Introduction to Cybersecurity
A Software/Hardware Approach

Application Level Attacks

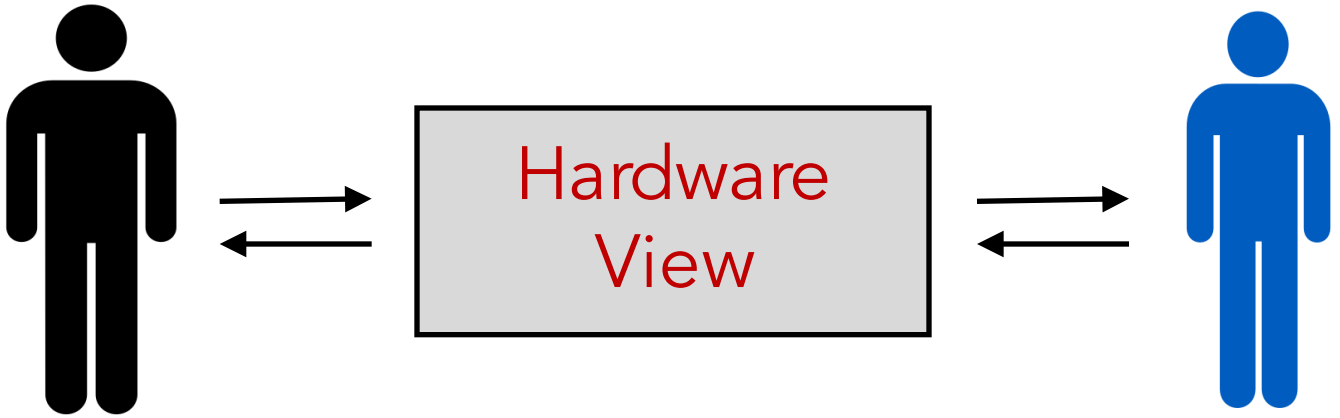
Prof. Michel A. Kinsy

BOSTON UNIVERSITY

Department of Electrical & Computer Engineering

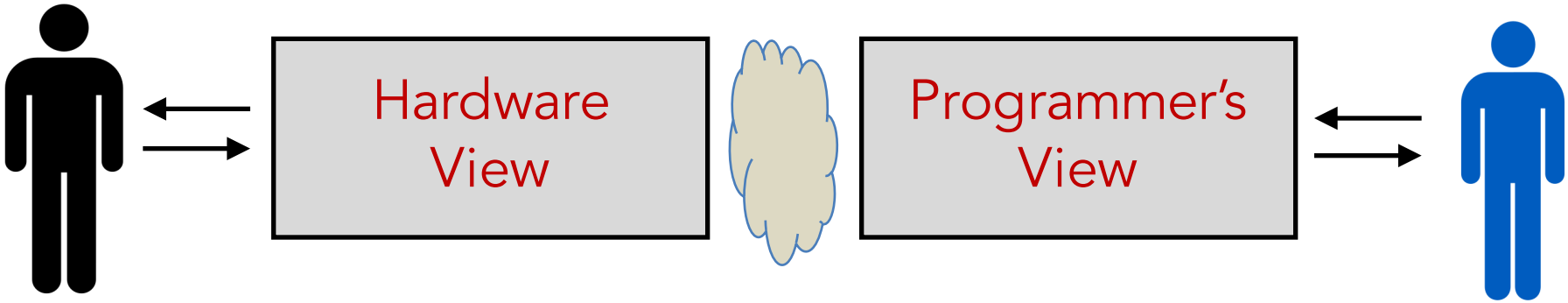
Programming Model

- The underlying programming model does not help either
- In the mid in 50's, the programmer's view of the machine was inseparable from the actual hardware implementation



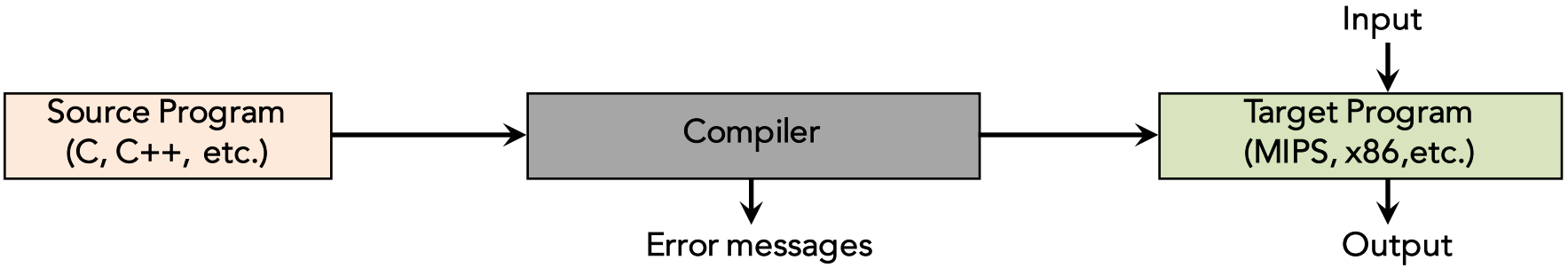
Programming Model

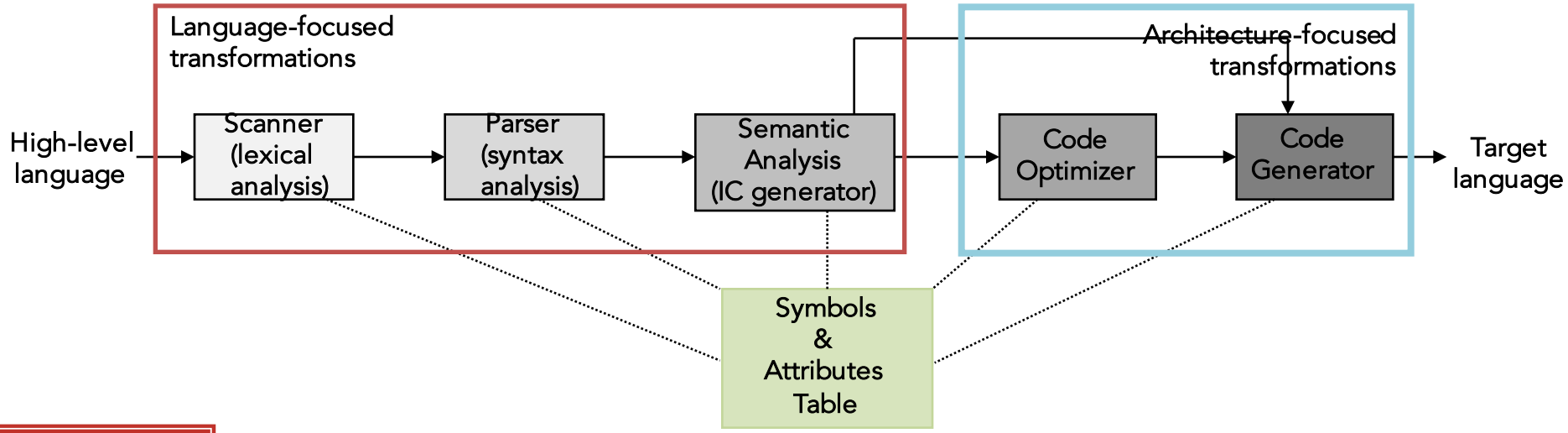
- Over time the programmer's view and the hardware implementation diverged
  - Programmer visible state of the processor (and memory) plays a central role in computer organization for both hardware and software
    - Software must make efficient use of it
- Programmer's machine model is a contract between the hardware and software



Application Compiling Process

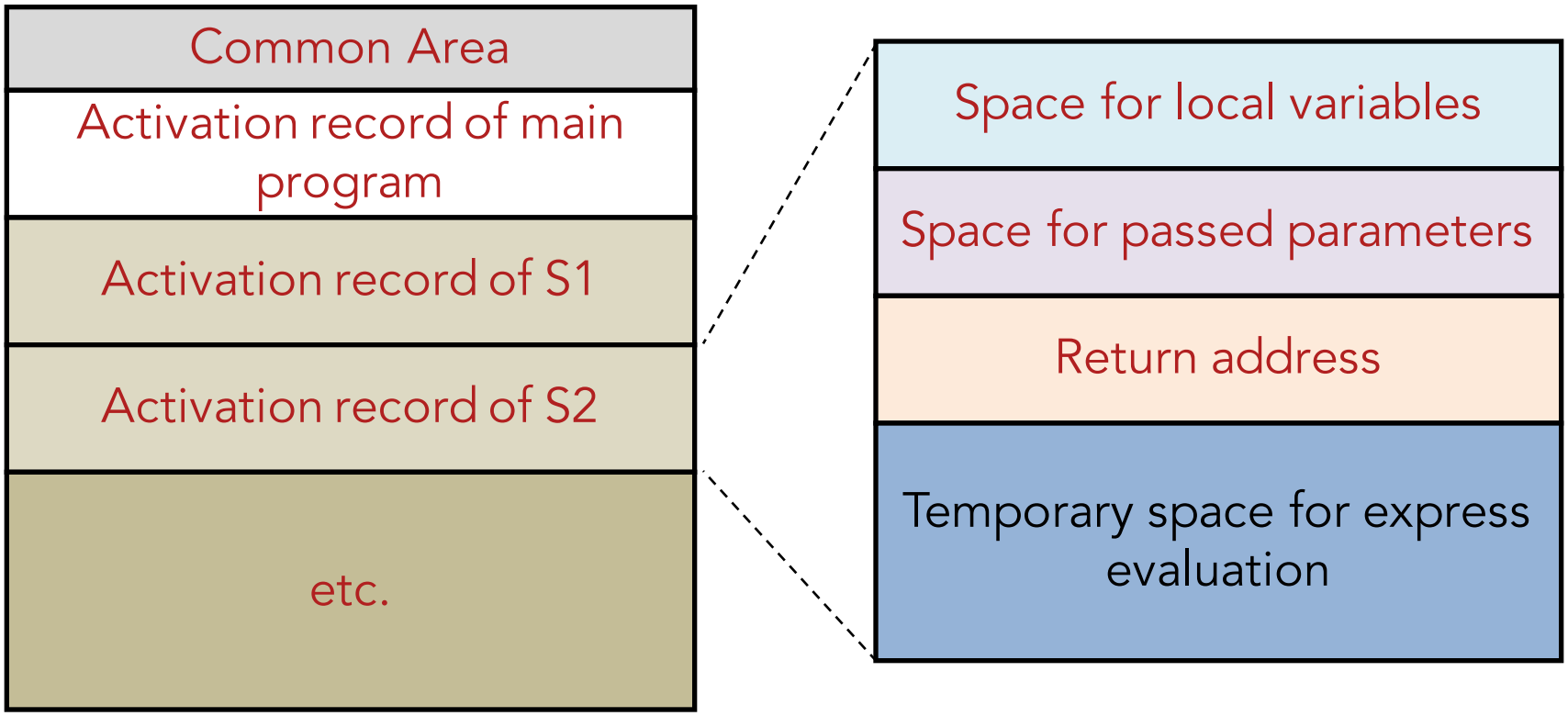
- Providing that layer of abstraction





Procedure Environment

- Activations and Allocations



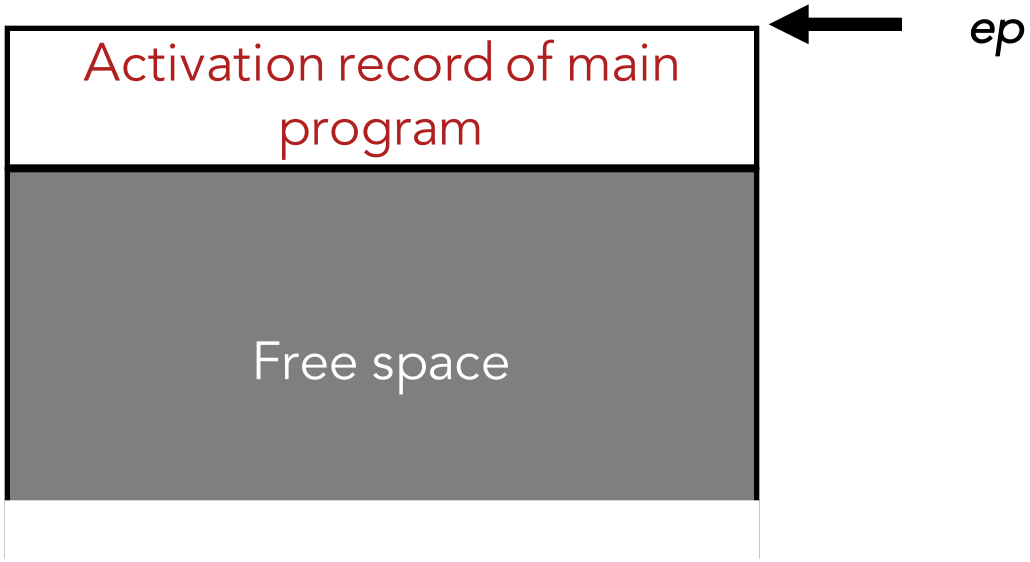
Stack-Based Environments

- Environment management is more dynamic
- Since procedures have no fixed locations for their activations, *environment pointer (ep)* is used to track the current activation
- Activations are in a stack, the pointer to the previous activation record is called *control link* or *dynamic link*

```
Void foo (void) {
    ....
}

Void bar (void) {
    ....
    foo()
}

Void main (void) {
    ....
    bar()
}
```

Stack-Based Environments

- With the execution of main



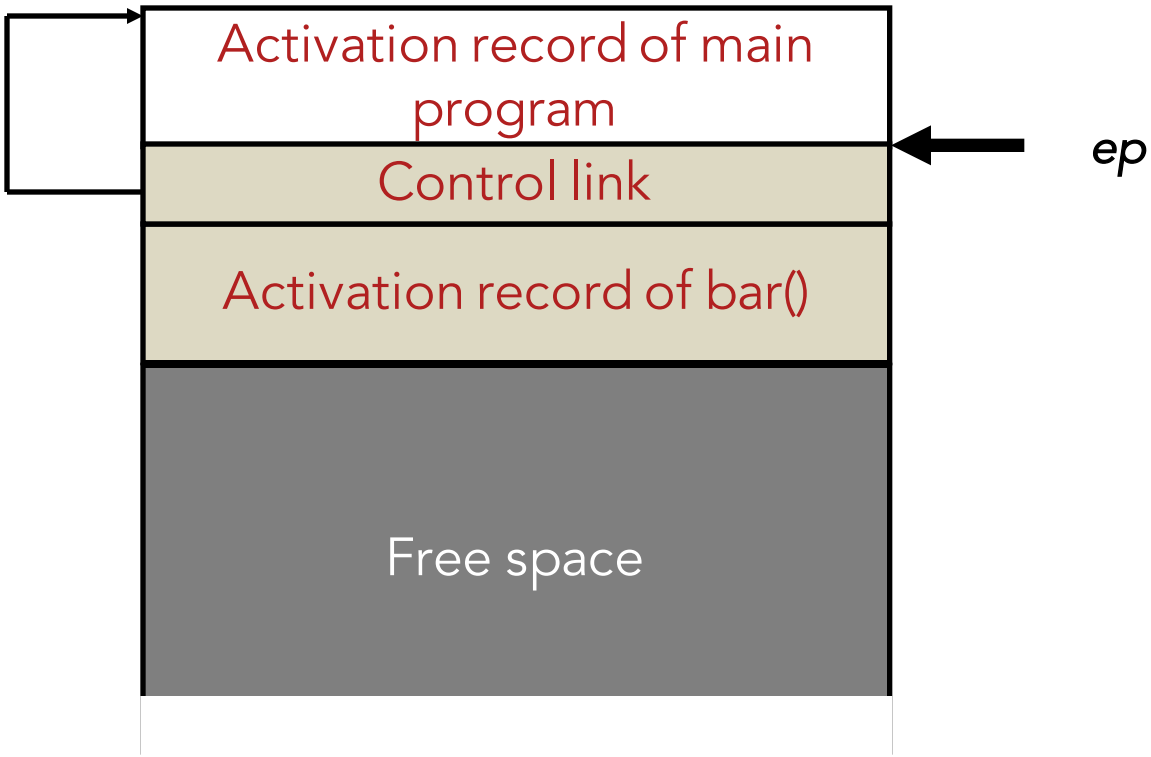

```
Void foo (void) {
      ....
}
```

```
Void bar (void) {
      ....
      foo()
}
```

```
Void main (void) {
      ....
      bar()
}
```

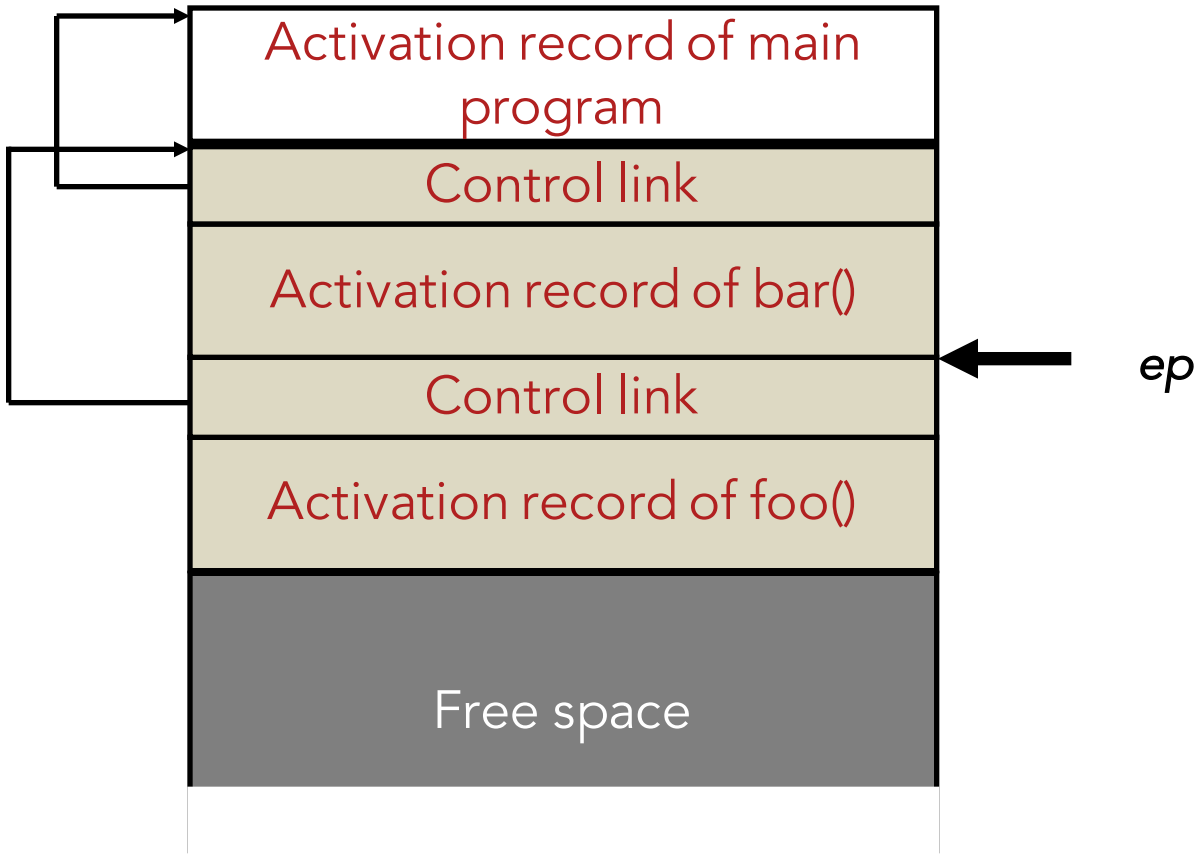
Stack-Based Environments

- After bar is called

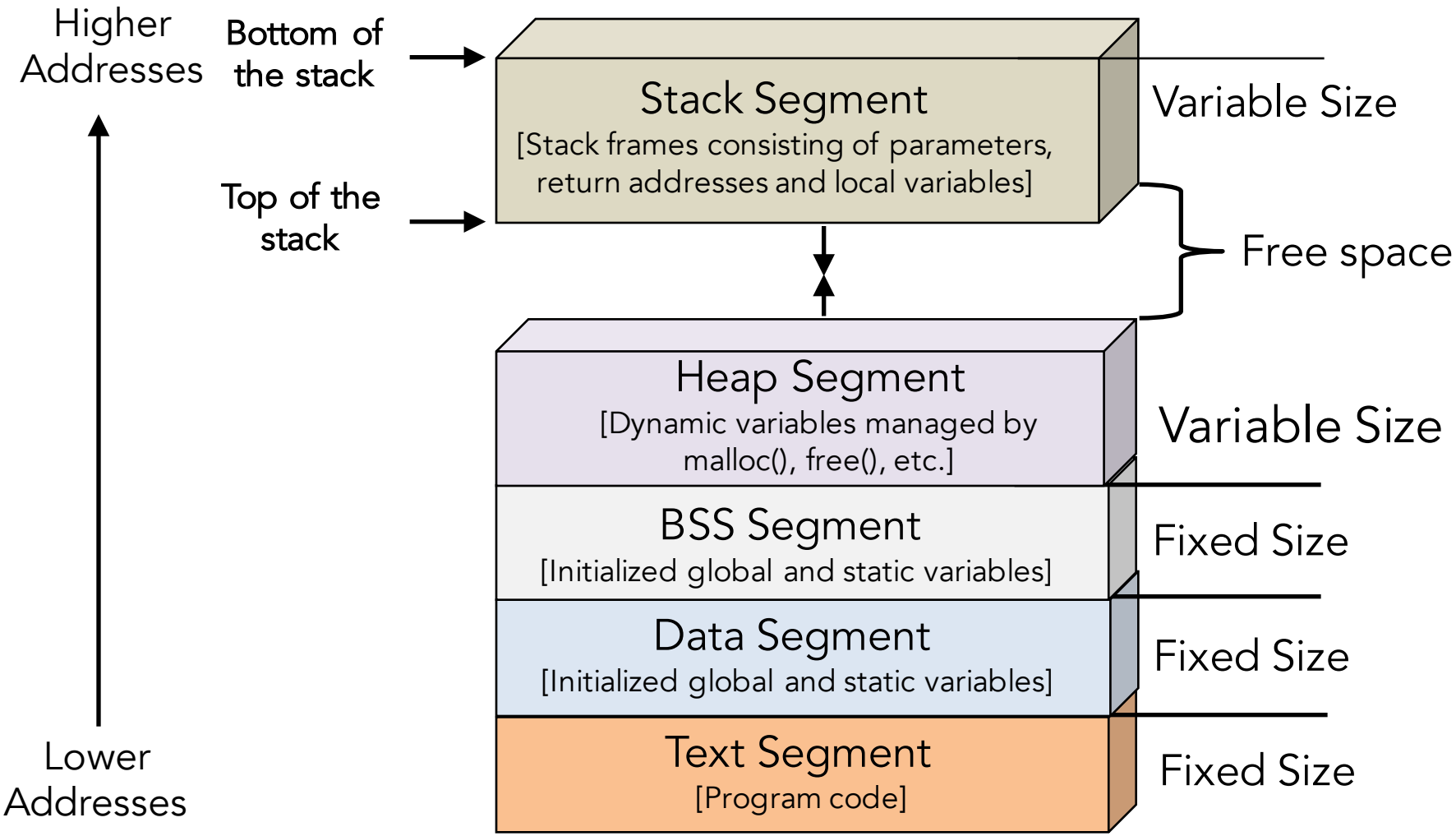


Stack-Based Environments

- Finally with the call of foo



Program memory management



Instruction Types

- Register-to-Register Arithmetic and Logical operations
- Control Instructions alter the sequential control flow
- Memory Instructions move data to and from memory
- CSR Instructions move data between CSRs and GPRs; the instructions often perform read-modify-write operations on CSRs
- Privileged Instructions are needed by the operating systems, and most cannot be executed by user programs

Attack Formalism

- An attack has three components
  - Channel
    - Delivery mechanism
  - Entry
    - Bug or vulnerability or even feature exploitation
    - Binary vulnerabilities
      - Stack overflow
      - Heap overflow
      - Null pointer dereference
  - Payload
    - The actual attack function
    - E.g., Get the ***Instruction Pointer*** to point to an attacker specified procedure

Code Injection

- Code injection can be used by an attacker to introduce (or "inject") code into another program to change the flow of the execution and to execute their own dedicated malicious code
- There are many types of code injection schemes
  - SQL injection
  - Script injection
  - Shell injection
  - OS command injection

Buffer Overflow

- There can overflow on both
  - The stack
  - The heap

```
void bar (char *str) {
    char array[256];
    strcpy(array, str);
    foo(array);
}
```

If `*str` is let us say 512 bytes long, then after `strcpy`, the function bar return address may be overwritten

Buffer Overflow

- There can overflow on both
  - The stack
  - The heap
- Many C functions like:
  - strcpy (char *dest, const char *src) are unsafe and their advertised safe versions, like strncpy(), are not either
    - strncpy() may leave buffer unterminated
    - Should be replaced by
      - strncpy(dest, src, sizeof(dest)-1)
      - dest[sizeof(dest)-1] = `\0`;

Buffer Overflow

- Example attack steps are
  - Inject attack code into buffer
  - Overflow return address
  - Redirect control flow to attack code
  - Execute attack code
- Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control
  - Use of Address Space Layout Randomization as protection
    - Arrange the positions of key data areas randomly in a process' address space

Buffer Overflow

- Function pointer overwritten
  - Overflowing function pointer
  - Harder to defend than return-address overflow attacks
- Smashing the Stack
  - Overflow the stack so that it overwrites the return address
  - When the function finishes, it will return to whatever address/value is written on the stack
  - A specific return/new address can be written to stack paired with code to perform hijack

Buffer Overflow

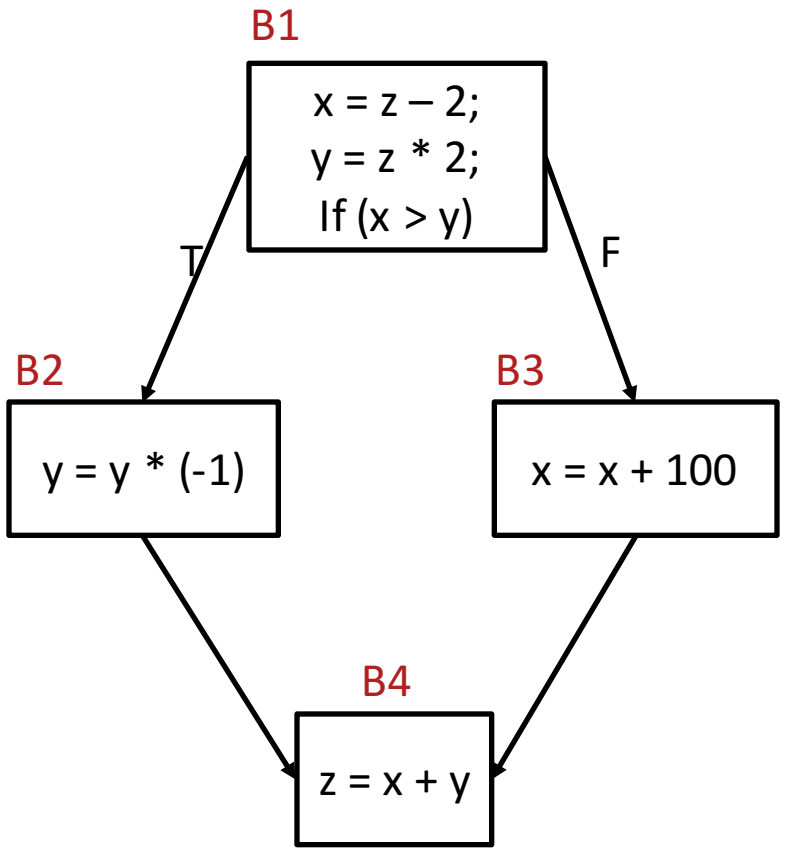
- Non-executable stacks
  - Can prevent many stack-based attacks
  - But cannot guard against return-to-libc attacks
  - Or protect against heap and function pointer overflows
- Canaries
  - Insert canaries in stack frames and verify their integrity during function returns
  - Have a canary for each frame and make as random as possible to make it hard on the attacker to guess or learn

Execution Control Flow Transfers

- Changes the control flow of a program in a specific way, conditionally or unconditionally
  - Direct transfer: Target is encoded as immediate offset in the instruction itself
  - Indirect transfer: Target depends on the runtime value of a register or memory reference
- Some of these control flow transfers are
  - Exceptions
  - Direct or conditional jumps or function calls
  - Indirect jumps or calls
  - Return instructions

Control-Flow Graph (CFG)

```
int x,y,z;
x = z - 2;
y = z * 2;
if (x > y) {
    y = y * (-1);
} else {
    x = x + 100;
}
z = x + y;
```



- CFG represents the control-flow execution of a program:
  - Nodes are basic blocks
  - Edges are possible flow control between blocks
  - Each block can have multiple incoming/outgoing edges

Control-Flow Hijacking

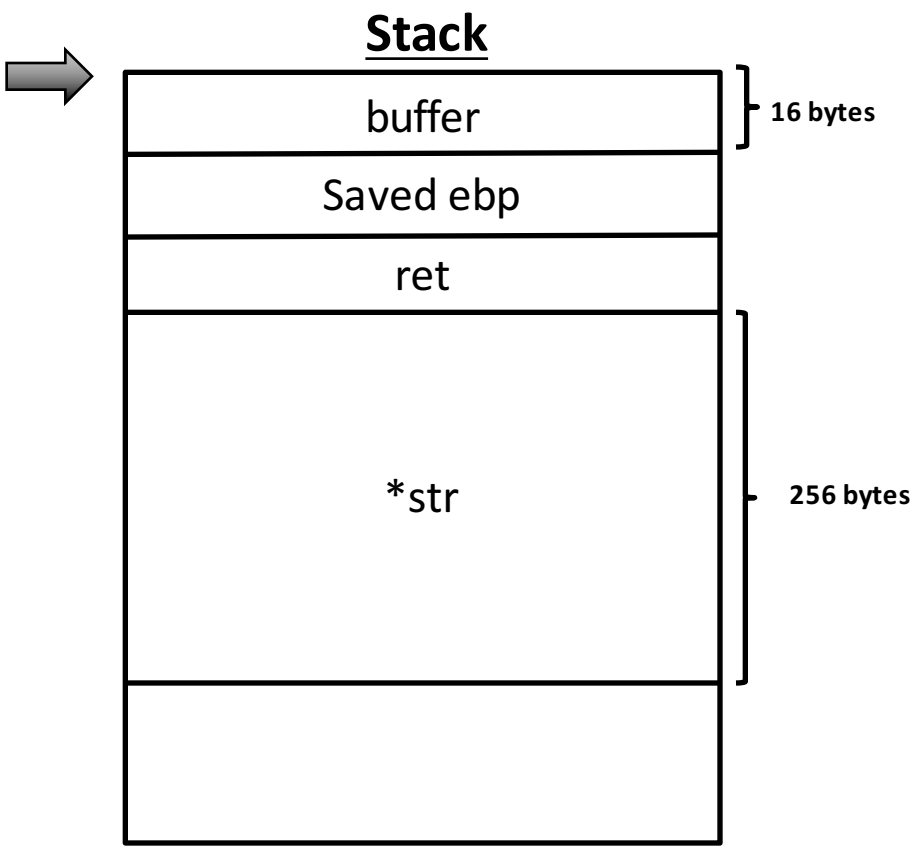
- Takes control over the victim by overwriting sensitive data structures to modify control flow of a program
- Considered one of the most dangerous class of security attacks
  - Exploit software vulnerabilities directly without asking for user actions
  - Used as basic building blocks to propagate between victim machines

Control-Flow Hijacking

- Tries to control an indirect control-flow transfer instruction in vulnerable program
  - Function pointers
  - Return addresses
- Often leads to code-reuse and code-injection attacks
  - Buffer overflow
  - Return-to-libc
  - Return-oriented programming (ROP)

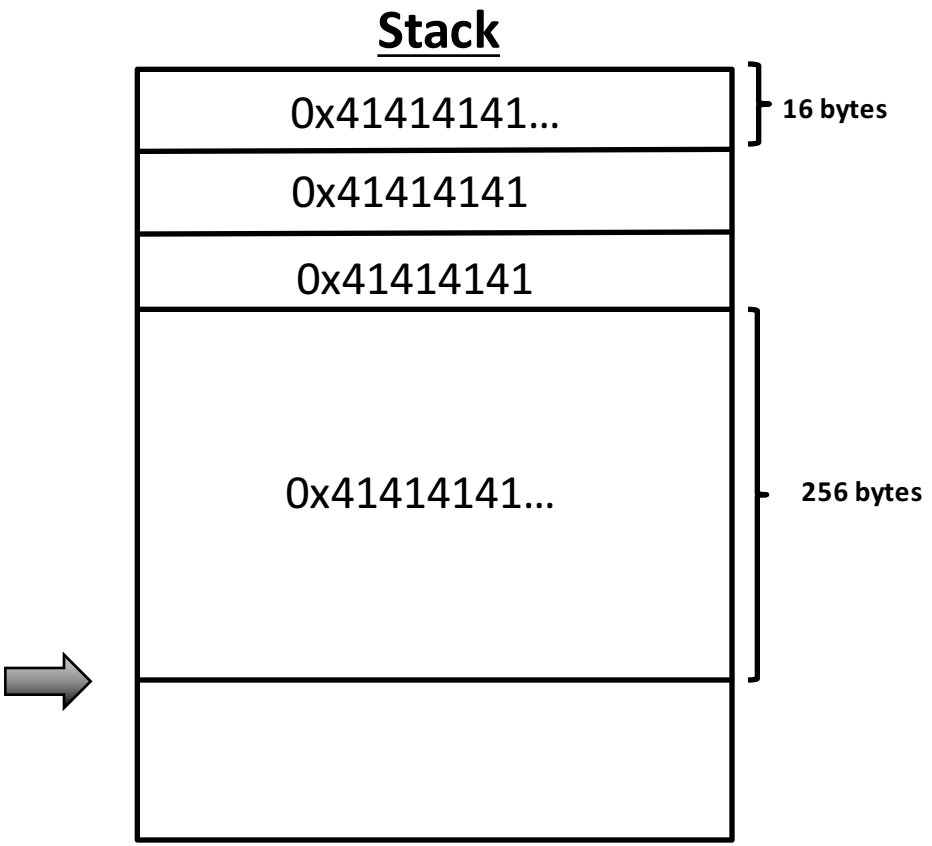
Buffer Overflow

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main(){
    char large_str[256];
    int i;
    for(i=0; i<255; i++) {
        large_str[i] = 'A';
    }
    foo(large_str)
}
```



Buffer Overflow

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main(){
    char large_str[256];
    int i;
    for(i=0; i<255; i++) {
        large_str[i] = 'A';
    }
    foo(large_str)
}
```



Control-Flow Integrity (CFI)

- Ensures the validity of control-flow graph (CFG) intended by the programmer
- Inserts checks before control-flow instructions to allow only valid targets
- **Problem**: Hard to be adopted in real-world applications
  - Requires complete and precise CFG of the protected application
  - Hinders incremental deployment in real systems
  - Results in high performance overhead

Next Class

- More Application Level Attacks