CS 5432:
Control Flow Defenses

Fred B. Schneider
Samuel B Eckert Professor of Computer Science

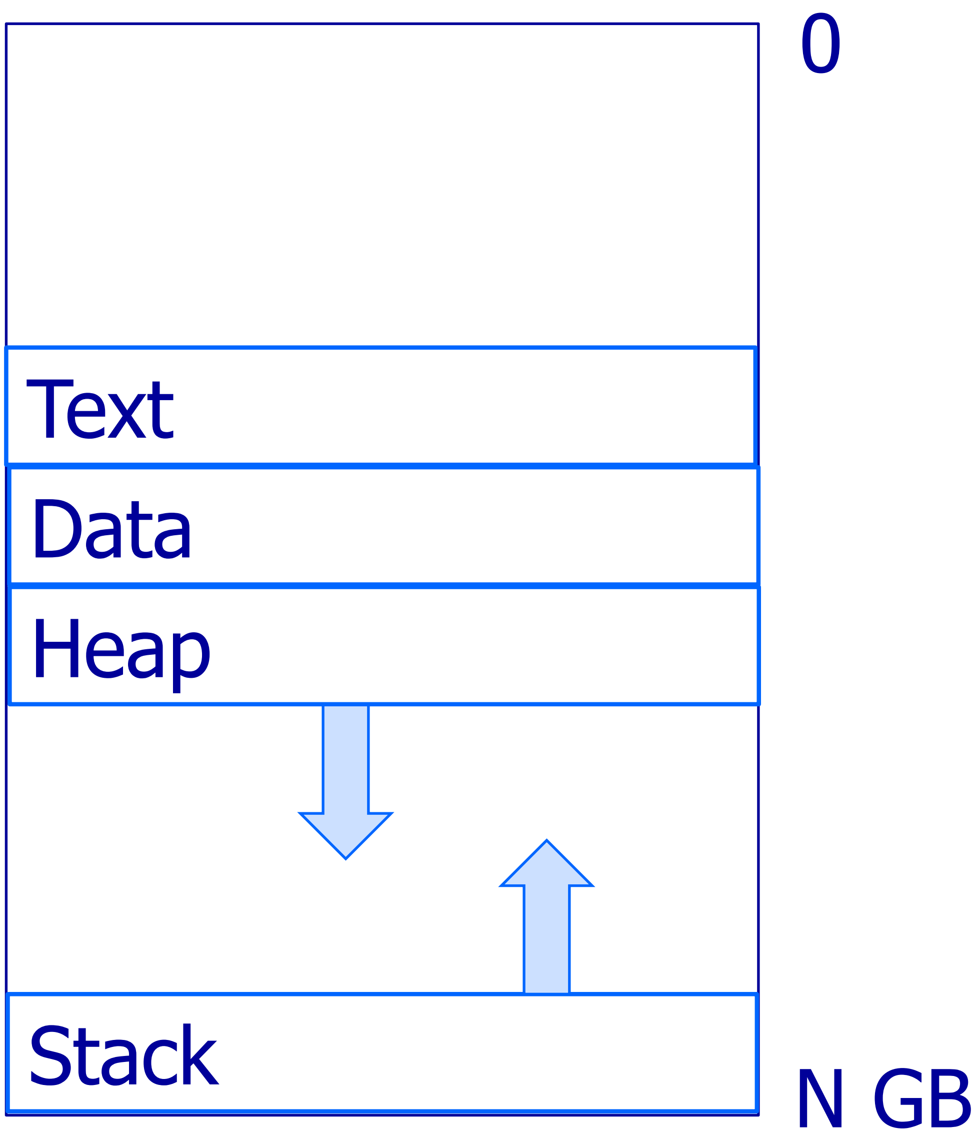
Department of Computer Science
Cornell University
Ithaca, New York 14853
U.S.A.

Cornell CIS
Computer Science

Attacks: High Level View

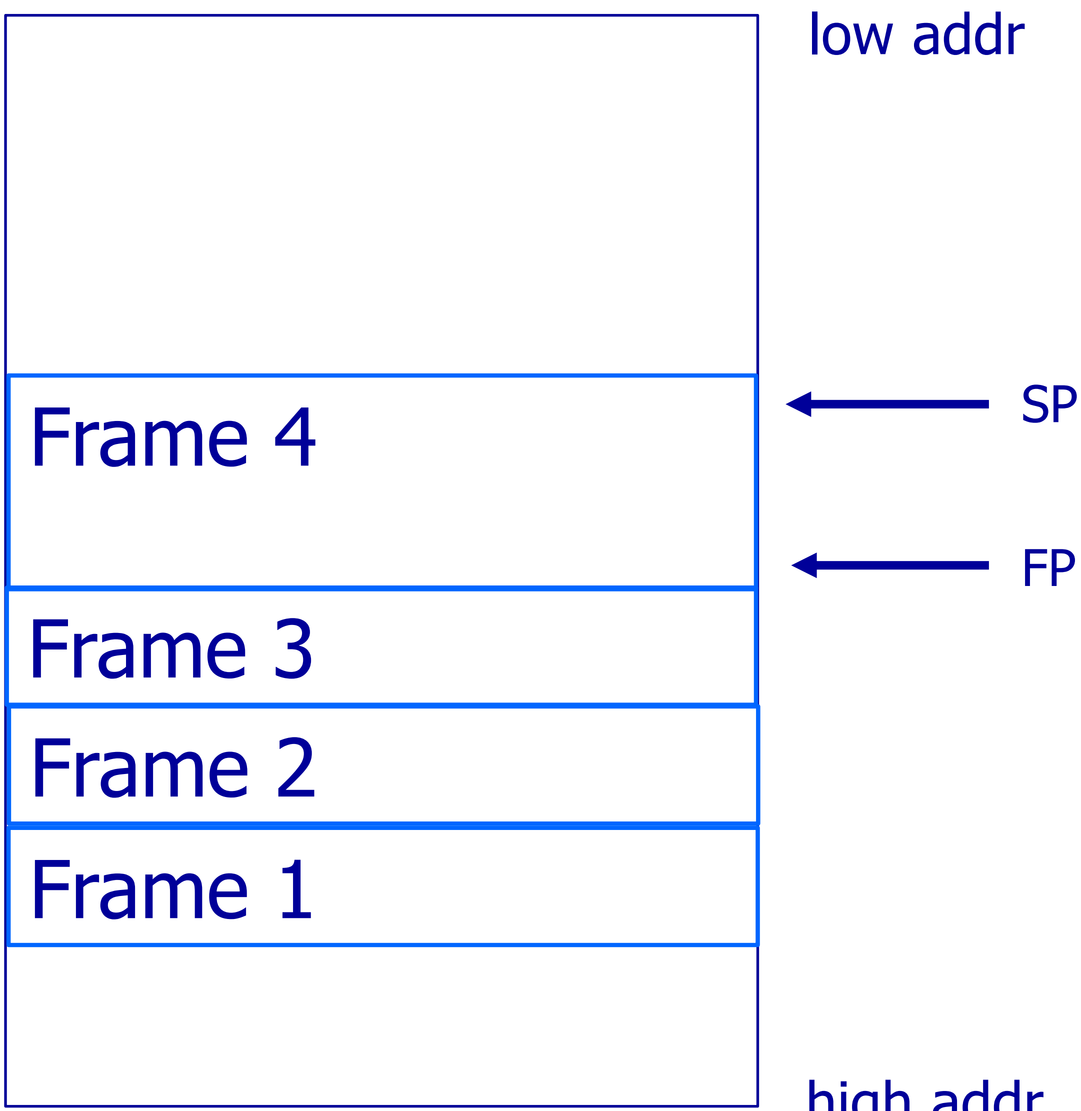
- Abuse existing functionality.
  - Code follows intended control flow.
- Inject code and execute that.
  - Code follows different control flow.

Memory Organization



Stack grows in direction of smaller addr in Intel, SPARC, MIPS, …

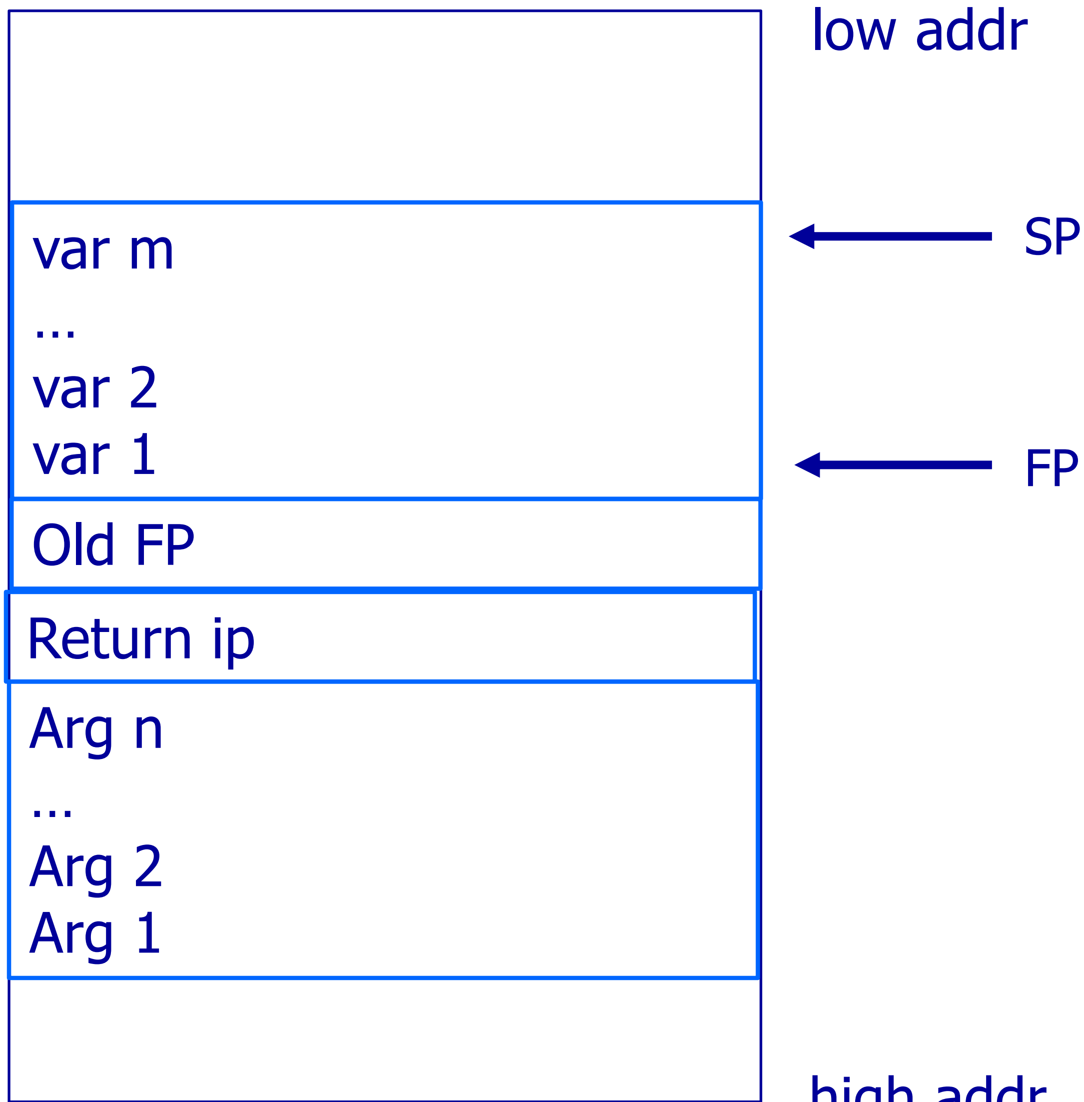
Runtime Stack: Frames



SP points to top data word in stack

FP points to start of frame.

Runtime Stack: Frame Layout

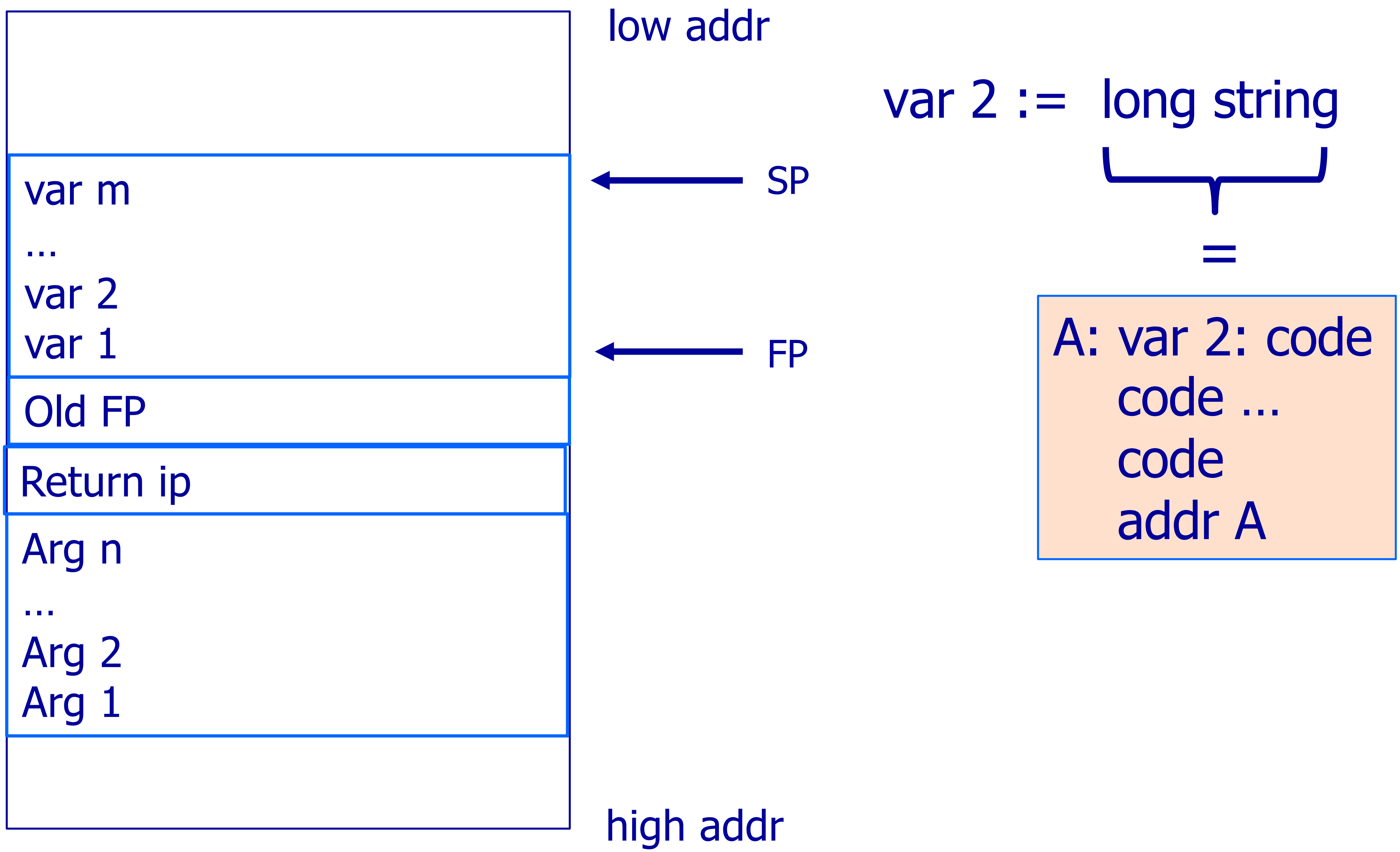



```
call f(x,y,z)
```

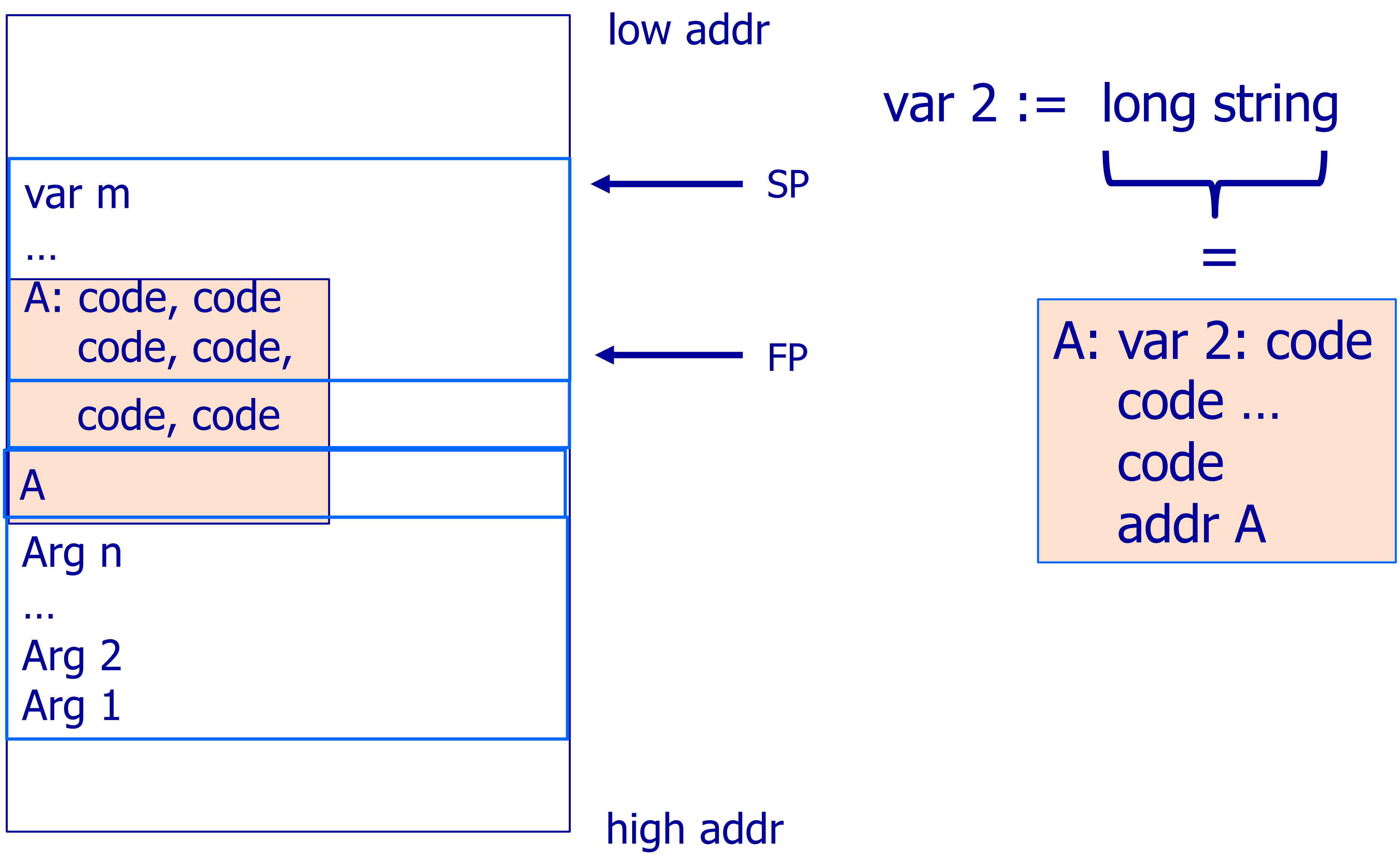
```
push x
push y
push z
call f
    push IP
    jmp f
```

```
f:  push FP
     FP := SP
     SP := SP – len(locals)
     …
     SP := SP – len(locals)
     pop FP
     jmp (*SP)
```

Buffer Overflow Attack



Buffer Overflow Attack



Defenses (?)

Protect return IP on stack

- Does not protect against:
  - Changes to other variables
  - Changes to function pointers

- Stackshield
- Stackguard
- Pointerguard
- Non executable stack (DEP or W+X)

Stackshield

Maintain shadow copy of stack in heap.

- Push return IP in function prolog
- Check return IP in function epilog

… assumes all library and applics are (re)compiled with this defense in place. Unreasonable assumption for apps.

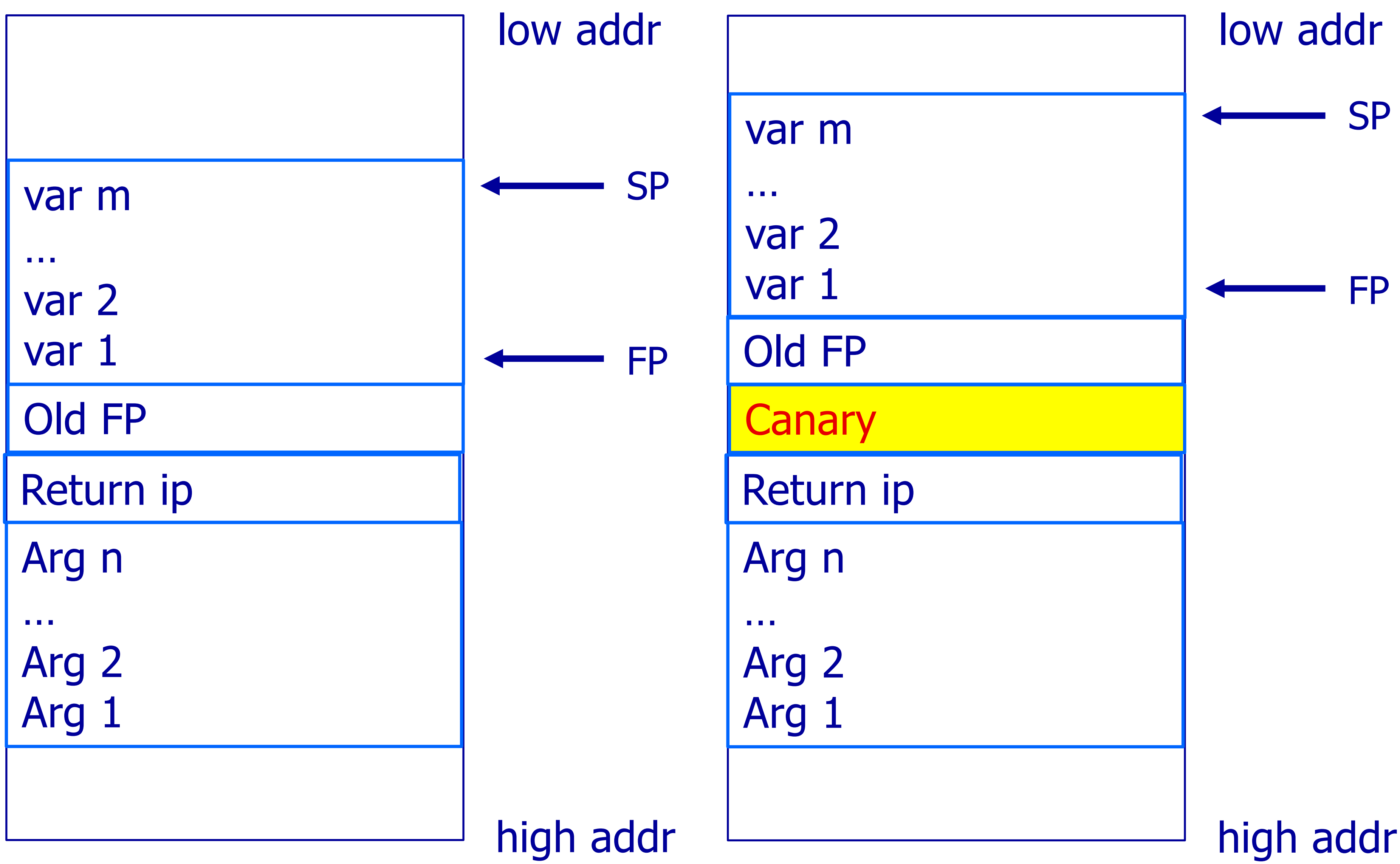
Stackguard [Cowan '98]

Compiler includes "canary" in the stackframe in order to protect return IP.

- Canary pushed onto stack by procedure prolog.
- Canary checked in procedure epilog

… Writing "up" from a variable will overwrite the canary, leading to detection at procedure exit.

Stackguard [Cowan '98]



Circumventing the Canary

Idea: Overwrite canary with a value that will be accepted by checking code in epilog.

- Easier if canary is public constant
- Harder if canary is not known to attacker.
  - … presumably canary value stored in system.

… this informs the design of canary.

Canary Implementations

- **Terminator canary**. Contains NULL (0x00), CR (0x0d), LF (0x0a), EOF (0xff).
  - *Either*: Attacker's copying will stop early, so overwrite will not reach and replace return IP address on stack.
  - *Or*: Copy operation will change contents of canary and, therefore, replace return IP address. But canary now has value that will fail test at epilog.
    - If multiple stack overruns possible: Attacker can then overwrite bogus canary (using multiple copy operations of different lengths) restoring a "terminator canary."

Canary Implementations

- **Random Canary**. Include value in DATA or TEXT:
  - Array RCan[0 .. 255] of random values
  - Stored in read/only page
  - Guarded by no-read pages
- Use as canary:

  RCan[(fn start addr) mod 255]

Canary Implementations

- **Random Function of IP**. Use as canary:

  return IP $\oplus$ random val

  At procedure epilog:

  – Check if Canary corresponds to planned return IP

  (Attacker could have copied pointer into return IP).

Defense: Prevent Data Execution

Defense: Prevent execution from writable memory.

- DEP (Data Execution Prevention) -also called-
- W^X aka W⊕X (writable or executable)

Implementations:

  - [older x86] Have separate segment for executable pages
  - [x86 64bit MMU] Use NX (AMD) or XD (Intel) bit in each page table entry.

Return-into-libc attacks

If execution of data is not possible…

Attack: Use code already present and executable.

- Return-into-libc attacks
  - Put onto stack as return IP: addr inside some libC function:
    - E.g., "call system(…).
  - May benefit from putting args on the stack, too.
  - May have IP point to a "call system" instruction <u>inside</u> of libc routine.

Note. Attack is restricted to invoking a single routine or a sequence of libc routines or their tails.

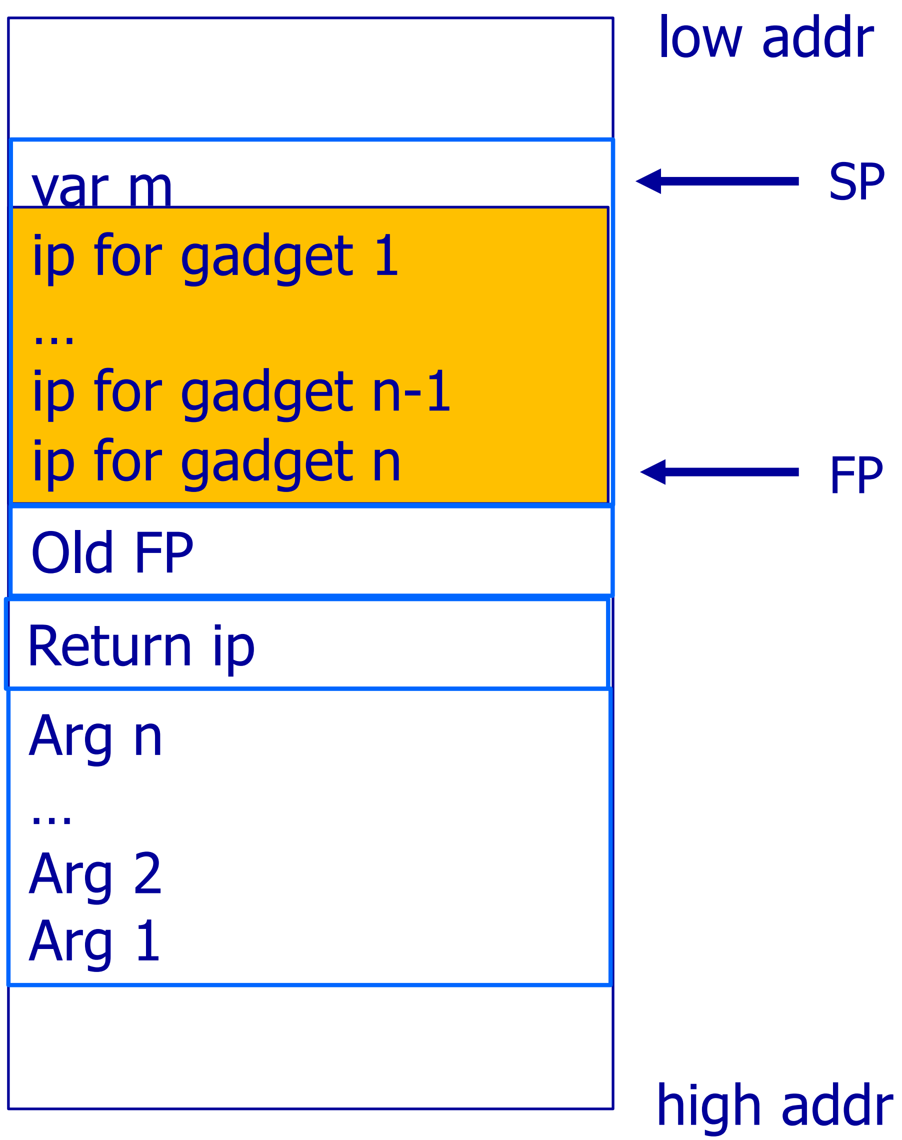
Defense: Return-into-libc attacks

- Make address of libc routines unpredictable.
  - Address Space Layout Randomization (ASLR)
    - Can be penetrated by brute force or certain invocations.
- Use ASCII armoring for address of libc routines.
  - Address of routine contains leading NULL byte (0x00), which prevents copying address onto stack.

Going beyond Return-into-libc attacks…

… use code but not functions.

Return-Oriented Programming



Gadget: Sequence of instructions that ends with **return** instruction (opcode: 0xc3).

Thesis: If instruction set is sufficiently dense then sys code includes Turing-complete set of gadgets.

Gadget Construction

- Start sequence at any instruction.
  - Do not include transfers of control.
- End sequence with a return (`ret`).
  - Fact: SP serves as the PC for sequencing

Fact: Every suffix of a gadget is a gadget.

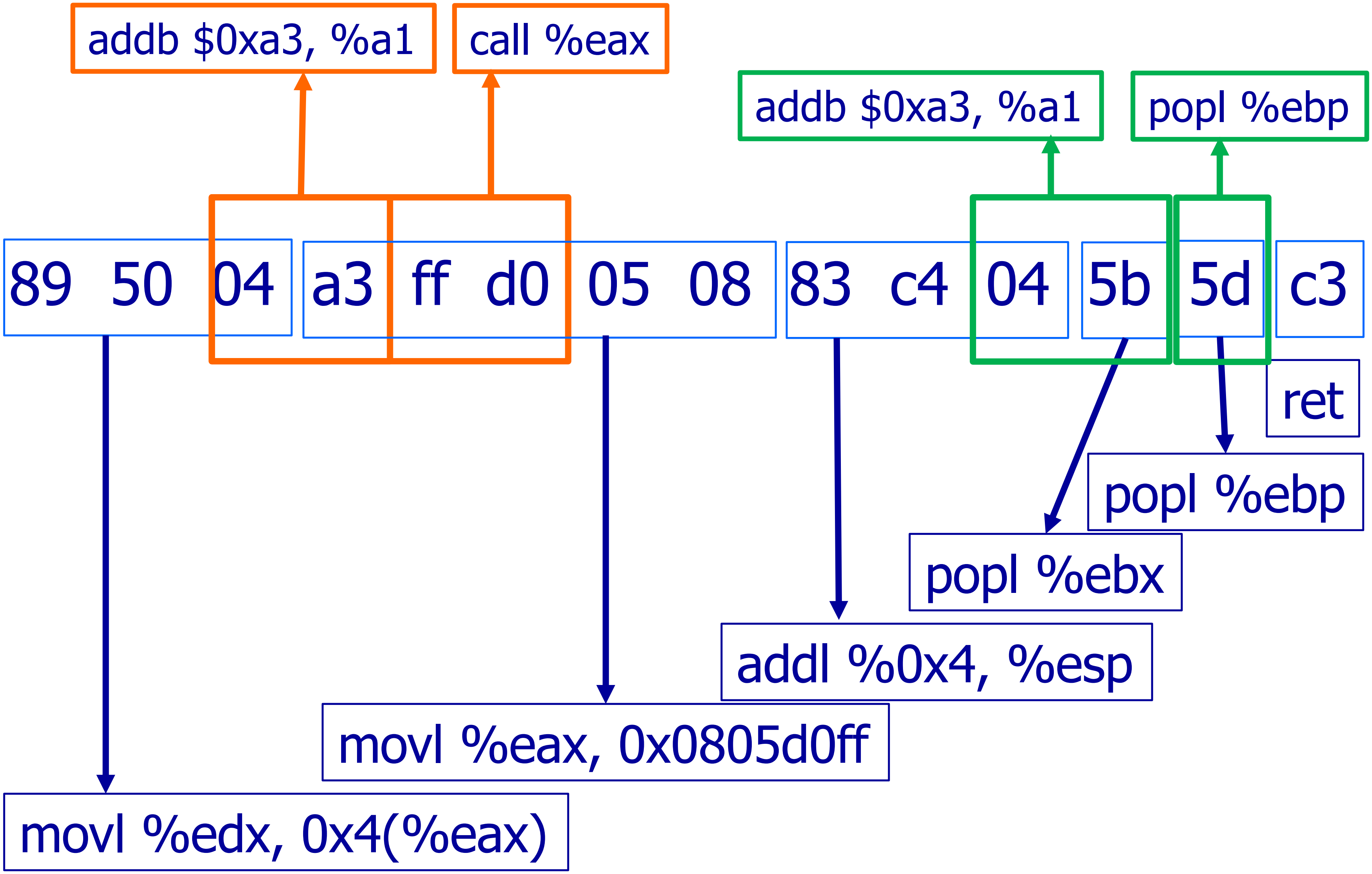
x86 Instruction “Geometry”

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

Shifted one byte…

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

Gadgets Galore!



In Search of Gadgets?

Existence of gadgets is helped by…

- Dense instruction set.
  - Increased chance a bit pattern is an instruction.
- Variable length instructions.
  - Each instruction admits many parses.
- Ambiguity in where instructions start.
  - Adding no-op padding can mitigate.

Example ROP Constructs

ip for gadget 1
constant
ip for gadget 2

high

reg := constant;

gadget 2

implemented by

pop %reg;

ret

Defending Against ROP

- Have separate stacks for variables vs return IP, so overflow of writes cannot change return IP.
  - StackShield [Cowan et al 1998], StackGhost [M. Frantsen and M. Shuey 2001], ROPdefender [Davii et al 2010]
- Pointer protection, so pointers cannot be forged.
  - Pointer protection codes, PointGuard.
- ASLR: Make gadget address unpredictable.
- G-Free: Generate code that does not include gadgets(!).

- CFI: Enforce control flow of original program.

PointGuard

Protects all pointers in programs.

Idea: Pointers stored in memory are encrypted.

Encryption: XOR with constant in global var

- Pointer must be in register for use.
- Do Decryption when pointer is loaded into register

Reference Monitors

Requirements

- Get control on relevant events.
- Able to perform remediation (eg kill process)
- Tamperproof.

Implementation

- External to monitored program (eg OS)
- Inlined into monitored program. (eg IRM, SFI)

Reference Monitors: Policies

Kinds of Polices: Must be safety properties.

- Allowed actions independent of program.
- Allowed data access for this program (SFI)
- Allowed control flow for this program (CFI)

Control Flow Integrity (CFI)

- Compute control flow graph before execution.
- Added run-time checks ensure all control transfers follow the graph.
  - Check precedes the control transfer (call/jmp/ret/….).

Adversary: Assumed to have full control over data memory of executing program.

CFI Implementation: Binary code rewriting. (IRM).

CFI Instrumentation

- Static analysis to obtain CFG
- Computed control transfers require run-time instrumentation.
- Posit instructions:
  - `label ID`.
  - `call ID,DST` xfers to addr `DST` only if that location contains instruction: `label ID.`
  - `ret ID`

… could be implemented in sw or hw.

Control Flow Graph

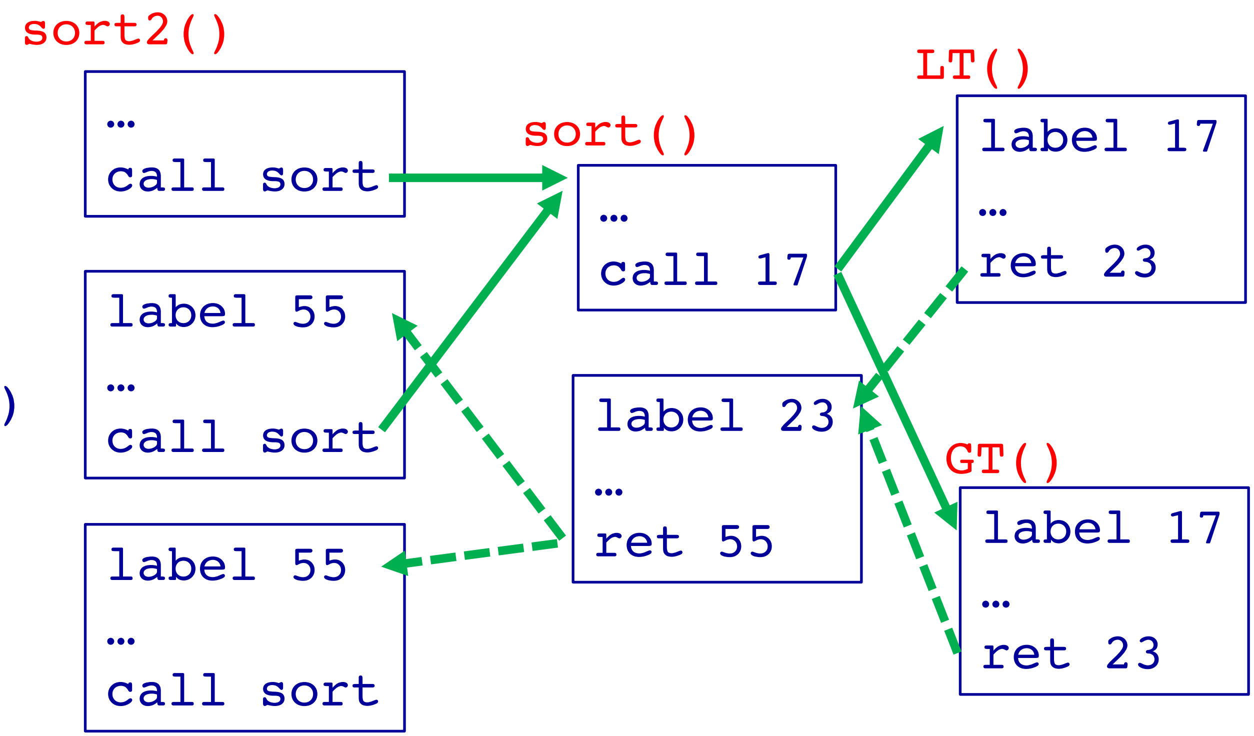
- Sources (store: call/jmp/ret)
- Destinations (store: label)
  - Equivalent destinations have the same set of in-bound edges.
- Edges (distinguish call from return)

Example CFG

```
bool LT(int x,y)
  {return x<y;}

bool GT(int x,y)
  {return x>y;}

sort2(int a[],b[],len)
  {sort(a, len, LT);
   sort(b, len, GT);}
```

Example CFG

```
bool LT(int x,y)
  {return x<y;}

bool GT(int x,y)
  {return x>y;}

sort2(int a[],b[],len)
  {sort(a, len, LT);
   sort(b, len, GT);}
```



CFI Instrumentation: Assumptions

Unique IDs. Patterns chosen are not present anywhere in code memory (except in IDs and ID checks). Probabilistic approximation possible.

Non-writable Code. Code cannot be modified at runtime).

Non-executable Data. Otherwise attacker could cause execution of an arbitrary ID.

CFI Instrumentation: `jmp ecx`

```
cmp [ecx],1234567h      id is at dest
jne error_lab           id check
lea ecx,[ecx+4]         first inst is past id
jmp ecx                 branch
```

Destination Equivalence

Control Flow Graph cannot distinguish between equivalent sources/destinations, so some illegal execution is not stopped.

- Use multiple ID's at a given destination.
- Duplicate code blocks.
- Employ a shadow stack.

Summary

- Code insertion → Code abuse
  - return-into-libc
  - return oriented programming (ROP)
- Corrupt the stack or some function pointer.
  - Protect stack from corruption
    - Canary
    - Shadow stack
  - Protect pointers from corruption
- Reference monitor for CFI (“ideal program”)