Memory safety: attacks, defenses, and principles

aslan@cs.au.dk

http://www.pl-enthusiast.net/2014/07/21/memory-safety/

BY MICHAEL HICKS | JULY 21, 2014 · 7:09 AM

↓ Jump to Comments

What is memory safety?

I am in the process of putting together a MOOC on software security, which goes live in October. At the moment I'm finishing up material on buffer overflows, format string attacks, and other sorts of vulnerabilities in C. After presenting this material, I plan to step back and say, "What do these errors have in common? They are violations of *memory safety*." Then I'll state the definition of memory safety, say why these vulnerabilities are violations of memory safety, and conversely say why memory safety, e.g., as ensured by languages like Java, prevents them.

No problem, right? Memory safety is a common technical term, so I expected its definition would be easy to find (or derive). But it's much trickier than I thought.

Traditional definitions of Memory Safety

e.g., based on [SoK: Eternal War in Memory]

- Bad things, *called memory access errors,* must not occur
  - buffer overflow — (PART OF TODAY'S LECTURE)
  - null pointer dereference
  - use after free
  - use of uninitialized memory
  - illegal free (of an already-freed pointer, or a non-malloced pointer)

  (ASSUMED TO KNOW; DO YOU?)

- Not a very satisfying definition: Wikipedia article for memory safety lists many other bad things that must not happen; such lists are non-exhaustive :(
  - Ideally, ruling out above errors out should be a *consequence* of a good definition!
    - Alas, we don't really have one; alternative definitions typically have semantic shortcomings of various flavors (ask me later in the course!). Hicks's principle of *no accesses to undefined memory* is a good approximation.

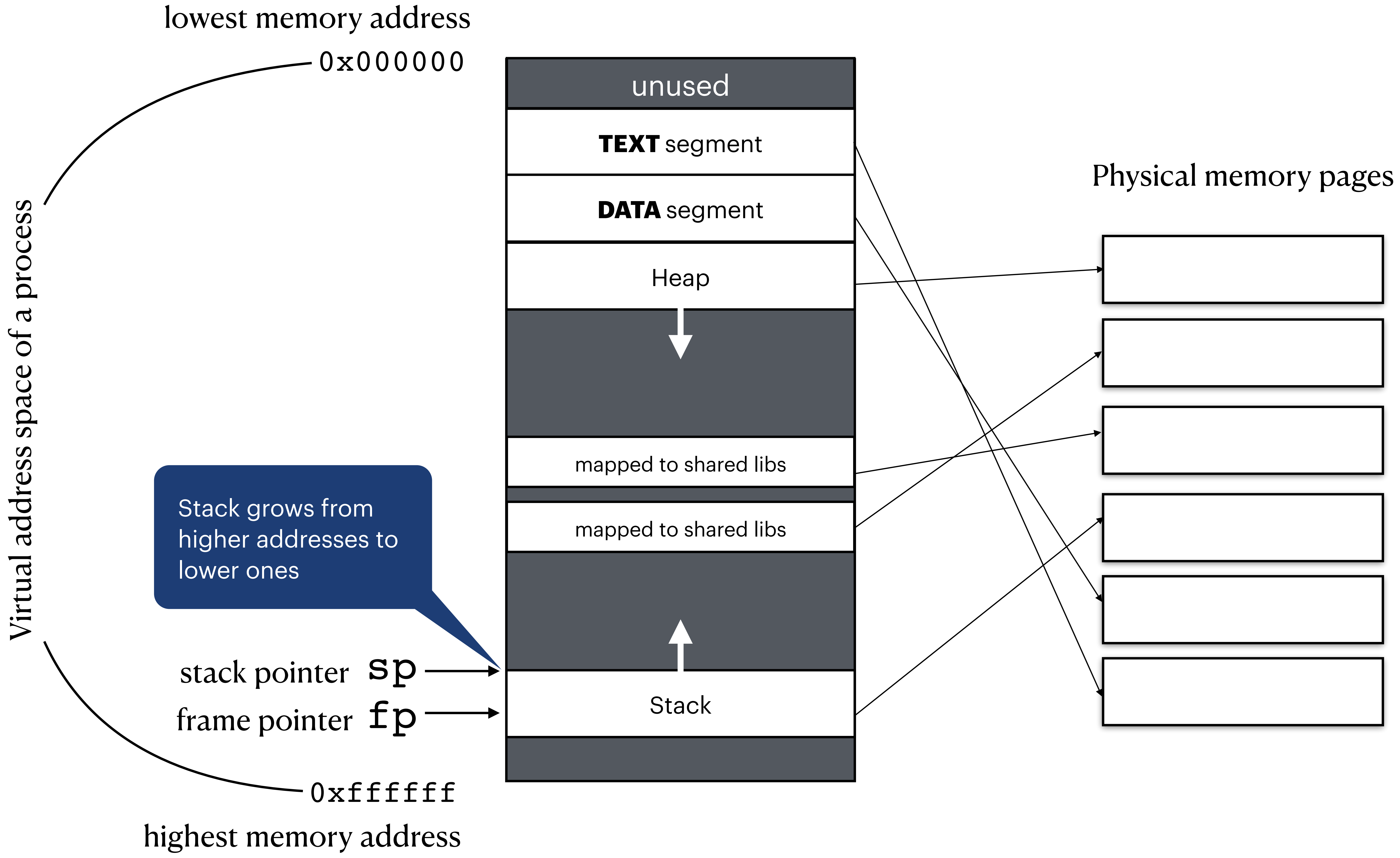
Memory Safety and Security

- Memory safety is paramount to security!
  - Lack of memory safety exposes low-level interiors of crucial abstractions, which leads to catastrophic attacks
- Memory safety by itself does not imply security!
  - Plenty of opportunities for higher-level bugs
    - For example, in Assignment #1, the backend is written in a memory-safe language (JavaScript)
- Today:
  - Introduction to classical memory safety attacks
    - Buffer overflows
    - Return-oriented programming (ROP)
  - Defenses
    - Page table protection (W⊕X)
    - Address-Space Layout Randomization (ASLR)
    - Control-Flow Integrity (CFI)

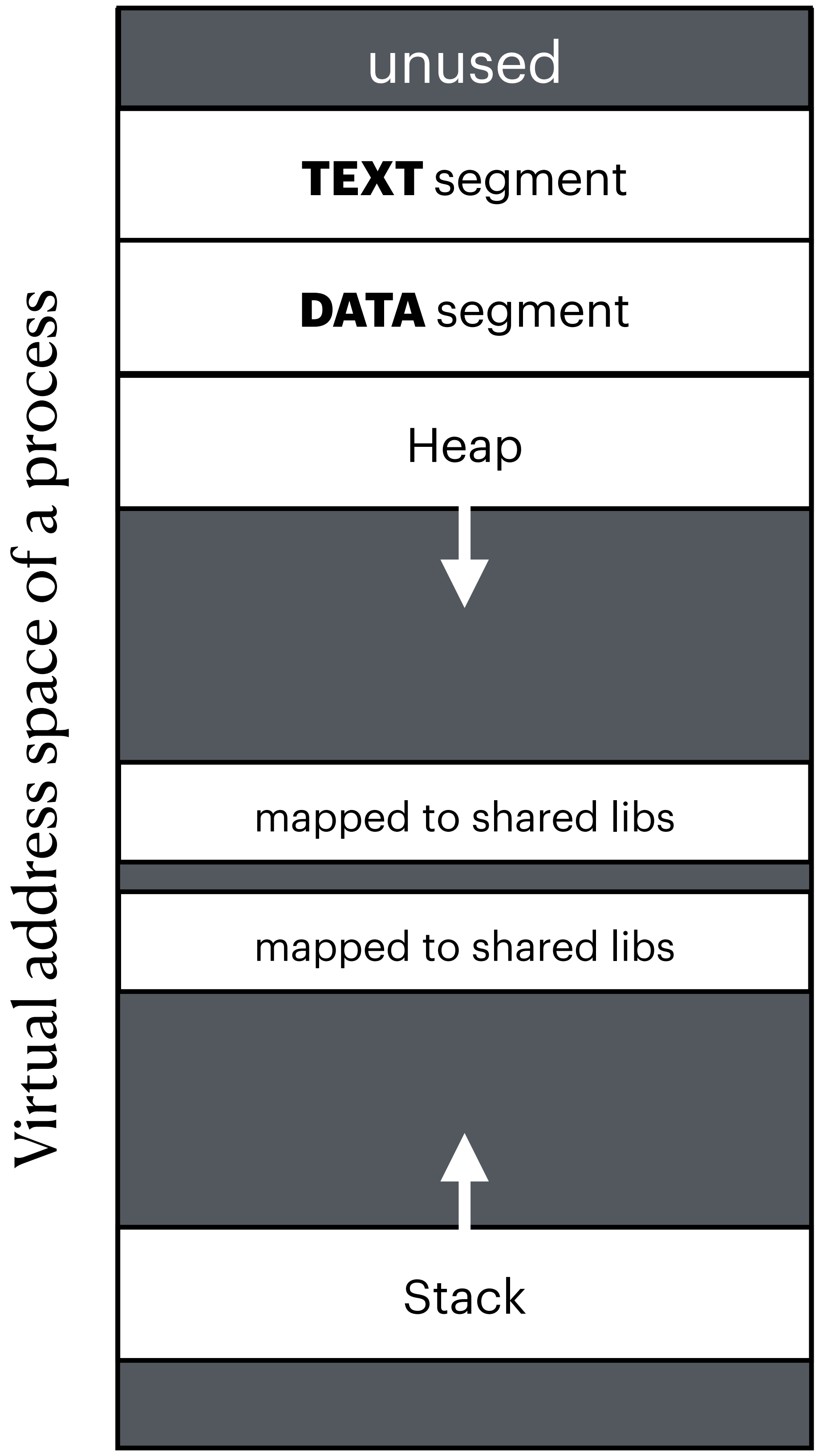
Buffer overflows

Acknowledgments: Andrei Sabelfeld

Background: Virtual Address Space

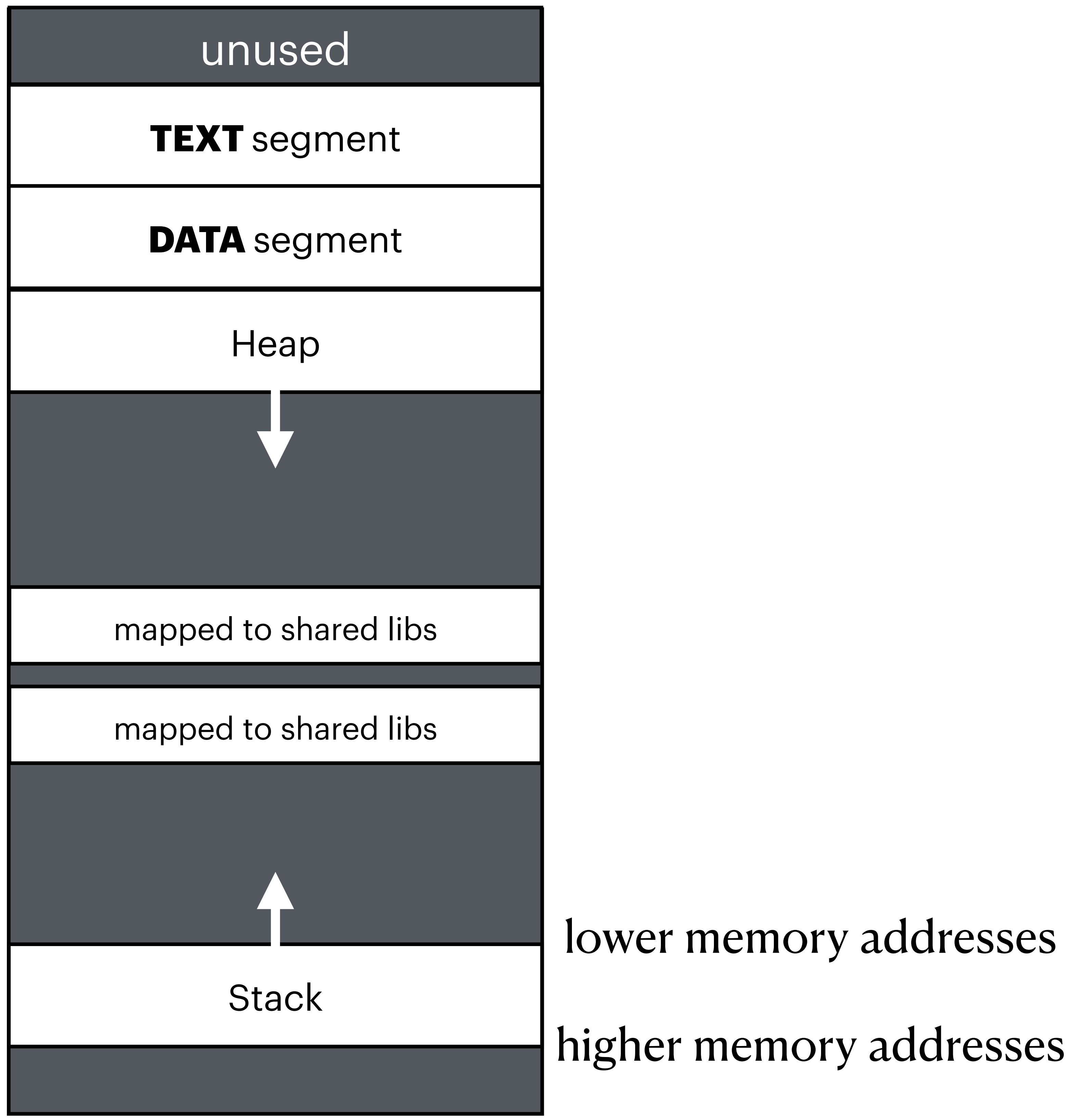


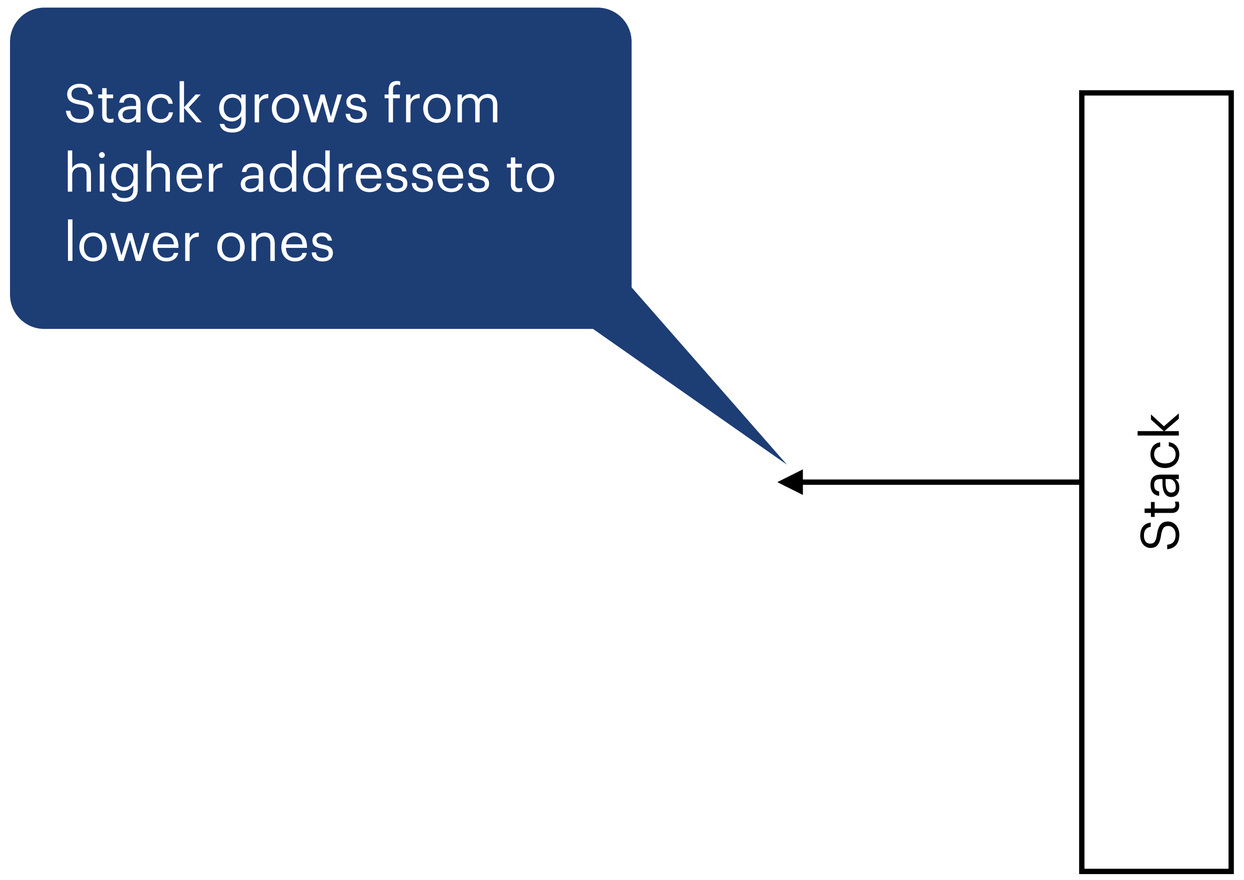
/proc/[proc_id]/maps



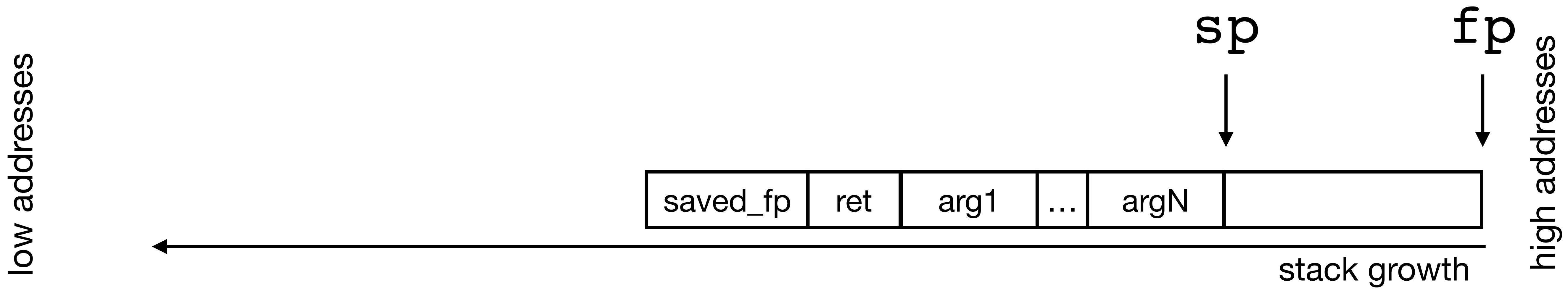

```
$ pmap -X 2448
2448:   ./a.out
         Address Perm   Offset Device  Inode Size Rss Pss Referenced Anonymous Swap Locked Mapping
        00400000 r-xp 00000000  fd:01 262808    4   4   4          4         0    0      0 a.out
        00600000 r--p 00000000  fd:01 262808    4   4   4          4         4    0      0 a.out
        00601000 rw-p 00001000  fd:01 262808    4   4   4          4         4    0      0 a.out
        00d32000 rw-p 00000000  00:00      0  132   4   4          4         4    0      0 [heap]
    7f5398b47000 r-xp 00000000  fd:01 393517 1792 248  20        248         0    0      0 libc-2.23.so
    7f5398d07000 ---p 001c0000  fd:01 393517 2048   0   0          0         0    0      0 libc-2.23.so
    7f5398f07000 r--p 001c0000  fd:01 393517   16  16  16         16        16    0      0 libc-2.23.so
    7f5398f0b000 rw-p 001c4000  fd:01 393517    8   8   8          8         8    0      0 libc-2.23.so
    7f5398f0d000 rw-p 00000000  00:00      0   16   8   8          8         8    0      0
    7f5398f11000 r-xp 00000000  fd:01 393333  152 128  10        128         0    0      0 ld-2.23.so
    7f5399128000 rw-p 00000000  00:00      0   12  12  12         12        12    0      0
    7f5399136000 r--p 00025000  fd:01 393333    4   4   4          4         4    0      0 ld-2.23.so
    7f5399137000 rw-p 00026000  fd:01 393333    4   4   4          4         4    0      0 ld-2.23.so
    7f5399138000 rw-p 00000000  00:00      0    4   4   4          4         4    0      0
    7ffc8725e000 rw-p 00000000  00:00      0  136  12  12         12        12    0      0 [stack]
    7ffc8735c000 r-xp 00000000  00:00      0    8   4   0          4         0    0      0 [vdso]
ffffffffff600000 r-xp 00000000  00:00      0    4   0   0          0         0    0      0 [vsyscall]
                                             ==== === === ========== ========= ==== ======
                                             4348 464 114        464        80    0      0 KB
```

Stack


unused
TEXT segment
DATA segment
Heap
mapped to shared libs
mapped to shared libs
Stack
lower memory addresses
higher memory addresses


Stack grows from higher addresses to lower ones
Stack

Call stack organization



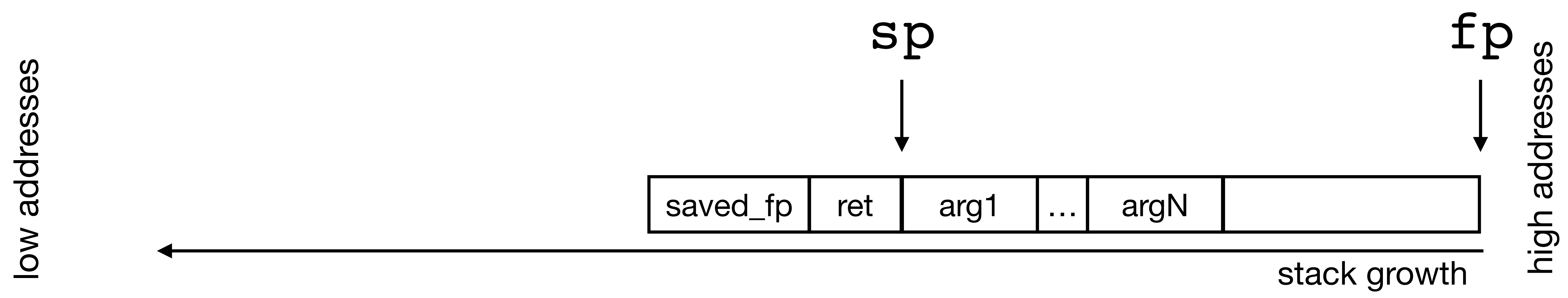
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



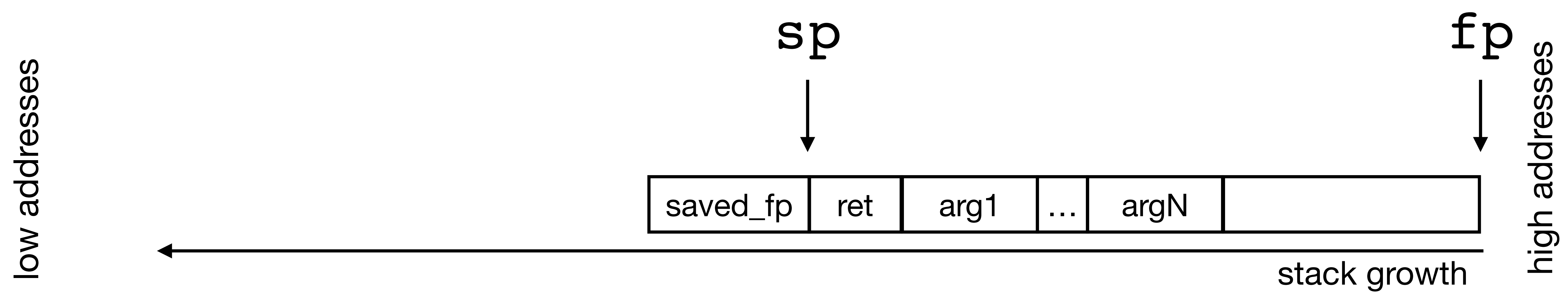
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



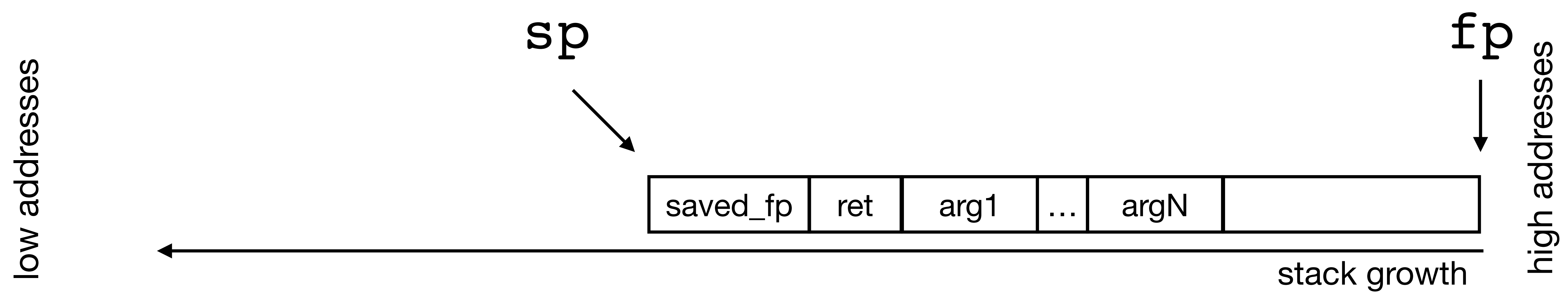
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



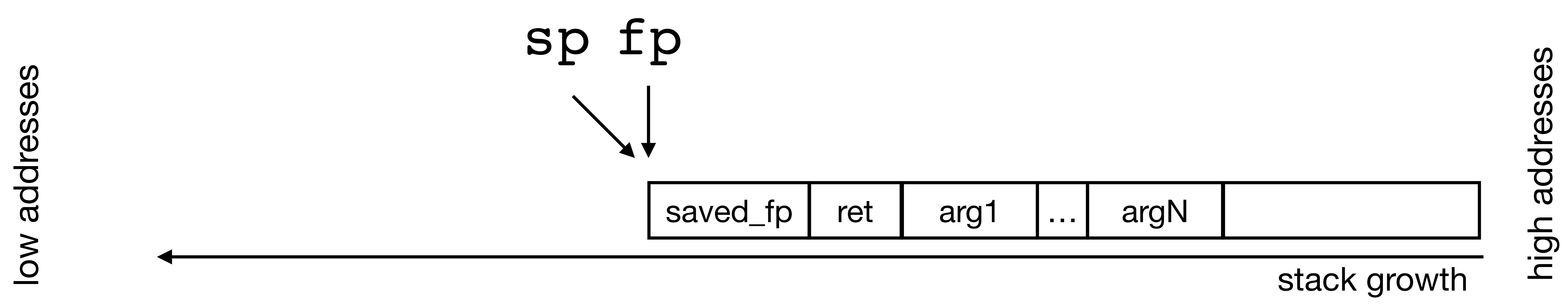
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



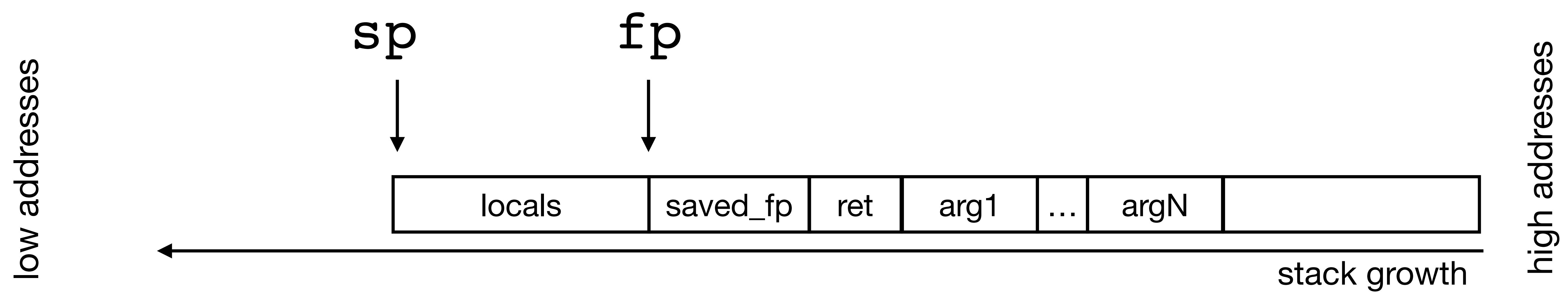
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



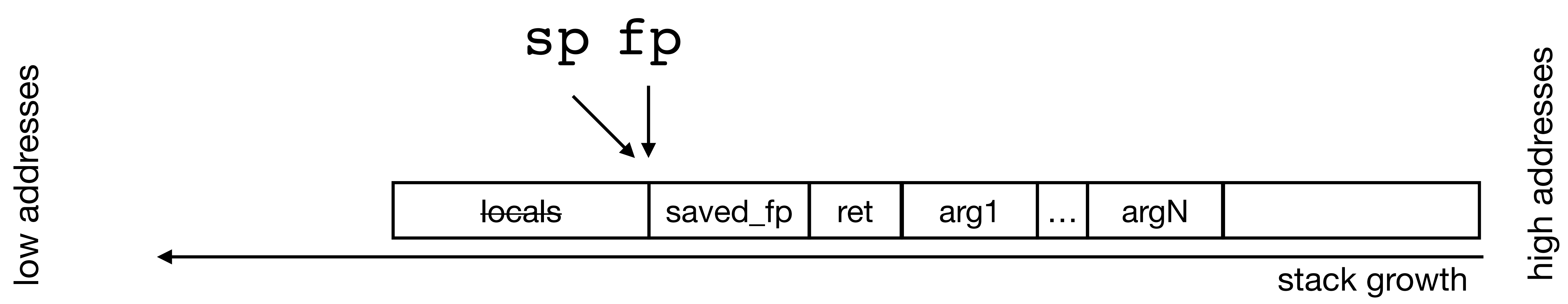
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



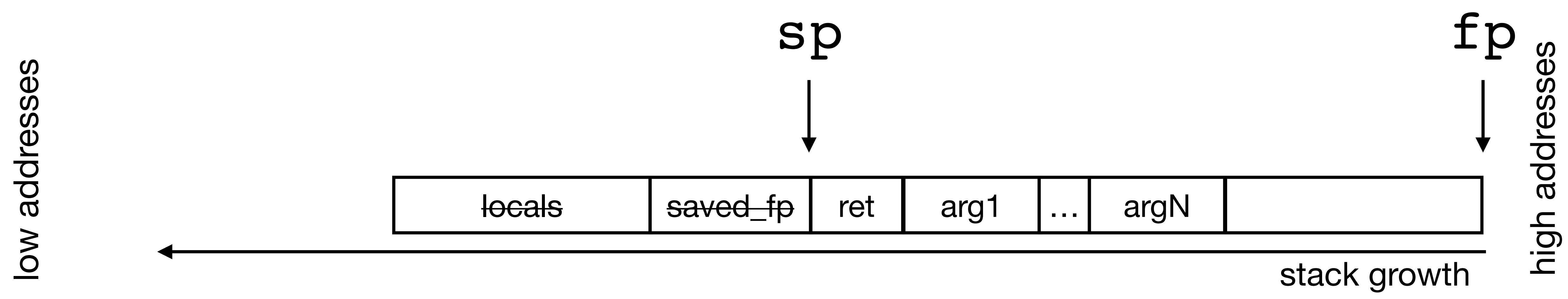
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



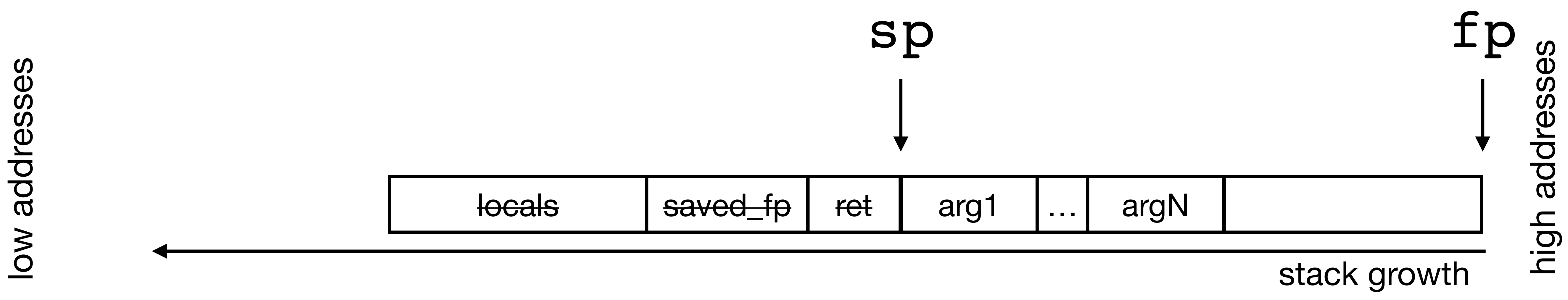
Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Call stack organization



Caller runs:

```
push argN; … ;push arg1;
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (locals)
// … body of the callee
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Buffer overflows

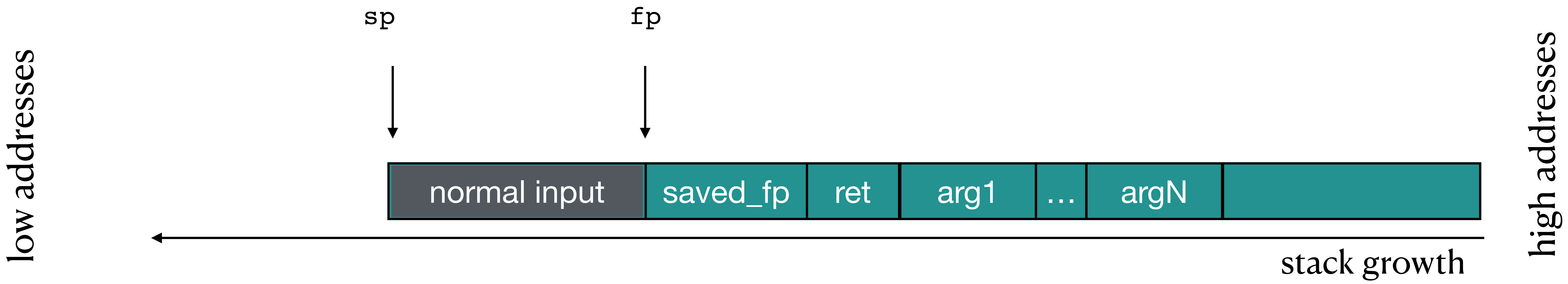
- Classical attack vector
- Extremely prevalent up to mid-2000s
  - variations of the attack still possible today
- Main cause: C and C++ do not perform array bound checks
- "Overflow" = writing past the end of an array/buffer
- Basic attack relies on accomplishing two tasks
  - Hijack control (by overwriting RET address)
  - Plant malicious code (payload)

Traditionally vulnerable C functions

- `strcpy, strcat, sprintf, scanf, sscanf, gets`
  - No bounds checked
- Example:

> - Reads a buffer from stdin
> - No checks for buffer sizes
> - \n (new line) or ^D (EOF) terminate the string

```c
#include <stdio.h>

int main () {
     char buffer [512];
     gets(buffer);
     return 0;
}
```

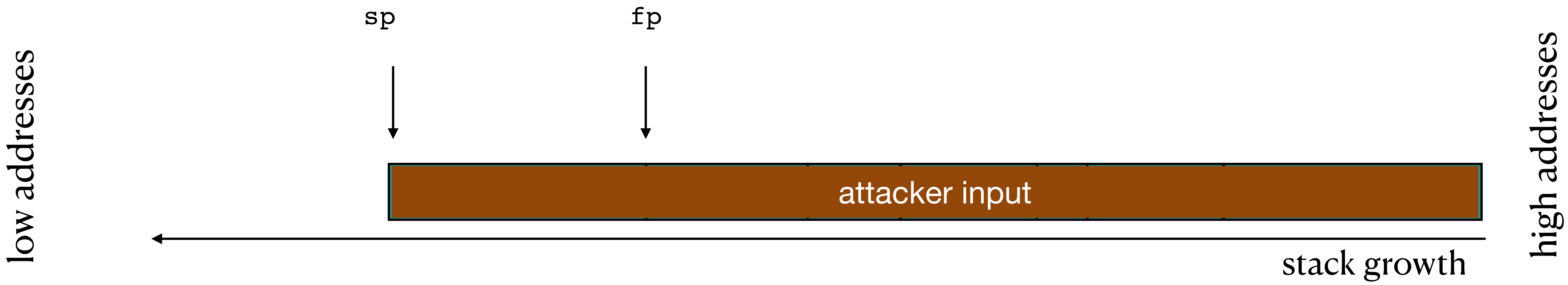


Traditionally vulnerable C functions

- `strcpy, strcat, sprintf, scanf, sscanf, gets`
  - No bounds checked
- Example:

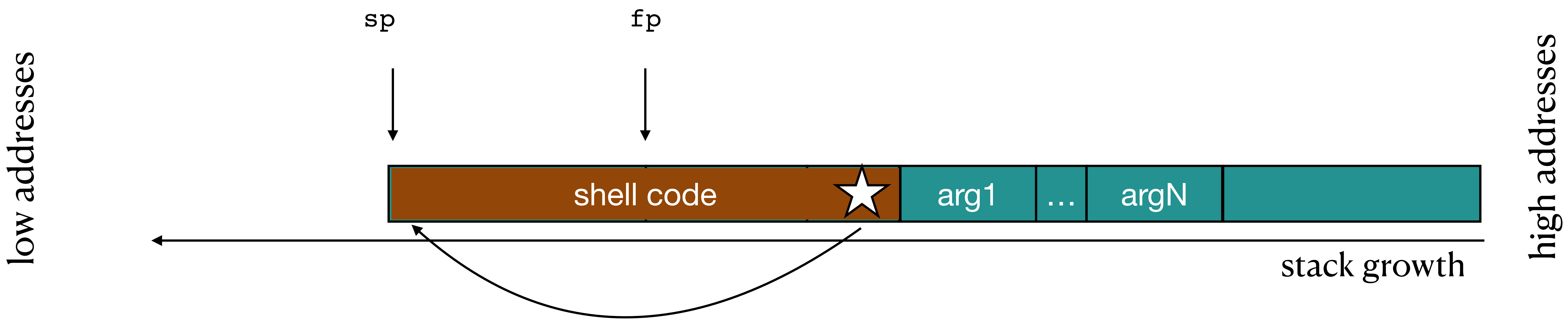
  - Reads a buffer from stdin
  - No checks for buffer sizes
  - \n (new line) or ^D (EOF) terminate the string

```c
#include <stdio.h>

int main () {
     char buffer [512];
     gets(buffer);
     return 0;
}
```



Shellcode

- Shellcode spawns a shell under the uid of the current process
  - If uid is elevated to root, this gives rootshell
- Attacker goals
  - Find how to embed the shellcode
    - in the simplest case: the buffer itself
  - Ensure that writing to the buffer overwrites the return address
  - Ensure that return pointer points to the shellcode



Stack smashing

Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

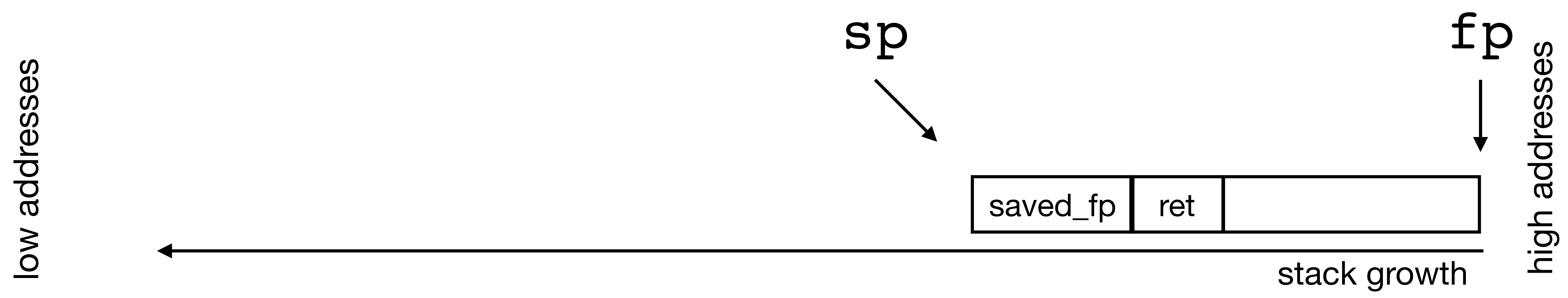
Stack smashing

Caller runs: `call F // push return_address; jmp F`

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



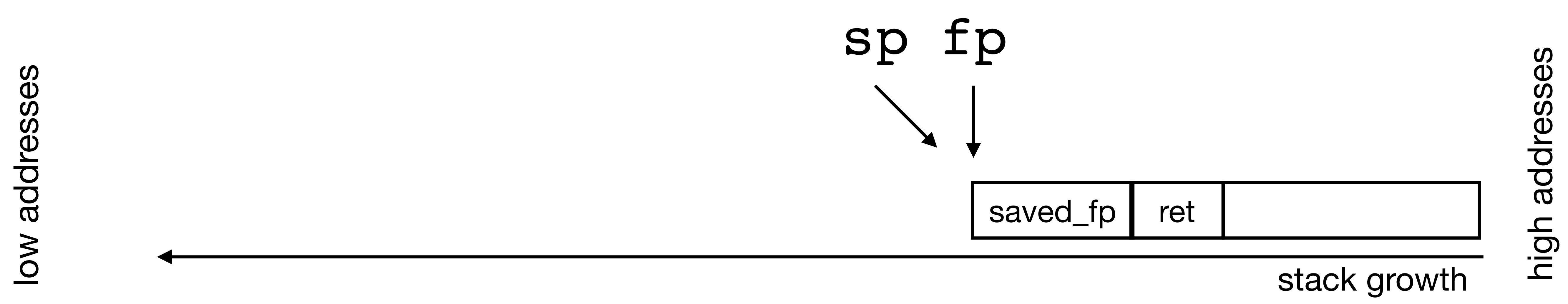
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



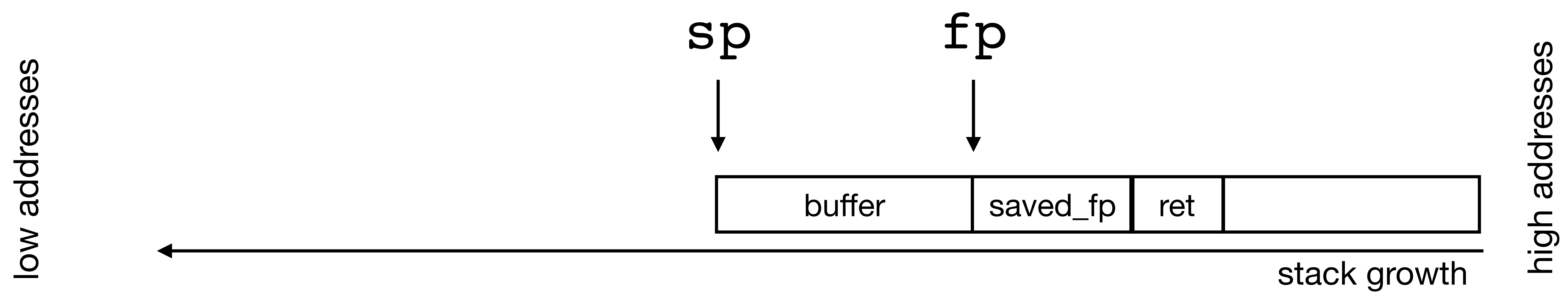
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



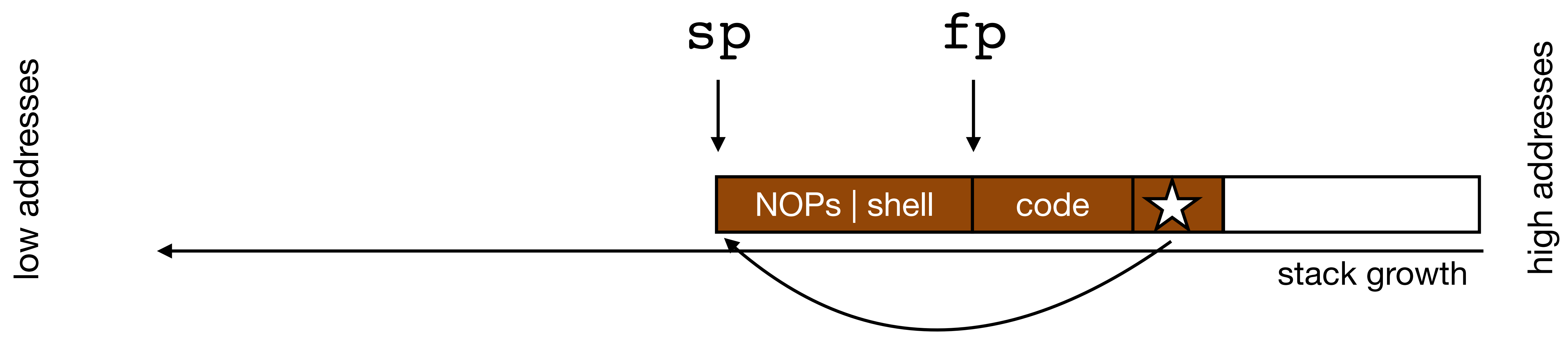
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



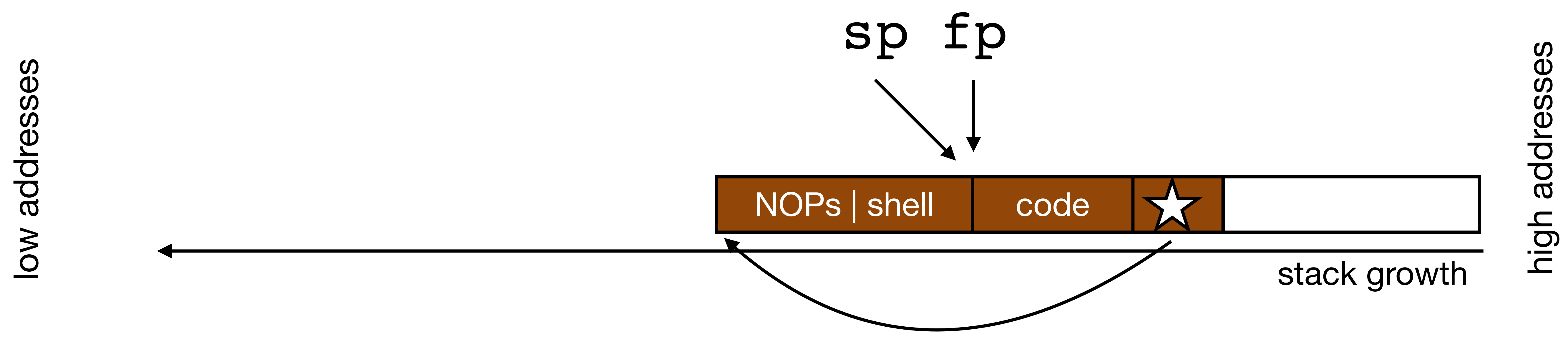
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



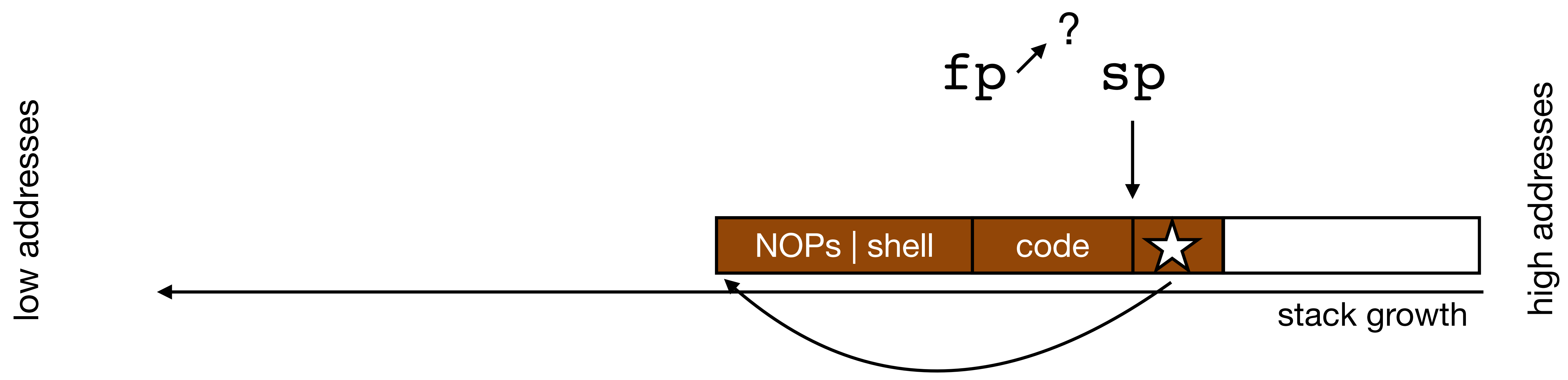
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



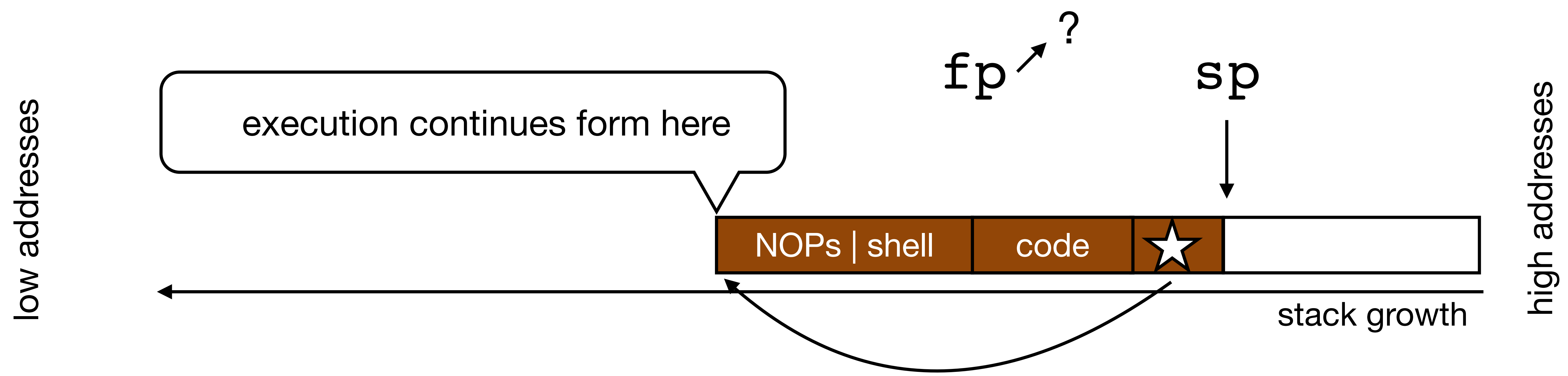
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

Stack smashing



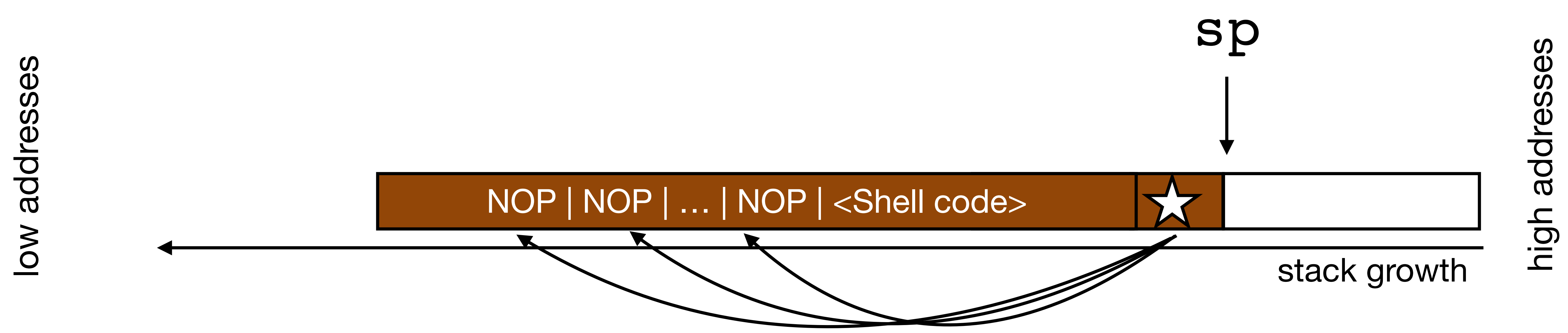
Caller runs:

```
call F // push return_address; jmp F
```

Callee runs:

```
push fp
fp := sp
sp := sp - sizeof (buffer)
gets(buffer)
sp := fp
fp := pop ()
ret // pops ret off the stack
```

NOP sled



Adding NOP instructions in front of the shell code makes it easier go guess the offset from the return address into the start of the buffer.

Protection: coding practice

- Type-safe languages
  - Type-safe systems languages: Rust
- If you have to use C/C++
  - strcpy → strncpy
  - safe libraries
  - restrict the scope of elevated privileges

Protection: basic defenses

- Stack canaries
  - writing a special value on the stack after RET address upon function entry, and checking it before returning
  - effective when attacker is limited to linear sequential writes
    - terminator canary `(0x000aff0d)` that is effective against strcpy/gets or randomized
      - question: why?
  - good performance
  - ineffective against overwrites of RET address and do not protect indirect calls and jumps
- Shadow stacks
  - save return address to separate shadow stack, and compare with it upon return
    - idea: corrupting two return addresses is harder than one
  - performance overhead: shadow stack itself may need to be protected:
    - NEW: hardware support in the modern (circa 2021) processors from Intel and AMD
  - issues with compatibility: extra complexity when unwinding stack in exception handling
  - still only protects backward (RET) edges

Protection: other defenses

- System support
  - Non-executable stack:
    - So-called W⊕X memory protection:
      - pages that can be written cannot be executed
      - executable pages cannot be written to
    - Rational: even if the stack is smashed, the memory page is marked as nonexecutable:
  - Only partial defense:
    - other attacks are possible: return-to-libc/ROP
- Address Space Layout Randomization
- Compiler level: Static Control-Flow Integrity

```
$ pmap -X 2448
2448:   ./a.out
         Address Perm   Offset Device  Inode Size Rss Pss Referenced Anonymous Swap Locked Mapping
        00400000 r-xp 00000000  fd:01 262808    4   4   4          4         0    0      0 a.out
        00600000 r--p 00000000  fd:01 262808    4   4   4          4         4    0      0 a.out
        00601000 rw-p 00001000  fd:01 262808    4   4   4          4         4    0      0 a.out
        00d32000 rw-p 00000000  00:00      0  132   4   4          4         4    0      0 [heap]
    7f5398b47000 r-xp 00000000  fd:01 393517 1792 248  20        248         0    0      0 libc-2.23.so
    7f5398d07000 ---p 001c0000  fd:01 393517 2048   0   0          0         0    0      0 libc-2.23.so
    7f5398f07000 r--p 001c0000  fd:01 393517   16  16  16         16        16    0      0 libc-2.23.so
    7f5398f0b000 rw-p 001c4000  fd:01 393517    8   8   8          8         8    0      0 libc-2.23.so
    7f5398f0d000 rw-p 00000000  00:00      0   16   8   8          8         8    0      0
    7f5398f11000 r-xp 00000000  fd:01 393333  152 128  10        128         0    0      0 ld-2.23.so
    7f5399128000 rw-p 00000000  00:00      0   12  12  12         12        12    0      0
    7f5399136000 r--p 00025000  fd:01 393333    4   4   4          4         4    0      0 ld-2.23.so
    7f5399137000 rw-p 00026000  fd:01 393333    4   4   4          4         4    0      0 ld-2.23.so
    7f5399138000 rw-p 00000000  00:00      0    4   4   4          4         4    0      0
    7ffc8725e000 rw-p 00000000  00:00      0  136  12  12         12        12    0      0 [stack]
    7ffc8735c000 r-xp 00000000  00:00      0    8   4   0          4         0    0      0 [vdso]
ffffffffff600000 r-xp 00000000  00:00      0    4   0   0          0         0    0      0 [vsyscall]
                                             ==== === === ========== ========= ==== ======
                                             4348 464 114        464        80    0      0 KB
```

rw-p … [stack]

Return-to-libc attacks

- If the stack is non-executable, but we can still smash it, where else can we point the return address to?
- Possibility: some existing function in libc (or other linked library) that is executable
  - Lots of "useful" functionality
  - The pages are marked as executable
- Example attack:
  - smash the stack and point "RET" to a libc function that already does what we want
- "Chained return-to-libc" calls:
  - calling multiple functions in succession

Return-Oriented Programming

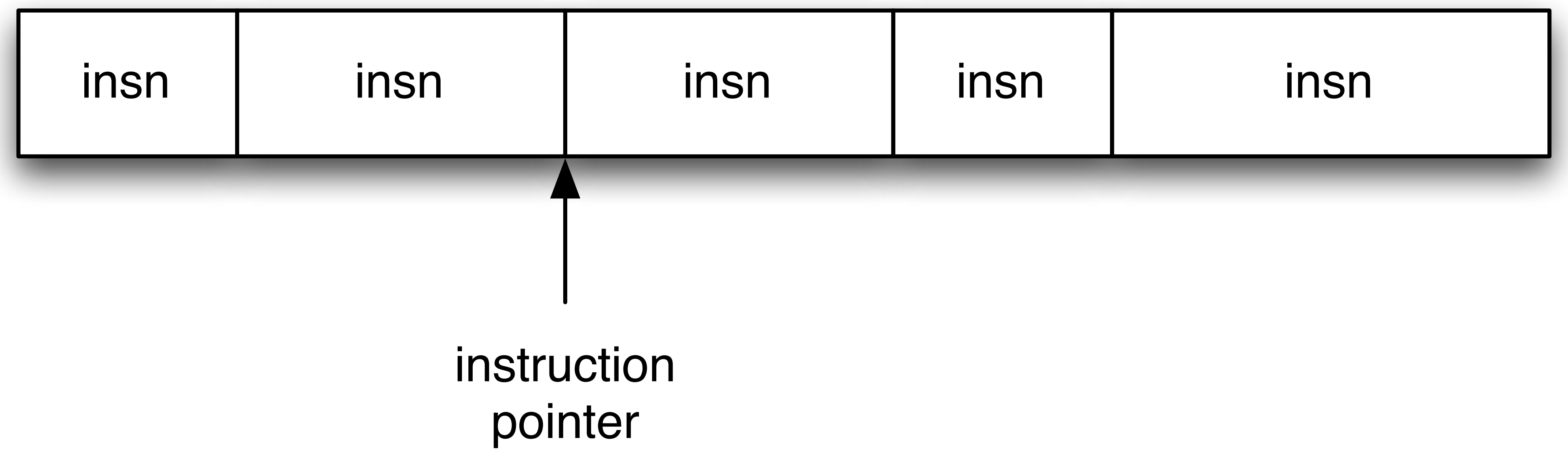
Based on the article by
Ryan Roemer, Erik Buchanan, Hovan Shacham and Stefan Savage

Return-oriented programming

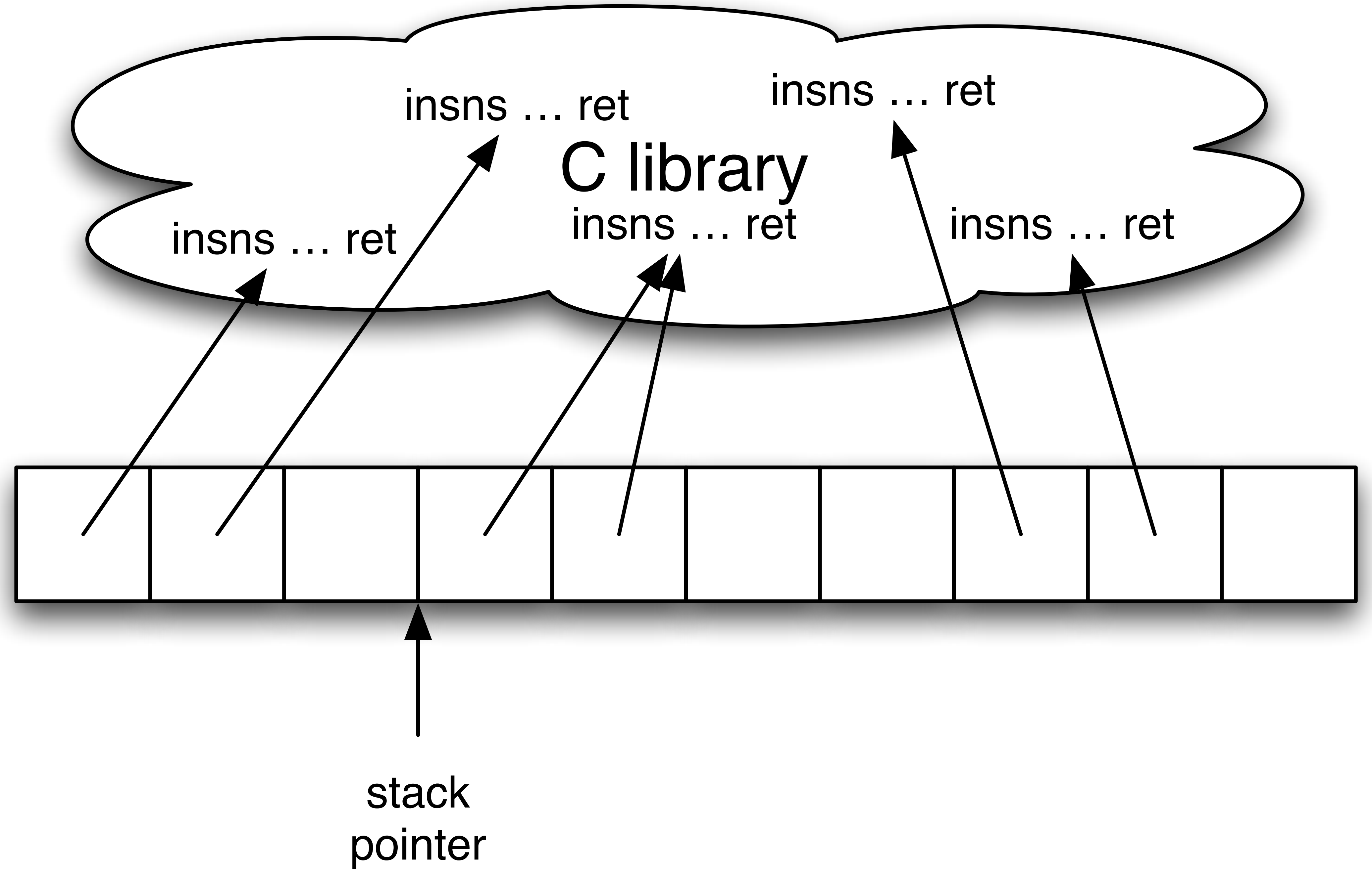
- Observation: attacker doesn't really need a whole libc- function; only a sequence of instructions followed by a return:
  - Example: `pop %edx; ret`
- All we need is chain these sequences to get the desired behavior
- How to chain them?
  - Use stack pointer as the "attack-level instruction pointer"

Ordinary and return-oriented programs

Layout of an ordinary program



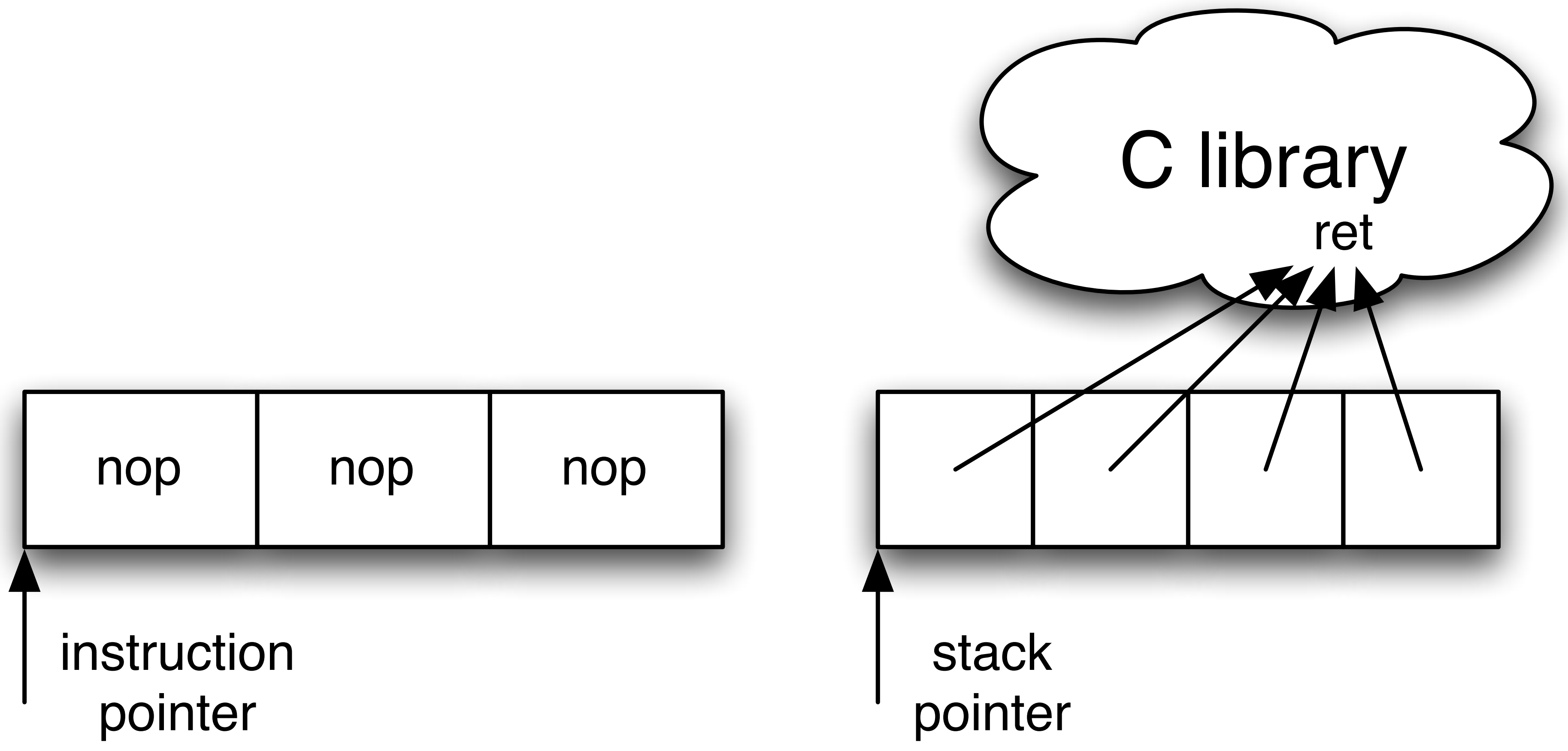
Layout of a return-oriented program



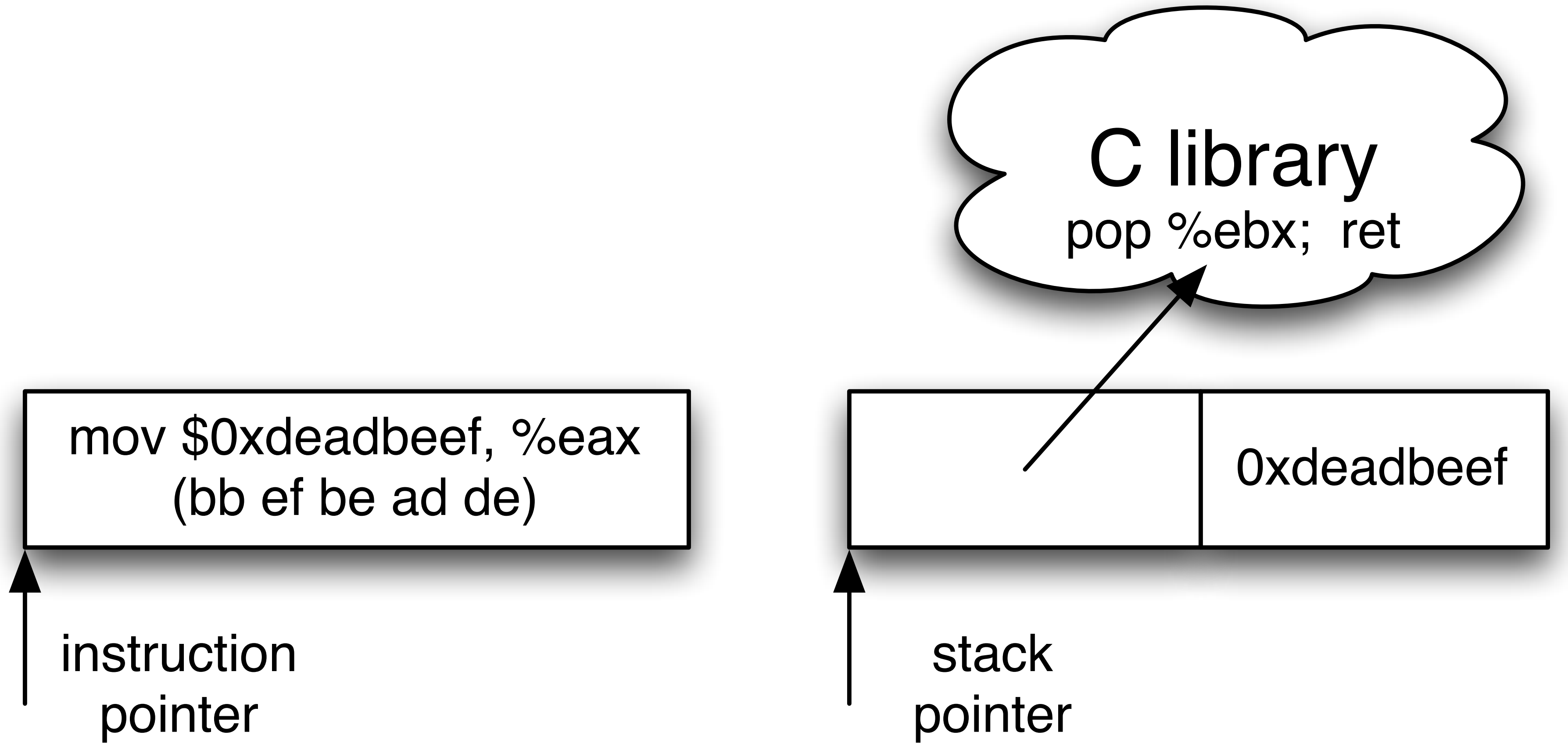
To kick-off an ROP program we start w/ a ret

Ordinary and return-oriented programs

NOP sleds



Immediates



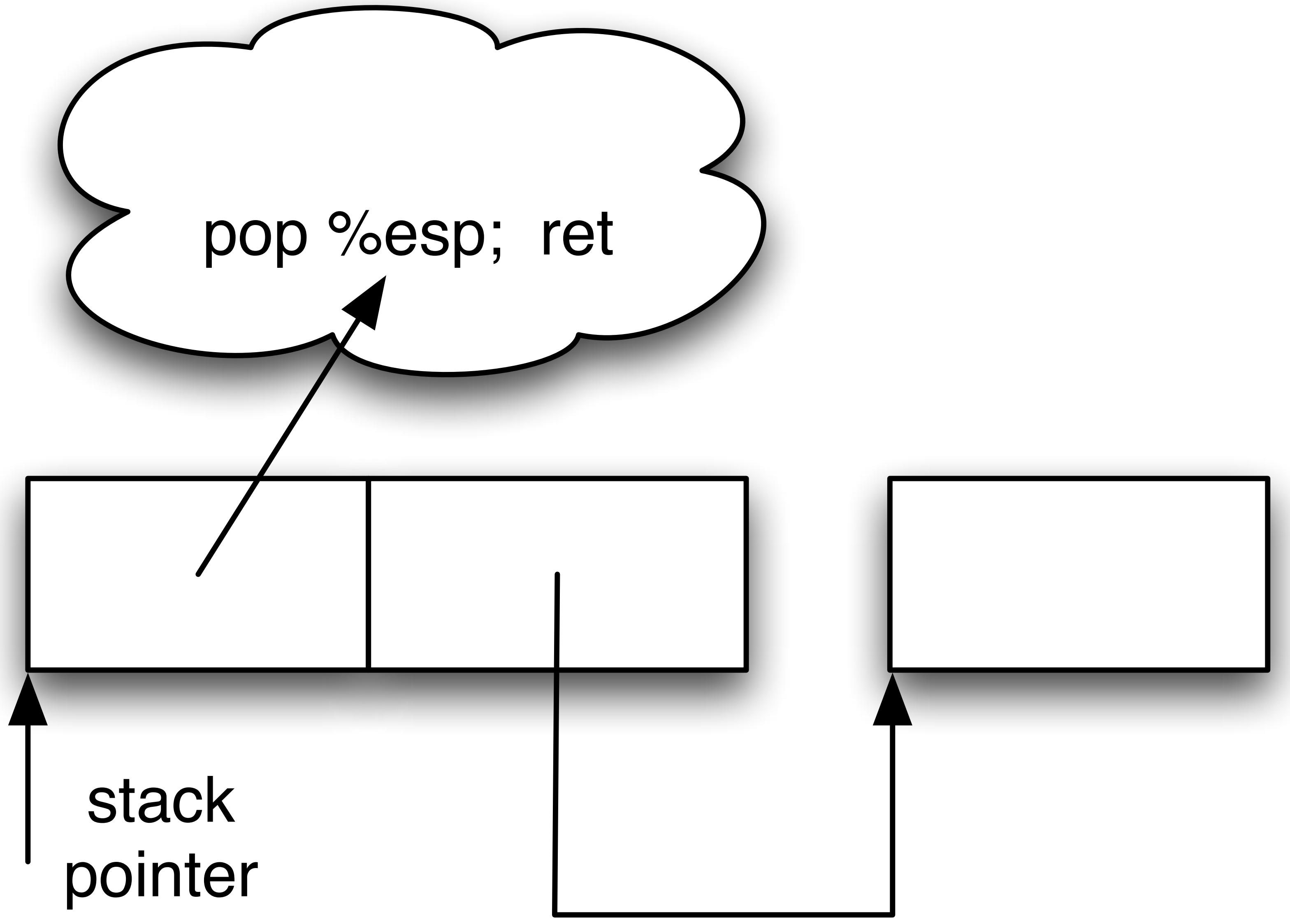
To kick-off an ROP program we start w/ a ret

Q: what does this ROP program do?

pop %esp; ret

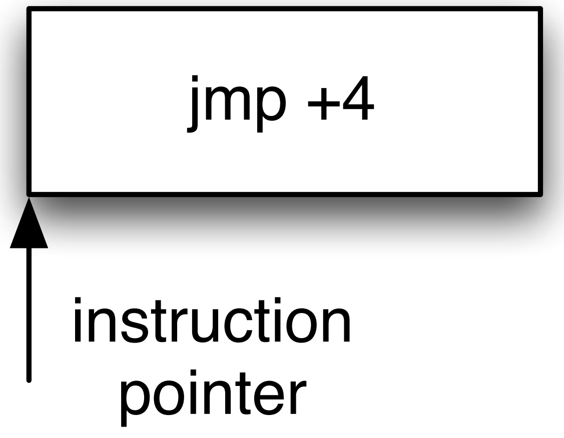
Hint: %esp is the stack pointer

Q: what does this ROP program do?



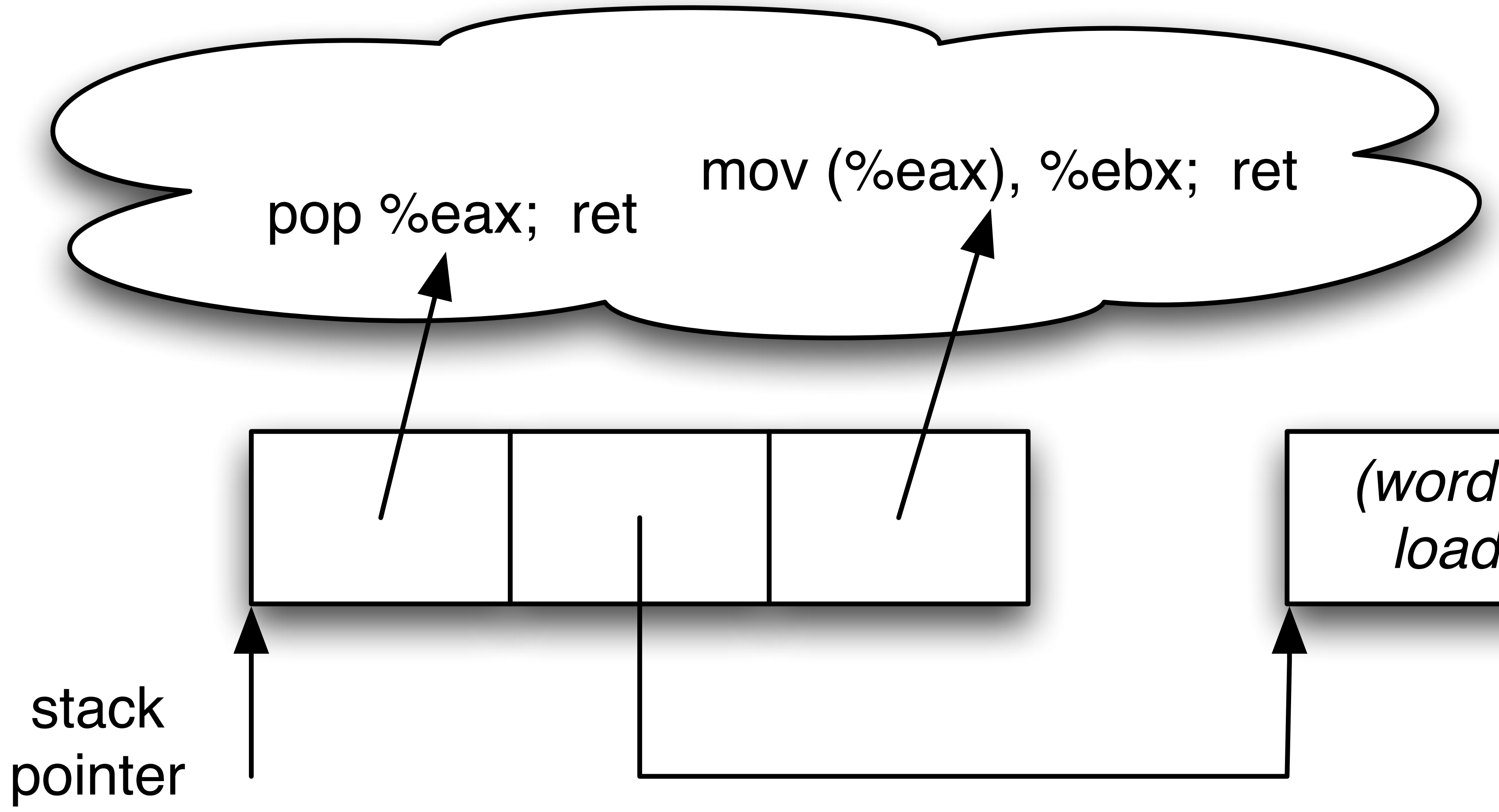
A: Direct jump

ordinary equivalent



Gadgets

An arrangement of values on the stack that causes several sequences to be executed



memory load gadget

Finding instruction sequences

- Intended instruction sequences – every sequence ending in a return
- Unintended instruction sequences
  - x86 uses variable-length encoding of instructions
    - Given a byte stream and a starting offset, the instruction at that offset can be disambiguously decoded; but different offsets will give different decoding
      - we can even start in the middle of an intended instruction
  - What's important is the c3 opcode (ret) and what's before it in the byte stream
  - Compare:

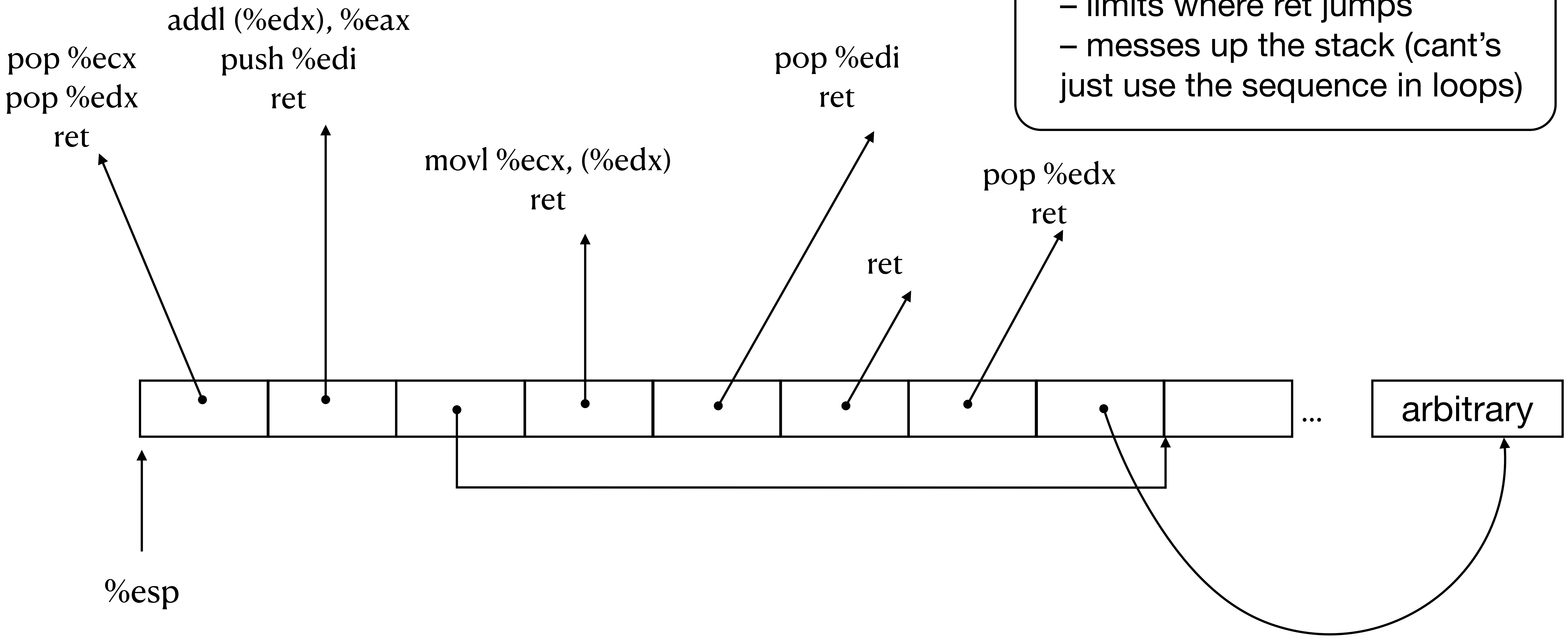
```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)


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

Example gadget: Add

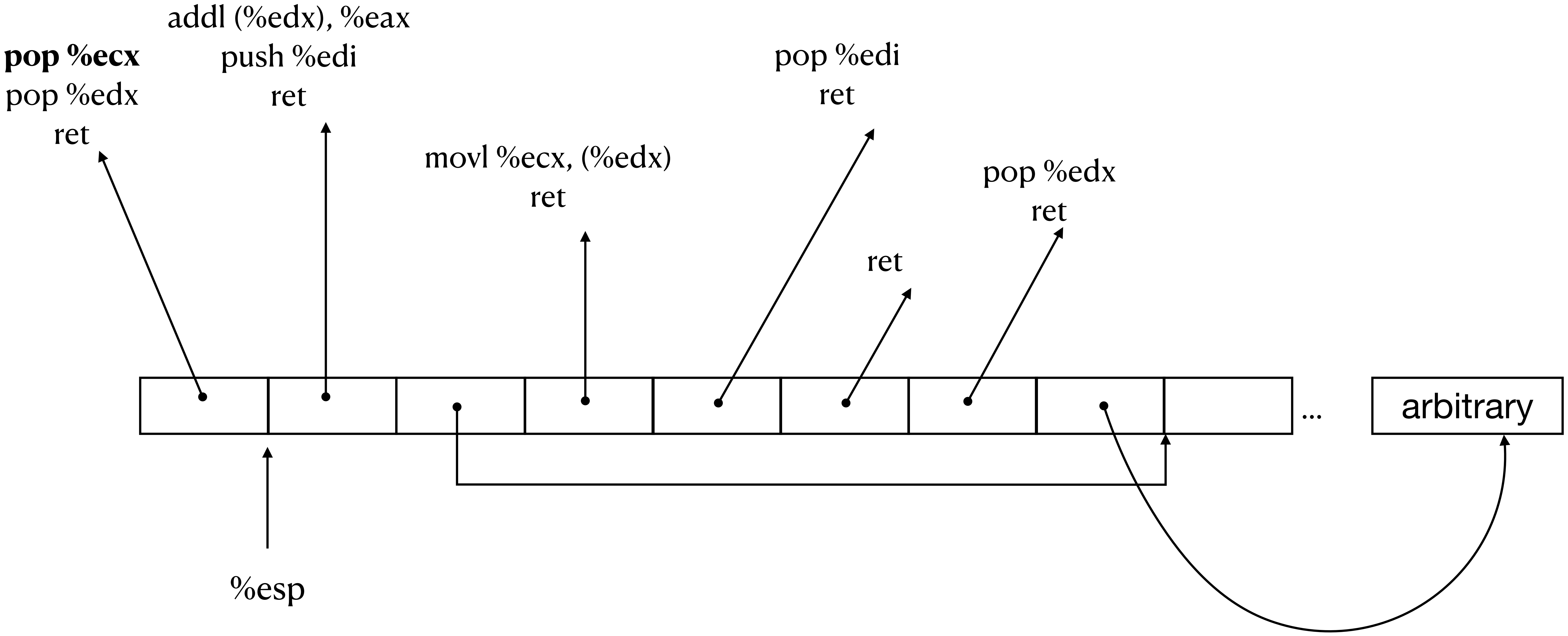
adds the word at %edx to %eax

The most convenient available sequence is `addl (%edx), %eax; push %edi; ret`

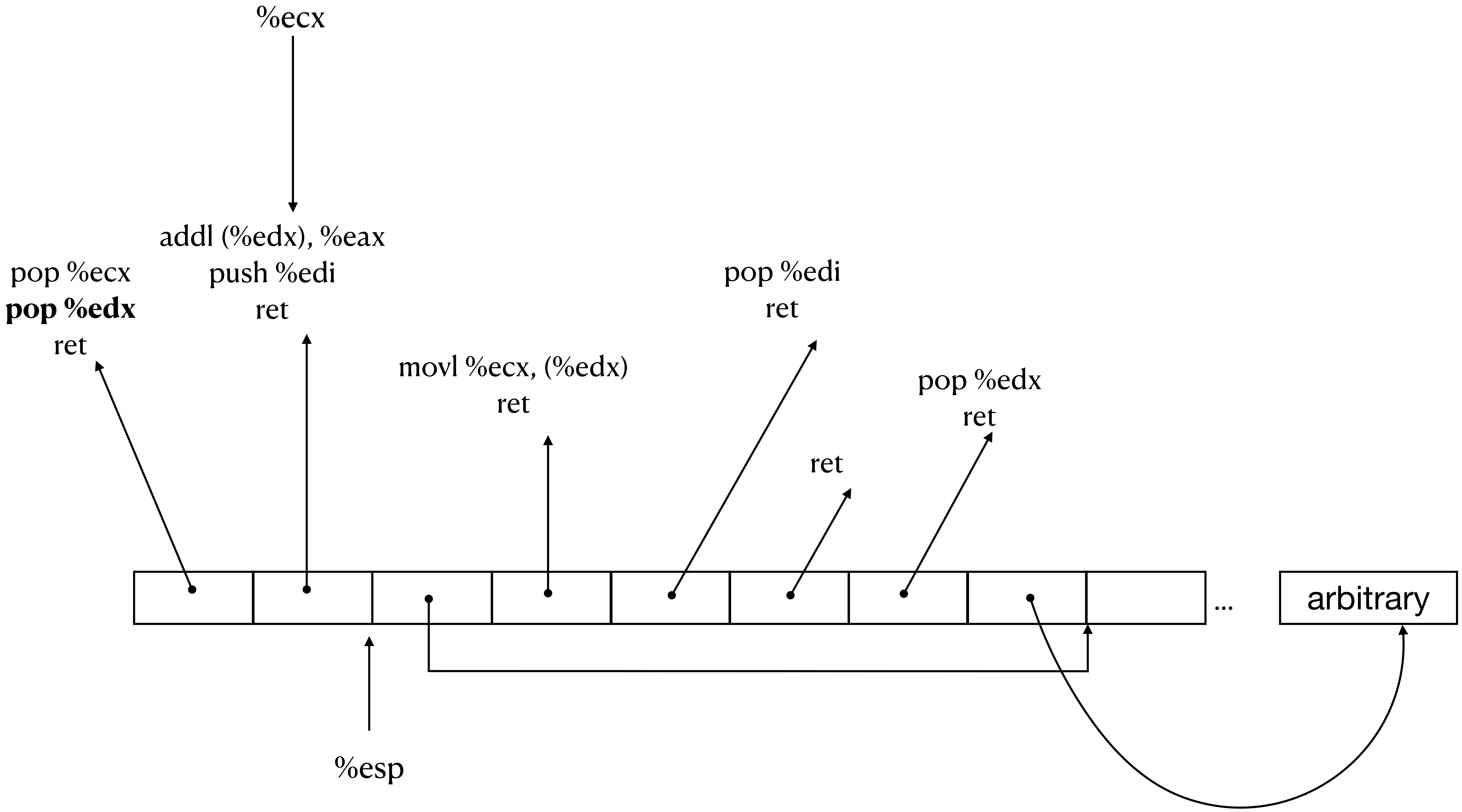
this is a problem
– limits where ret jumps
– messes up the stack (cant's just use the sequence in loops)



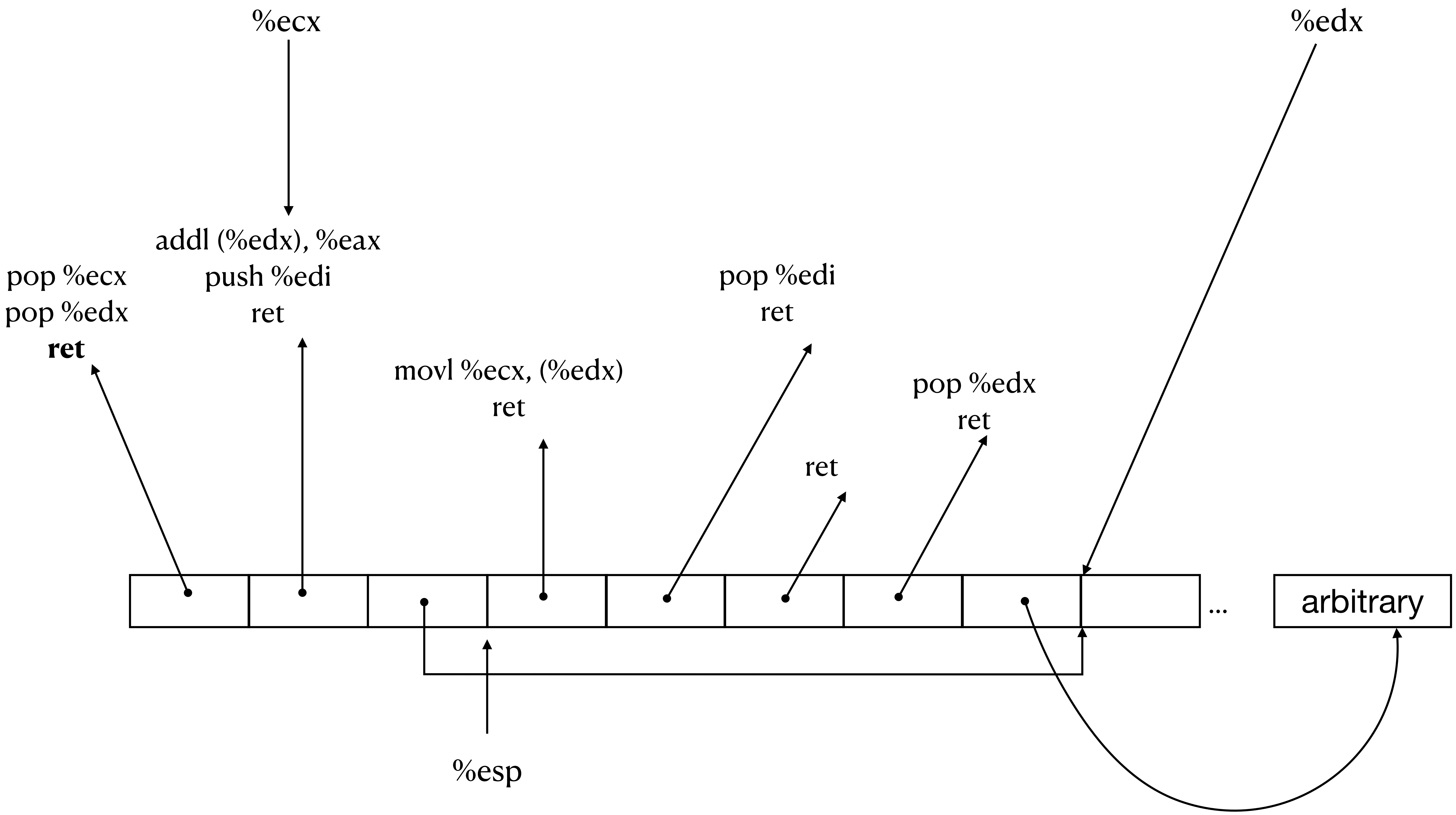
Example gadget: Add



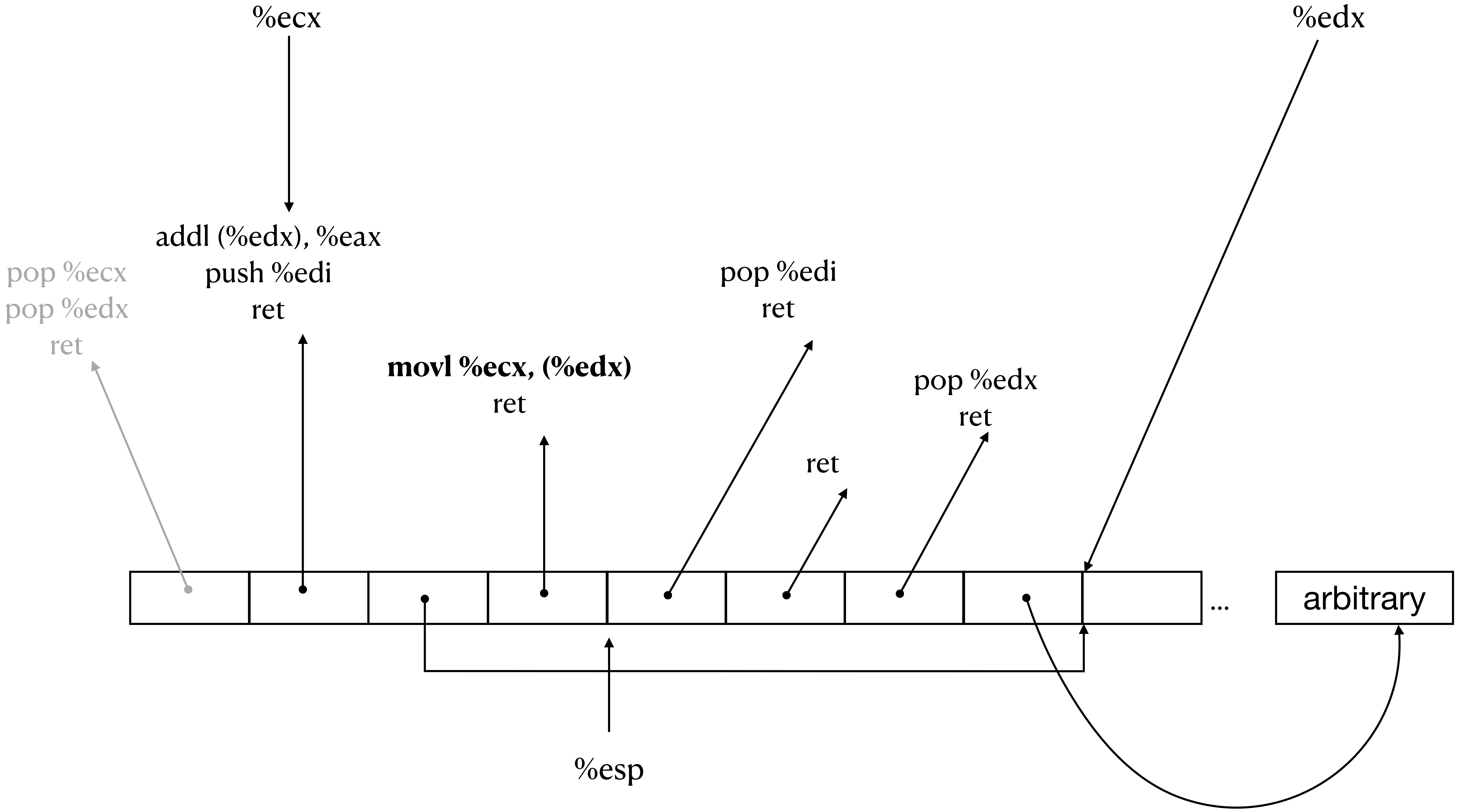
Example gadget: Add



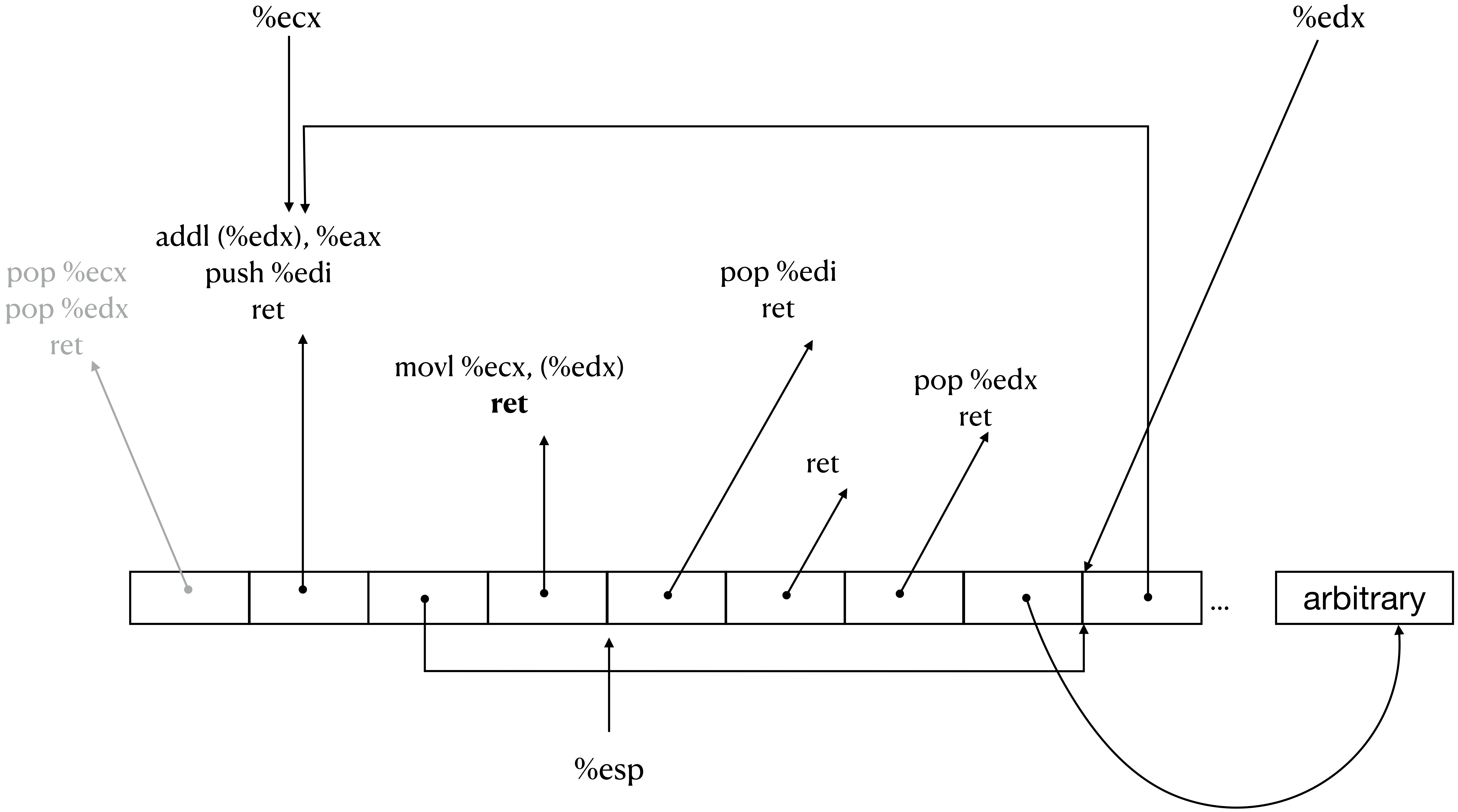
Example gadget: Add



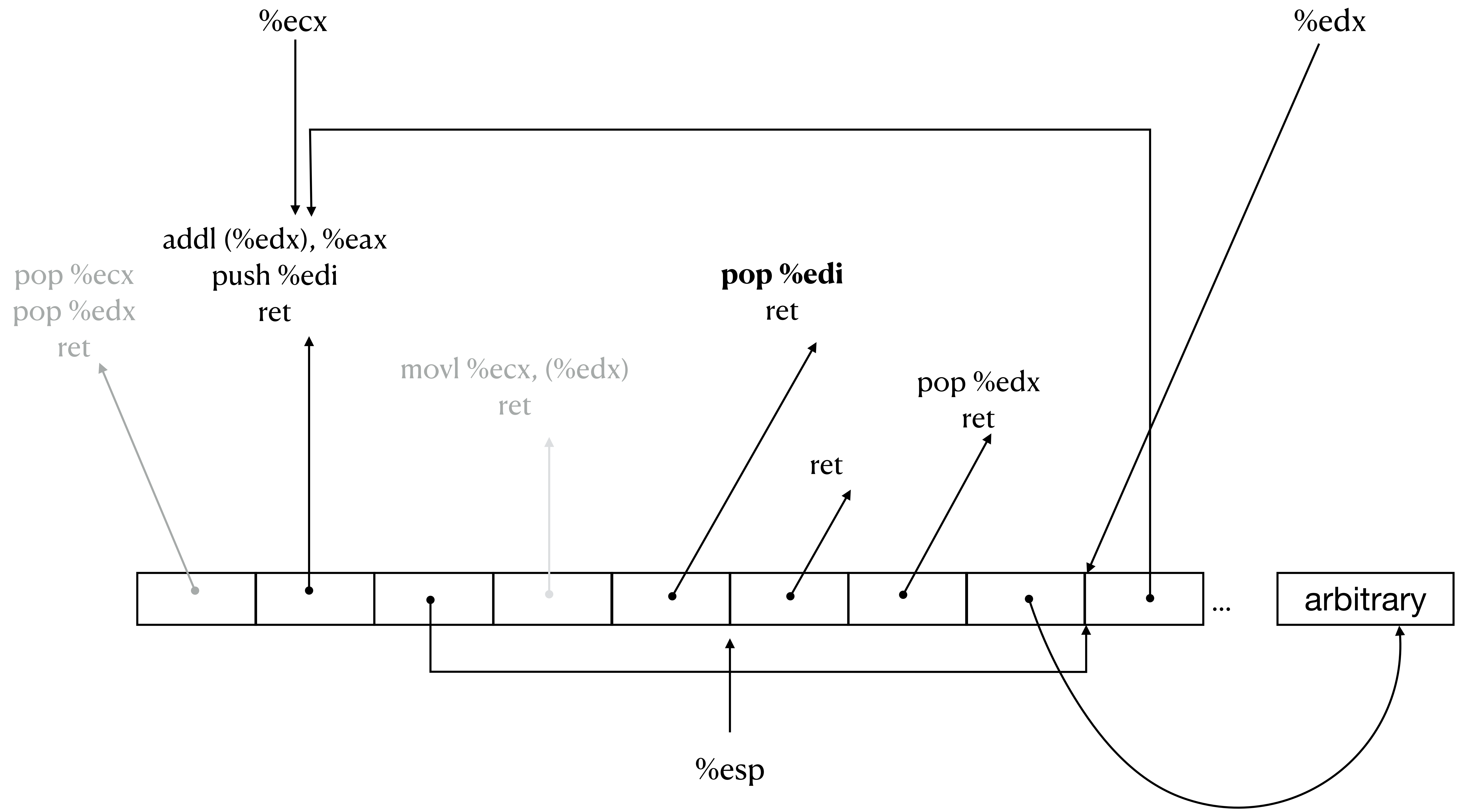
Example gadget: Add



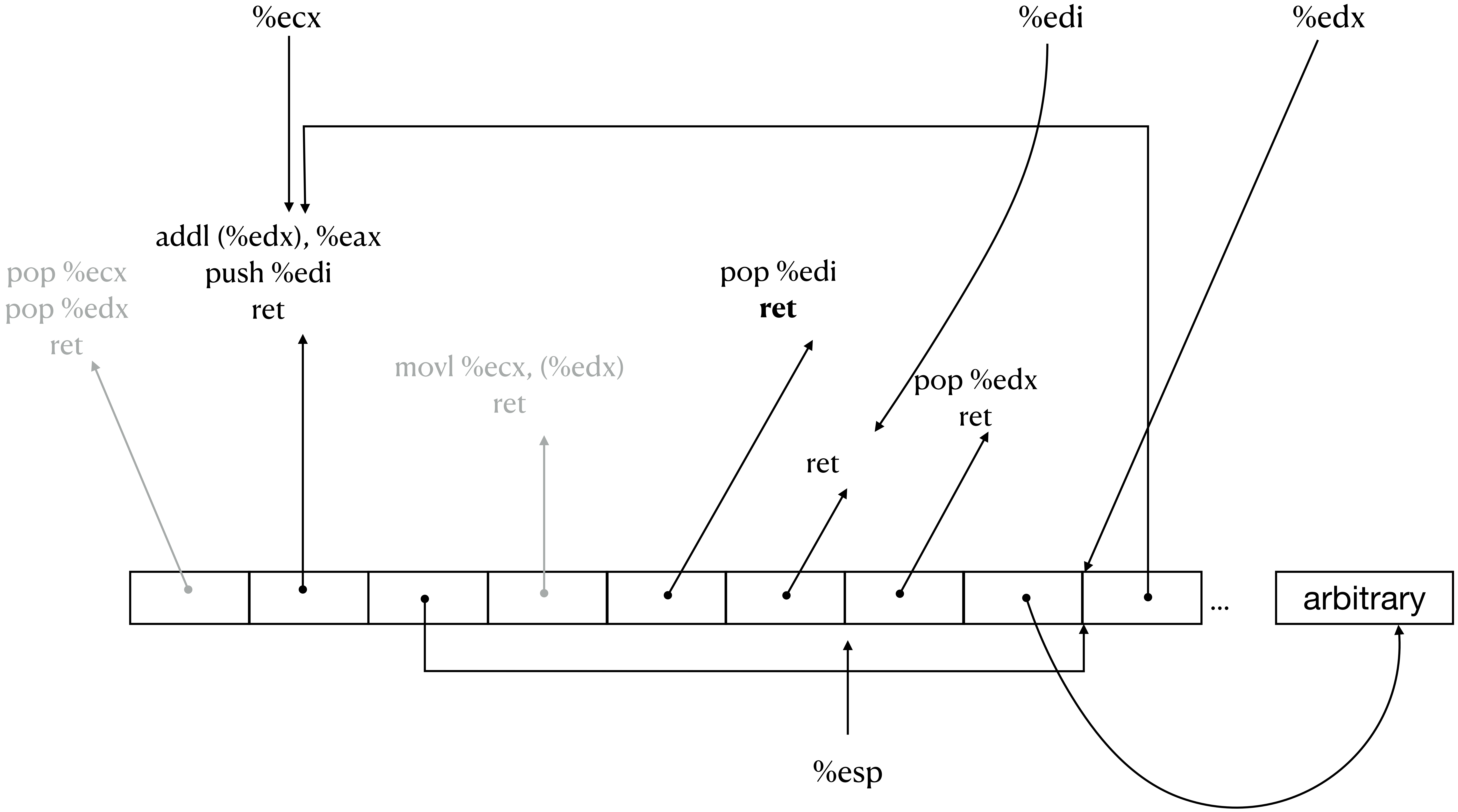
Example gadget: Add



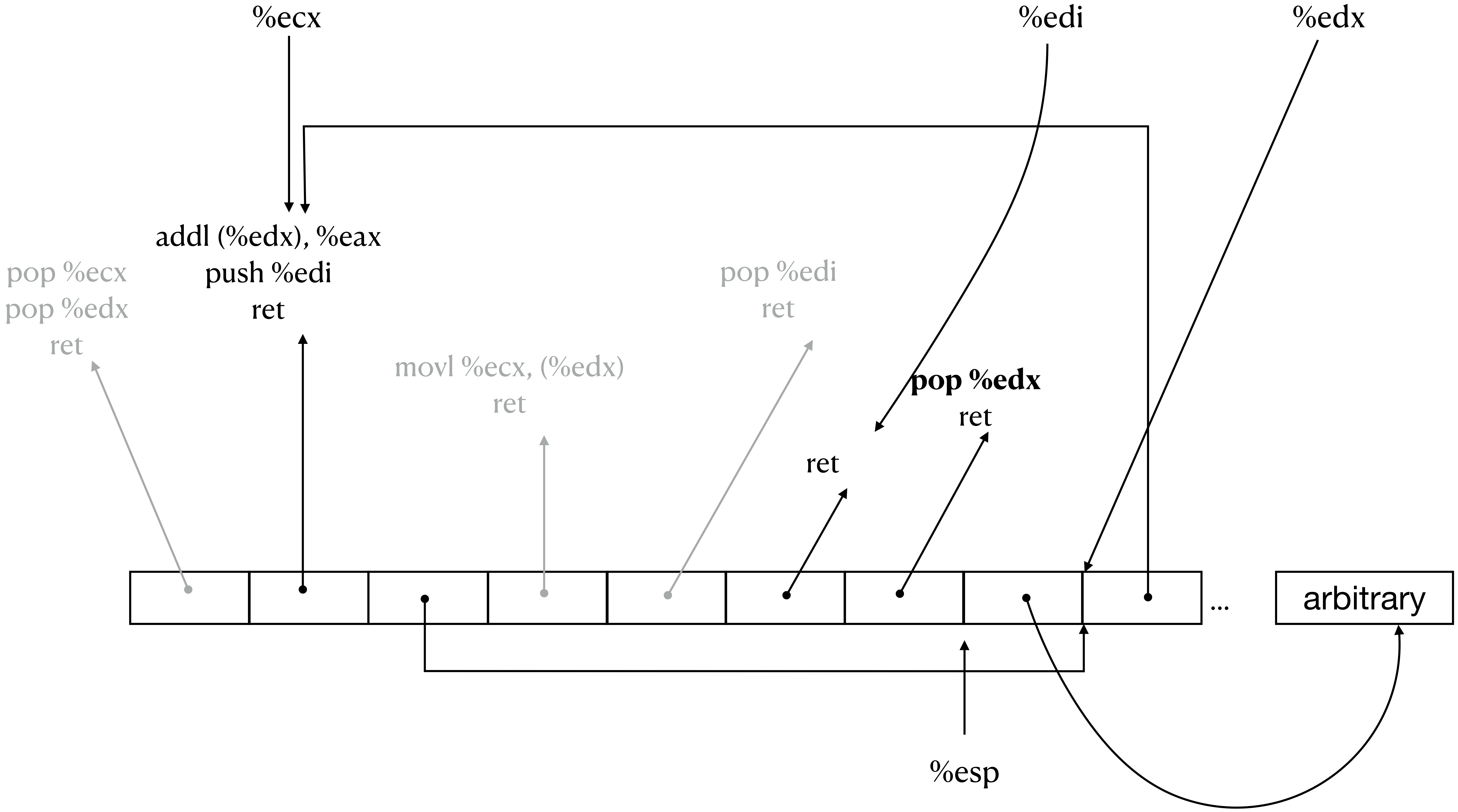
Example gadget: Add



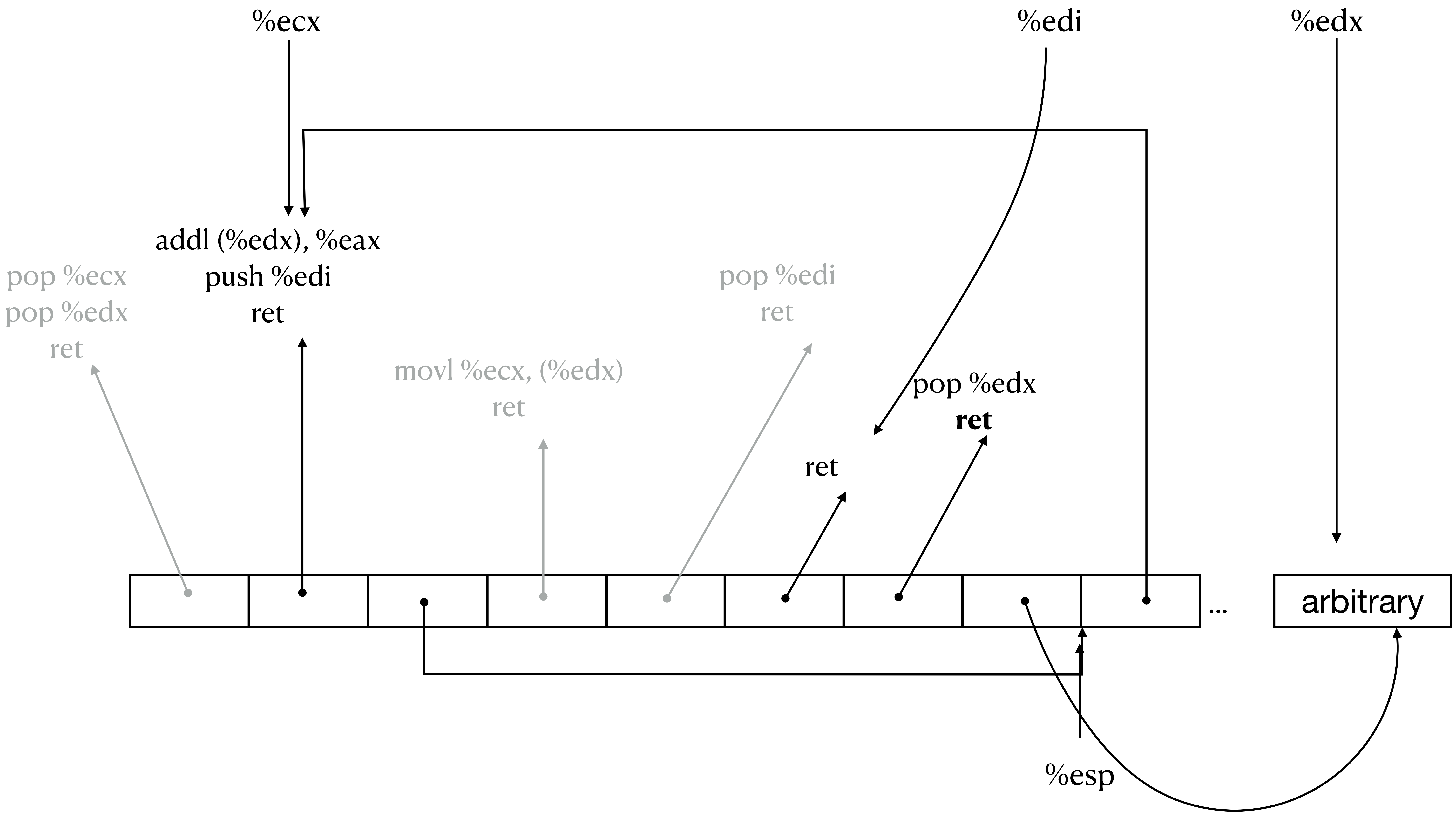
Example gadget: Add



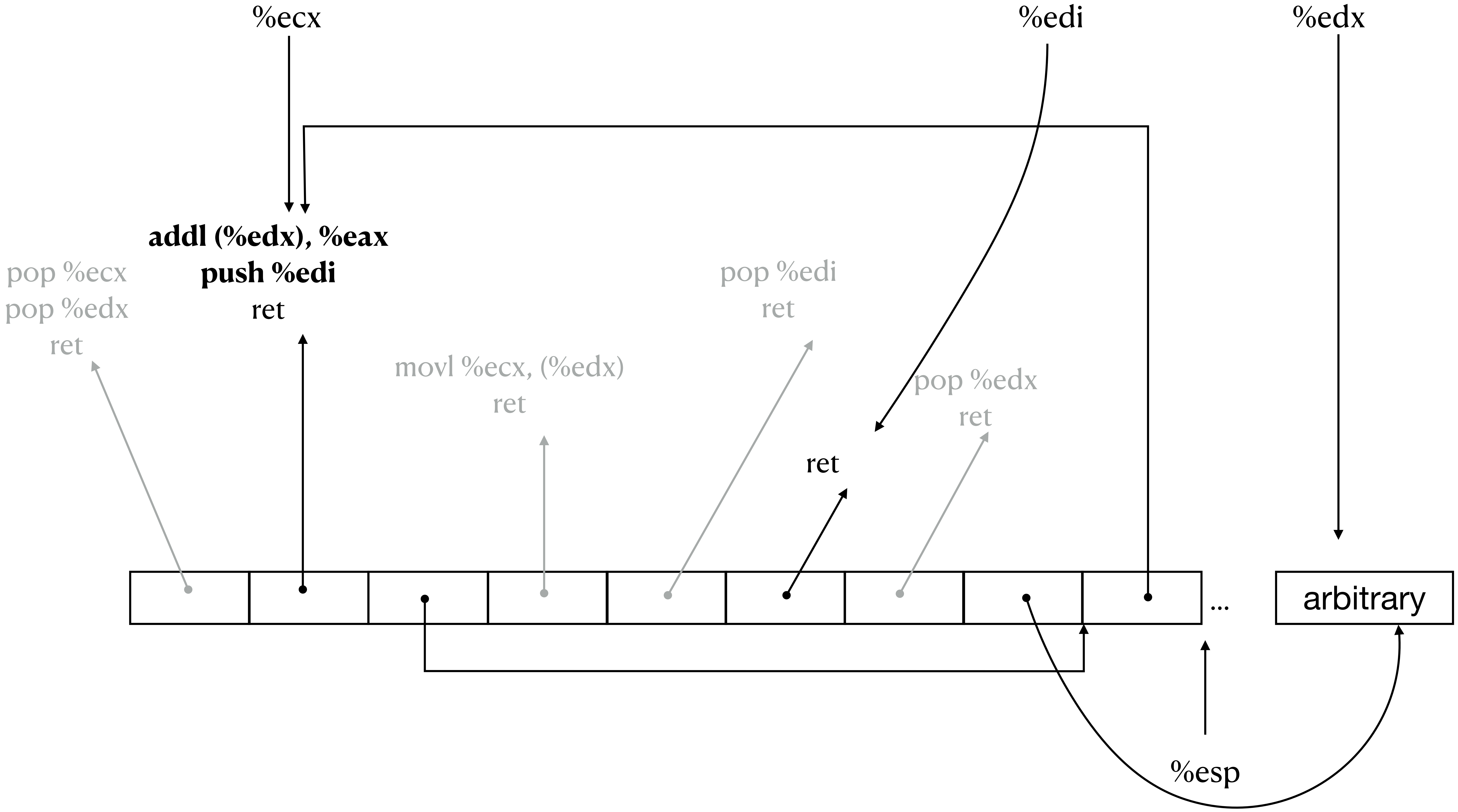
Example gadget: Add



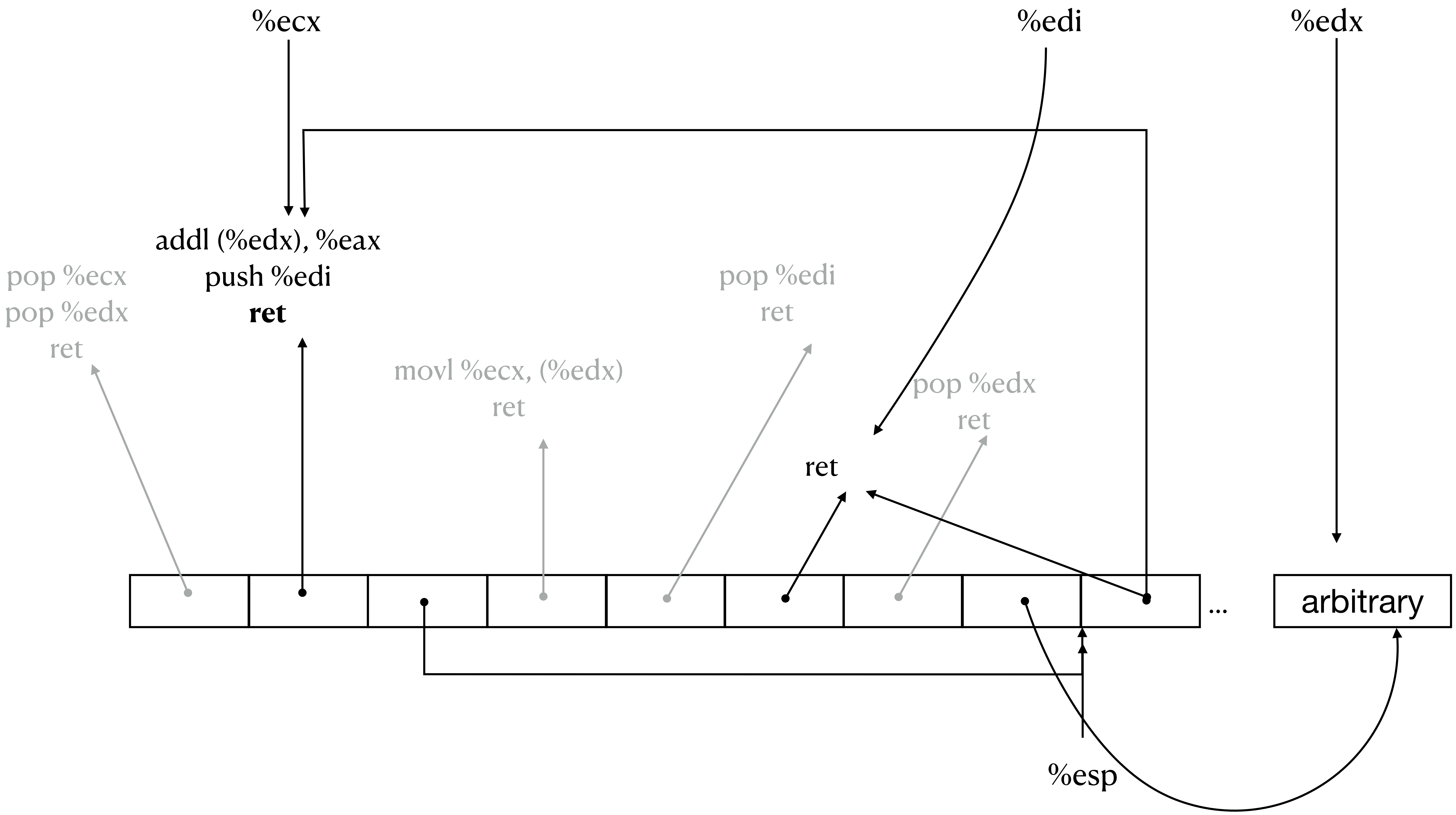
Example gadget: Add



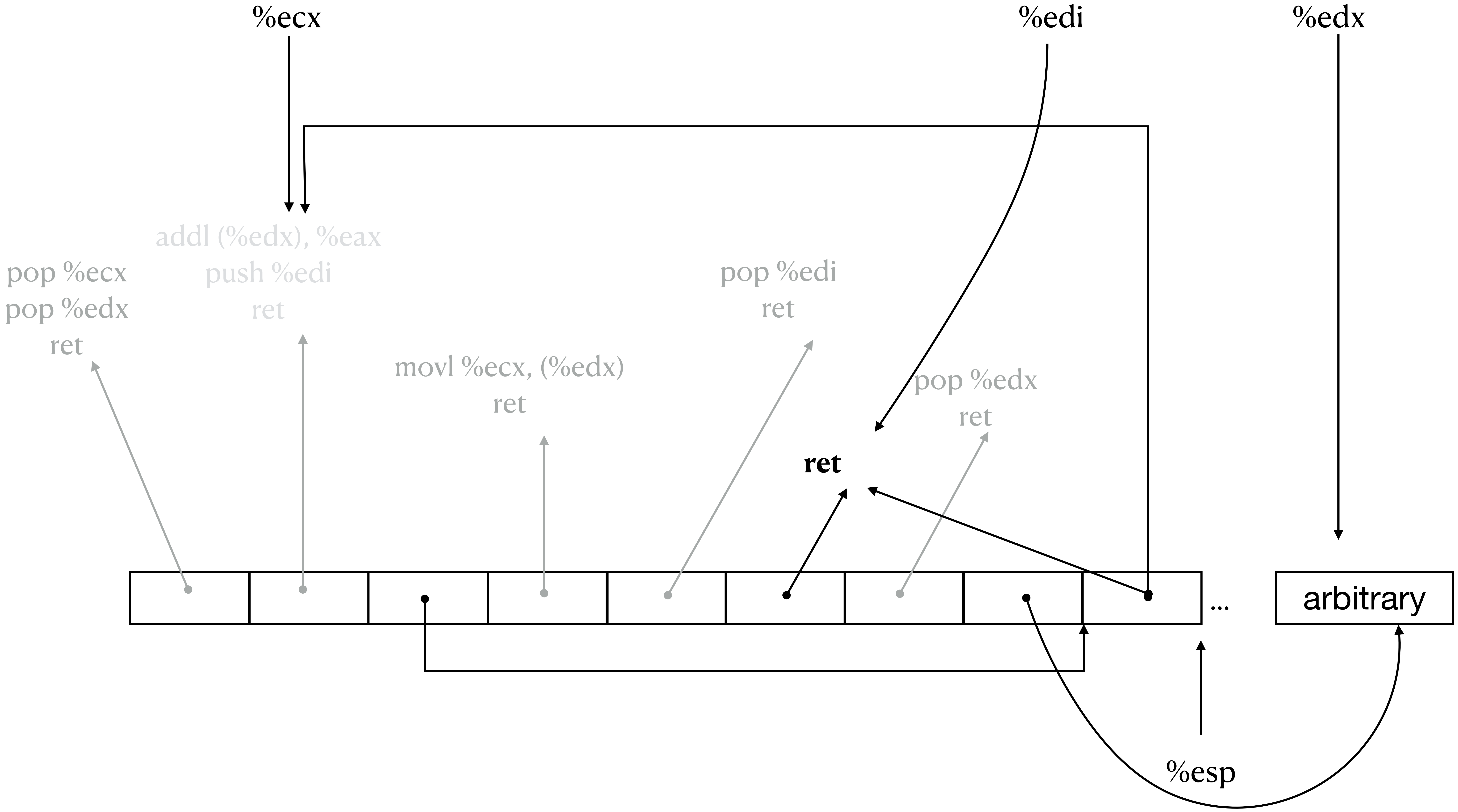
Example gadget: Add



Example gadget: Add



Example gadget: Add

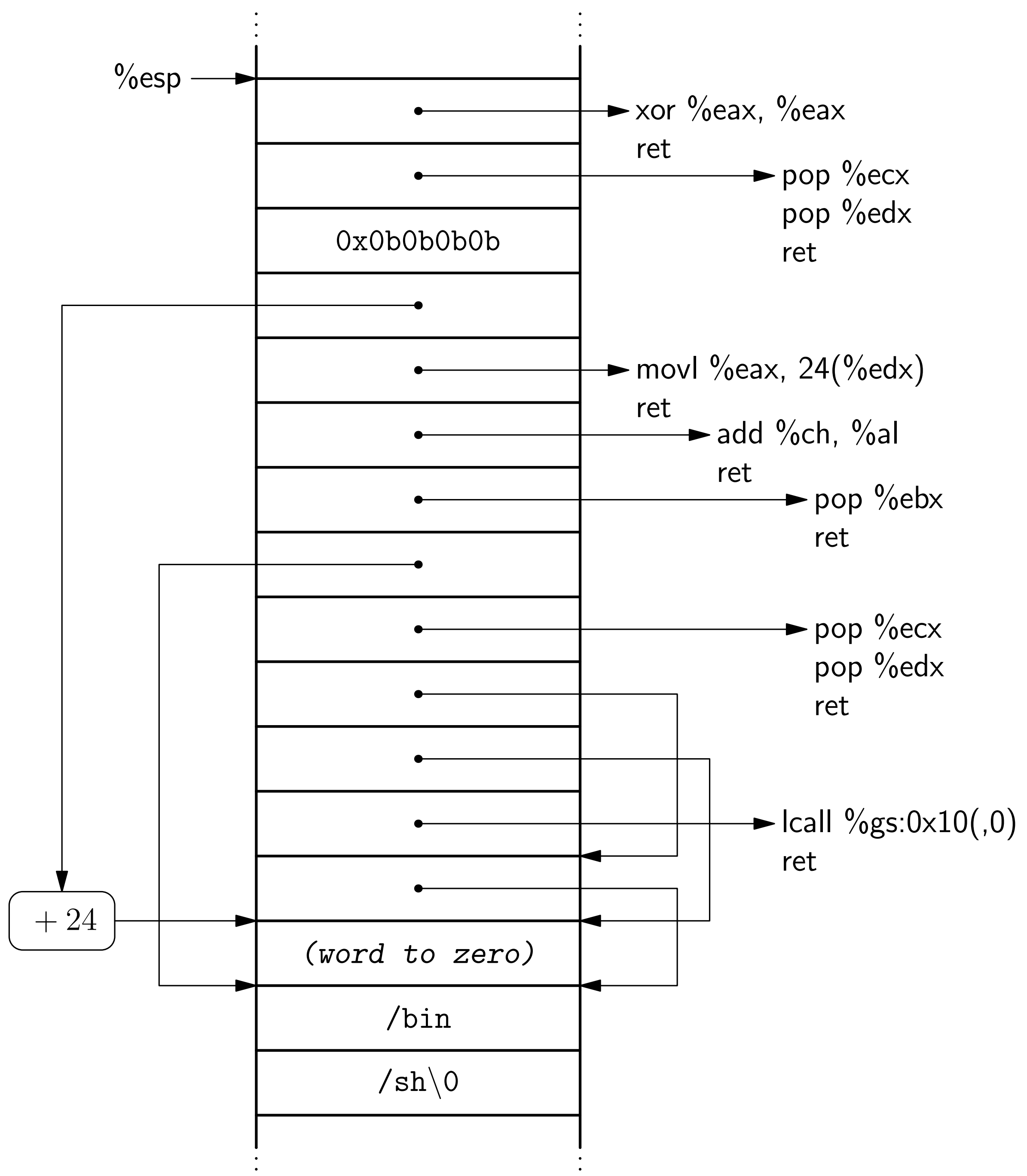


Requirements on gadgets

- Conditions that guarantee correct execution of an ROP program
  - Precondition:
    - %esp points to the first word in the gadget and the processor executes a ret instruction
  - Postcondition:
    - When the ret instruction in the last instruction sequence of the gadget is executed, %esp points to the next gadget to be executed

Building on top of this

- Other building blocks:
  - More complicated gadgets for control flow
  - Also: system calls, and "regular" function calls
- Example: shellcode gadget on the right
- Claim: the gadget collection is Turing-complete
- Exploit framework:
  - a compiler from a high-level language to gadgets



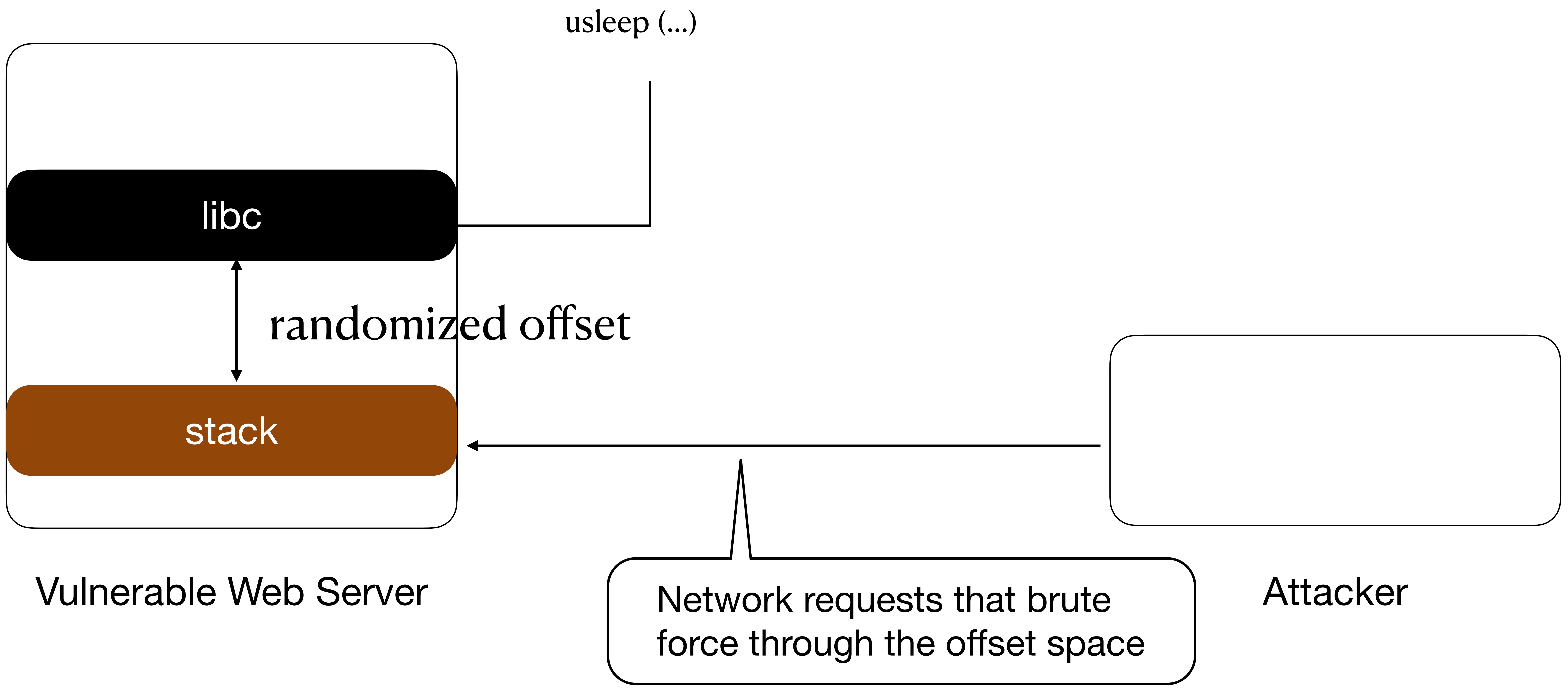
ROP: the takeaways

- Malicious computation ≠ Malicious code
- W⊕X memory protection by itself is insufficient
  - Need something more principled

Address-Space Layout Randomization

- Idea: Randomize the layout of the address space
  - The offsets will differ at each invocation
  - Forces the attacker to brute-force through the offsets
- Limitation:
  - On 32 bit machines there is not much entropy for this to be a realistic defense
  - Forks preserve the layout
  - Coarse-grained
    - Difficult to randomize offsets within functions

Attack on a 32 bit ASLR



Invalid offsets terminate connection immediately

Valid offset terminates connection after sleeping

The offset is preserved across forks

ASLR Conclusion

- Need 64 bit architectures to make randomization effective
- Need something more principled

Control Flow Integrity

Direct vs indirect control transfer

Direct	**Indirect**
Direct jump – jumping to a statically determined constant. *Examples*: if-then-else, loops, most local (w.r.t. a function body) control flow	Jump to a dynamically computed target: *Examples*: switch statement implemented via a dispatch table, Procedure Linkage Tables (PLT), etc
Direct call – calling to a statically determined target, e.g., static function call	Indirect call – call to a computed, i.e., dynamically determined target. *Examples:* function pointers in C, various implementations of dynamic dispatch
	Function return (OBS: regardless of the whether the call is direct or indirect)
	Complex control-flow due to exception handling (stack unwinding)
	Complex control-flow to support dynamic linking or separate compilation

in OO-languages, dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.

Indirect jumps as the source of all troubles

- It is the indirect jumps that make it possible to hijack the program control flow

CFI: Control-Flow Integrity (2005)

- Observation: there is gap between a machine-level jump and the original source-level target
- An indirect jump at a machine level can land *anywhere,* including middle of an instruction (on x86)
- Even the lowest of the systems programming languages rule out most of those targets
  - Definitely NOT middle of an instruction
  - Only a handful of possible targets: functions, switch statements, exception handlers
- This gap is the source of many problems: buffer overflows, ret-to-libc, ROP
- Goal of the CFI:
  - make the gap smaller, by reducing the set of machine-level jump targets to the intended subset

CFI is a 2-phase process

- Phase 1: Analysis – identify possible targets for all indirect jumps
  - This means we need to compute the Control-Flow Graph (CFG) of the program
  - Recall: control flow graph of a program is a graph where nodes correspond to basic blocks and edges correspond to jumps/branches.
  - Different ways to compute CFG:
    - Statically from the source of the program
    - Statically from the binary of the program
    - Dynamically, through profiling under “normal” input

CFG and its targets

Example C program with indirect calls and function pointers

```
void foo(int a){
    return;
}
void bar(int a){
    return;
}
void baz(void){
    int a = input();
    void (*fptr)(int);
    if(a){
      fptr = foo;
      fptr();
    } else {
      fptr = bar;
      fptr();
    }
}
```

`fptr` is a function pointer

Indirect call

Another indirect call

Q: what are the possible targets of `fptr()` call

Dynamically, it is either foo or bar

CFG imprecisions

Example C program with indirect calls and function pointers

```
void foo(int a){
    return;
}
void bar(int a){
    return;
}
void baz(void){
    int a = input();
    void (*fptr)(int);
    if(a){
      fptr = foo;
      fptr();
    } else {
      fptr = bar;
      fptr();
    }
}
```

Imprecise ≠ Useless. In this example, knowing that there are at most 2 possible targets is much better than not knowing anything at all (which would mean that the target of the `fptr()` call is anywhere in address space!)

Q: what are the *statically* possible targets of `fptr()` call?

Depends on how sensitive our analysis is

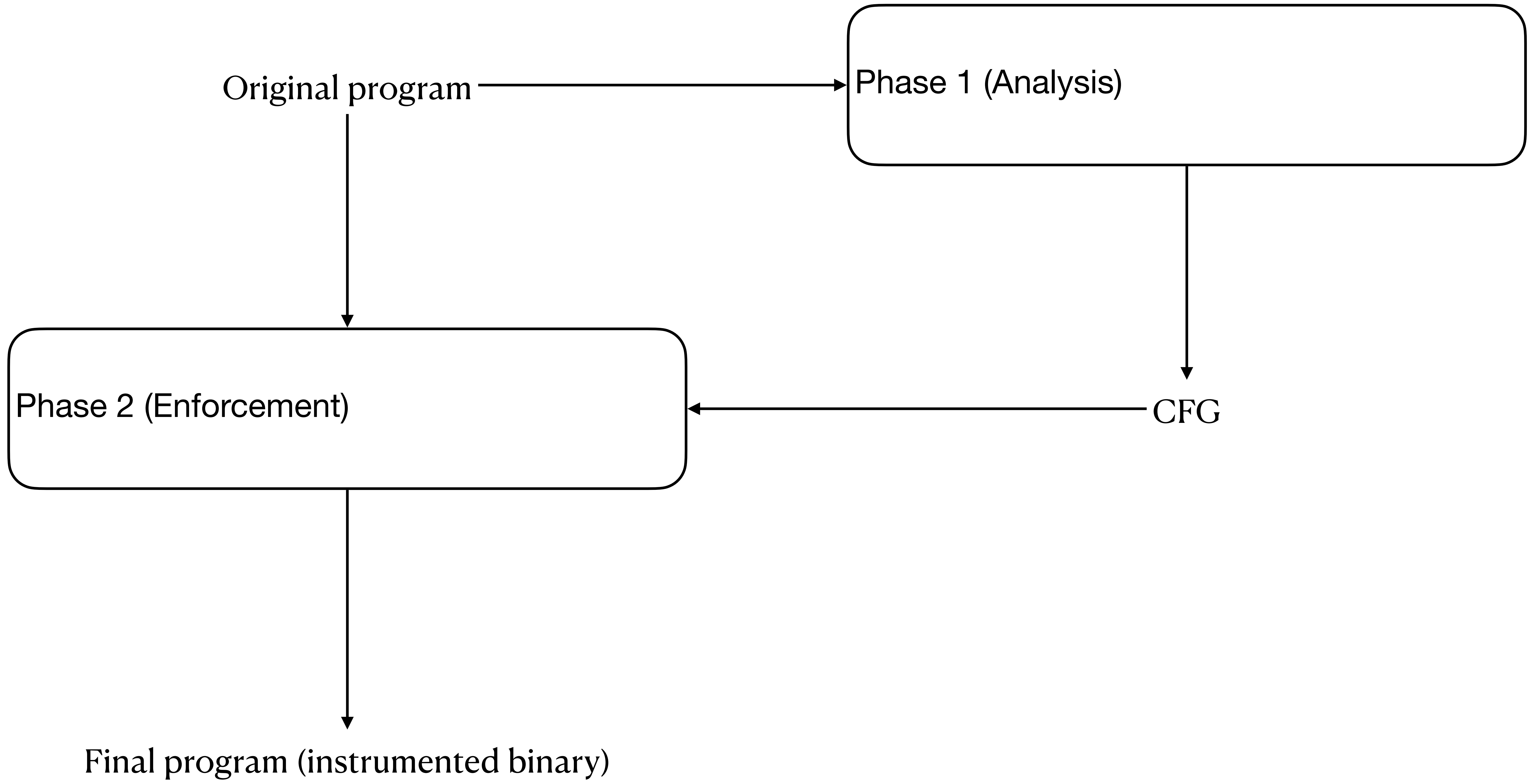
For example, a simple analysis will mark both `foo` and `bar` as possible targets for all indirect function calls of `fptr()`

Static analyses get only better, but we know that even the best ones will have imprecisions

CFI is a 2-stage process

- Phase 2: Enforcement – ensuring that all executed branches correspond to the edges in CFG
- At least three ways to implement
  - Static rewriting by the source-level compiler
  - Static binary rewriting
  - At runtime, through binary translation

CFI stages


Original program
Phase 1 (Analysis)
Phase 2 (Enforcement)
CFG
Final program (instrumented binary)

Attacker model

- The attacker knows the source/binary of the program, including the final (instrumented) binary
  - Important: no security through obscurity (cf. principles of secure design from [Seltzer & Schroeder'75])
- W⊕X memory protection
  - Attacker has write access to program memory, but cannot modify the program code
- A handful of tamper-resistant registers
- Possibility of creating bit patterns that do not appear anywhere in code memory (i.e., do not conflict with opcodes)

Ordinary CALL

DST register: 0x12002020

```
…
call DST
…
```

```
                 …
0x12002020:      // body of foo
                 …
```

source instruction

destination instruction

OBS: the call will succeed no matter where DST points to

Magic assembly

- Suppose we have three new assembly instructions
  - `label ID - has no effect`
  - `call ID, DST` – transfers control to address at register `DST` only if that code starts with `label ID`
  - `ret ID` – return to the call point only if that point starts with `label ID`

These instruction could be in principle added in hardware, but are practically implemented in software

CALL with CFI

DST register: 0x12002020

```
…
call 12345678 DST
…
```

source instruction

```
             …
0x12002020:  label 12345678
             // body of foo
             …
```

destination instruction

The call succeeds because the labels match

The idea of instrumentation: modify each source instruction and each possible destination instruction by adding IDs that correspond to potential targets from CFG

CALL with CFI

DST register: 0x12002028

```
…
call 12345678 DST
…
```

source instruction

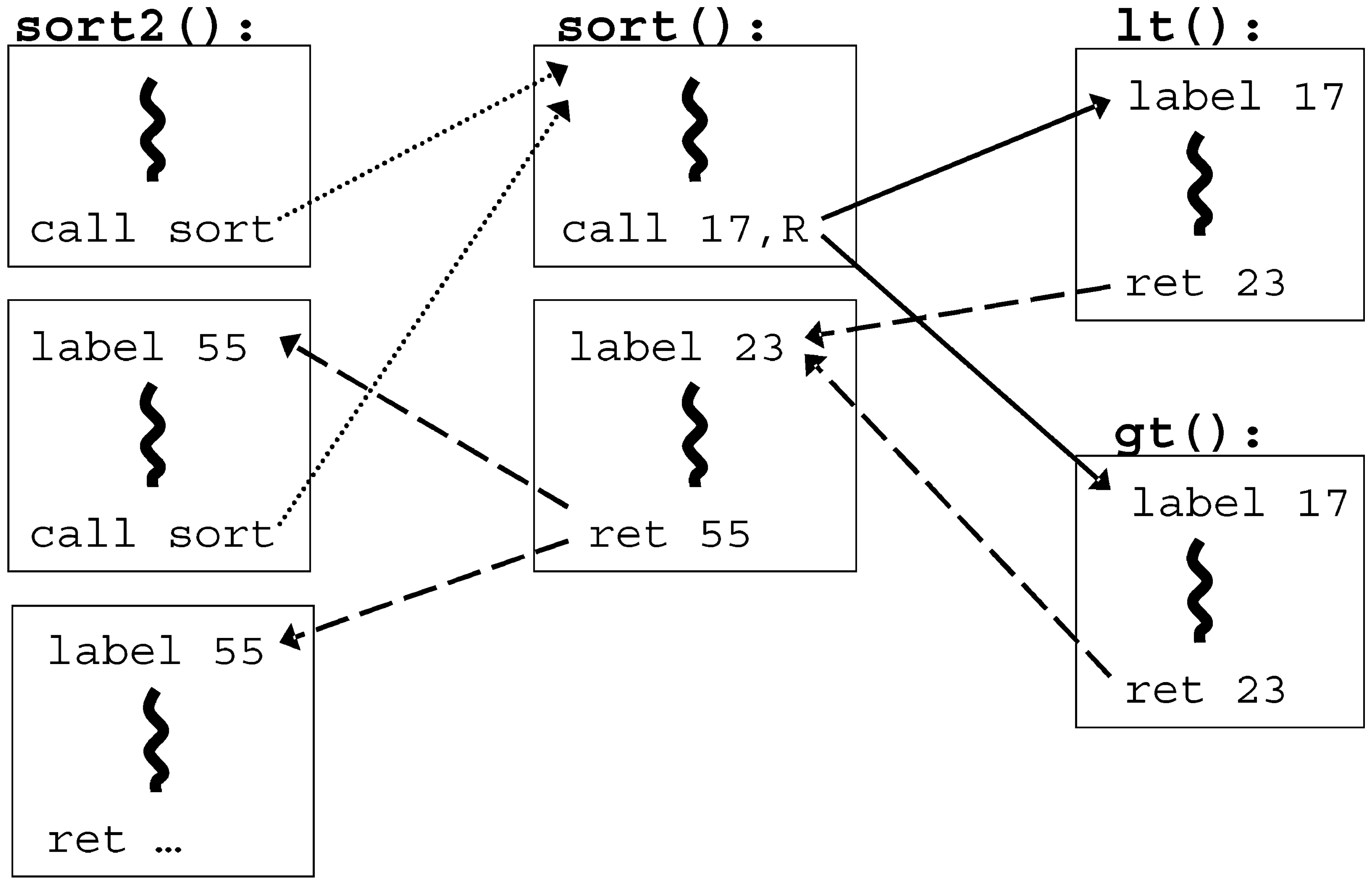
```
0x12002028:  …
             // no label here
             // some other code
             …
```

destination instruction

A call to a destination where there is no label or the label is wrong will fail

Example: a C program and CFG

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



dotted arrows - direct calls
edges from source - straight arrows
dashed arrows - return edges

CFI soft… call site

The ID bit pattern is embedded within the ID-check cmp opcode bytes. As a result, an attacker that can somehow affect the value of the ecx register might be able to cause a jump to the jne instruction instead of the intended destination.

Bytes (opcodes)

```
FF E1                jmp  ecx                ; a computed jump instruction
```

can be instrumented as (a):

Avoids the above problem with ecx register being manipulated by the attacker

```
   …p  [ecx], 12345678h  ; compare data at destination
   …e  error_label       ; if not ID value, then fail
   …a  ecx, [ecx+4]      ; skip ID data at destination
   …p  ecx               ; jump to destination code
```

CFI software instrumentation – callee site

Bytes (opcodes)	x86 assembly code	Comment
`8B 44 24 04`	`mov  eax, [esp+4]`	`; first instruction`
`...`		`; of destination code`

can be instrumented as (a):

`78 56 34 12`	`DD 12345678h`	`; label ID, as data`
`8B 44 24 04`	`mov  eax, [esp+4]`	`; destination instruction`

Precision issue

- The more precise the CFG the better we are
- Precision could be added by code duplication

Overhead

Binary size increase: average 8%

Runtime overhead: average 16%

2007

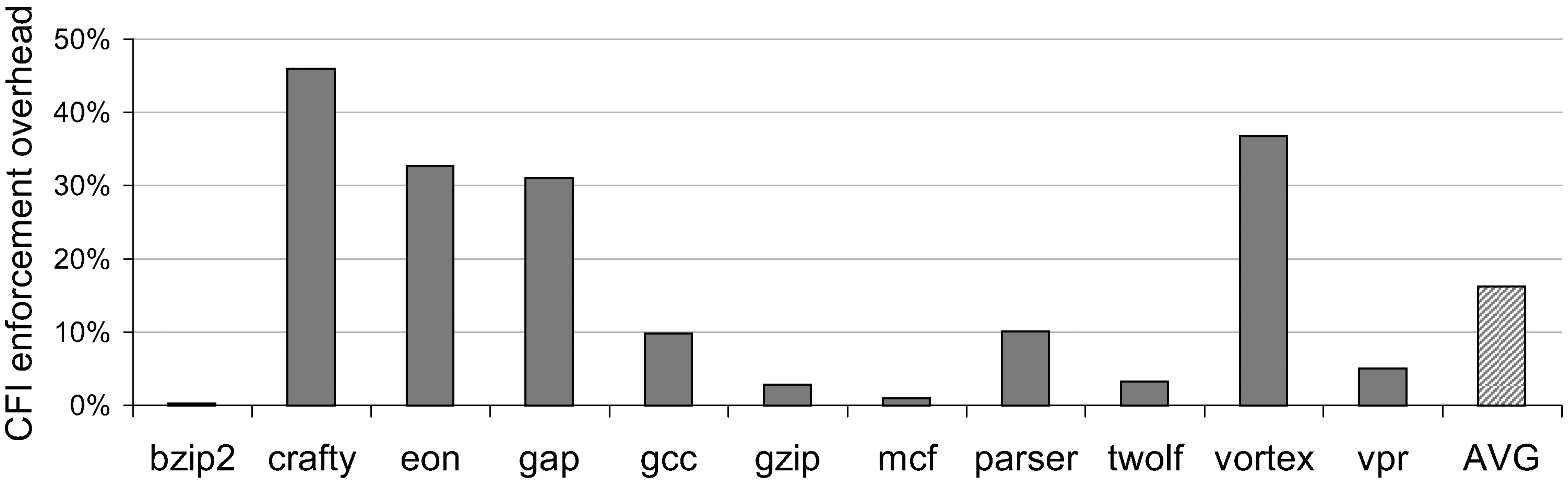


Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

2013

Seekers et al.

Basic CFI: Average: 16; Max: 45

CFI w/ shadow stack : Average: 21; Max: 56

Security Experiments: GDI+JPEG flaw in Windows (2004)

Example vulnerable code

```
int median( int* data, int len, void* cmp )
{
   // must have 0 < len <= MAX_LEN
   int tmp[MAX_LEN];
   memcpy( tmp, data, len*sizeof(int) );
   qsort( tmp, len, sizeof(int), cmp );
   return tmp[len/2];
}
```

qsort_with_cfi:

```
...
push   ebx
mov    eax, esi
call   shortsort
prefetchnta [AABBCCDDh]
add    esp, 0Ch
...
push   edi
push   ebx
mov    eax, [esp+comp_fp]
cmp    [eax+4], 12345678h  ; CFI check
jne    error label         ; prevents
call   eax                 ; going to X
prefetchnta [AABBCCDDh]
add    esp, 8
test   eax, eax
jle    label_lessthan
...
```

regular_qsort:

```
   ...
   push   ebx
   mov    eax, esi
   call   shortsort
   add    esp, 0Ch
   ...
   push   edi                ; an attack is
   push   ebx                ; possible by
   call   [esp+comp_fp]      ; going to X
   add    esp, 8
   test   eax, eax
   jle    label_lessthan
   ...
```

regular_library_function:

```
   mov    edi,edi
   push   ebx
   mov    ebx,esp
   push   ecx
   ...
   pop    ebp
X: mov    esp,ebx
   pop    ebx
   ret
```

Formal study

$Instr ::=$ instructions

$label\ w$	label (with embedded constant)
$add\ r_d, r_s, r_t$	add registers
$addi\ r_d, r_s, w$	add register and word
$movi\ r_d, w$	move word into register
$bgt\ r_s, r_t, w$	branch-greater-than
$jd\ w$	jump
$jmp\ r_s$	computed jump
$ld\ r_d, r_s(w)$	load
$st\ r_d(w), r_s$	store
$illegal$	illegal

Fig. 12. Instructions.

S is a state – (M, R, pc)

$$\frac{S \rightarrow_{\mathrm{n}} S'}{S \rightarrow S'} \qquad \frac{S \rightarrow_{\mathrm{a}} S'}{S \rightarrow S'}$$

$$(M_c|M_d, R, pc) \ \rightarrow_{\mathrm{n}} \ (M_c|M_d, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc + 1),$$

example: normal step transition for **add r_d r_s r_t**

$$(M_c|M_d, R, pc) \ \rightarrow_{\mathrm{a}} \ (M_c|M_d{}', R, pc).$$

attacker transition

Assumptions: the code is well-instrumented for CFI

- Direct jump targets
  - all targets must be valid according to CFG
- IDs
  - There must be an ID right after every entry point
  - No IDs by accident
- ID checks
  - There must be a validation check before every control transfer
  - Each check must respect CFG

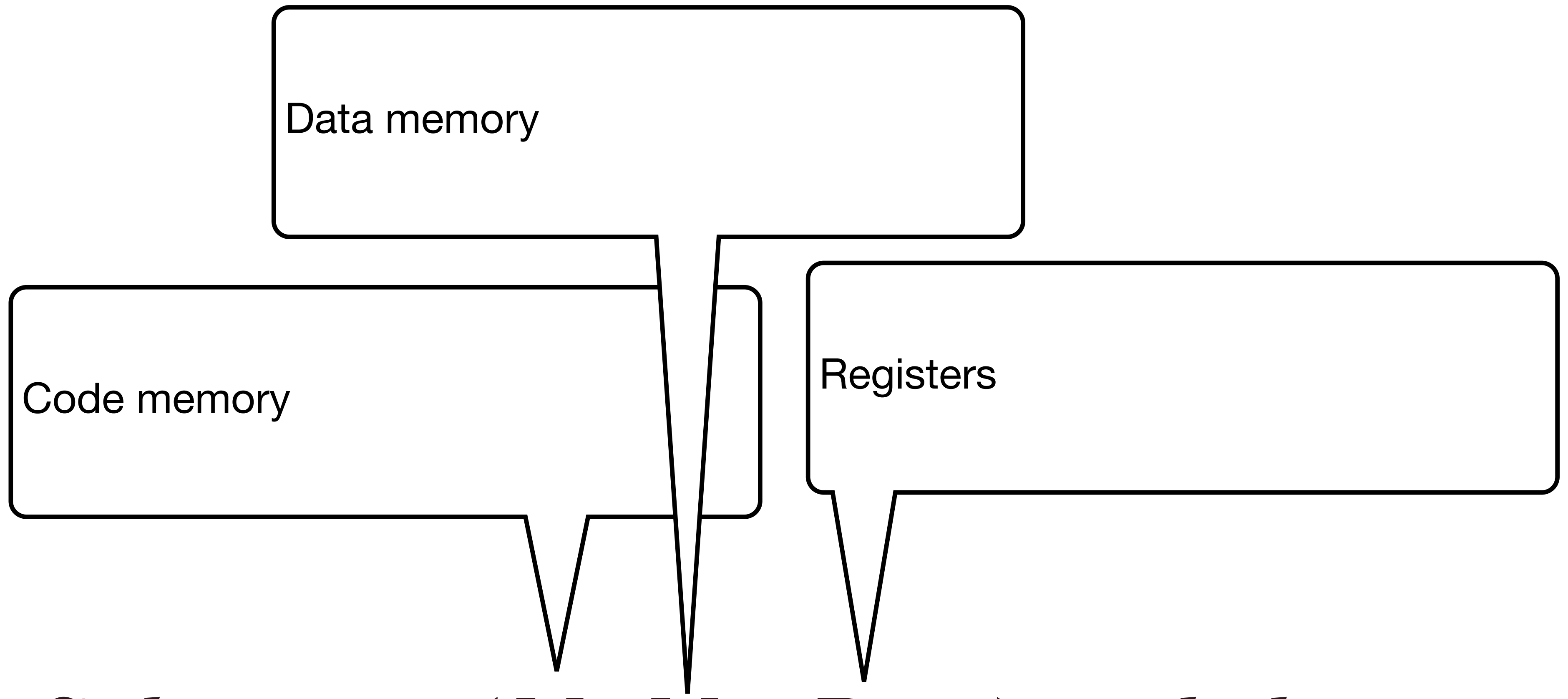
Example validation check

If $w_0 \in \mathrm{dom}(M_c)$ holds a *jmp* instruction, then this instruction is *jmp* r_0 and it is preceded by a specific sequence of instructions, as follows:

$$
\begin{array}{l}
addi\ r_0, r_s, 0 \\
ld\ r_1, r_0(0) \\
movi\ r_2, IMM \\
bgt\ r_1, r_2, HALT \\
bgt\ r_2, r_1, HALT \\
jmp\ r_0,
\end{array}
$$

where r_s is some register, *HALT* is the address of the *illegal* instruction specified in Condition (1), and *IMM* is the word w, such that $Dc(w) =$ *label* $\mathrm{dst}(w_0)$. This code compares the dynamic target of a jump, which is initially in register r_s, to the *label* instruction that is expected to be the target statically. When the comparison succeeds, the jump proceeds. When it fails, the program halts.

Theorem



THEOREM 1. *Let* S_0 *be a state* $(M_c|M_d, R, pc)$*, such that* $pc = 0$ *and* $I(M_c, G)$*, where* G *is a CFG for* M_c*, and let* $S_1, \ldots, S_n$ *be states, such that* $S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n$*. Then, for all* $i \in 0..(n-1)$*, either* $S_i \rightarrow_{\mathrm{a}} S_{i+1}$ *and* $S_{i+1}.pc = S_i.pc$*, or* $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, G, S_i.pc)$*.*

Idea behind formal proof: induction on executions with an invariant – constrain values of the distinguished registers (0 - 2) within the instrumentation sequences