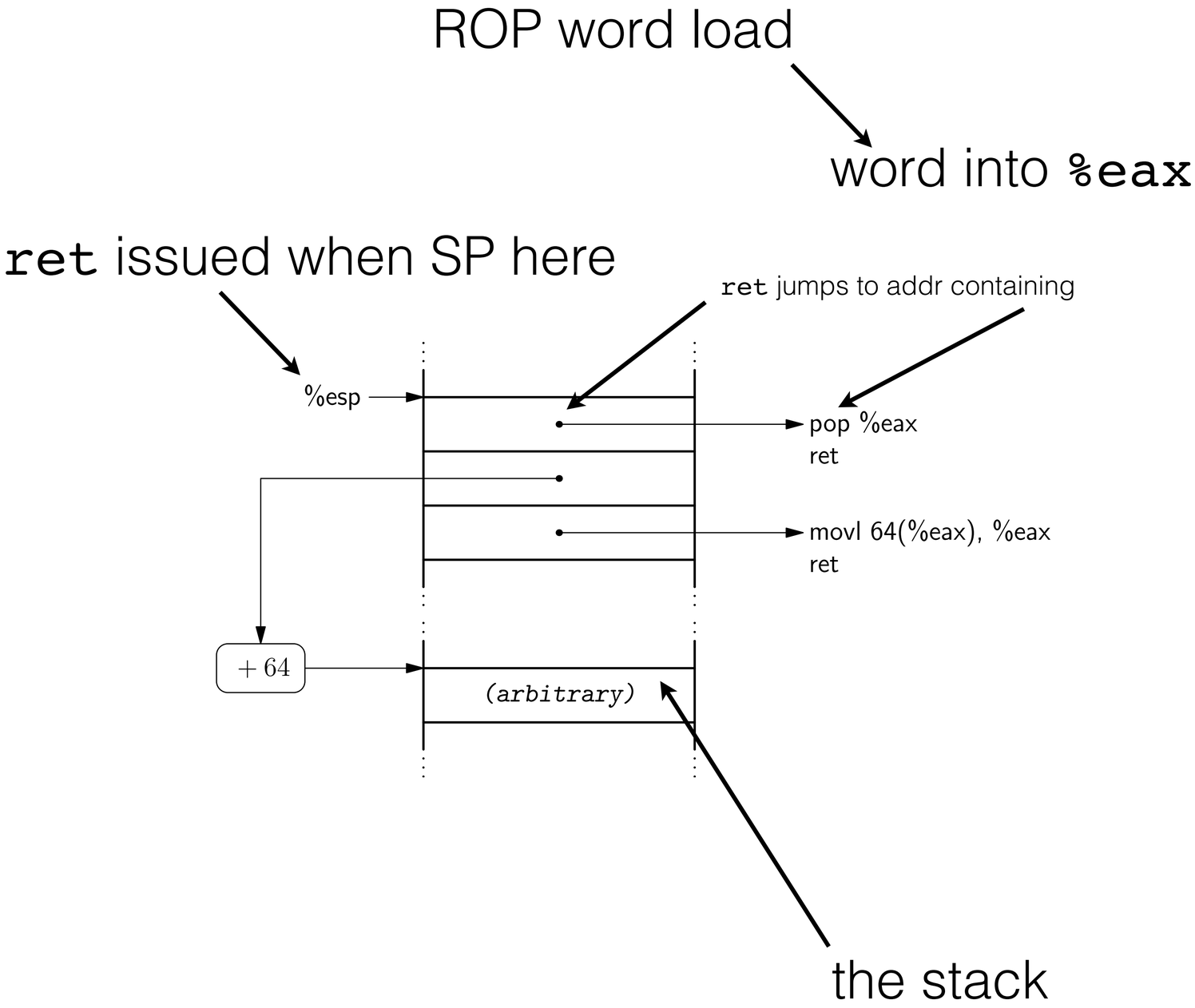
word into `%eax`

`ret` issued when SP here



the stack

dot represents constructed stack

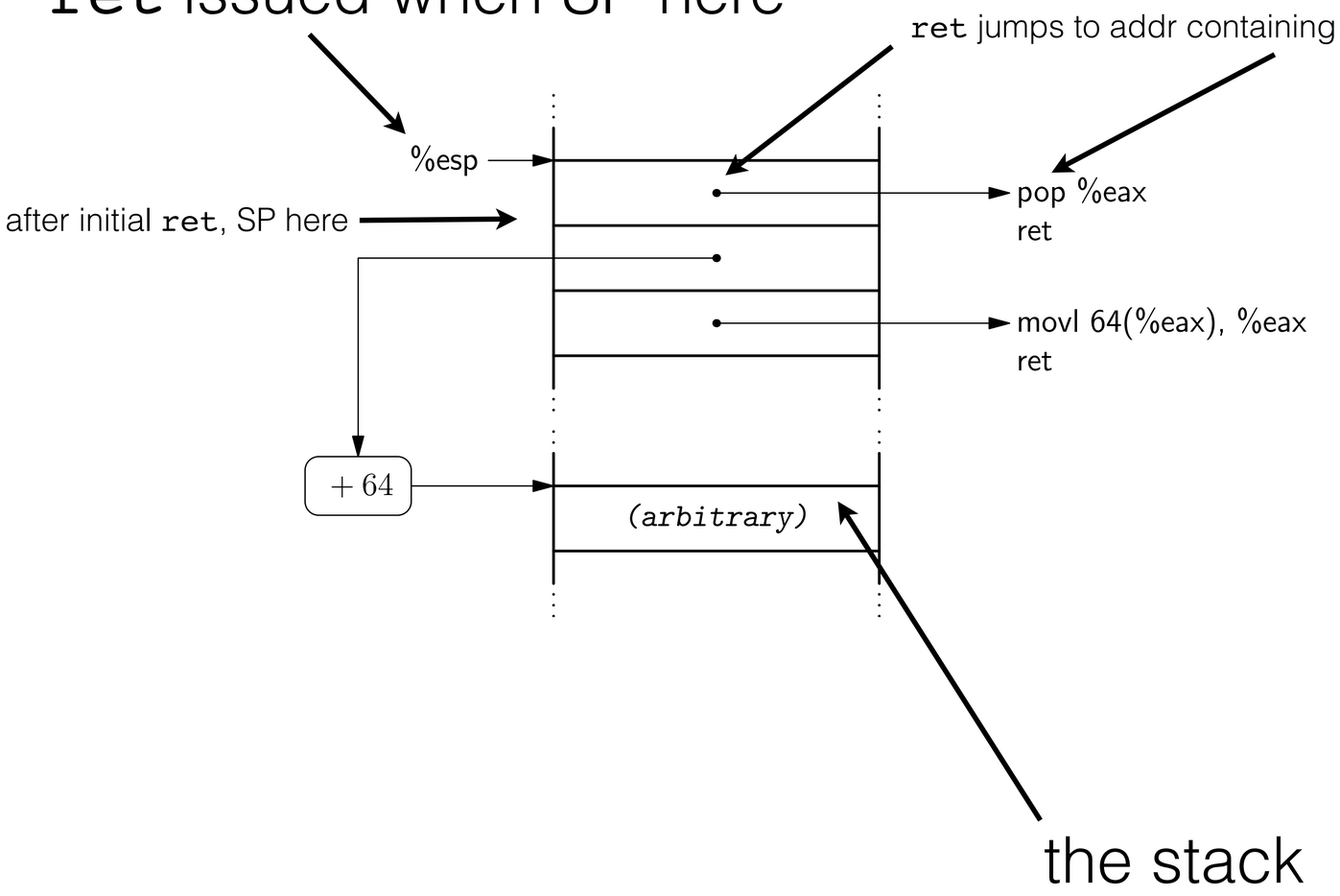


dot represents constructed stack

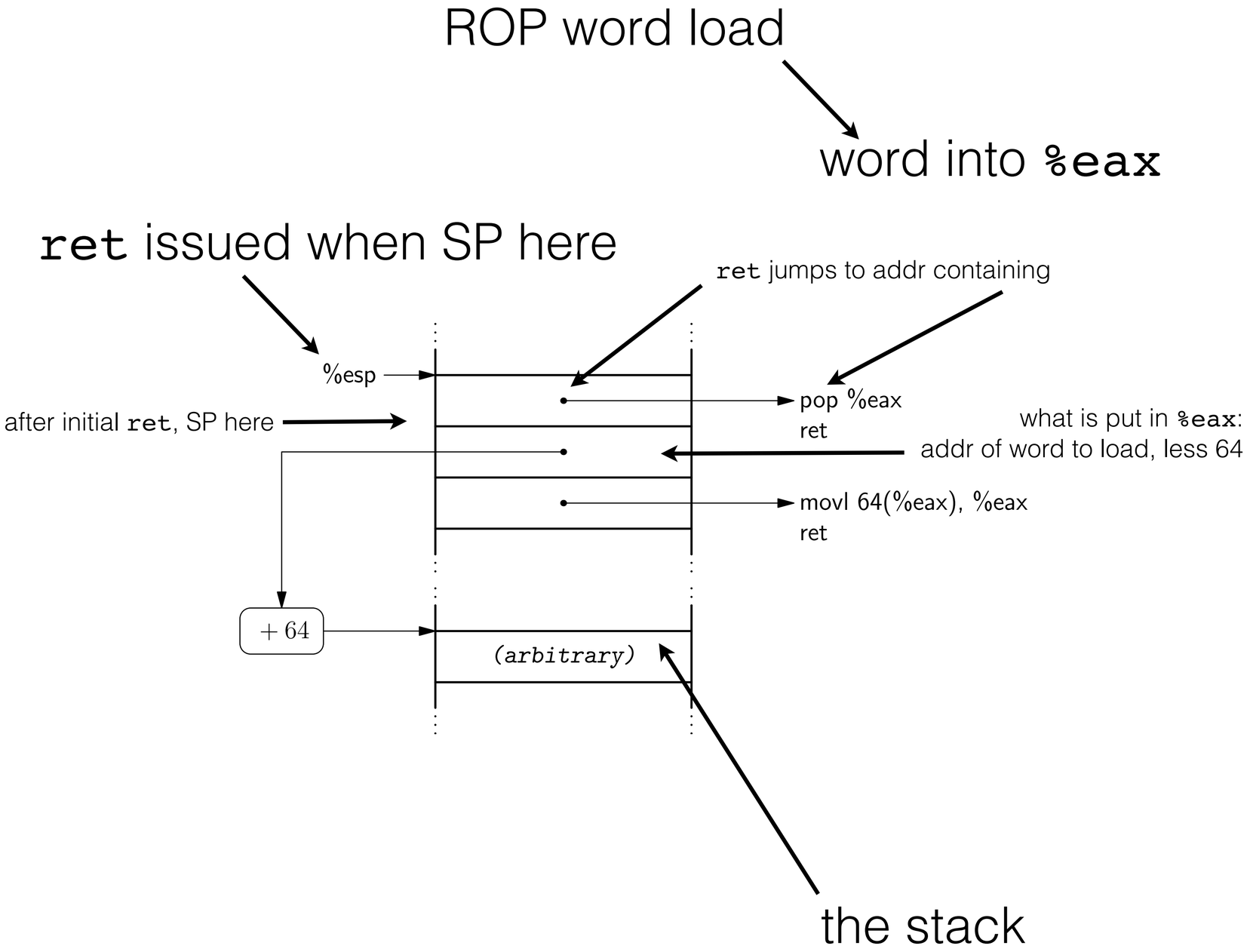
ROP word load

word into `%eax`

`ret` issued when SP here



dot represents constructed stack

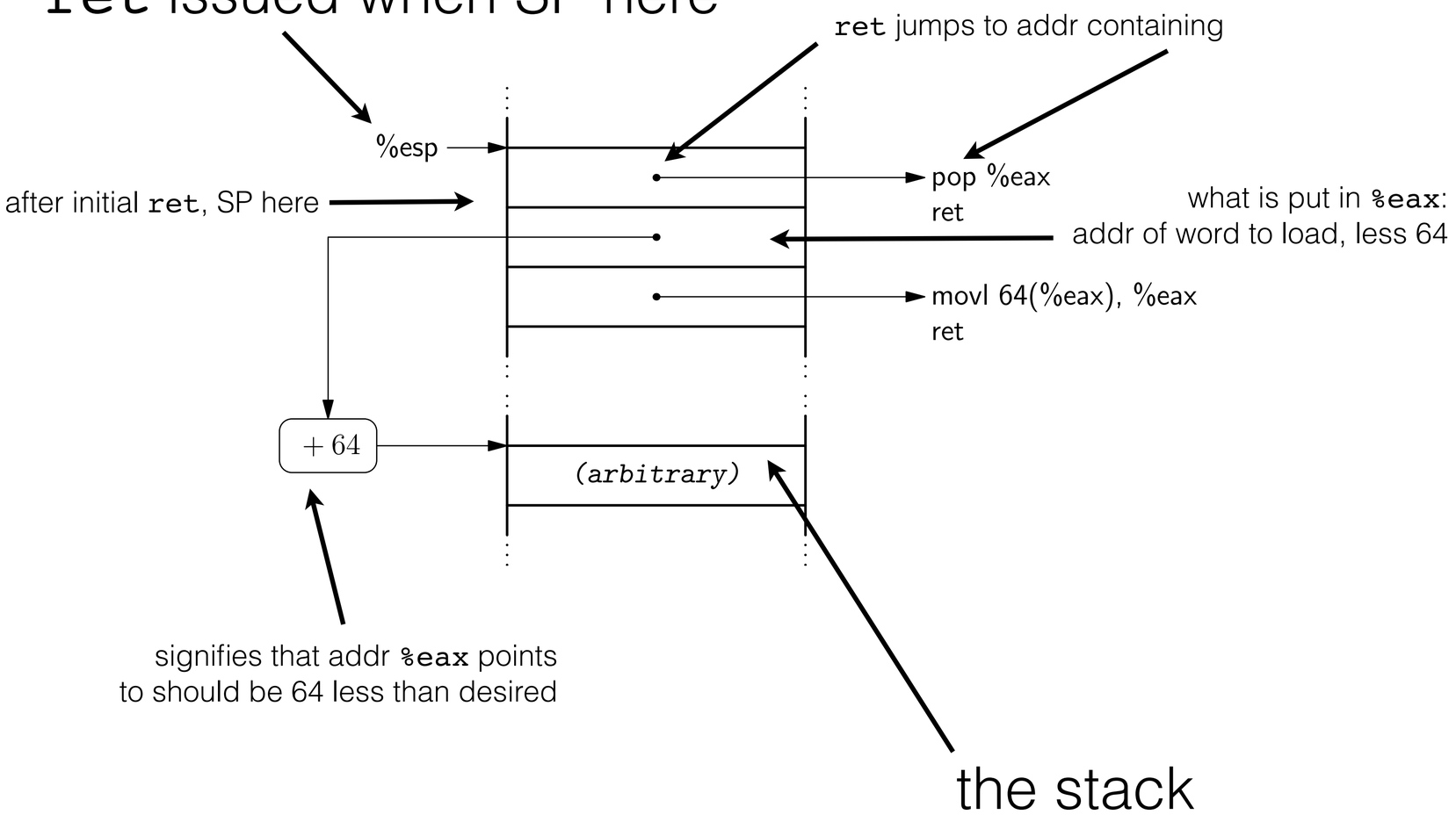


dot represents constructed stack

ROP word load

word into `%eax`

`ret` issued when SP here

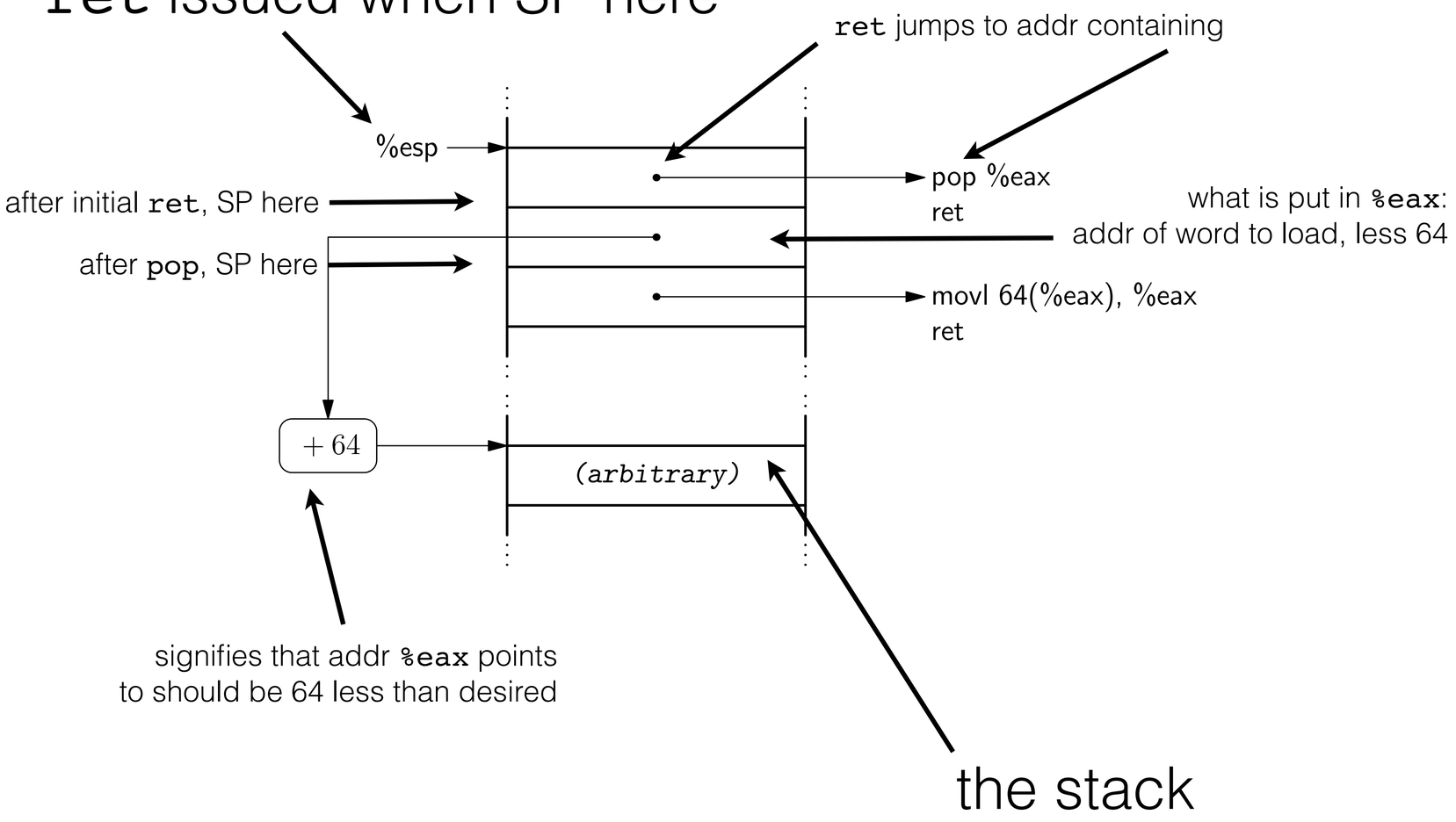


dot represents constructed stack

ROP word load

word into `%eax`

`ret` issued when SP here

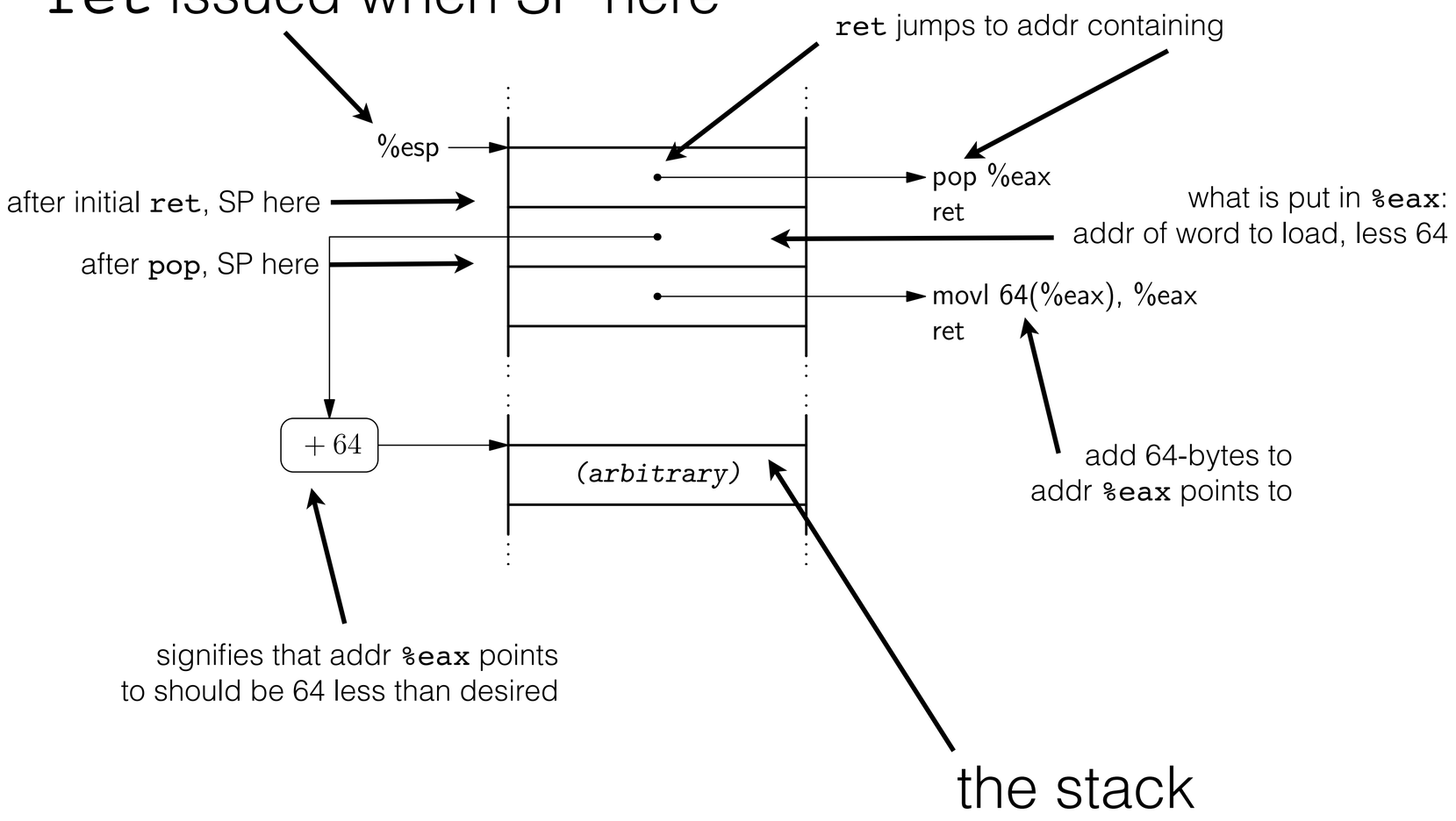


dot represents constructed stack

ROP word load

word into `%eax`

`ret` issued when SP here

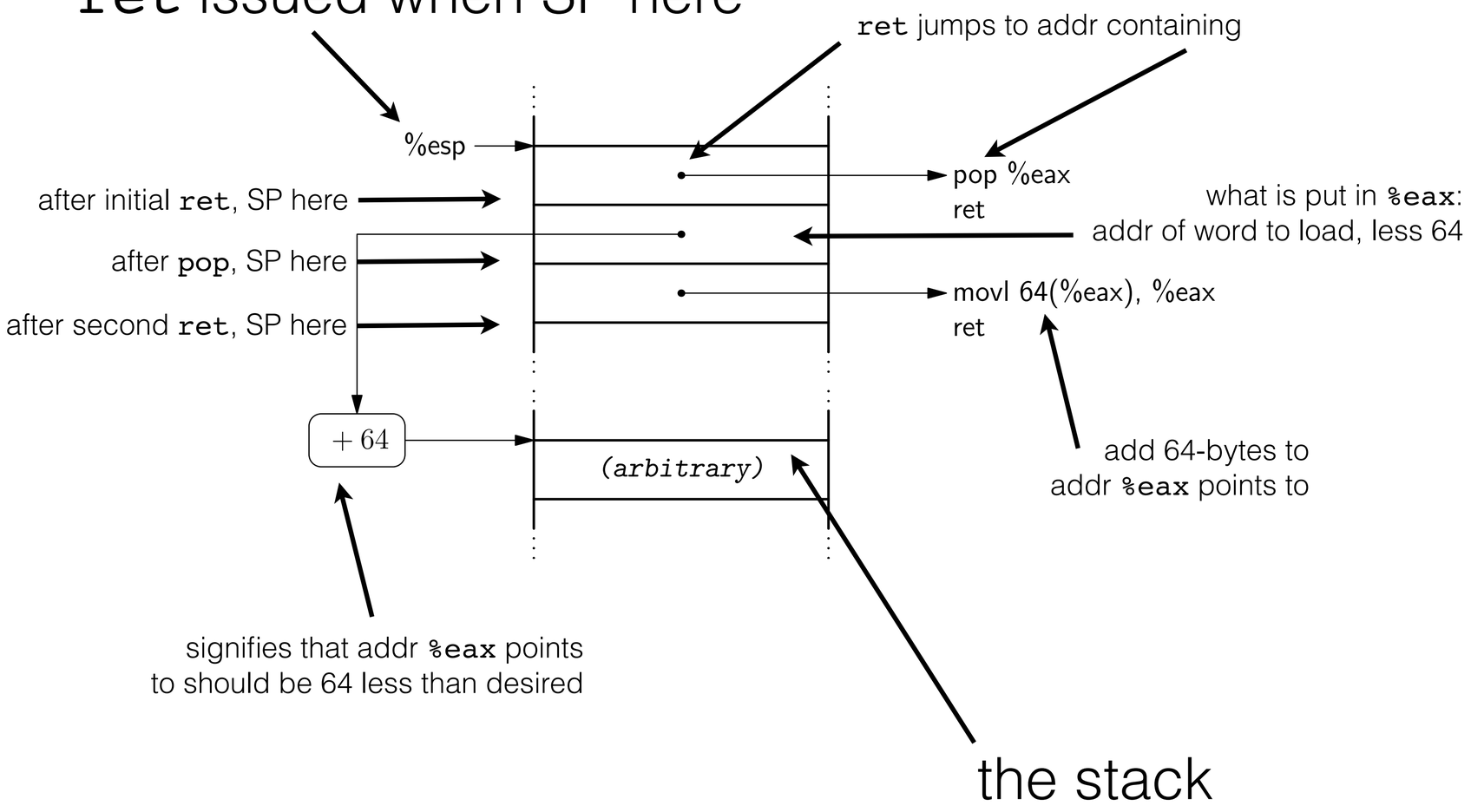


dot represents constructed stack

ROP word load

word into `%eax`

`ret` issued when SP here



dot represents constructed stack

can sequence gadgets s.t. Turing complete instruction set...

Q: is this necessary for exploitation in all instances?

A: no, let's build gadgets for what we need (shellcode)

can sequence gadgets s.t. Turing complete instruction set...

Q: is this necessary for exploitation in all instances?

A: no, let's build gadgets for what we need (shellcode)

system calls

`umask` system call:

```
89 da                   mov %ebx, %edx
8b 5c 24 04             movl 4(%esp), %ebx
b8 3c 00 00 00          mov $0x0000003C, %eax
65 ff 15 10 00 00 00    lcall %gs:0x10(,0)
89 d3                   mov %edx, %ebx
c3                      ret
```

system call wrappers in **`libc`**

system calls

`umask` system call:

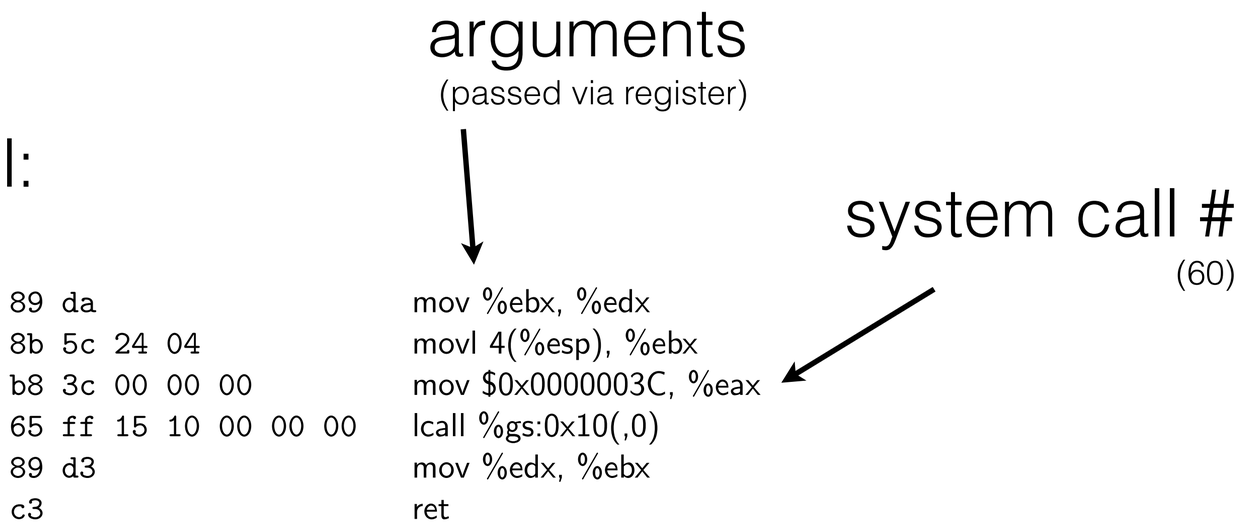
arguments
(passed via register)

```
89 da                   mov %ebx, %edx
8b 5c 24 04             movl 4(%esp), %ebx
b8 3c 00 00 00          mov $0x0000003C, %eax
65 ff 15 10 00 00 00    lcall %gs:0x10(,0)
89 d3                   mov %edx, %ebx
c3                      ret
```

system call wrappers in **`libc`**

system calls

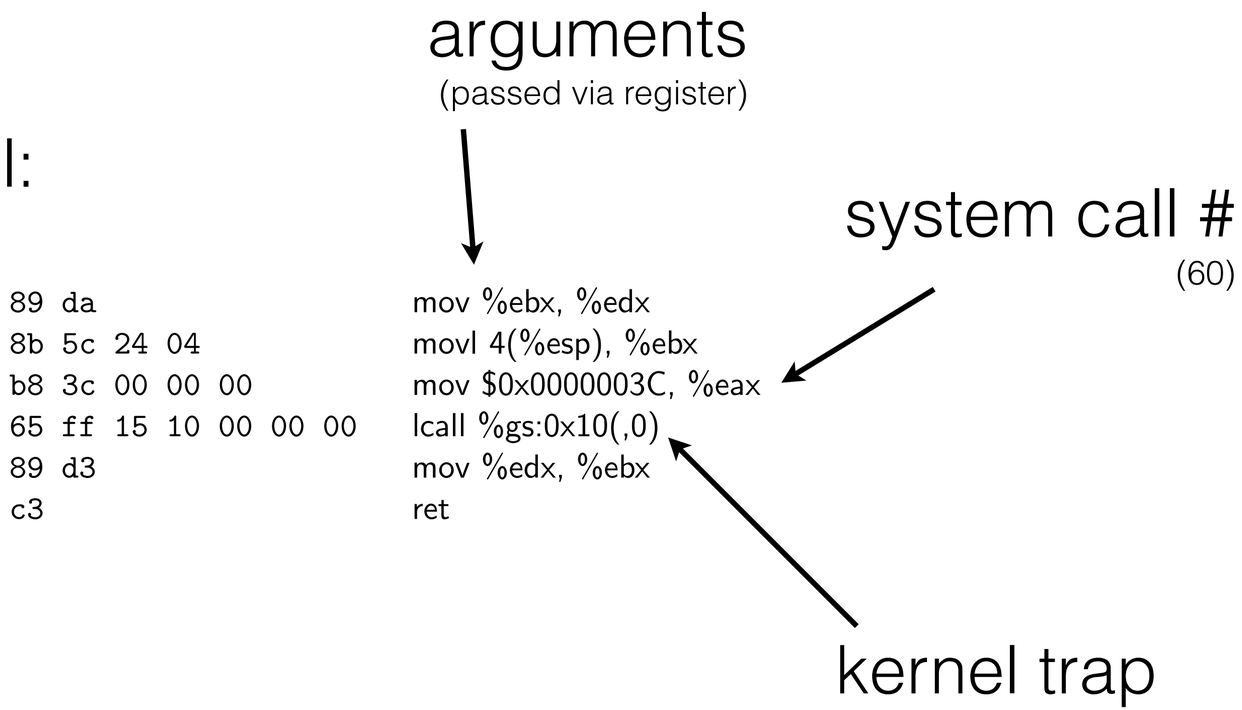
`umask` system call:



system call wrappers in **`libc`**

system calls

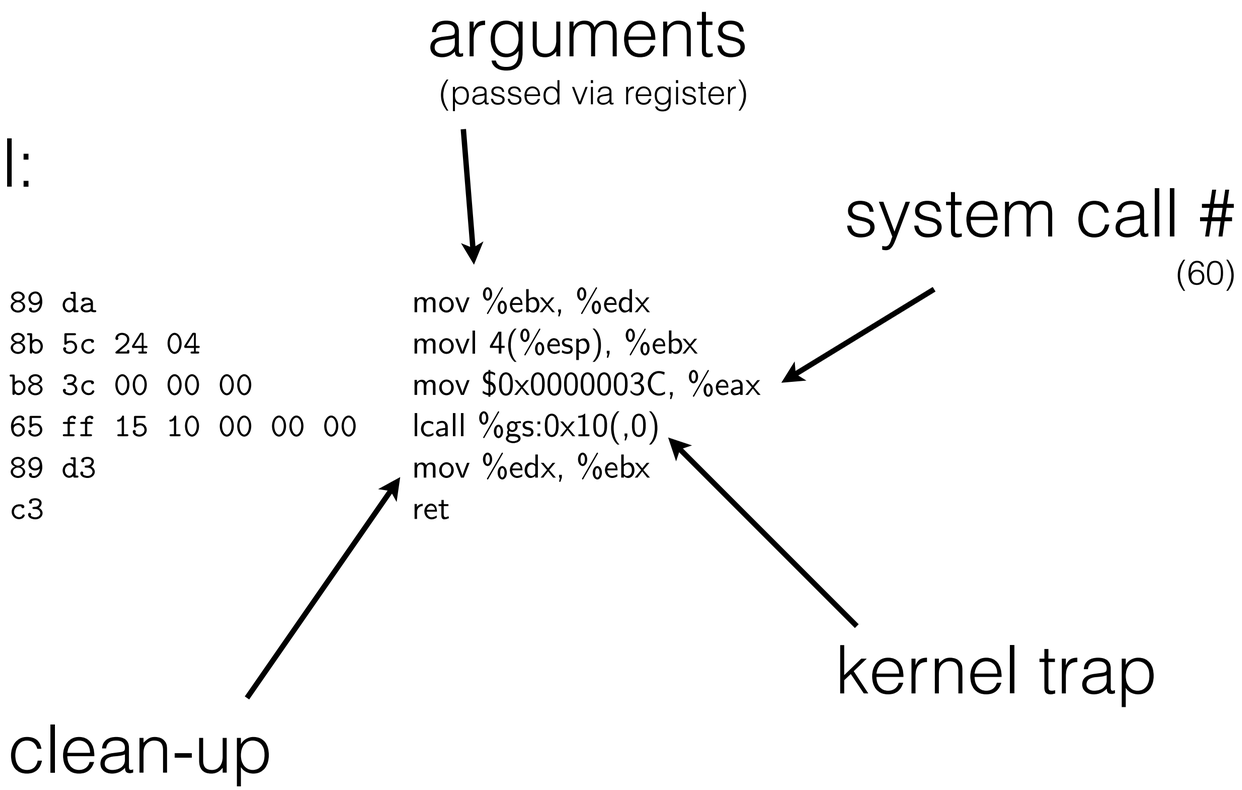
`umask` system call:



system call wrappers in **`libc`**

system calls

`umask` system call:



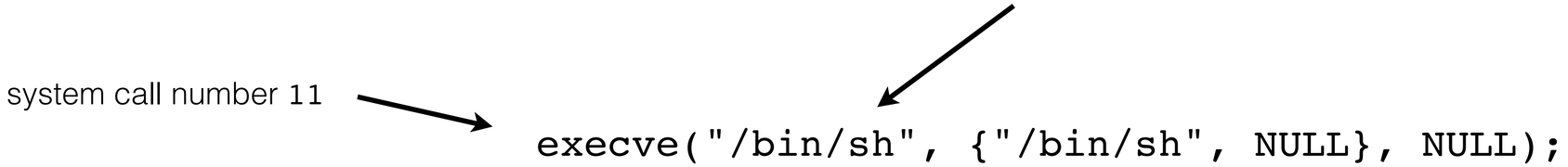
system call wrappers in **`libc`**

constructing ROP shellcode

system call number 11

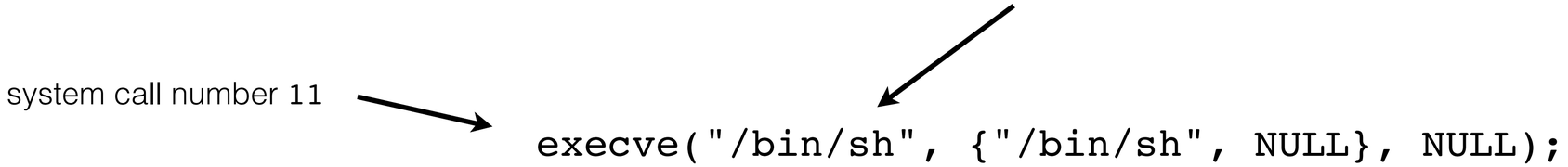
```
execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

constructing ROP shellcode



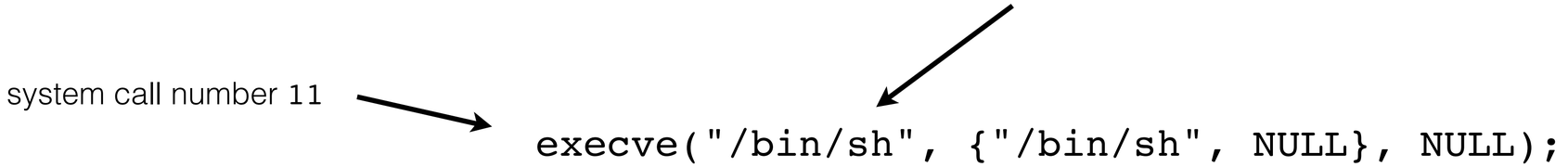
NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

constructing ROP shellcode



NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

constructing ROP shellcode

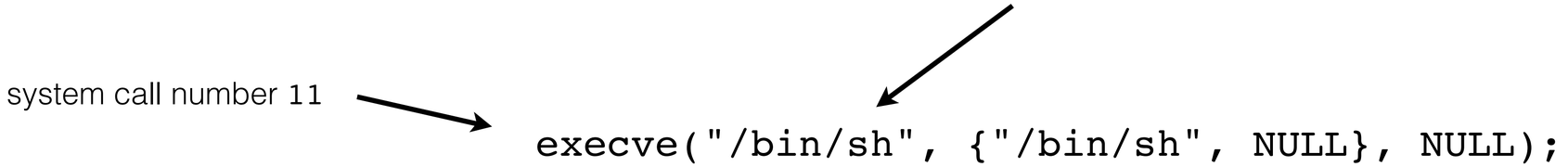


NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

path to program

args for program

constructing ROP shellcode



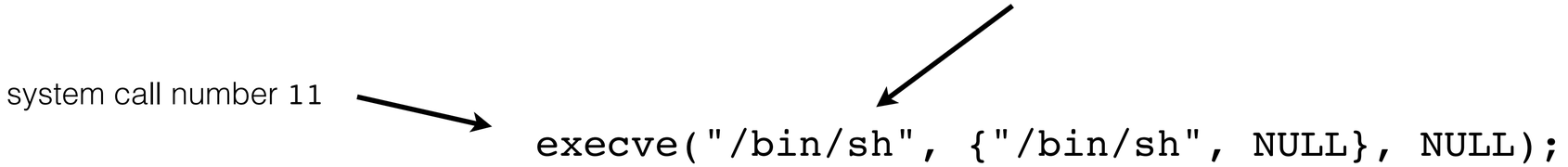
NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

path to program

args for program

env vars. for program

constructing ROP shellcode



NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

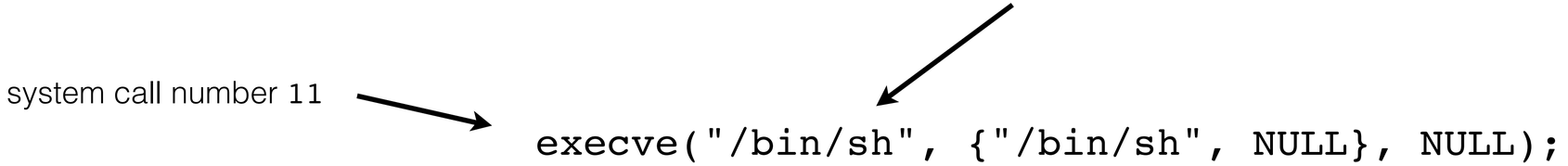
path to program

args for program

env vars. for program

procedure:

constructing ROP shellcode



NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

path to program

args for program

env vars. for program

procedure:

1. call # (`0xb`) into `%eax`

constructing ROP shellcode

system call number `11`

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

NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

path to program

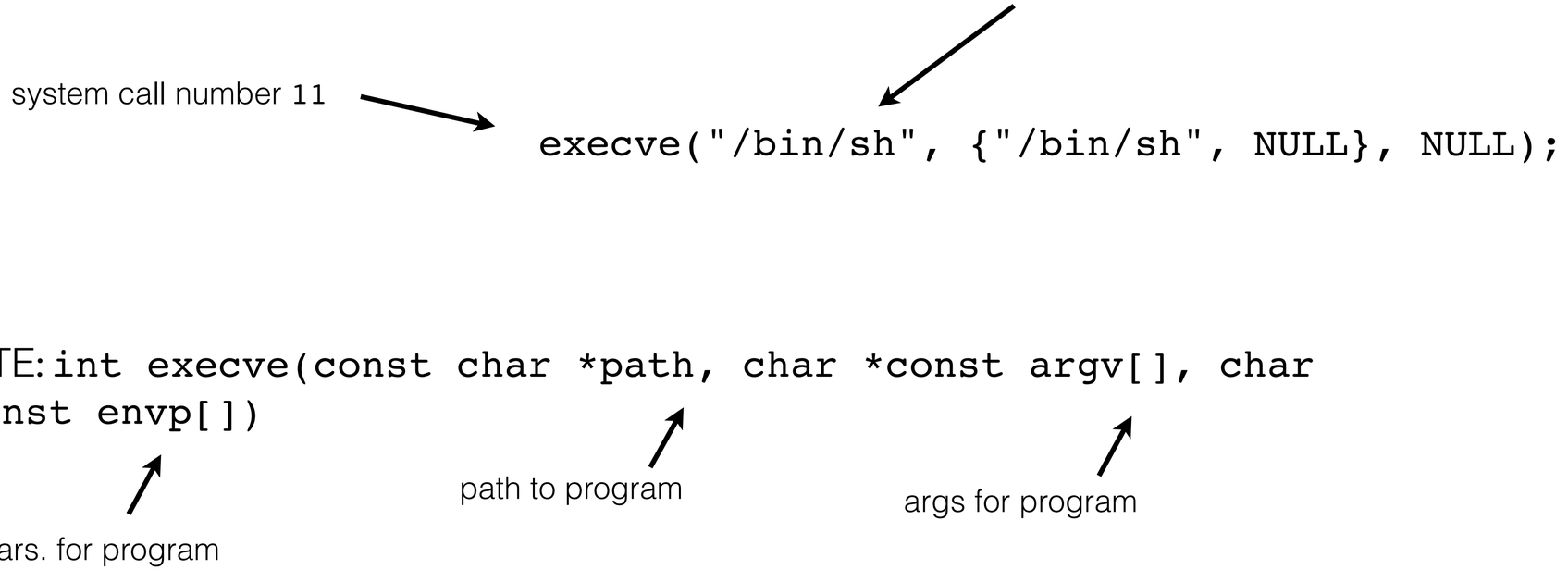
args for program

env vars. for program

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (‘`/bin/sh`’) in `%ebx`

constructing ROP shellcode



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (‘`/bin/sh`’) in `%ebx`
3. set argv: pointer to ptr. array (‘`/bin/sh`’ and `NULL`) in `%ecx`

constructing ROP shellcode

system call number `11`

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

NOTE: `int execve(const char *path, char *const argv[], char *const envp[])`

path to program

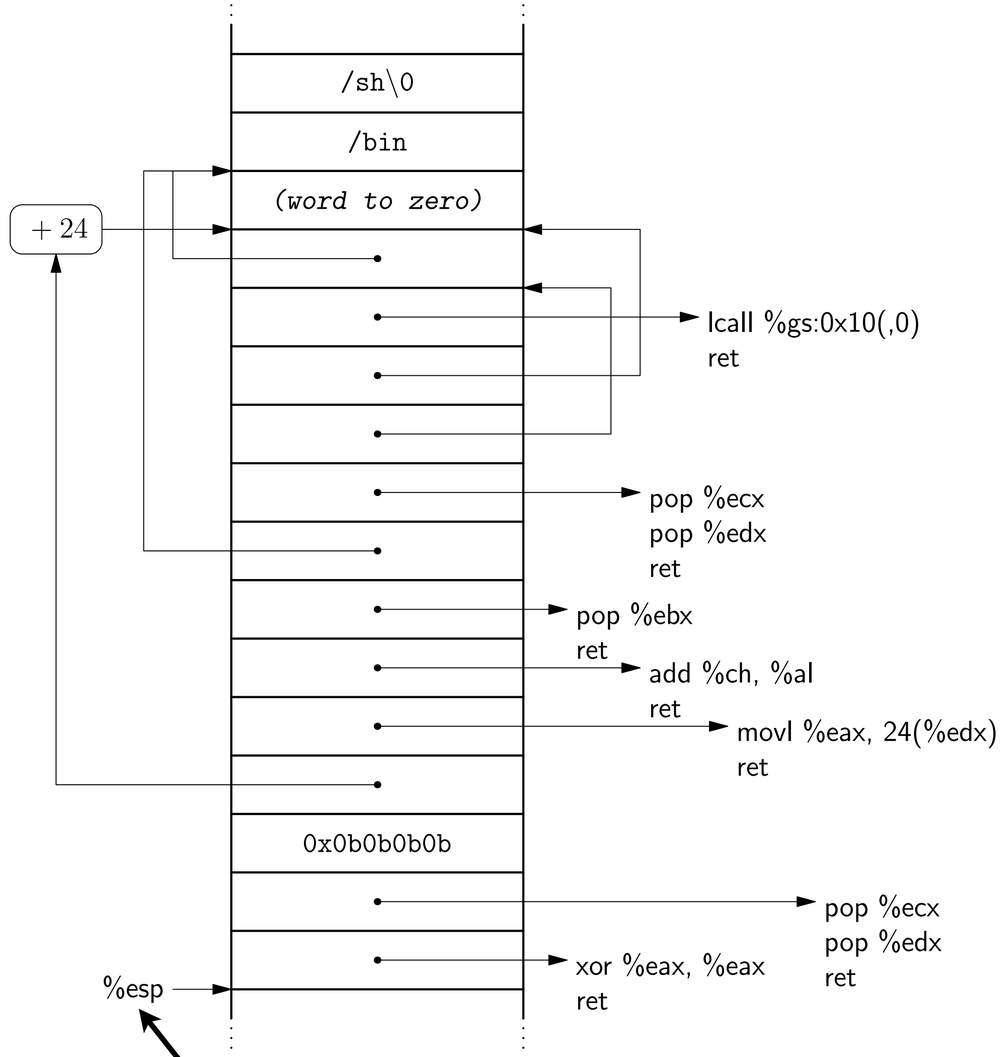
args for program

env vars. for program

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (‘`/bin/sh`’) in `%ebx`
3. set argv: pointer to ptr. array (‘`/bin/sh`’ and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

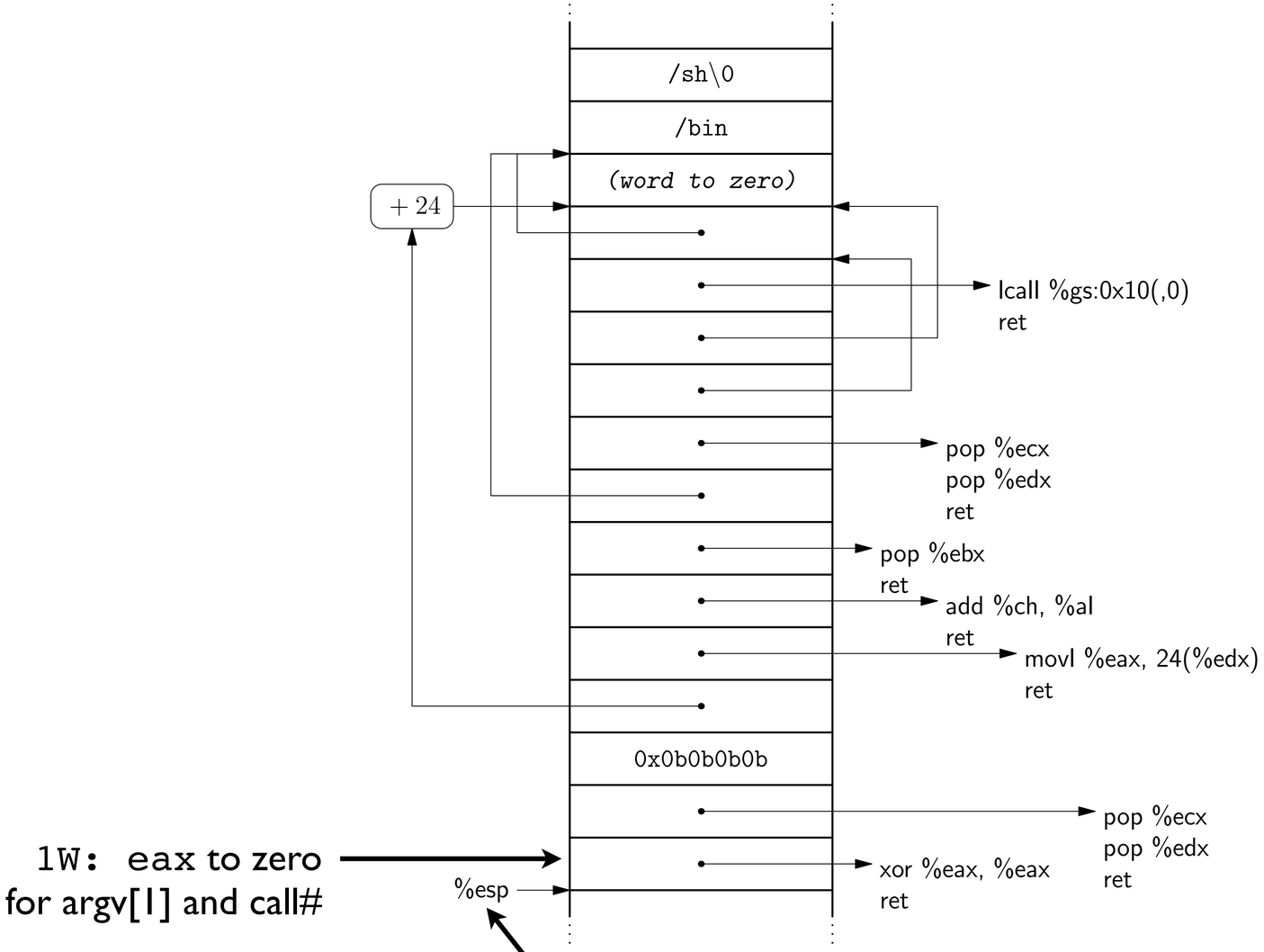


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

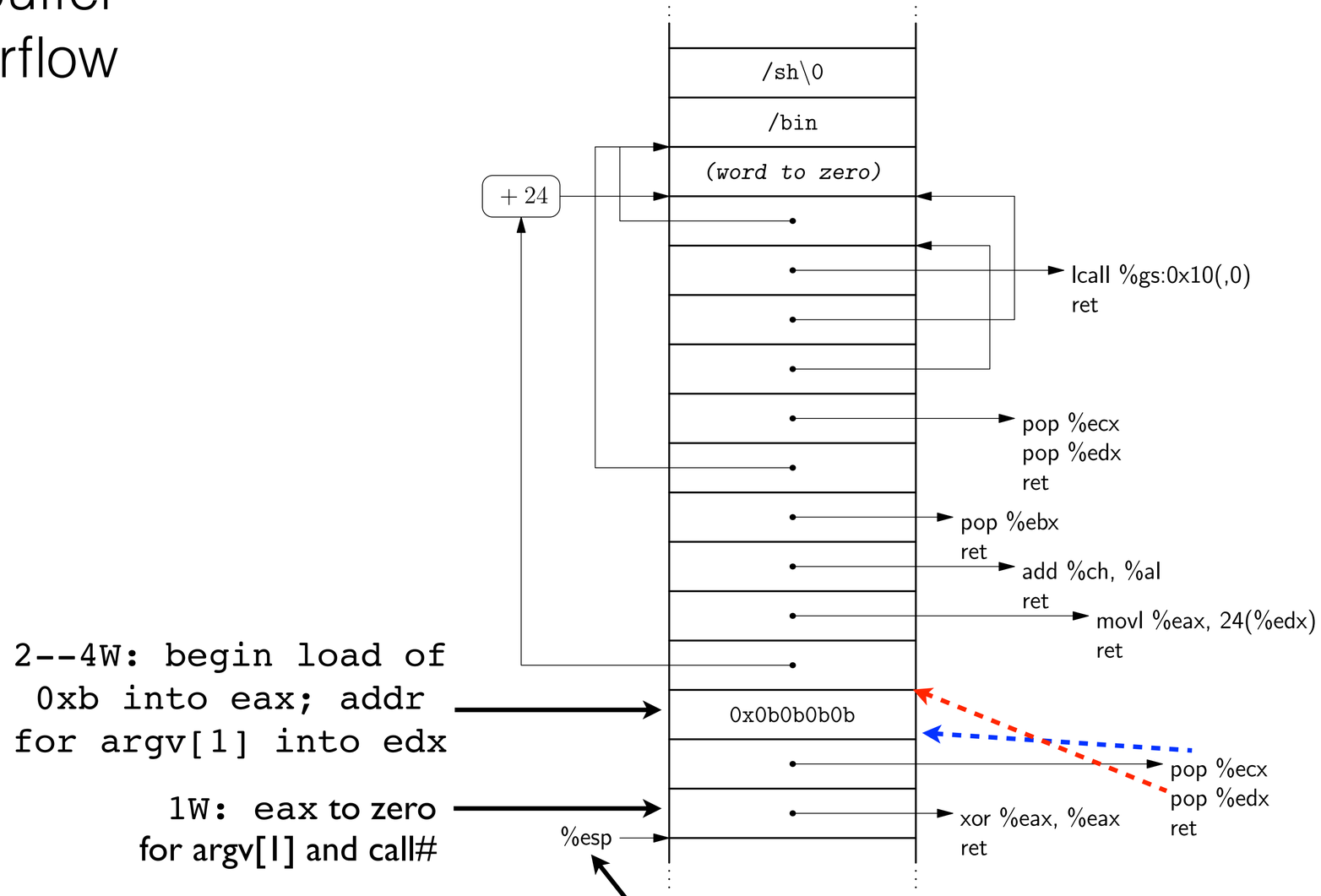
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

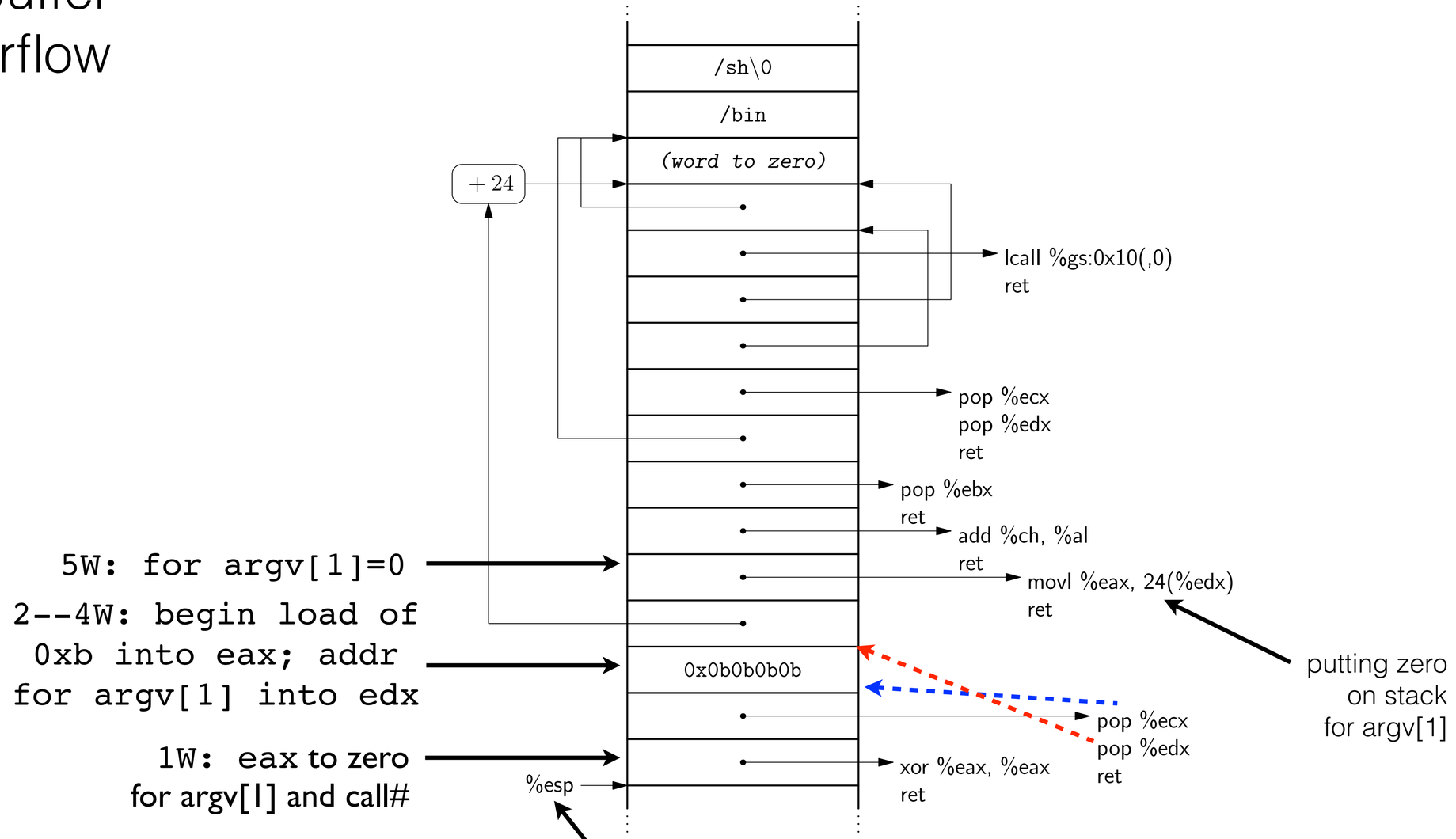
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

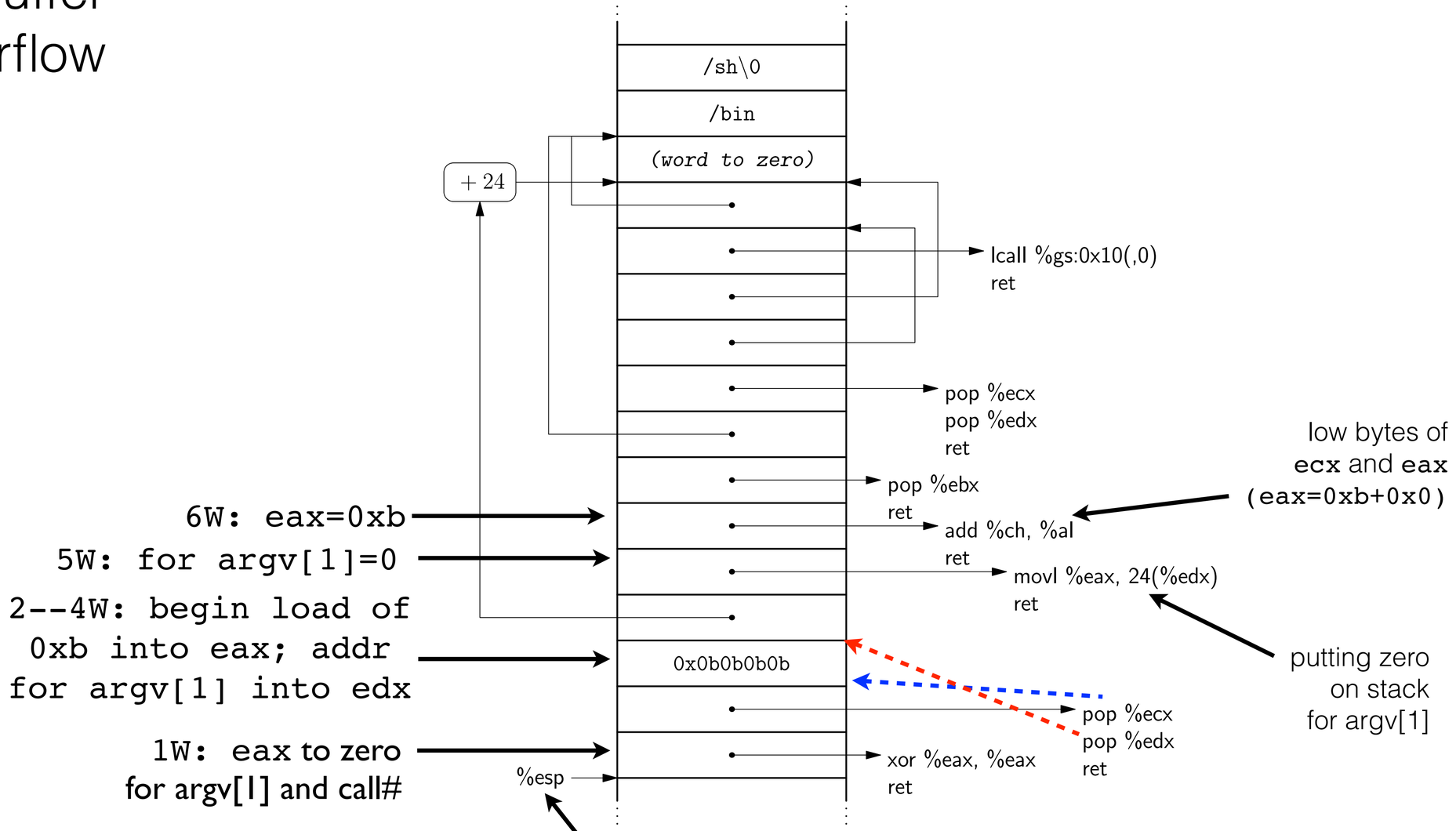
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

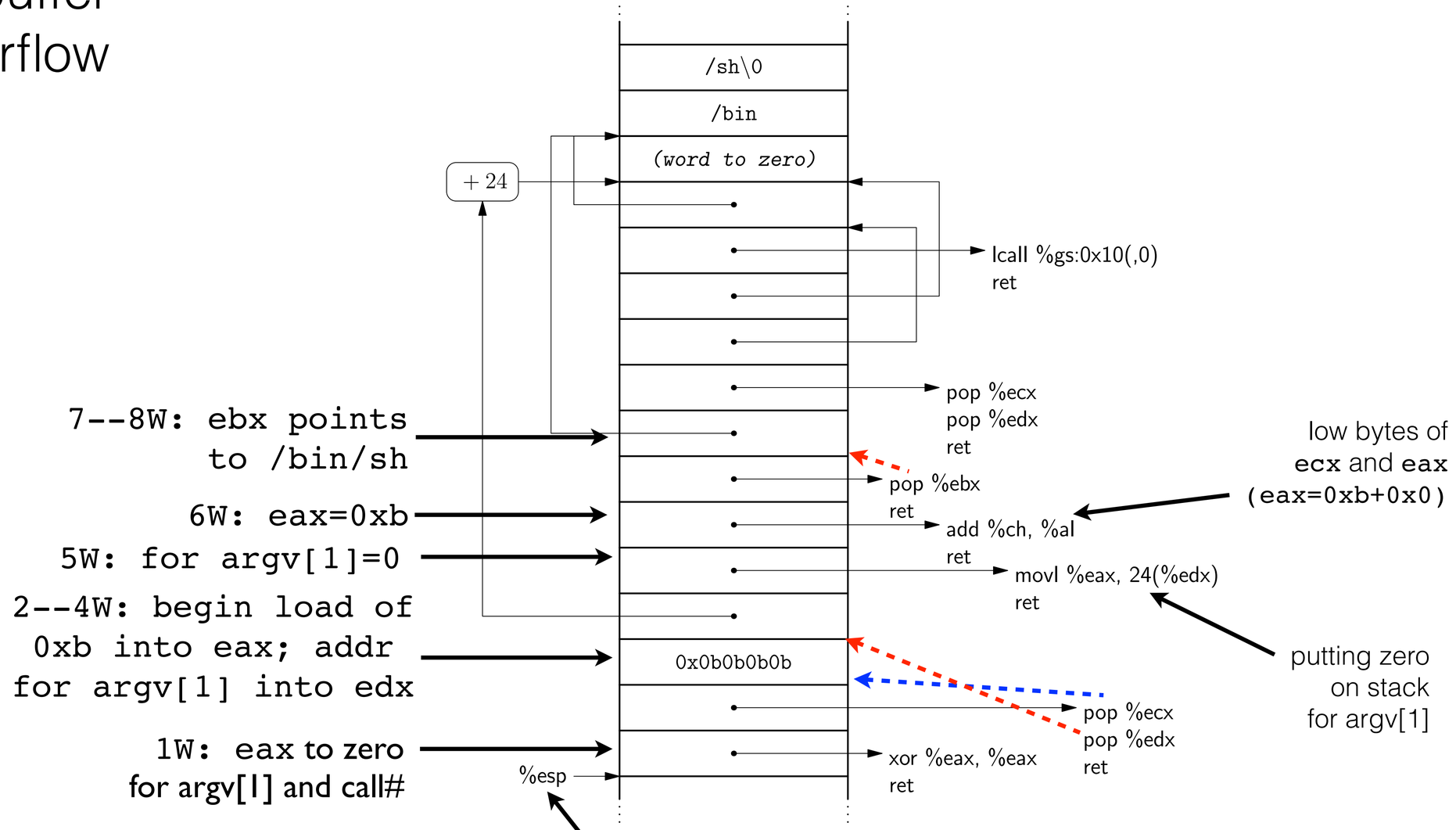
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

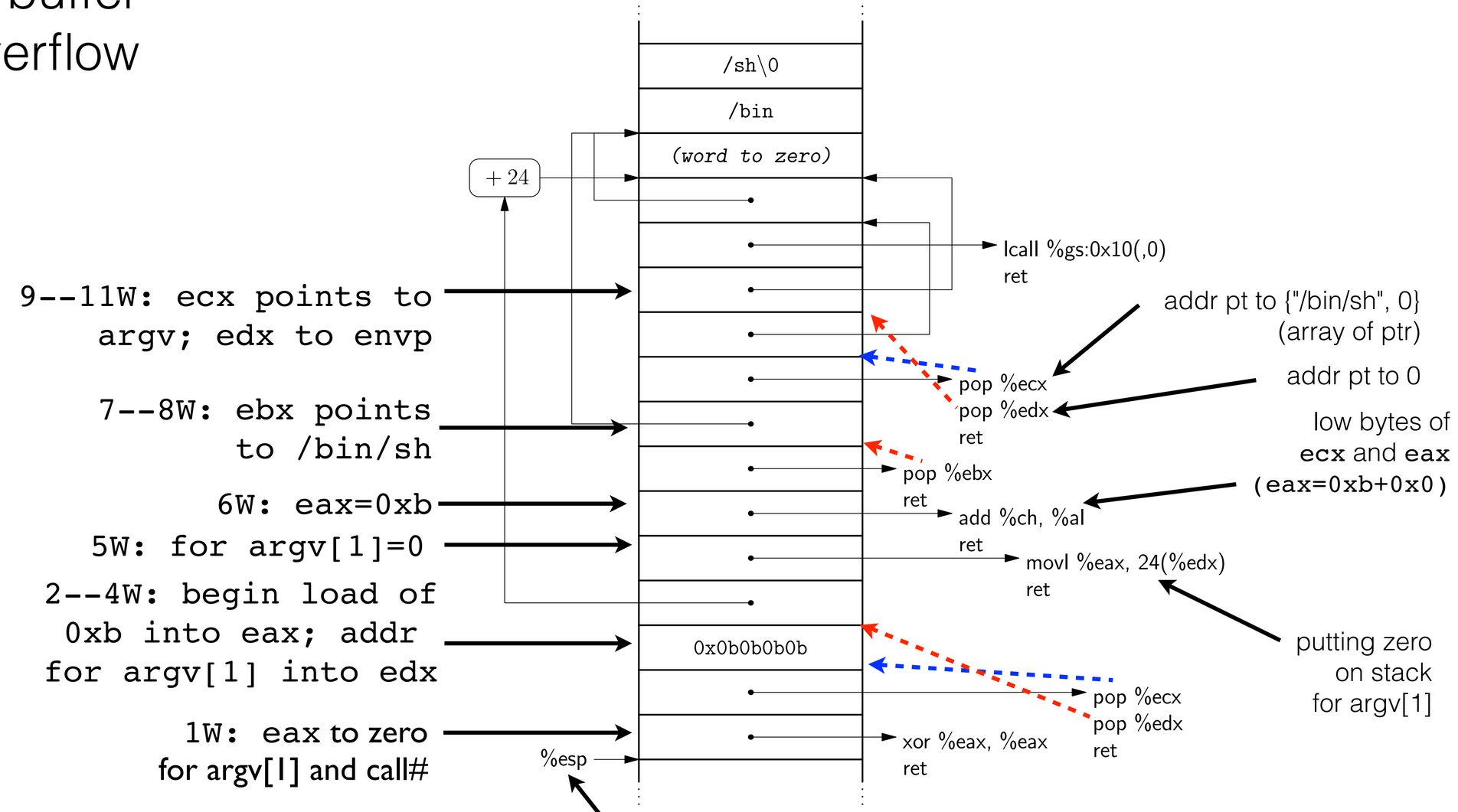
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

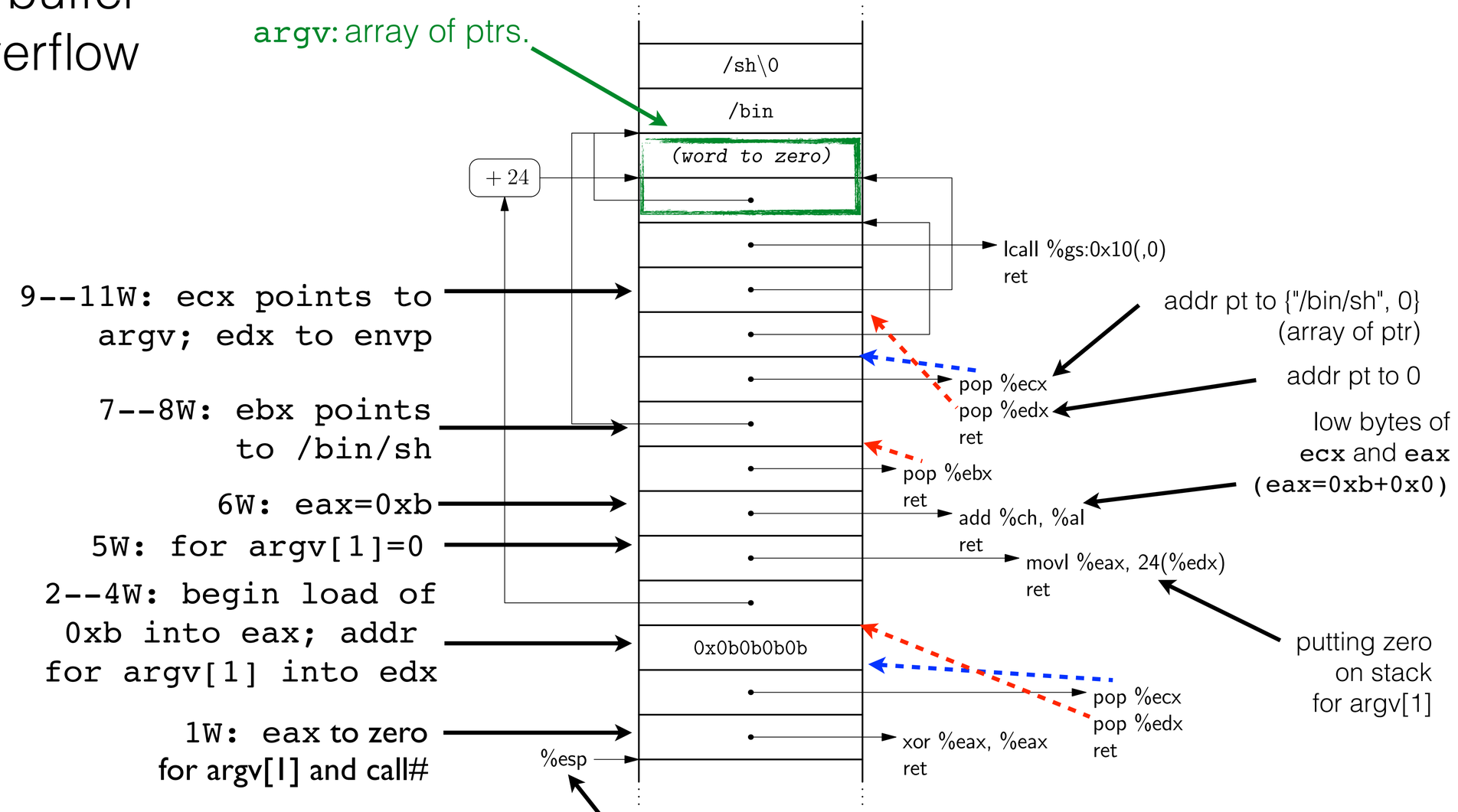
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

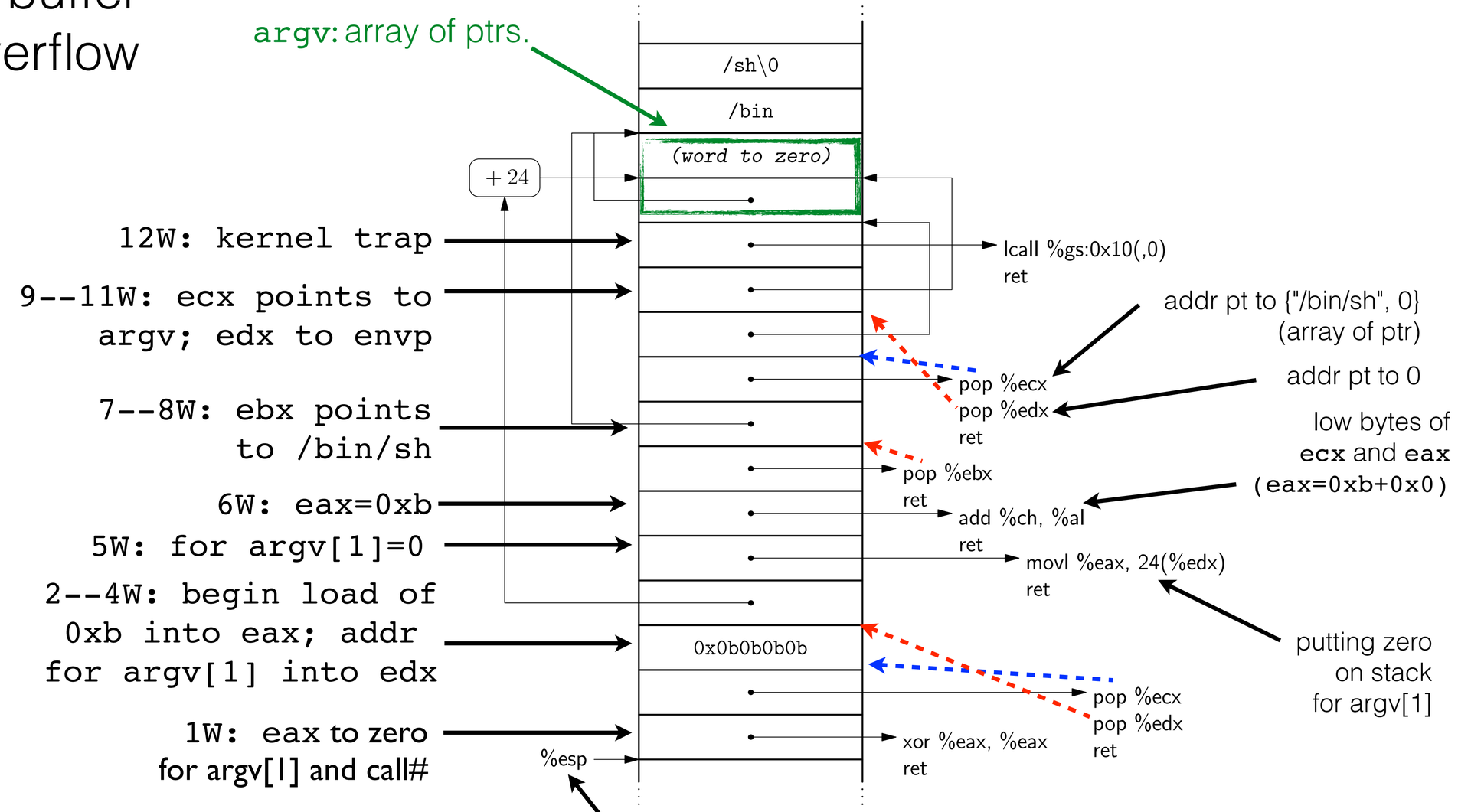


SP before epilog (ret)

procedure:

1. call # (0xb) into %eax
2. (ptr. to) program to run ('/bin/sh') in %ebx
3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx
4. envp: pointer to NULL in %edx

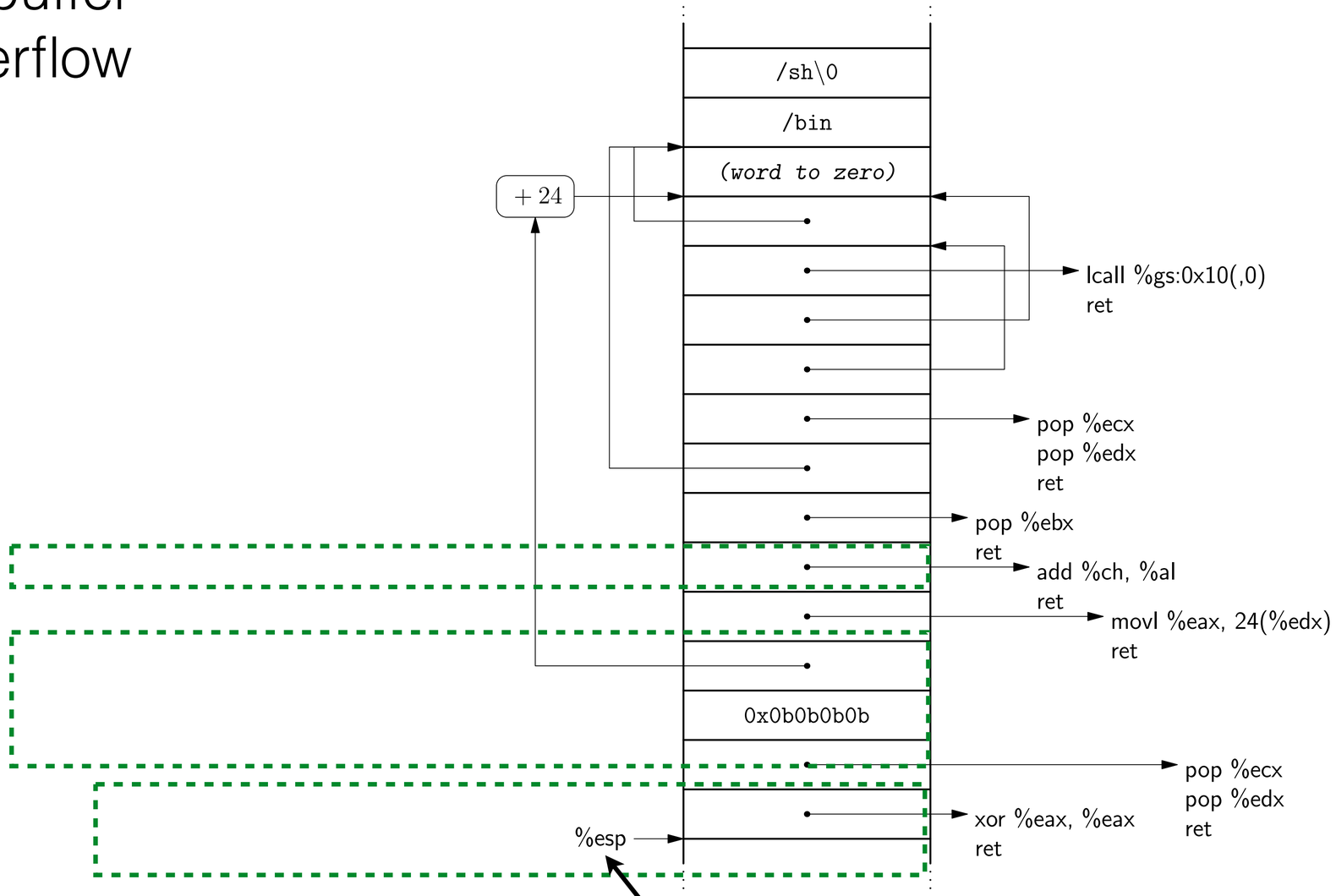
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

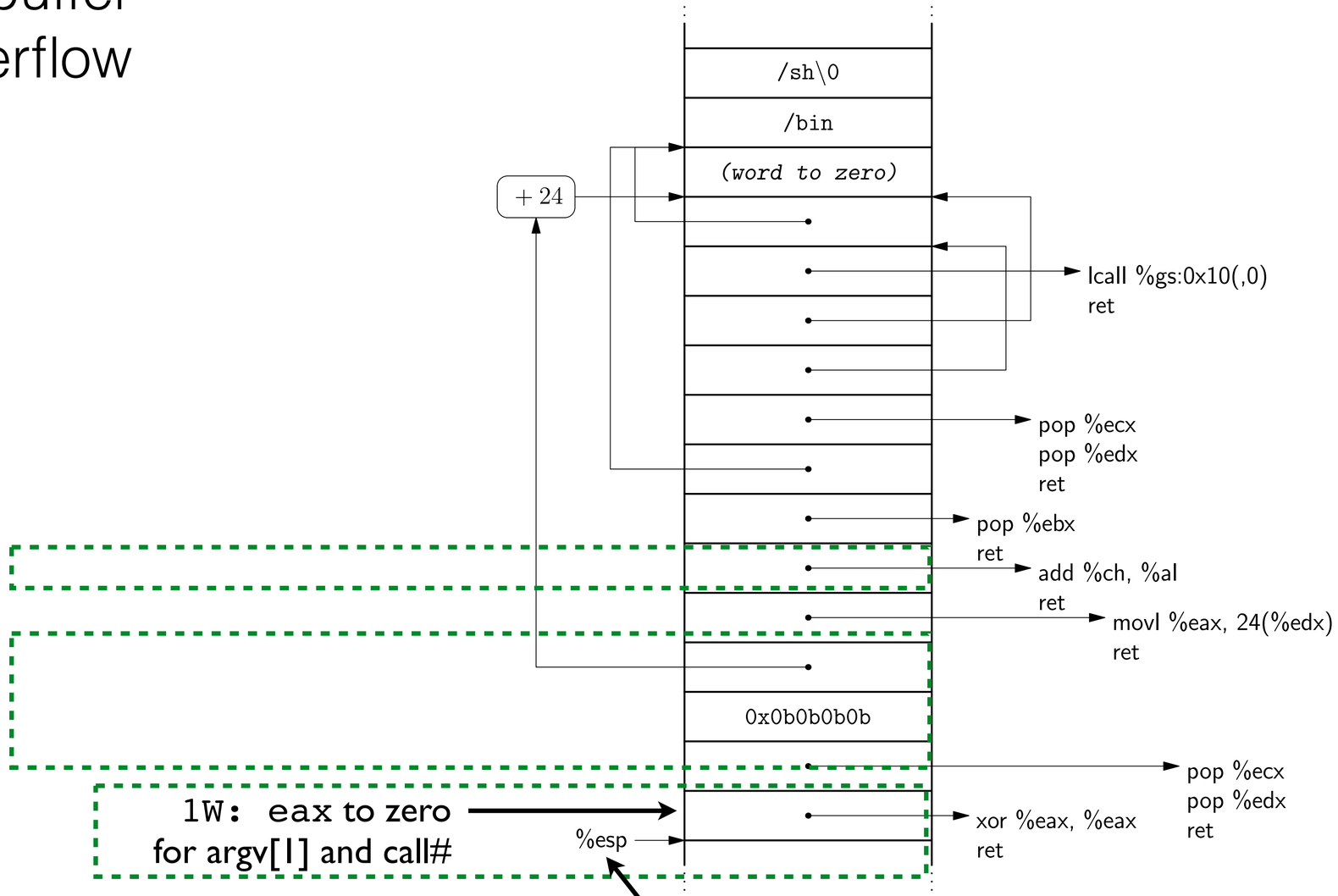


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

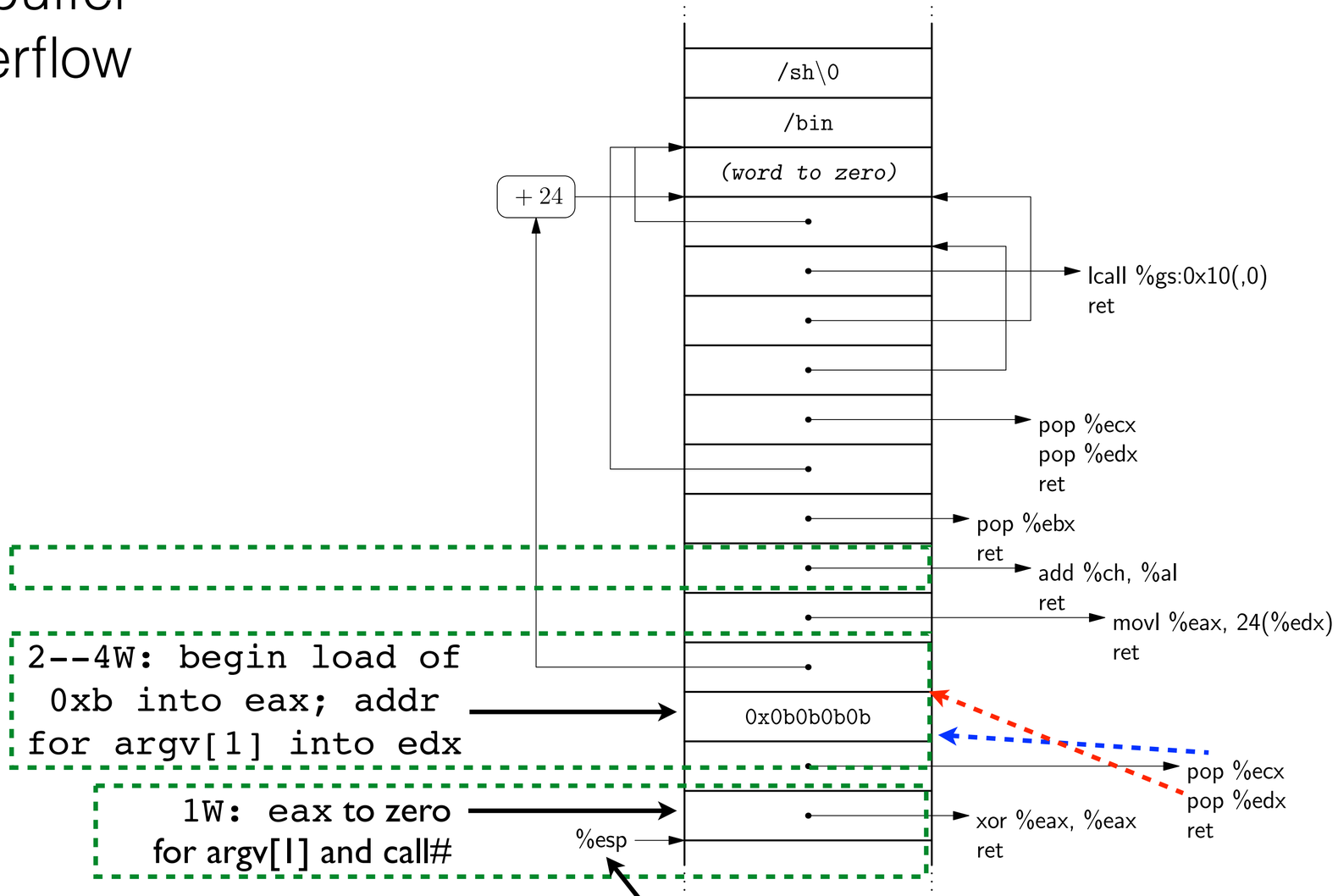


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

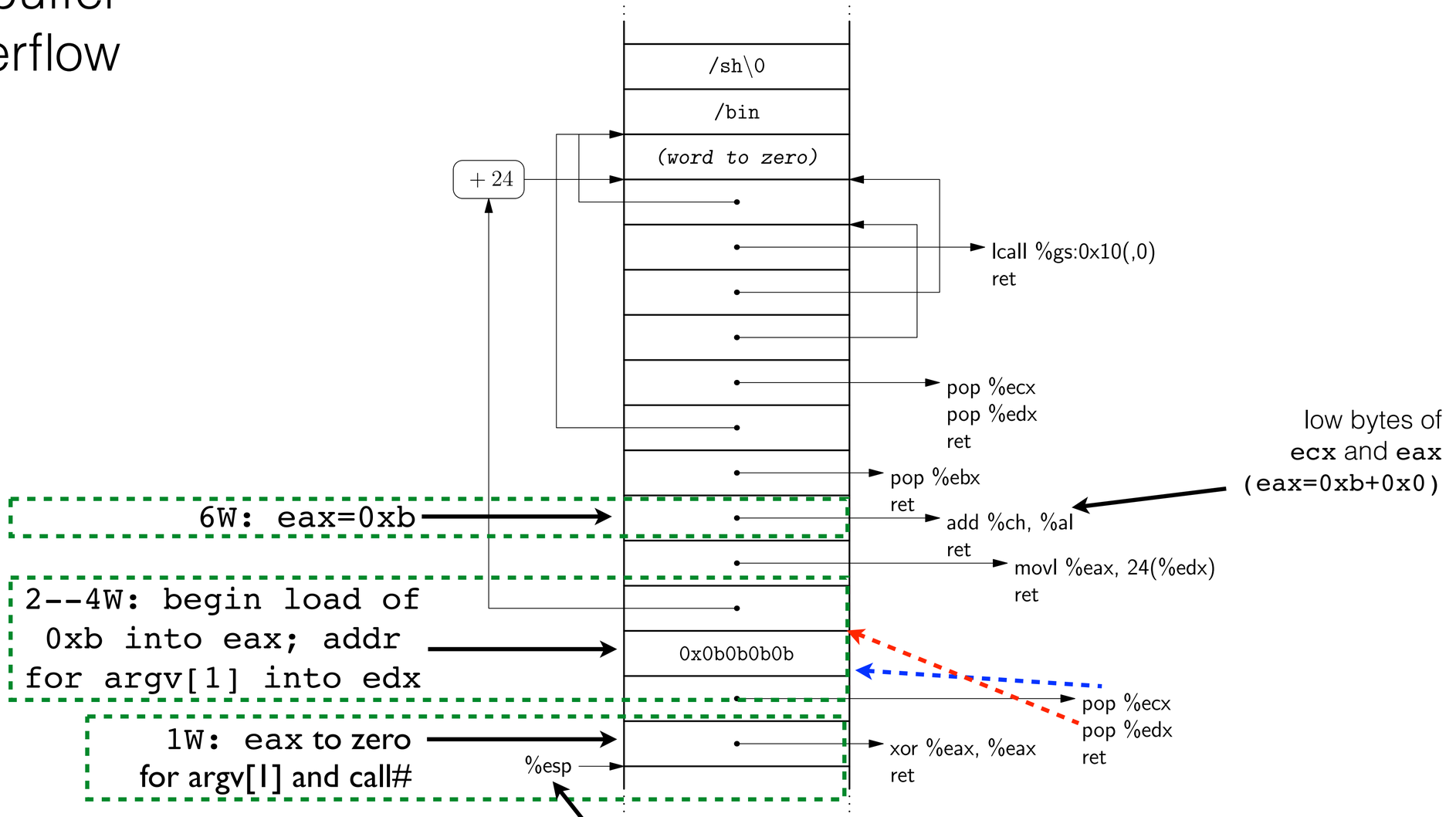


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

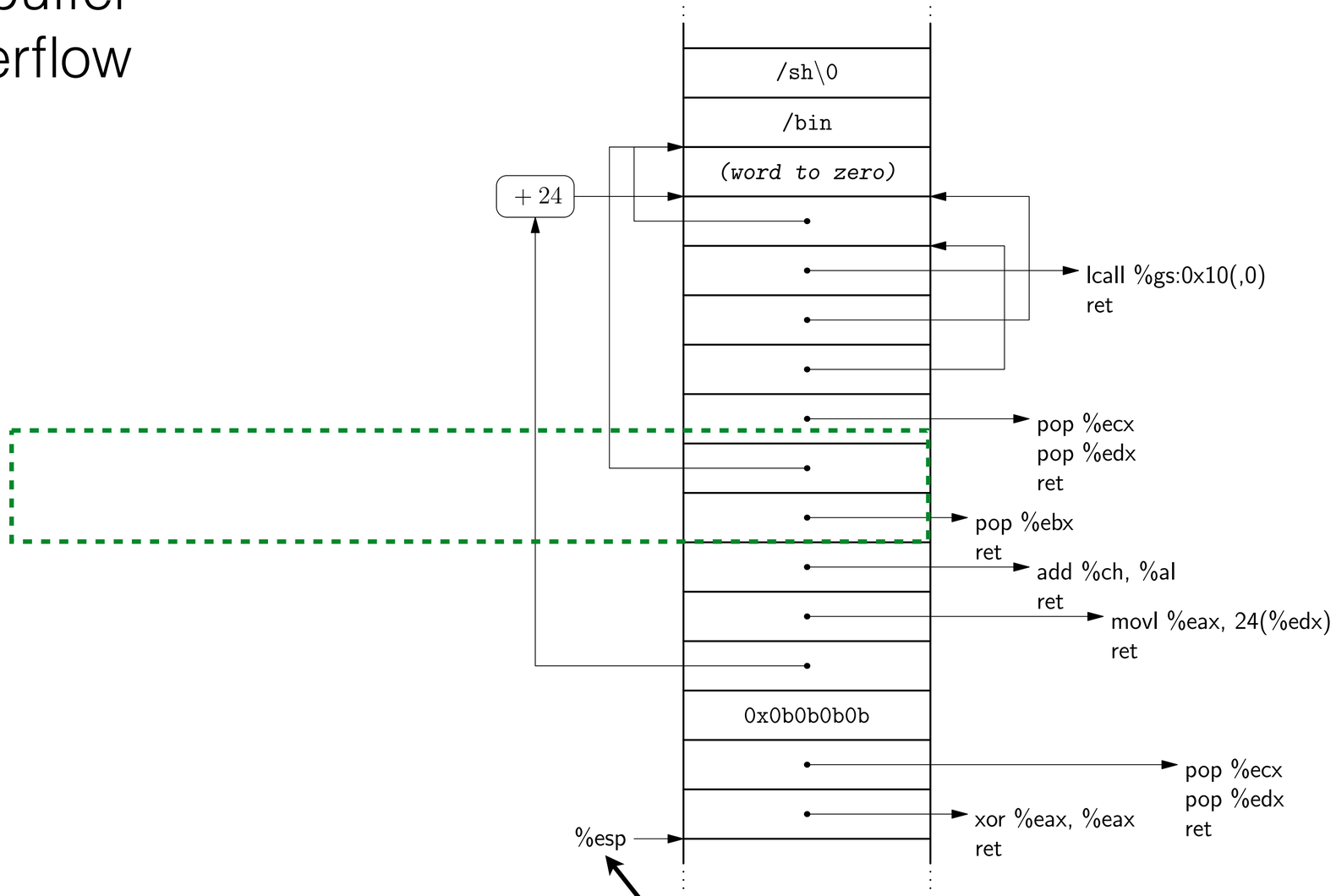
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

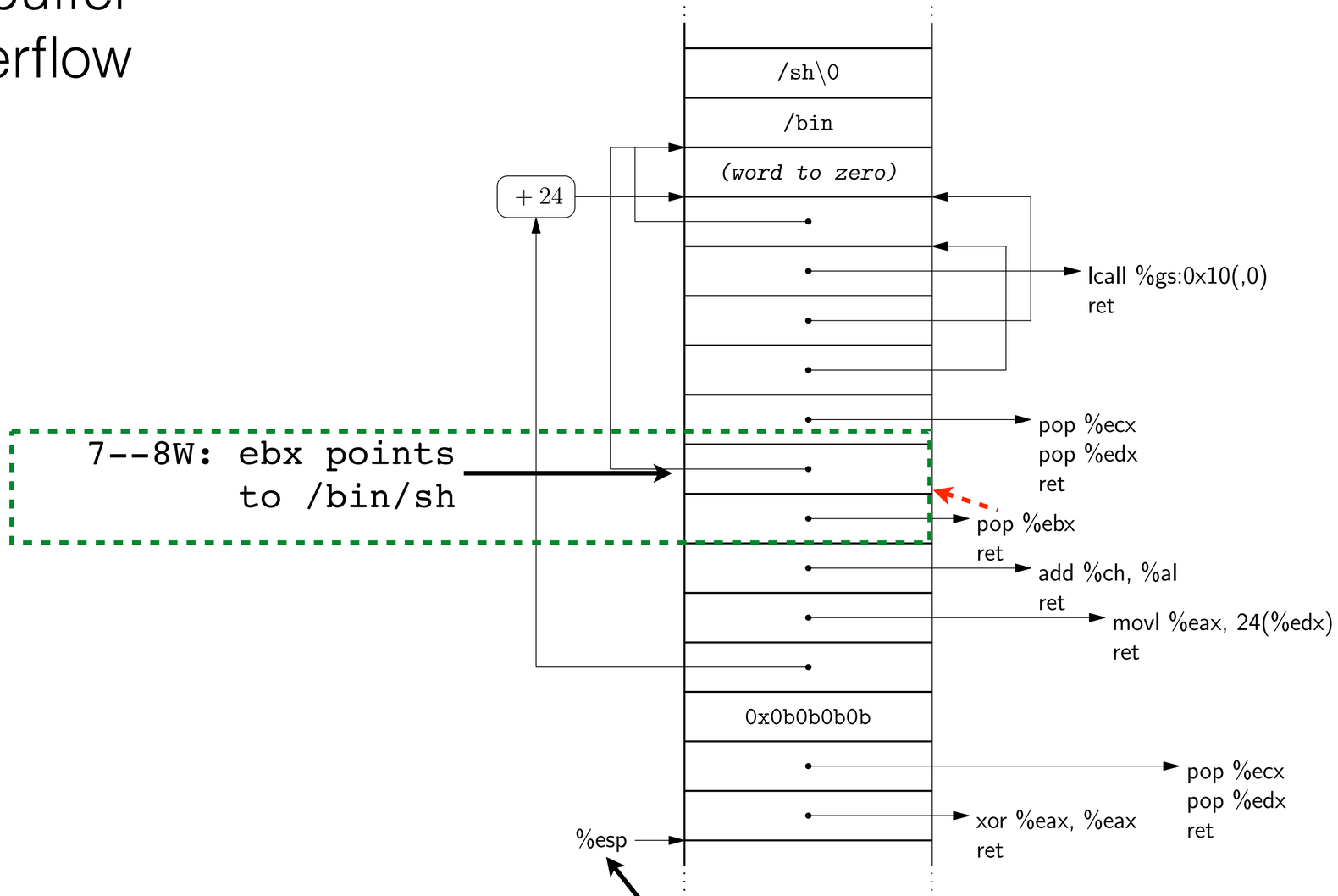


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

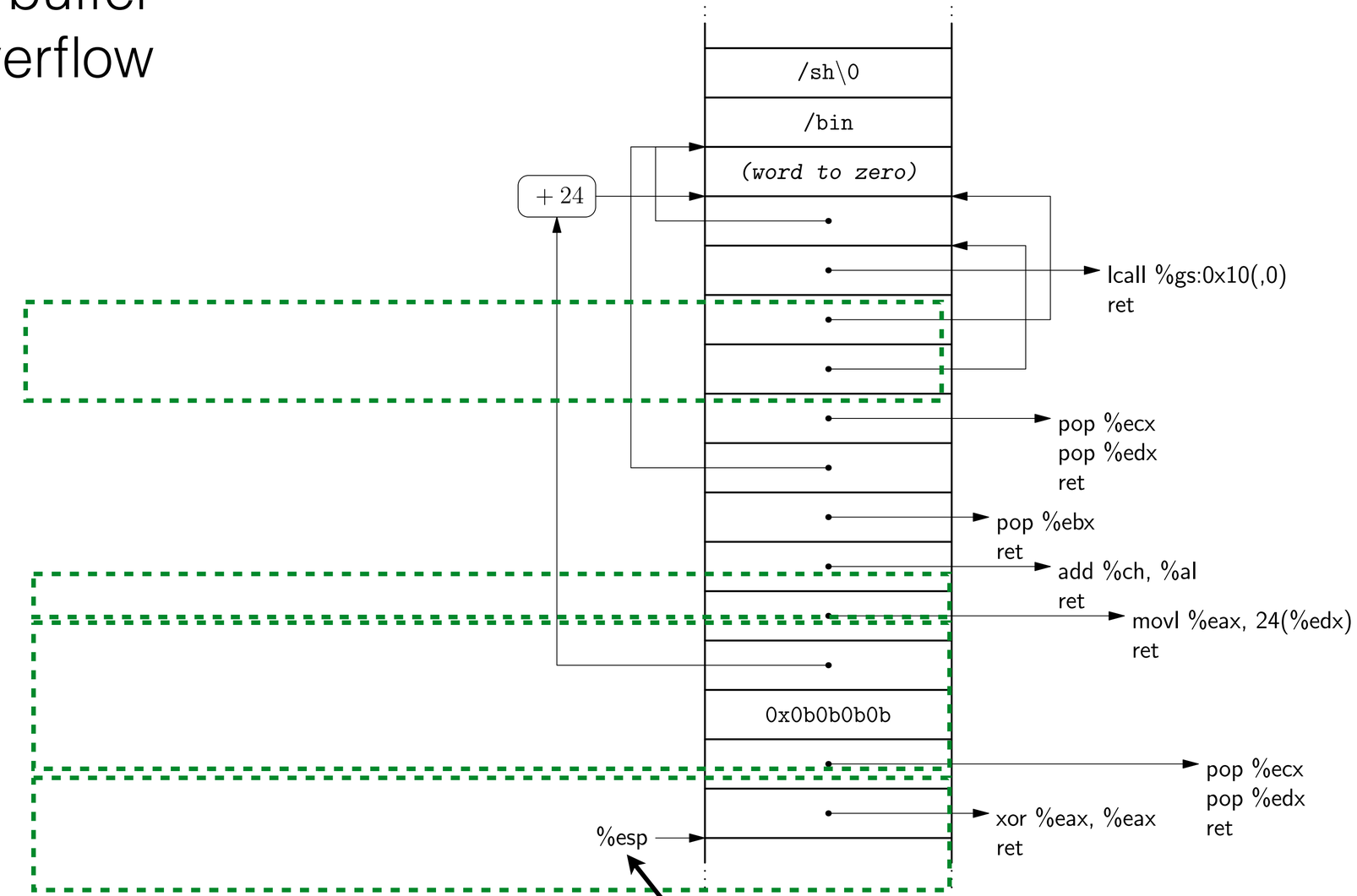
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

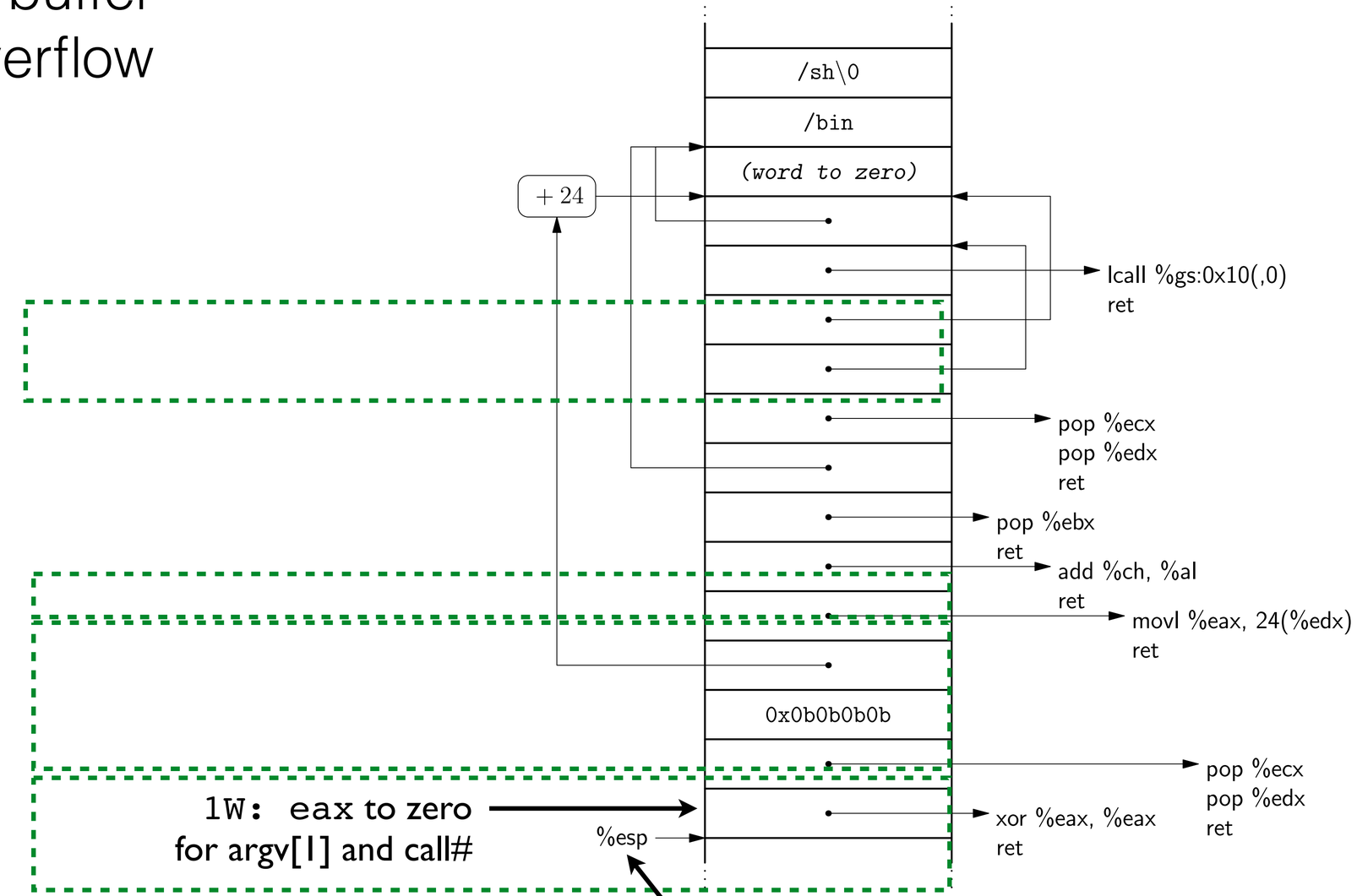


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

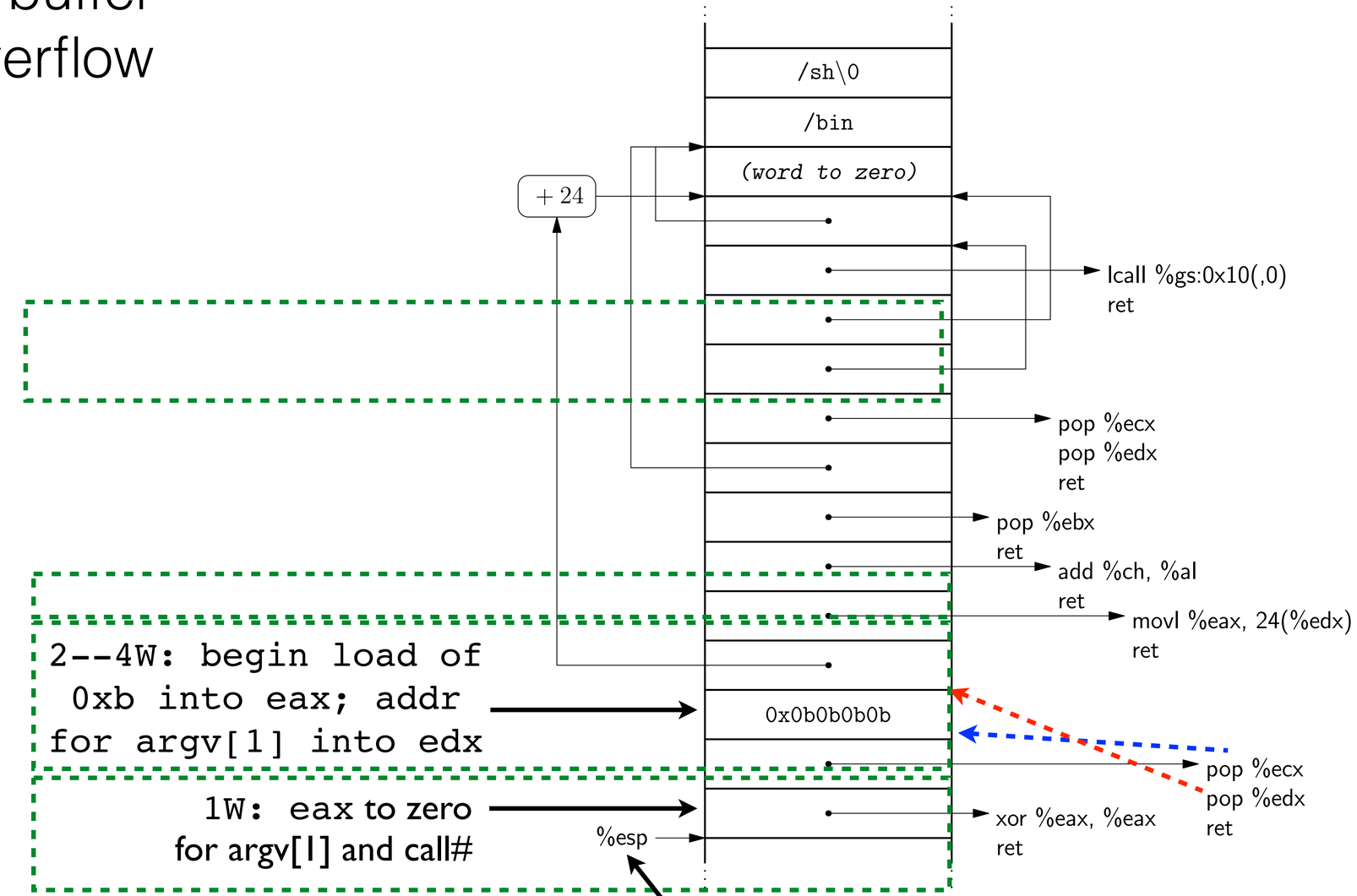


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

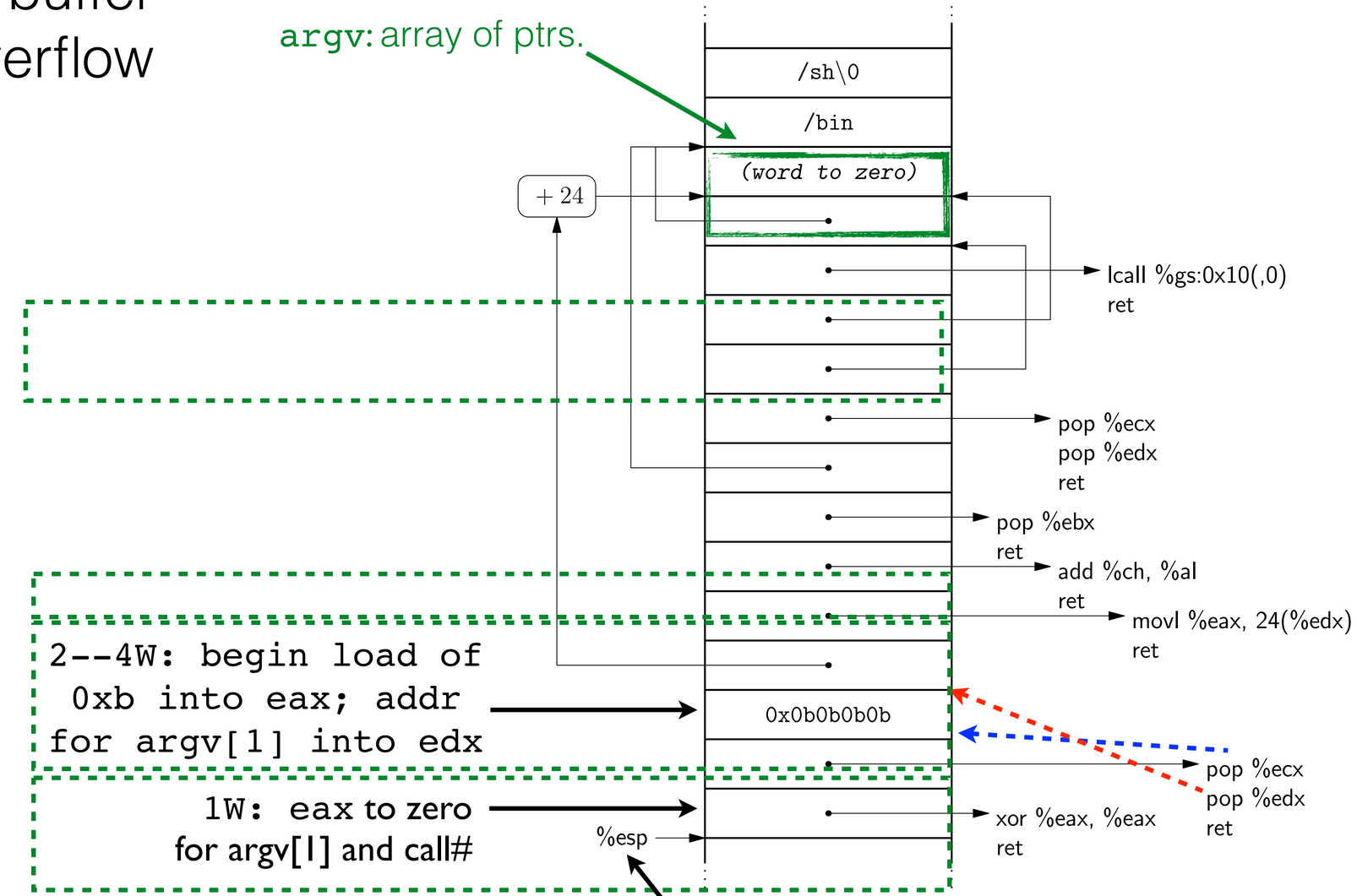


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

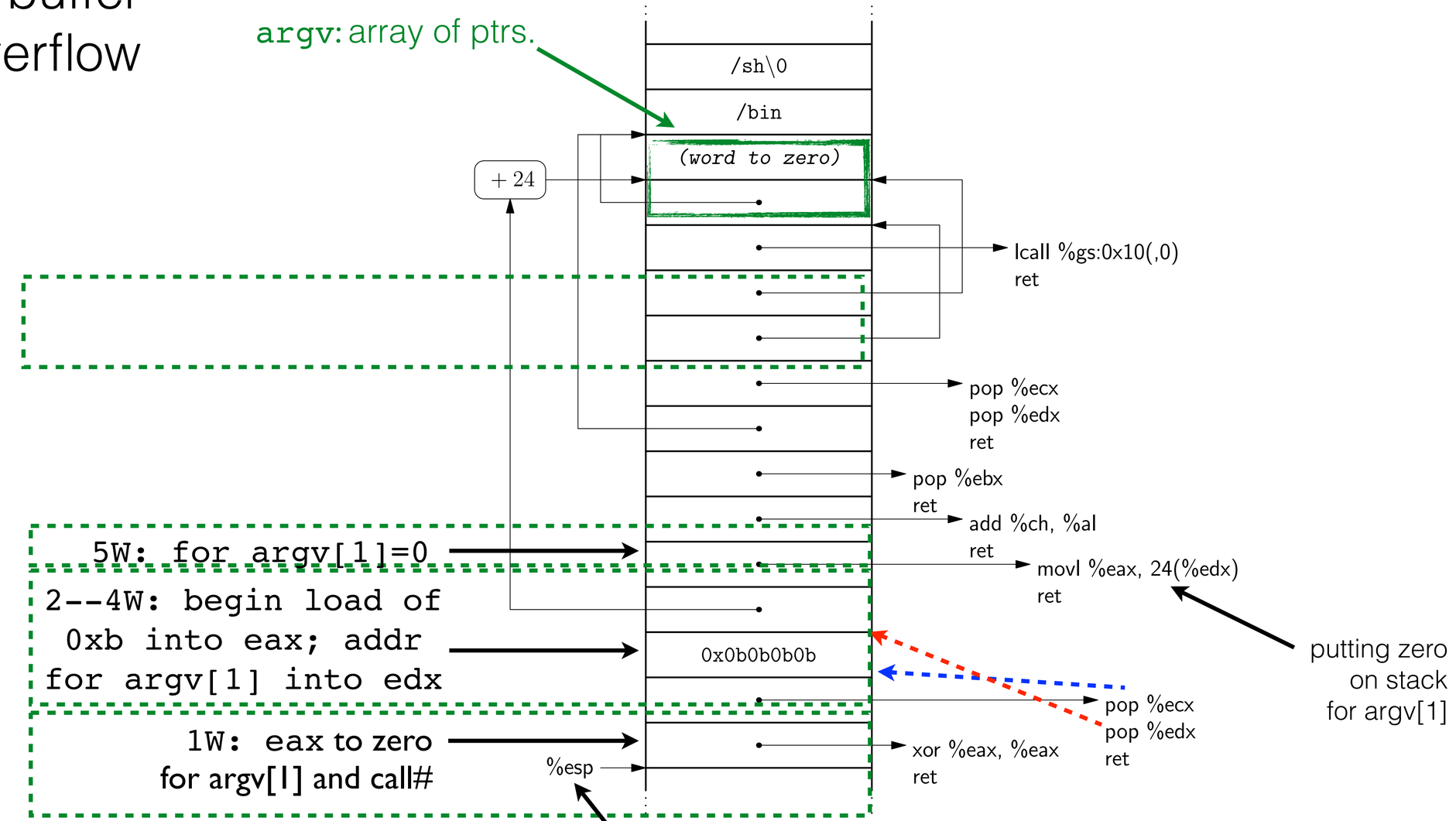


SP before epilog (ret)

procedure:

1. call # (0xb) into %eax
2. (ptr. to) program to run ('/bin/sh') in %ebx
3. set argv: pointer to ptr. array ('/bin/sh' and NULL) in %ecx
4. envp: pointer to NULL in %edx

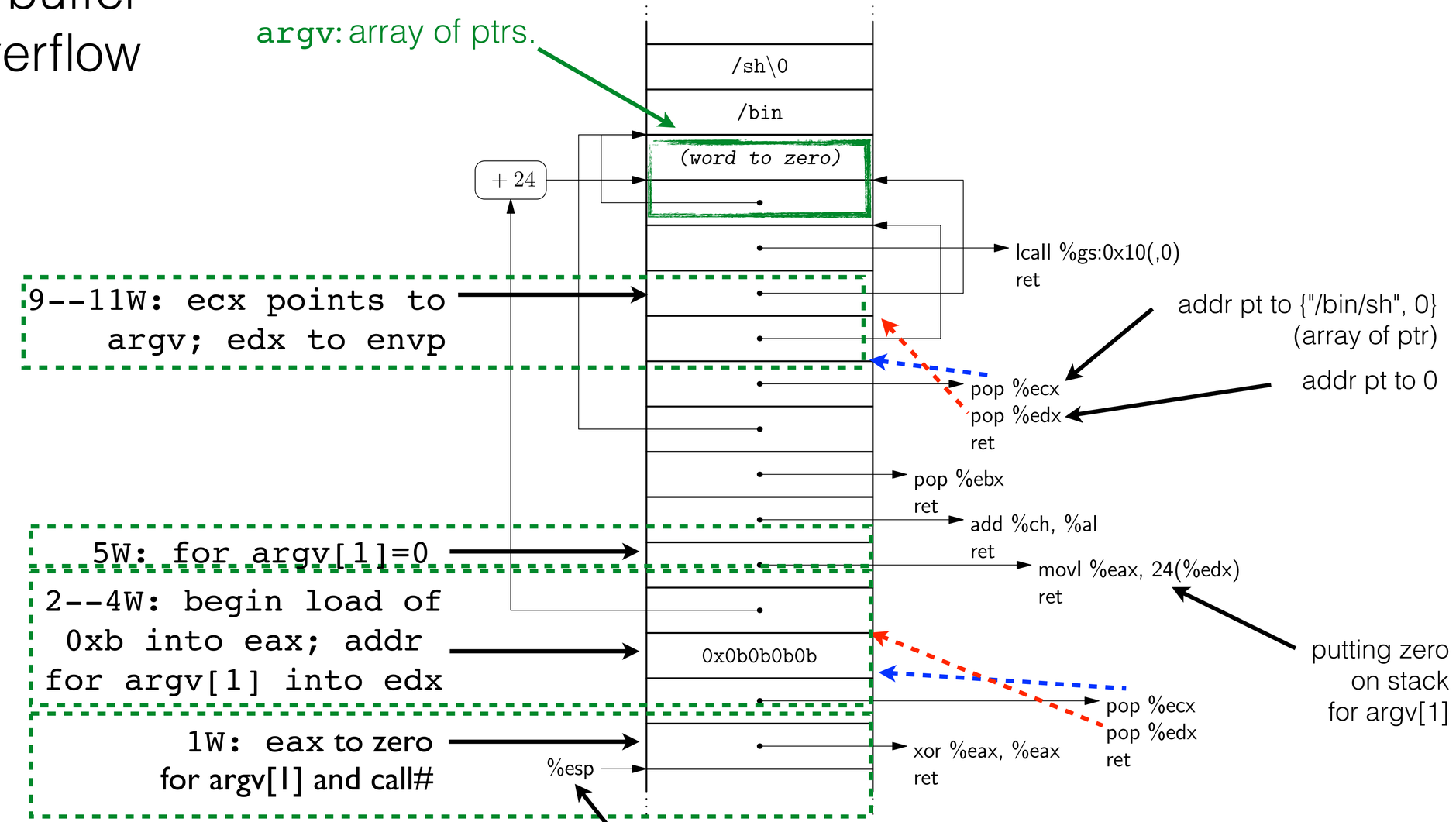
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

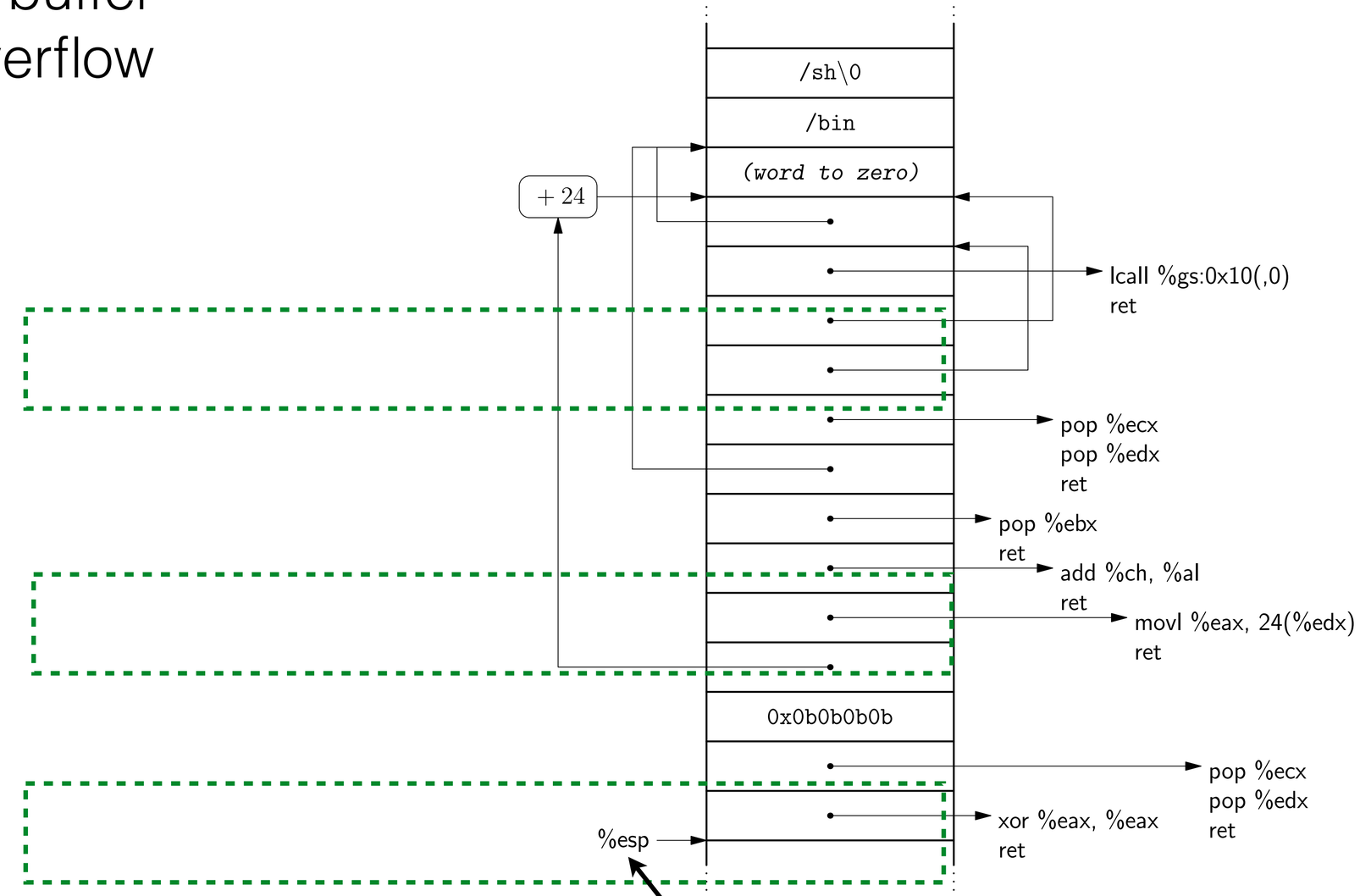
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

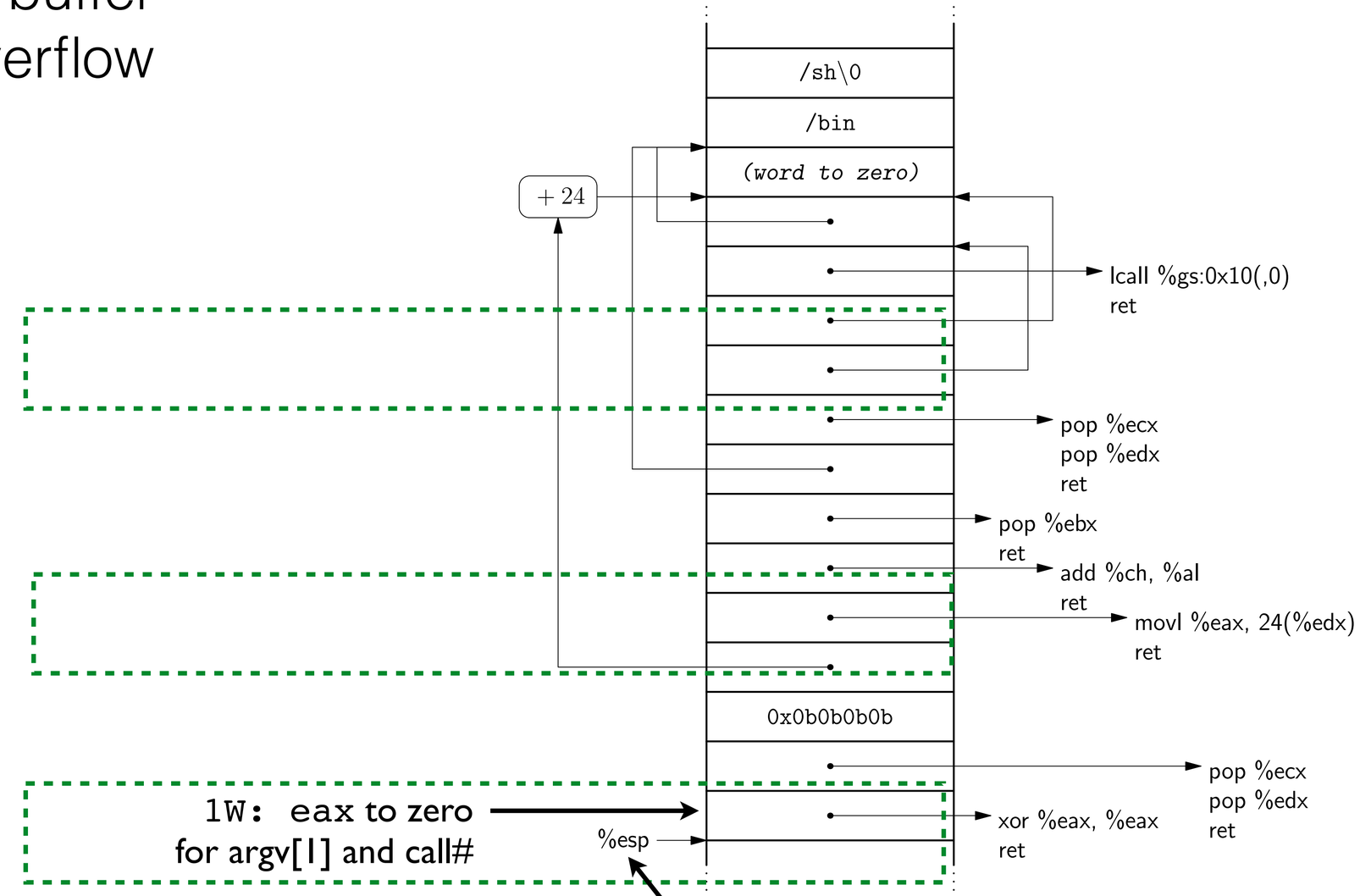


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

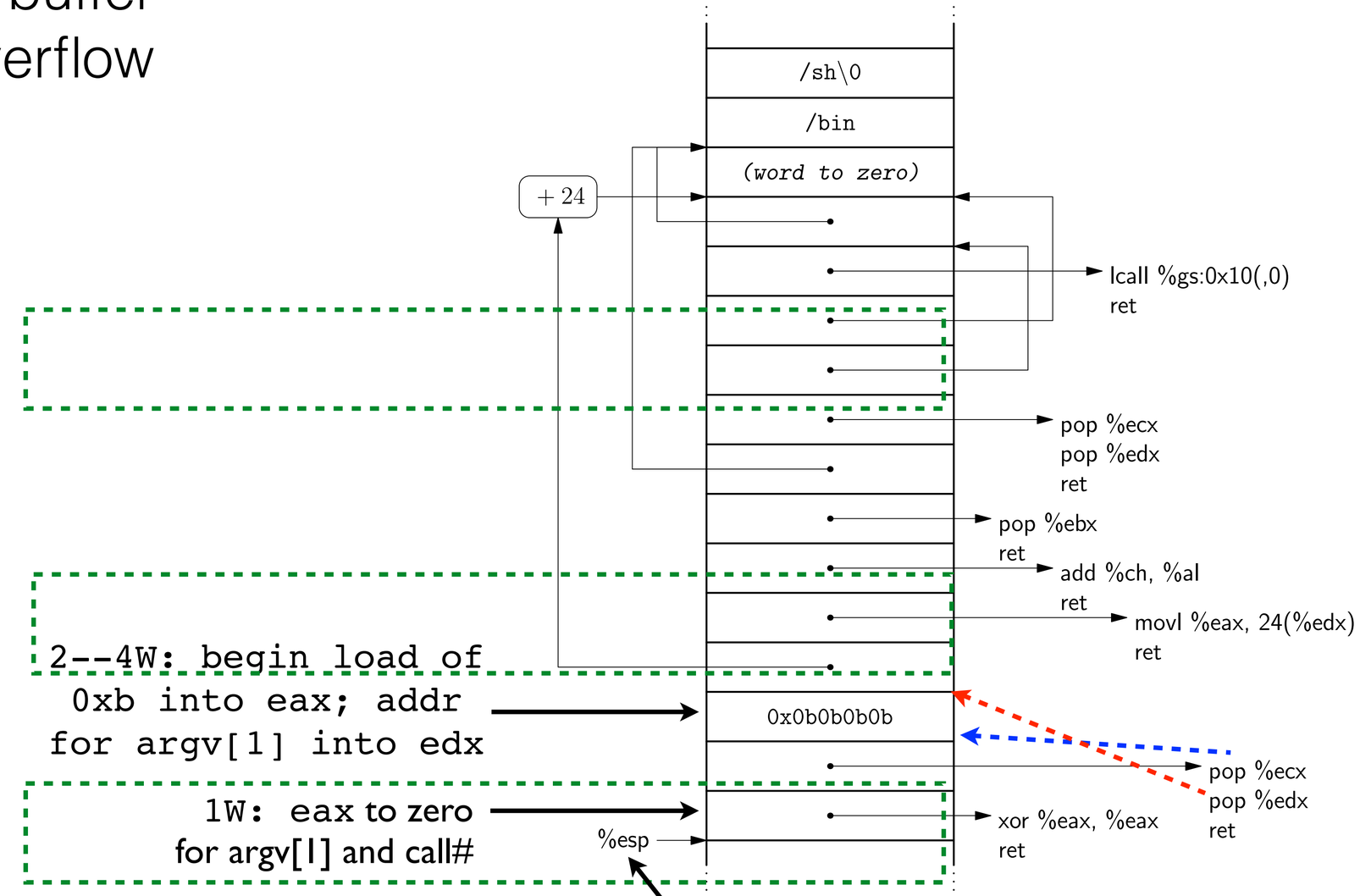


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow

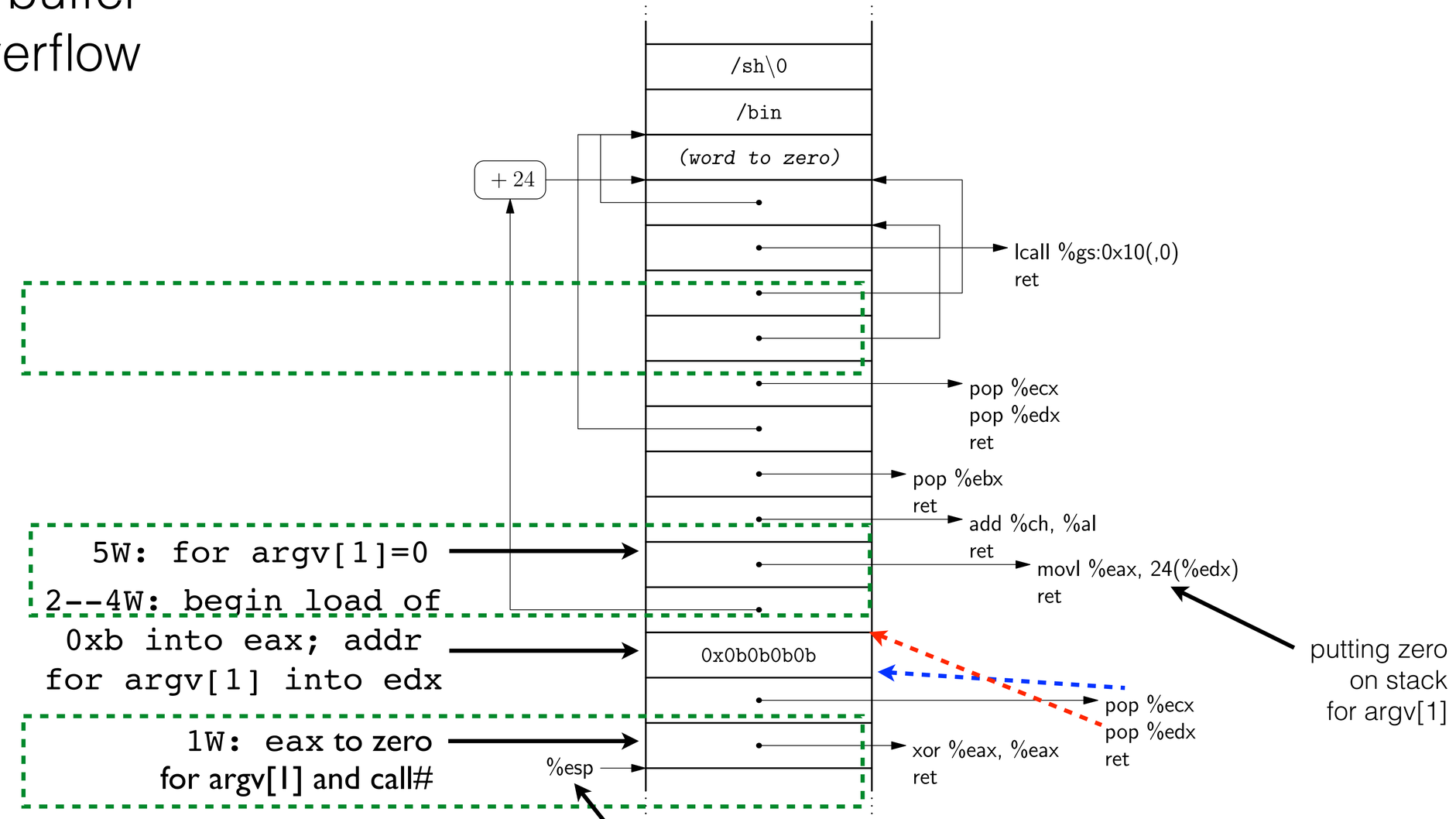


SP before epilog (`ret`)

procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

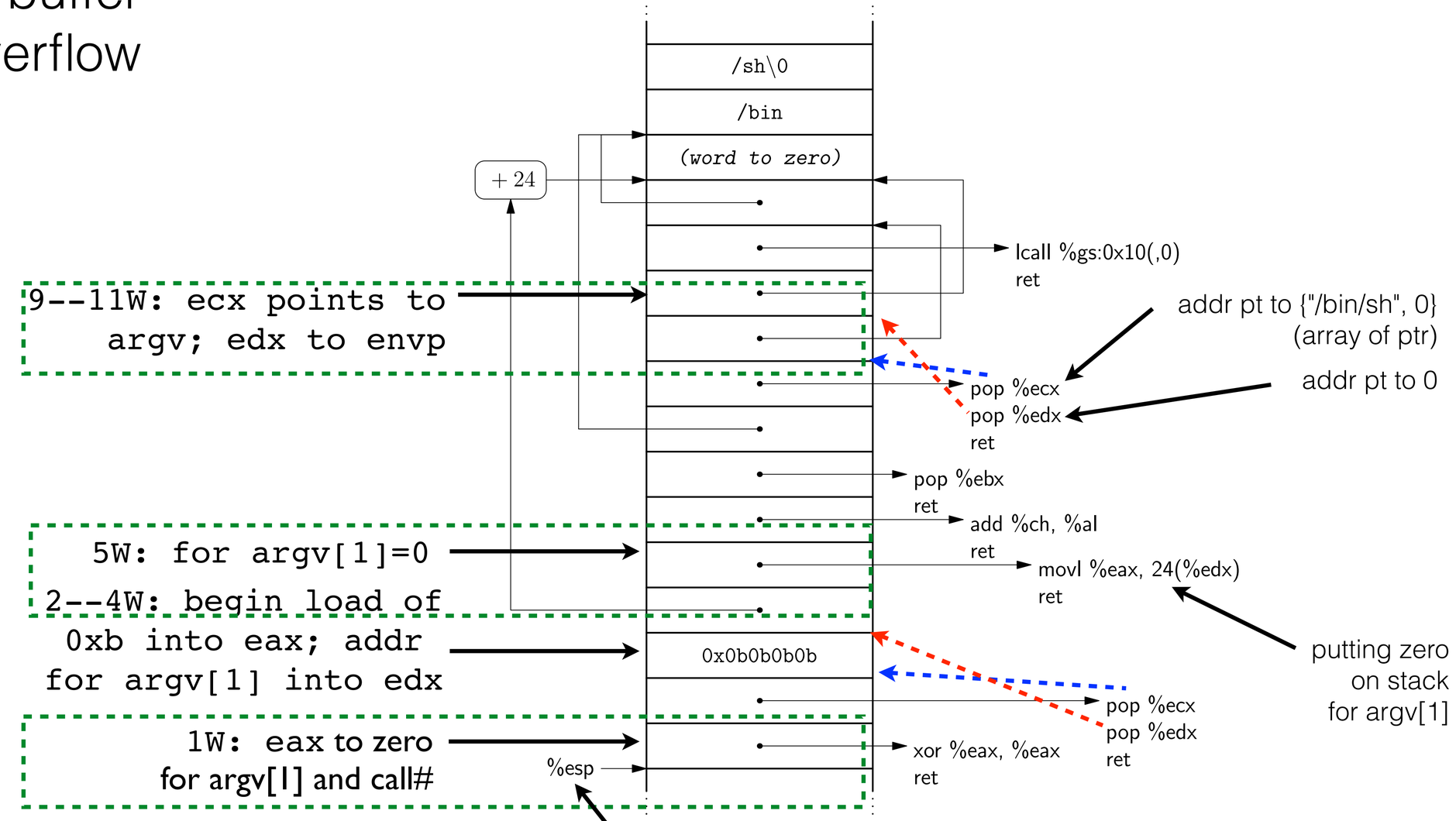
stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

stack via buffer overflow



procedure:

1. call # (`0xb`) into `%eax`
2. (ptr. to) program to run (`'/bin/sh'`) in `%ebx`
3. set argv: pointer to ptr. array (`'/bin/sh'` and `NULL`) in `%ecx`
4. envp: pointer to `NULL` in `%edx`

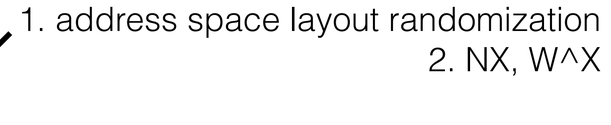
Virginia Tech
Invent the Future

stack protection methods:

1. make the overflow difficult to exploit
2. make overflow noticeable
3. make it impossible

stack protection methods:

1. make the overflow difficult to exploit
2. make overflow noticeable
3.make it impossible



stack protection methods:

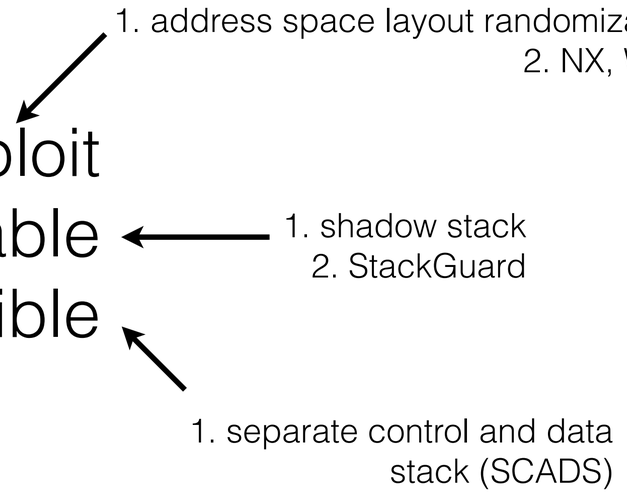
1. make the overflow difficult to exploit
2. make overflow noticeable
3.make it impossible

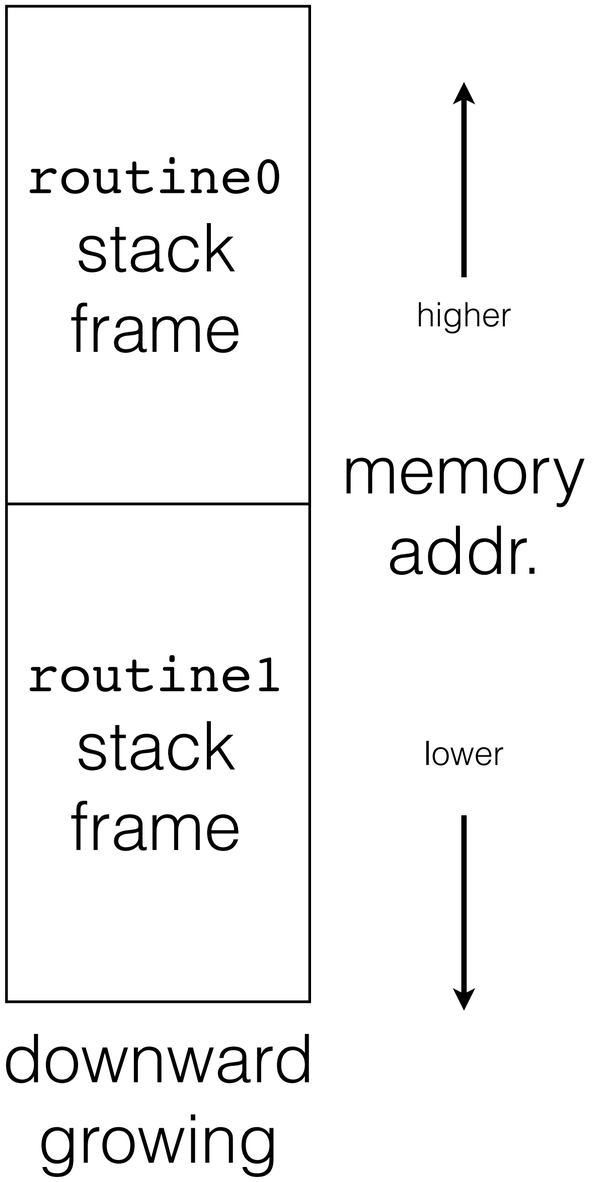
1. address space layout randomization
2. NX, W^X

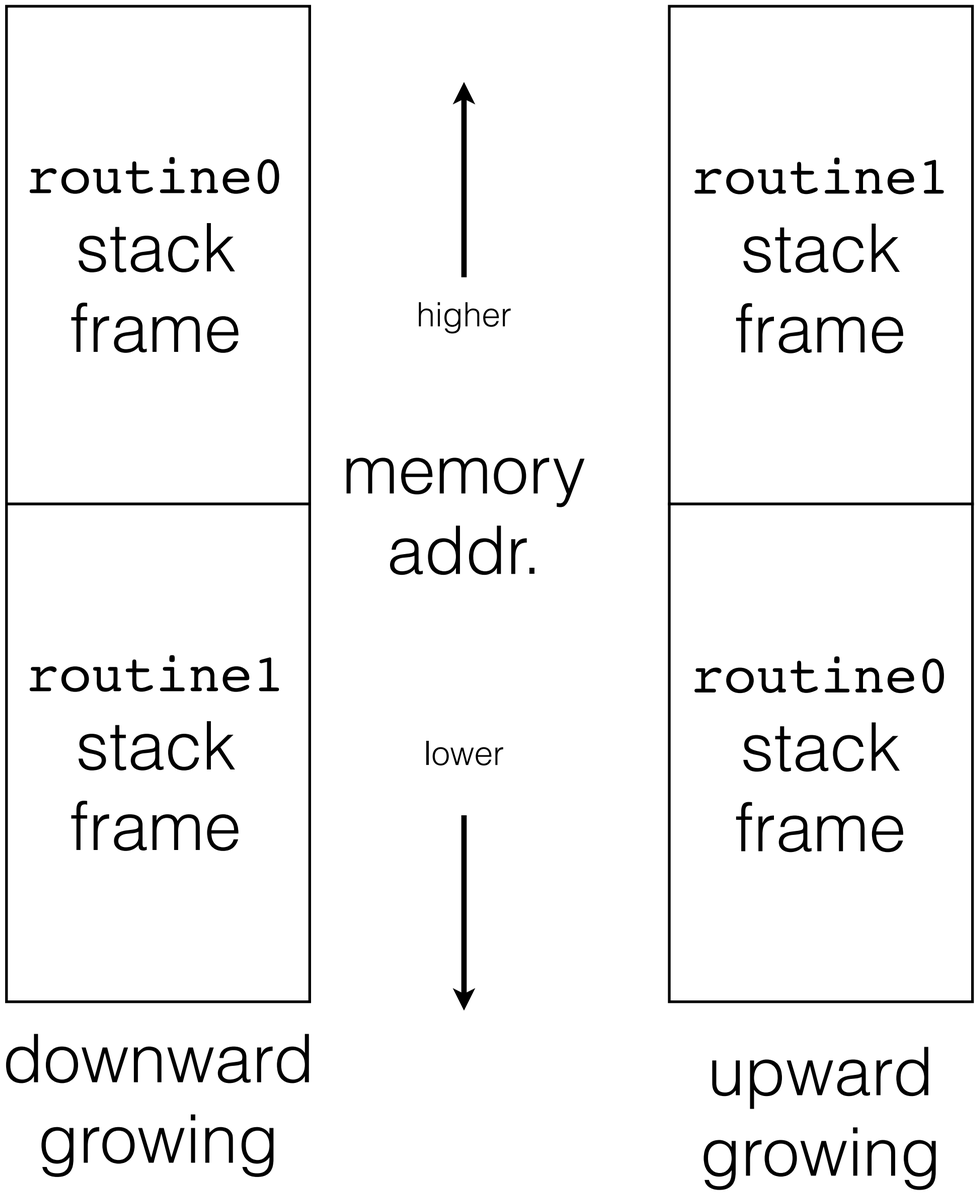
1. shadow stack
2. StackGuard

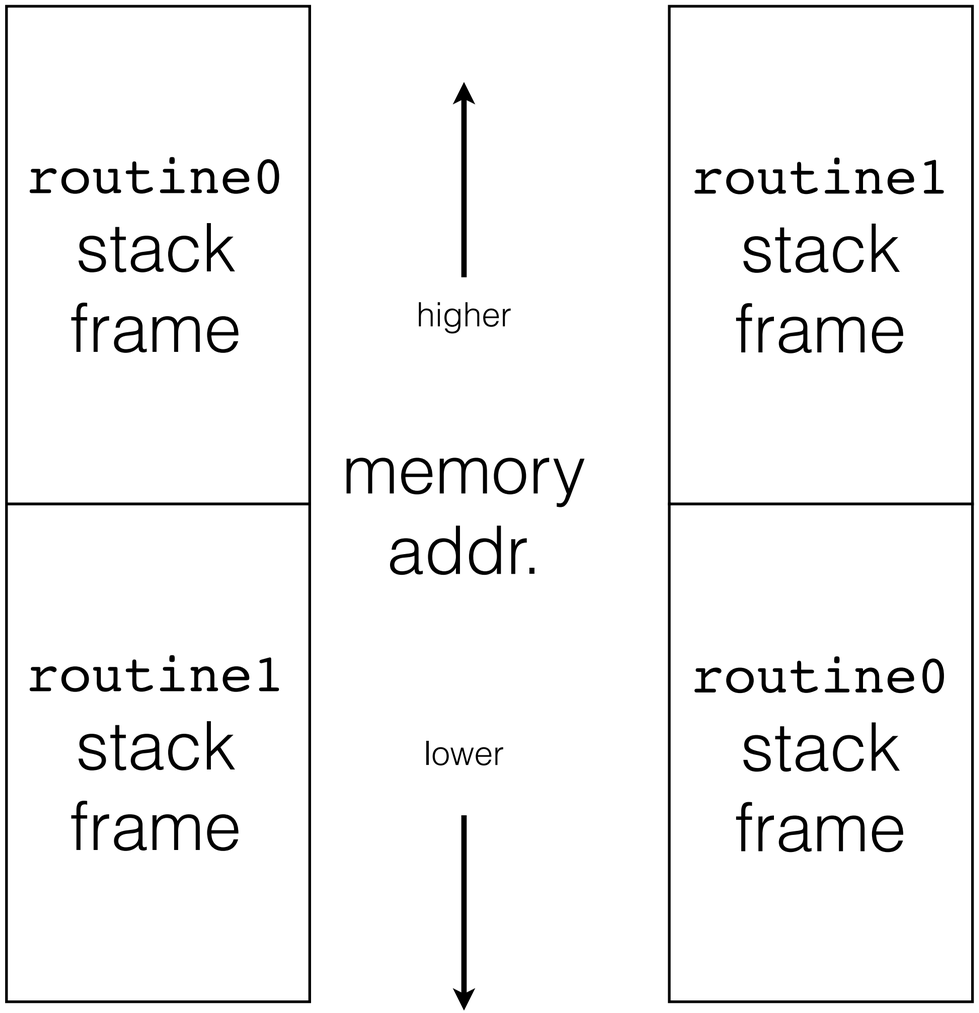
stack protection methods:

1. make the overflow difficult to exploit
2. make overflow noticeable
3.make it impossible









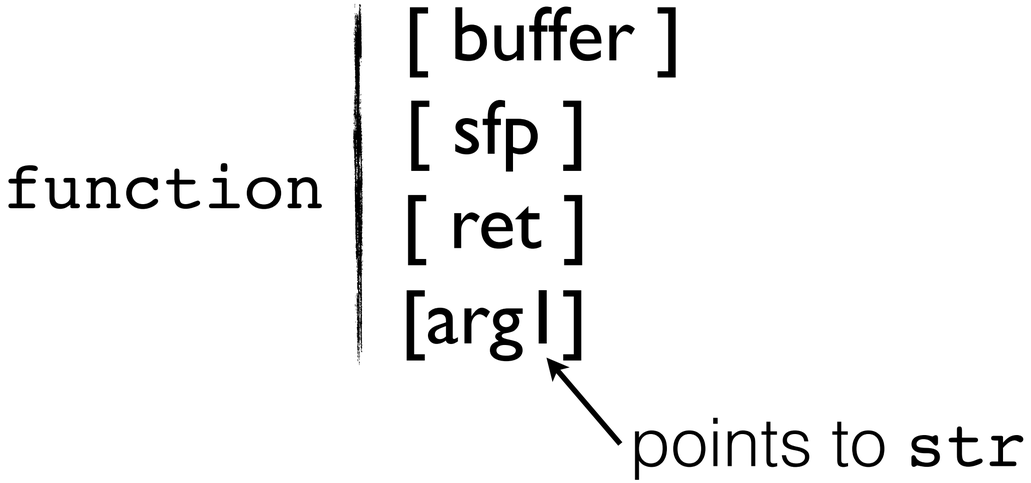
Q: would an upward growing stack prevent exploit of buffer overflow?

upward growing stack buffer overflow

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

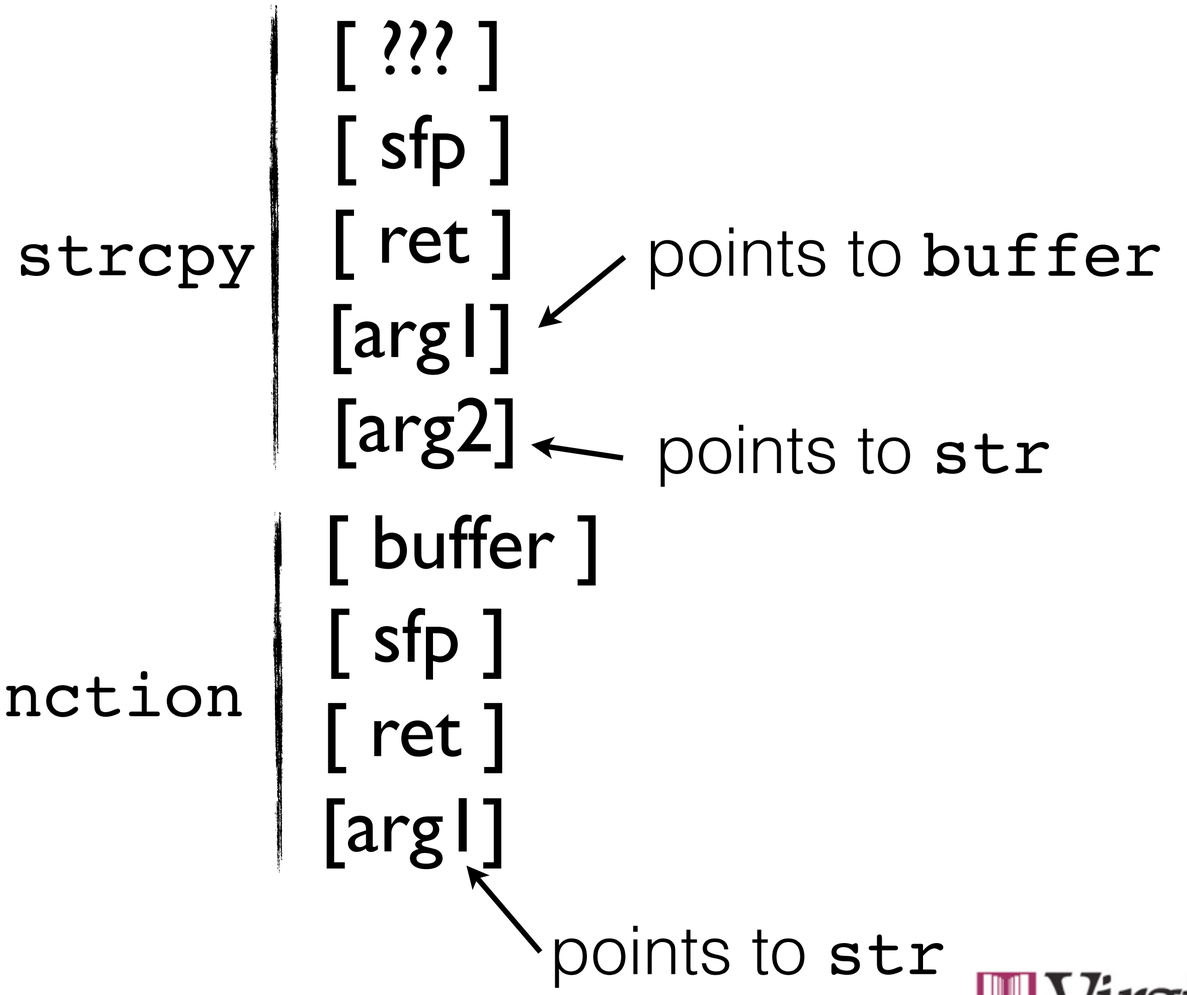
upward growing stack buffer overflow

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```



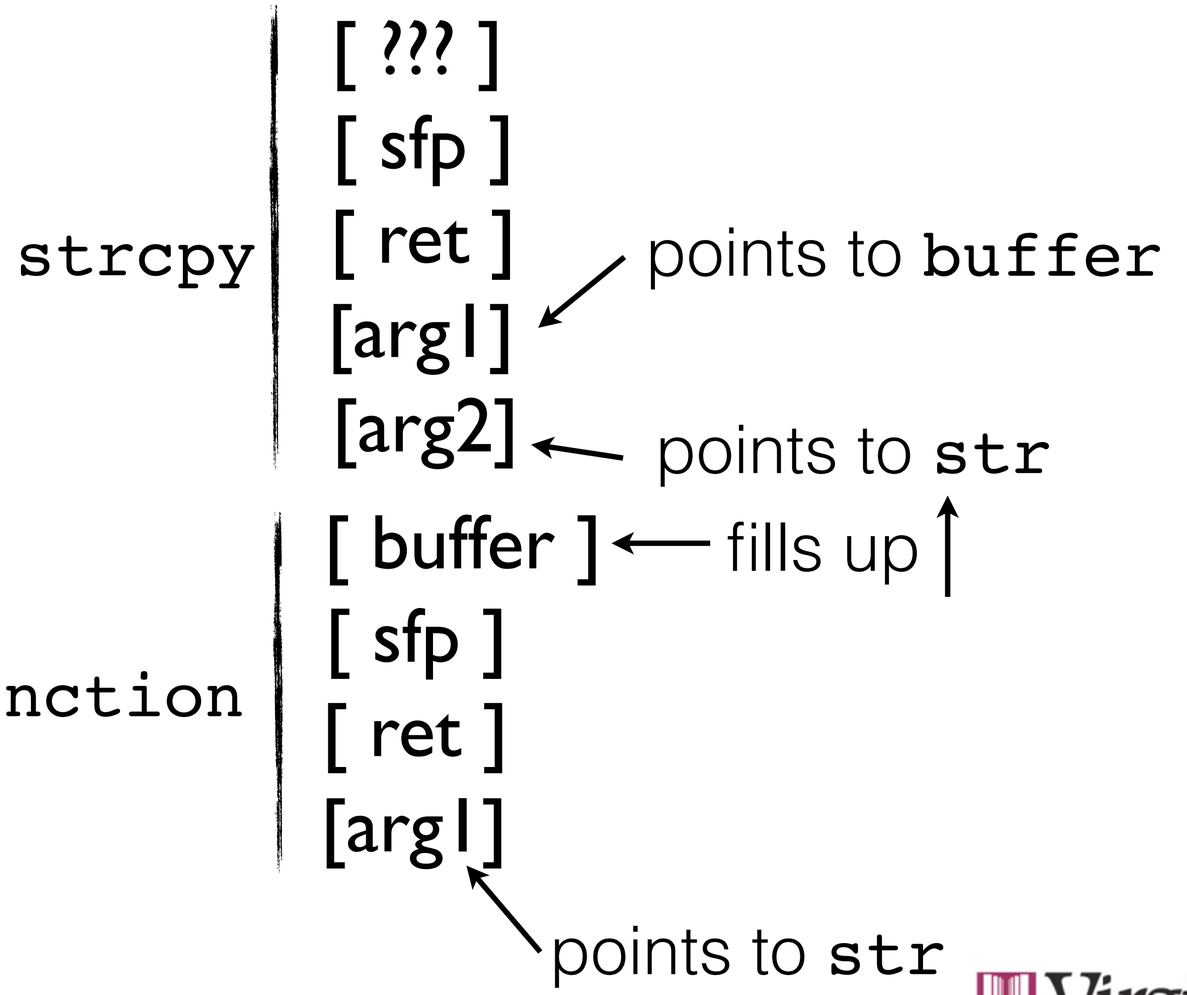
upward growing stack buffer overflow

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

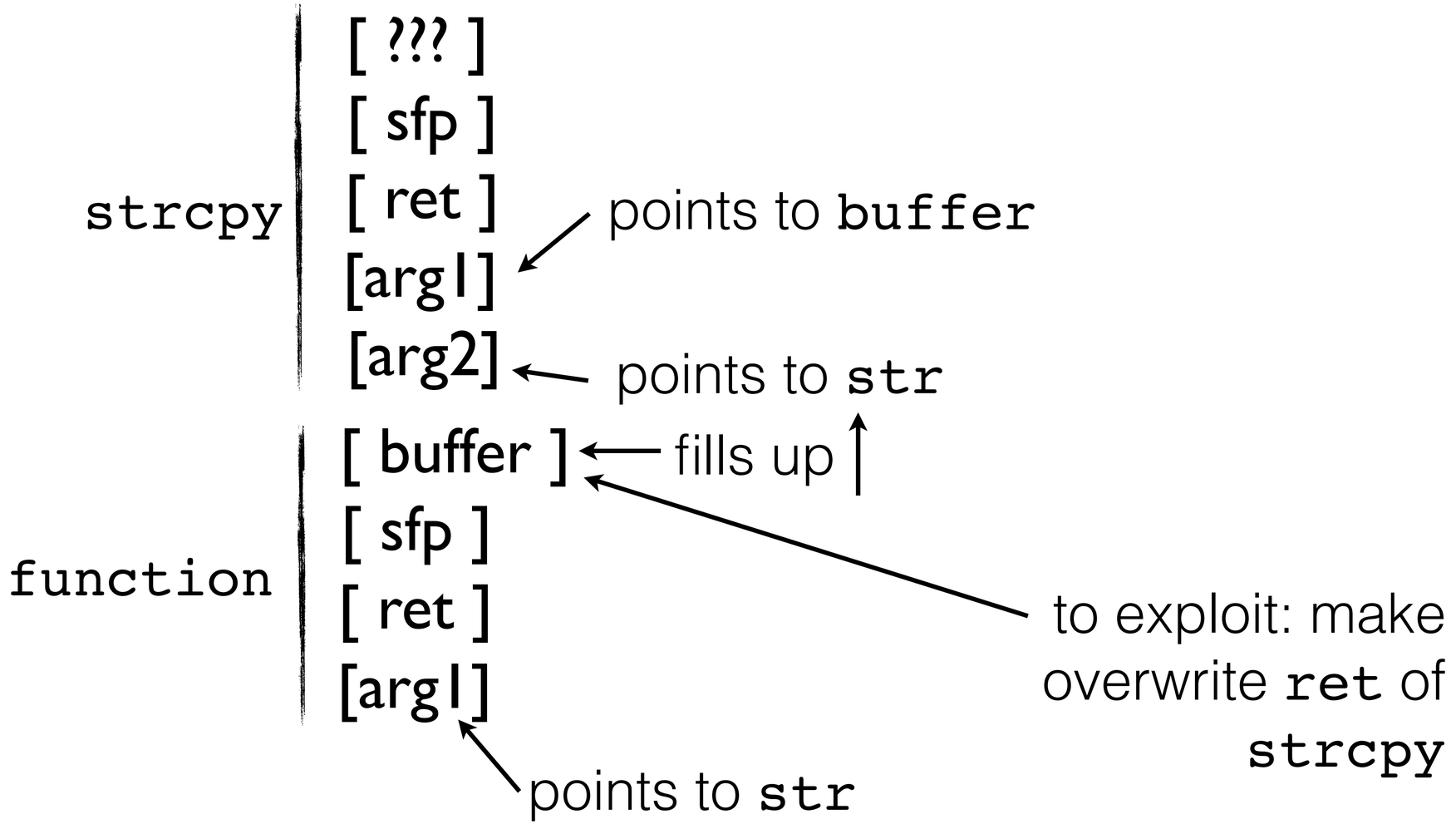


upward growing stack buffer overflow

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```



upward growing stack buffer overflow




```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```

Address Space Layout Randomization (ASLR)

motivation: attacker must know address of libraries/instructions to return to

Address Space Layout Randomization (ASLR)

motivation: attacker must know address of libraries/instructions to return to

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution

Address Space Layout Randomization (ASLR)

motivation: attacker must know address of libraries/instructions to return to

address of X, randomised

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution

Address Space Layout Randomization (ASLR)

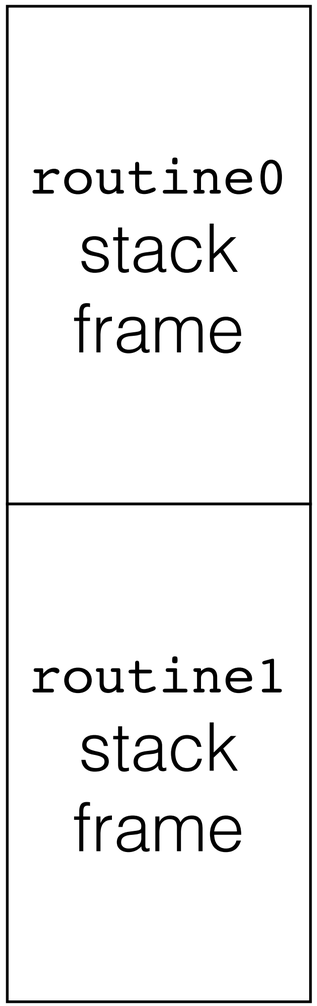
motivation: attacker must know address of libraries/instructions to return to

position independent executable/code

address of X, randomised

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution

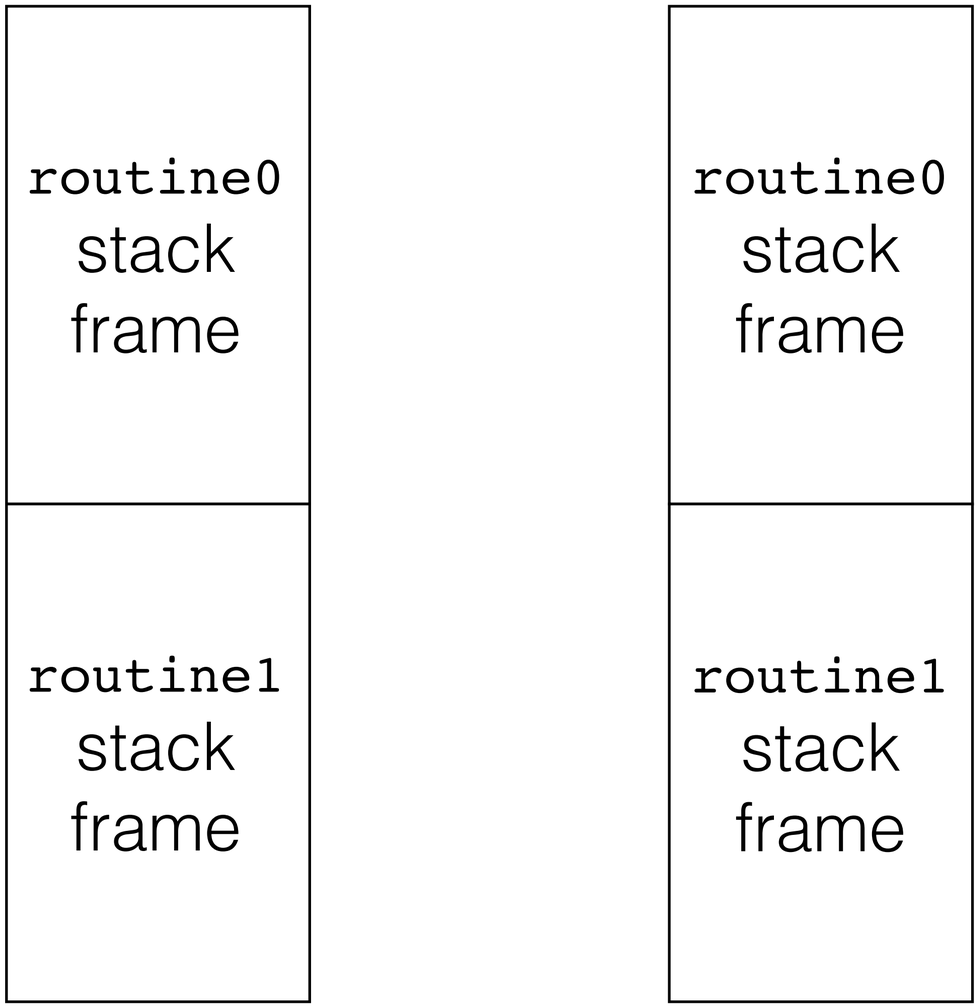
shadow stack



working stack

(overflow occurs here)

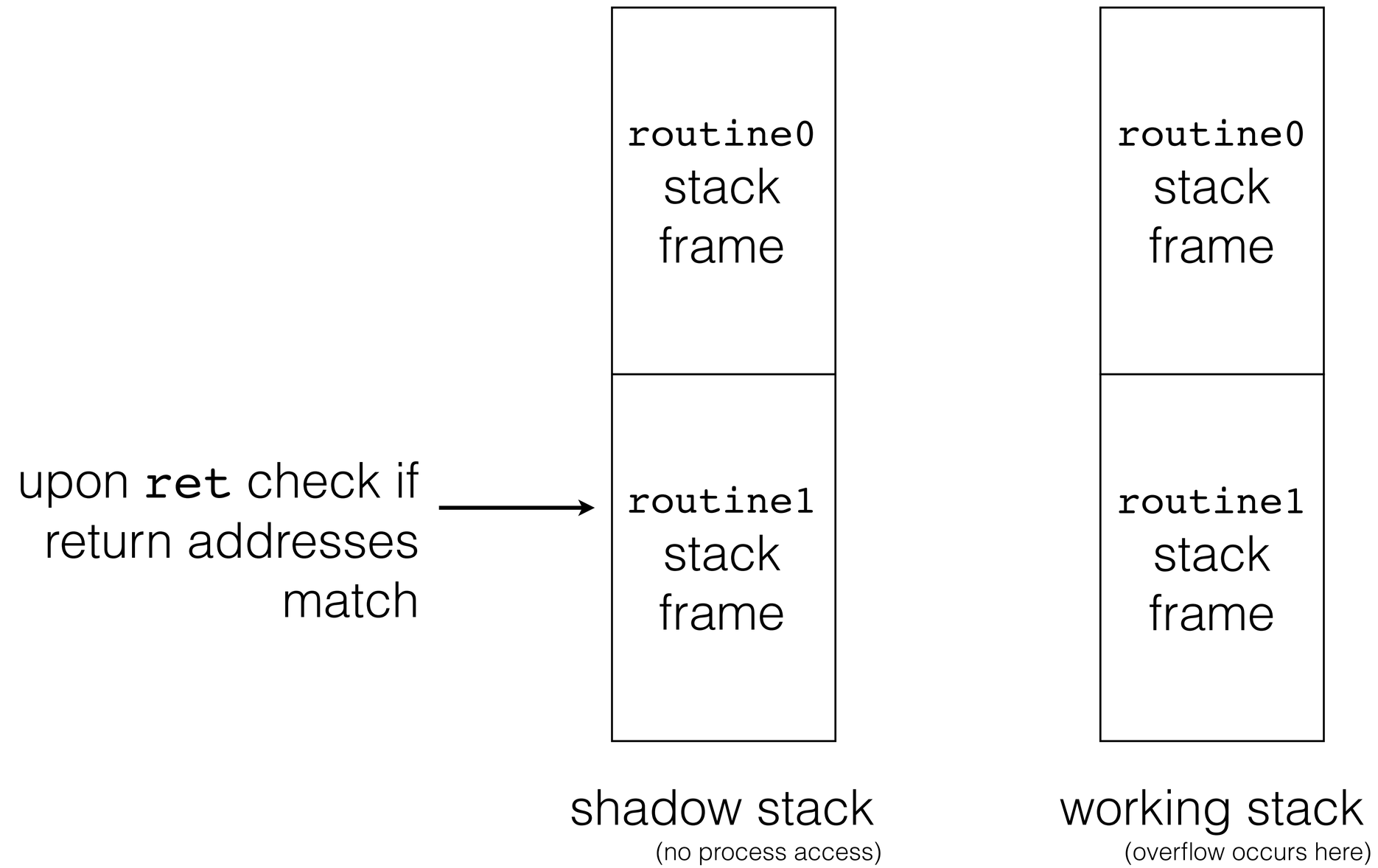
shadow stack



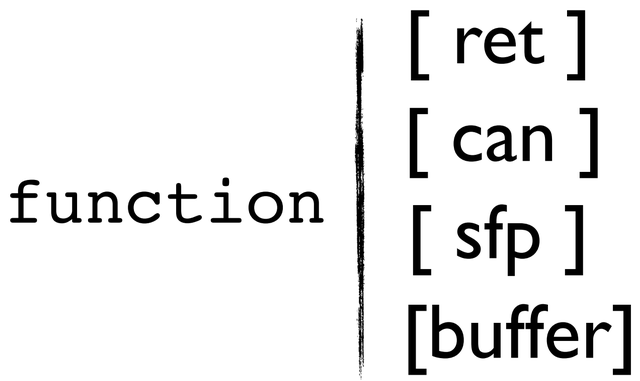
shadow stack
(no process access)

working stack
(overflow occurs here)

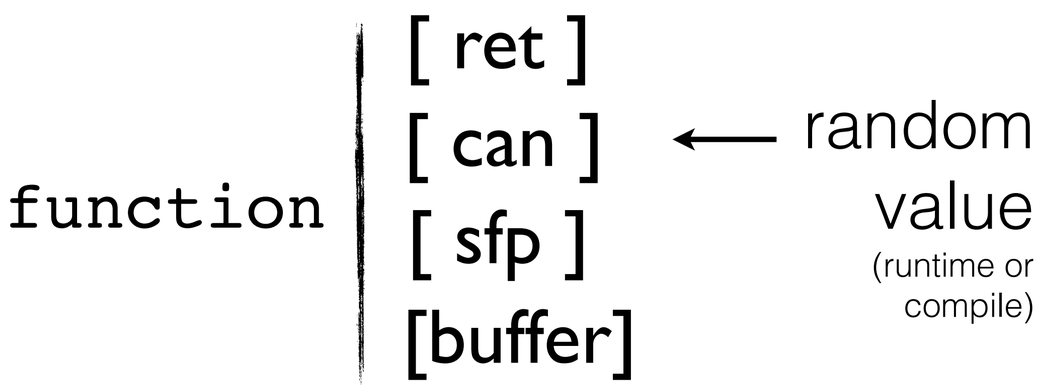
shadow stack



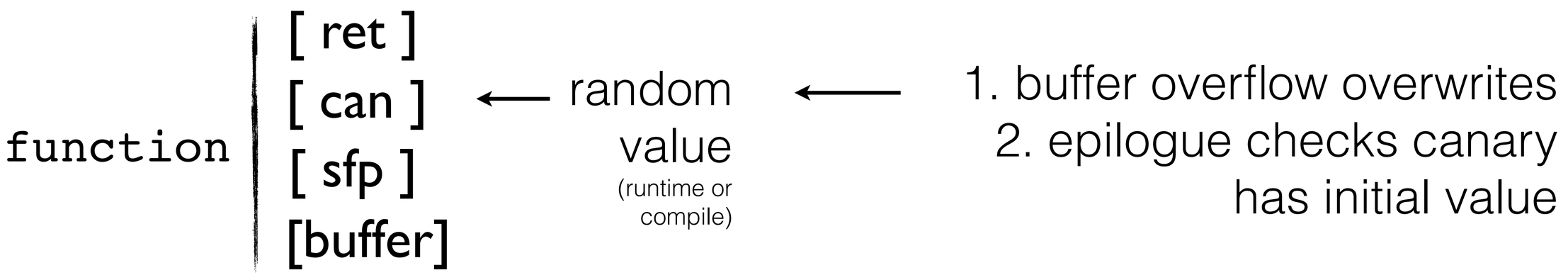
StackGuard (canary)



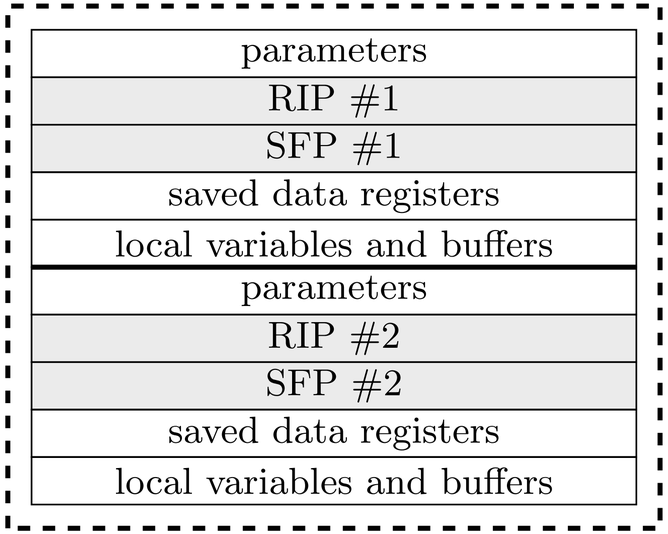
StackGuard (canary)



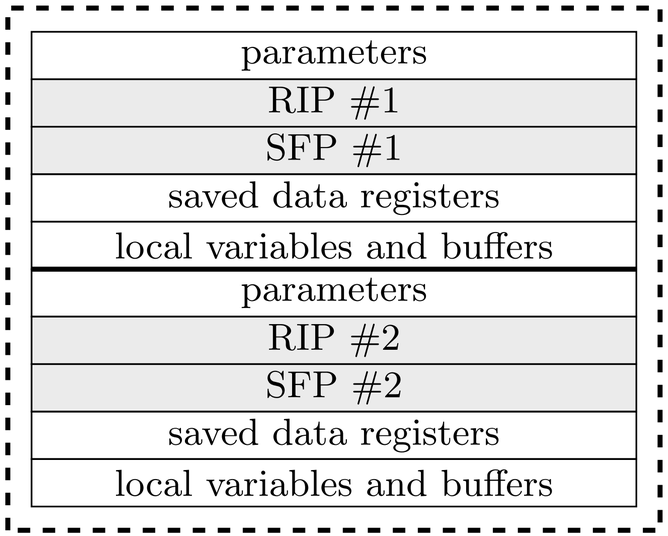
StackGuard (canary)



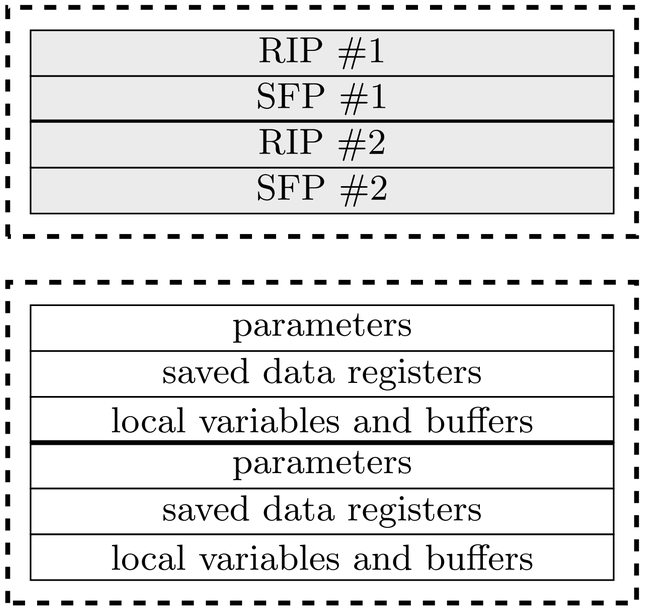
SCADS



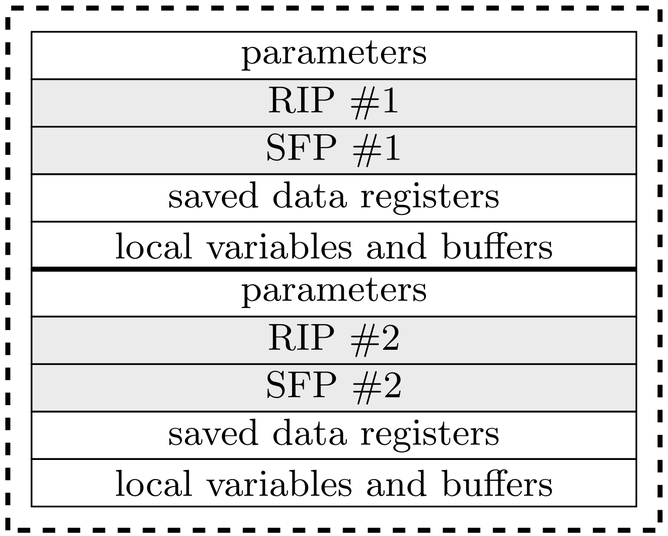
(a) Single call stack.



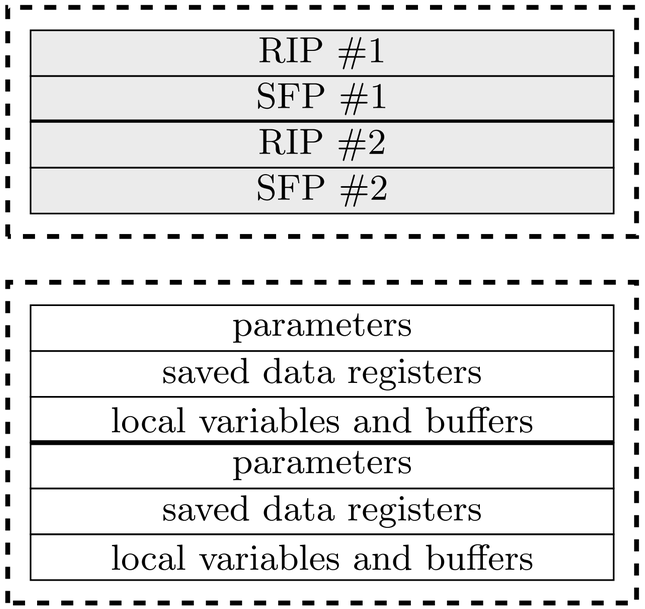
(a) Single call stack.



(b) Separated CS and DS.



(a) Single call stack.



(b) Separated CS and DS.

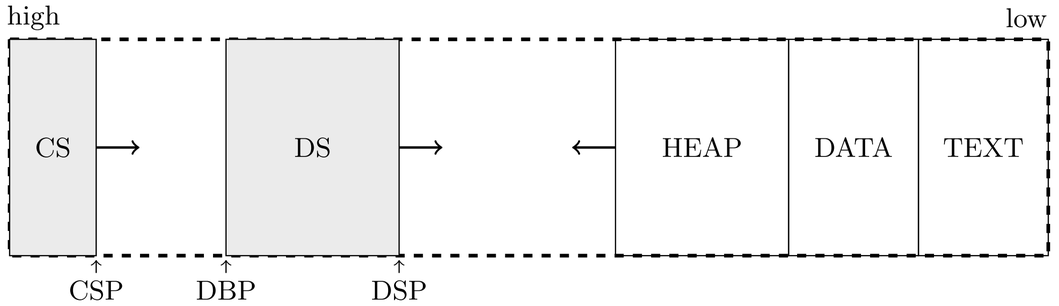


Fig. 2. Virtual address space layout of a user mode process compiled with SCADS.