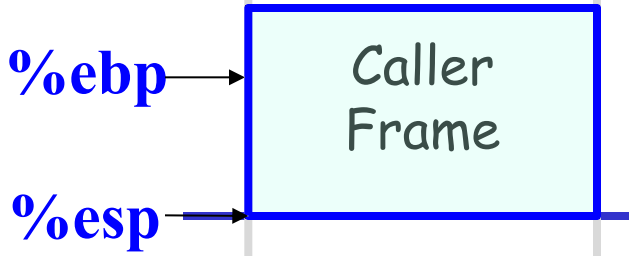
程序的机器级表示（6）

谢旻晖

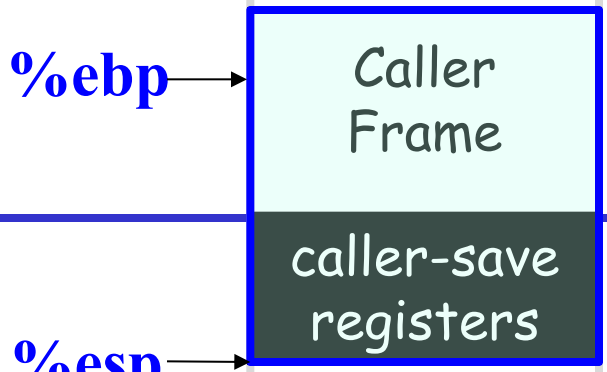
2025.11

Put it Together



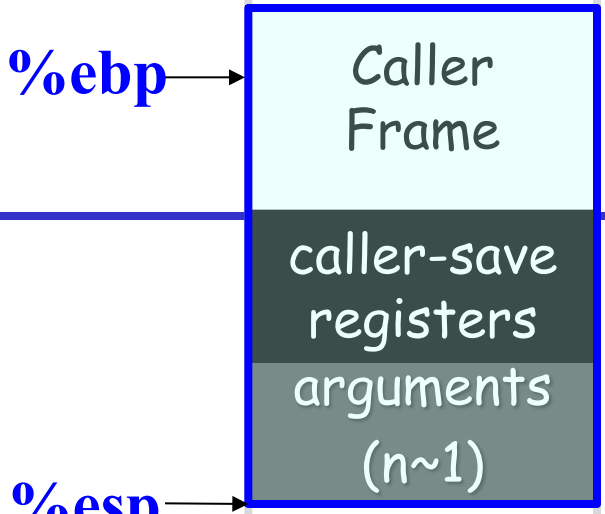
Put it Together

1. Save caller-save registers
   (%rbx, %rbp, %r12~15,
   %rsp以外的寄存器)



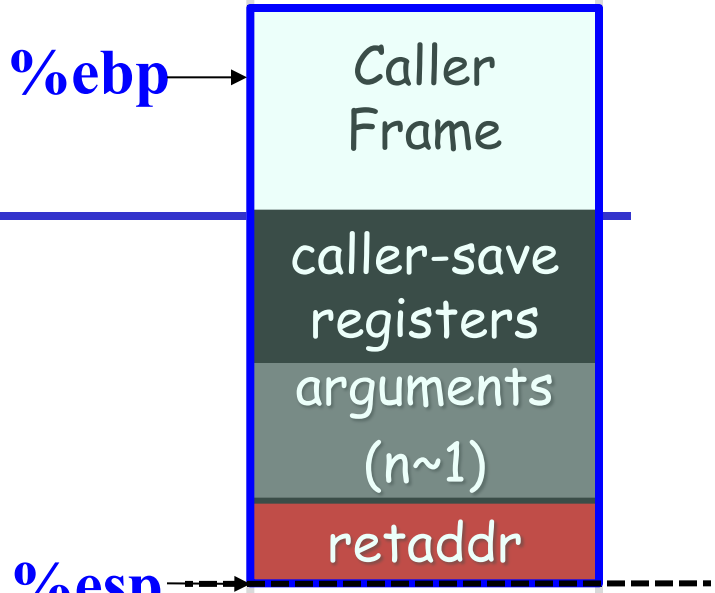
Put it Together

1. Save caller-save registers
2. Push actual arguments from right to left



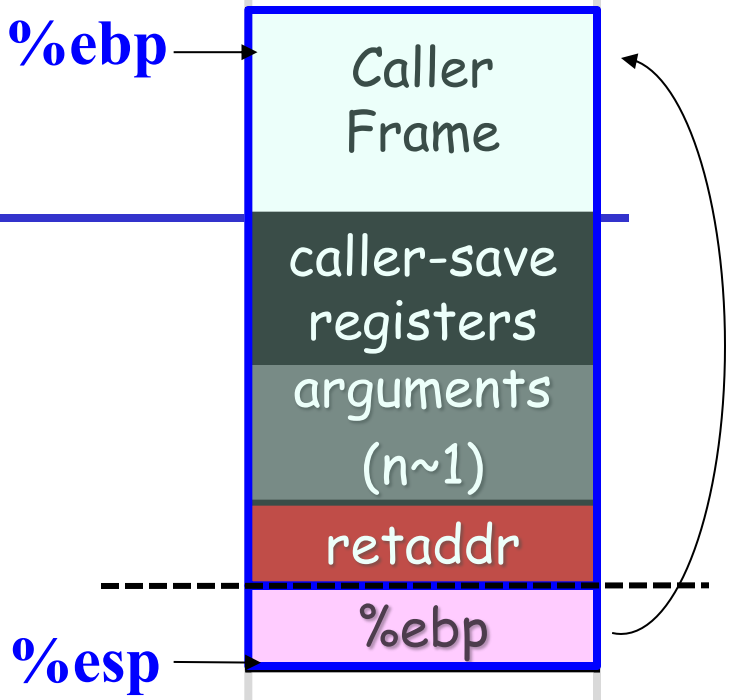
Put it Together

1. Save caller-save registers
2. Push actual arguments from right to left
3. Call instruction
   - Save return address
   - Transfer control to callee



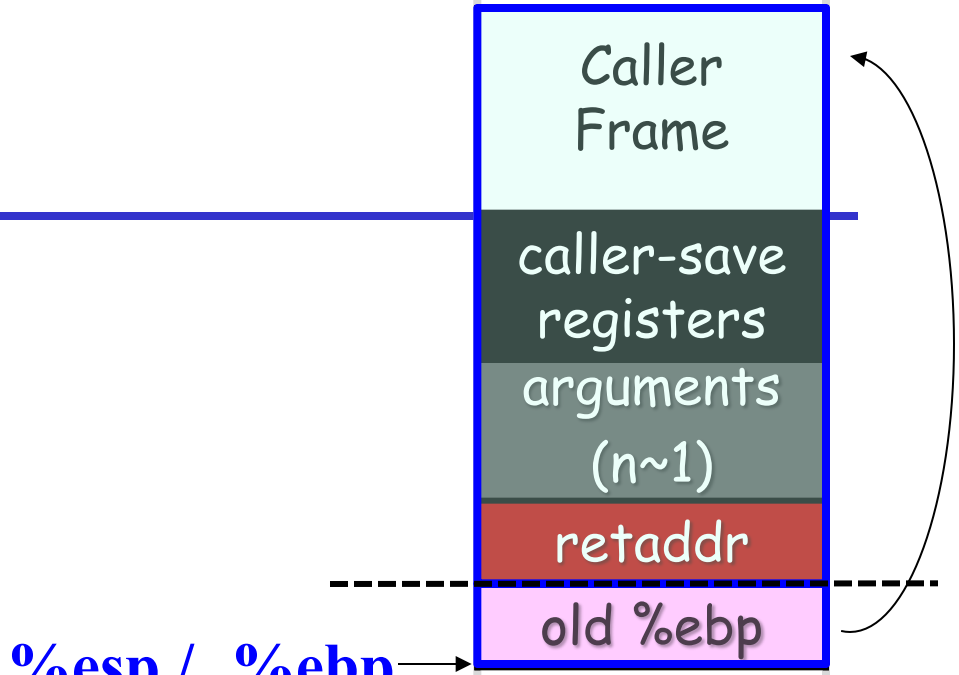
Put it Together

4. Save caller %ebp



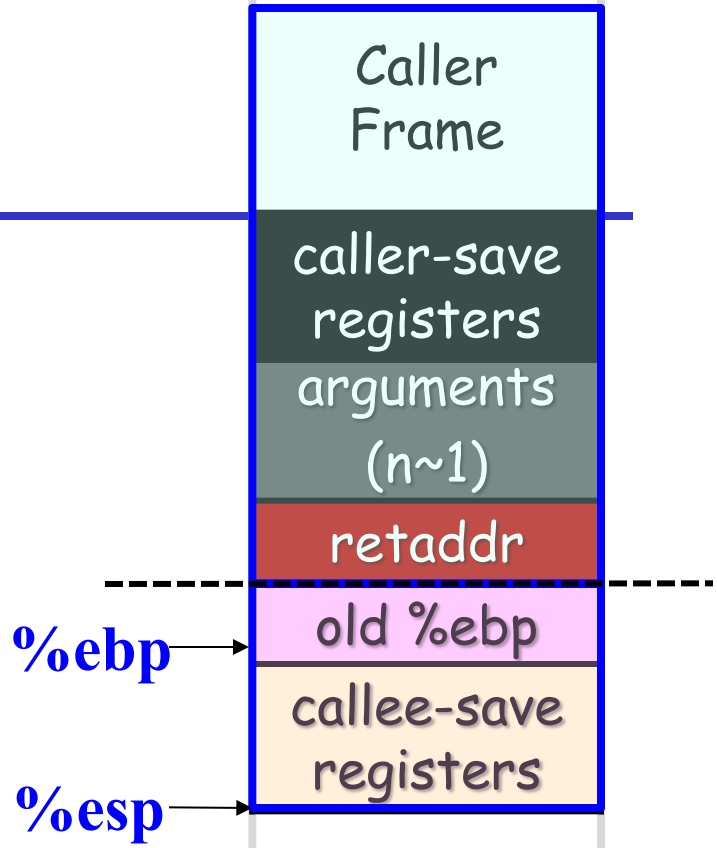
Put it Together

4. Save caller %ebp
5. Set callee %ebp



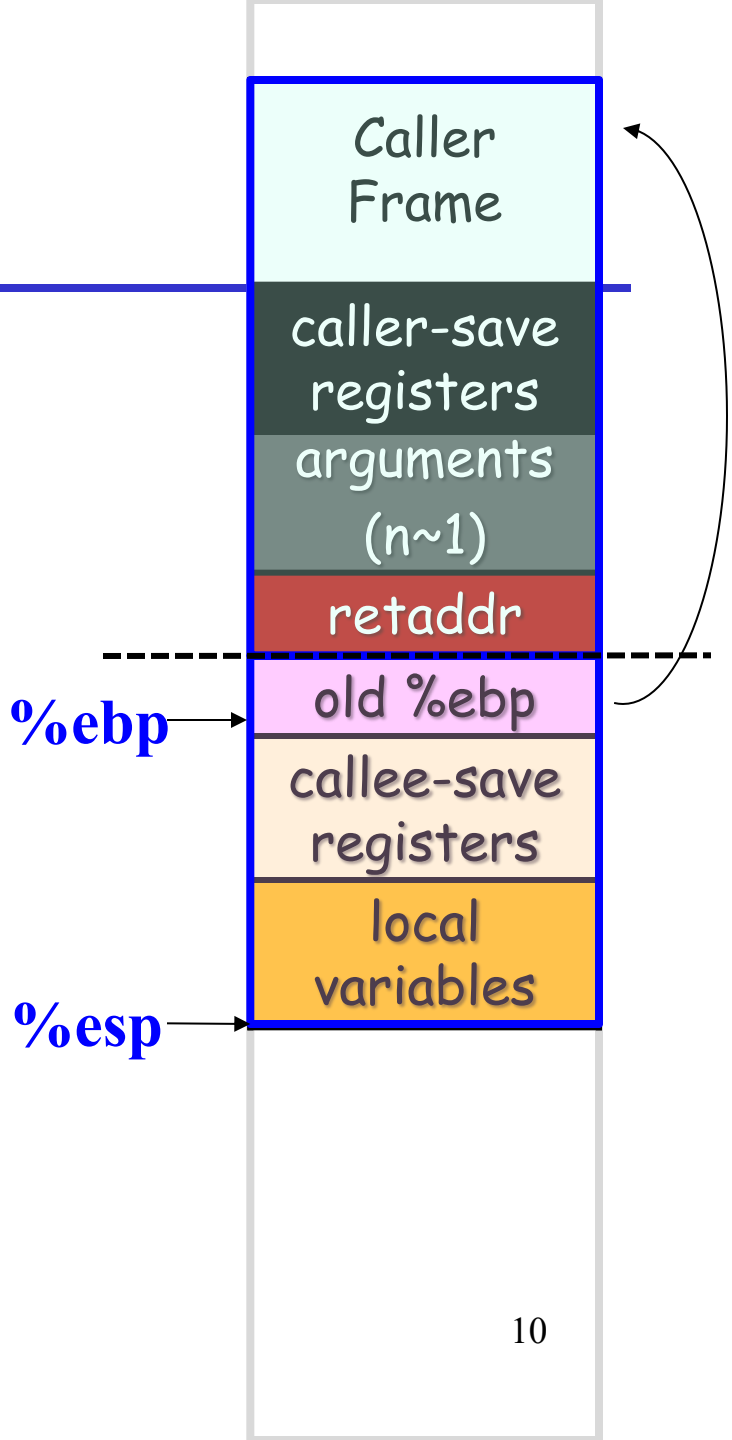
Put it Together

4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers (%rbx, %rbp, %r12~15)



Put it Together

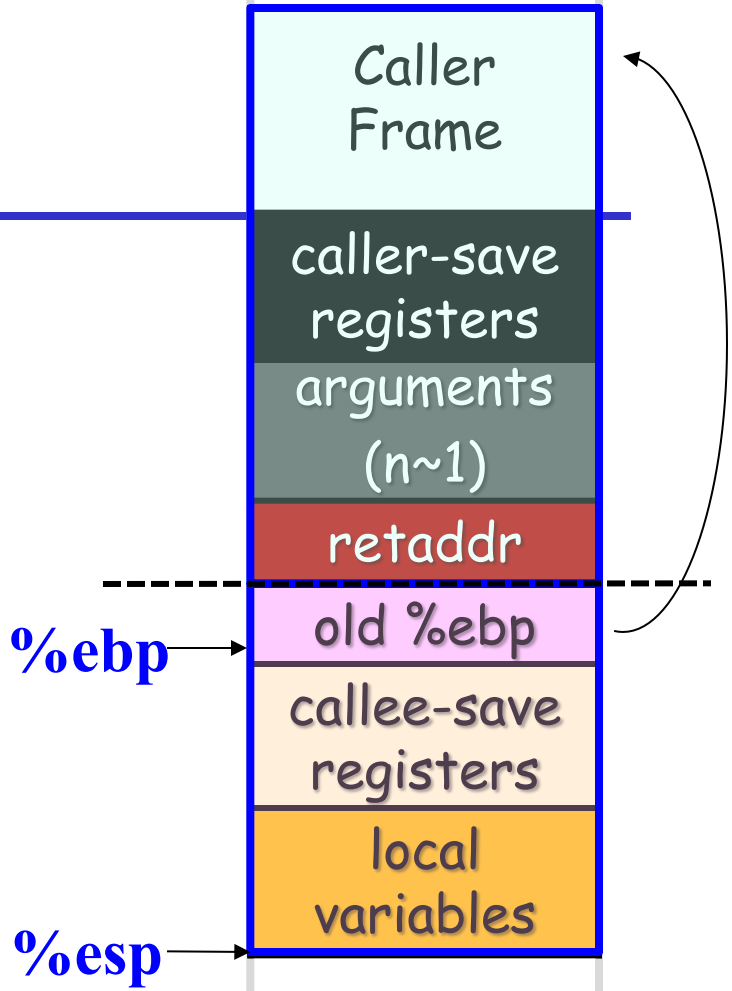
4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers
   (%rbx, %rbp, %r12~15)
7. Allocate space for local variable



Put it Together

. . .

n-4. save return value in %eax

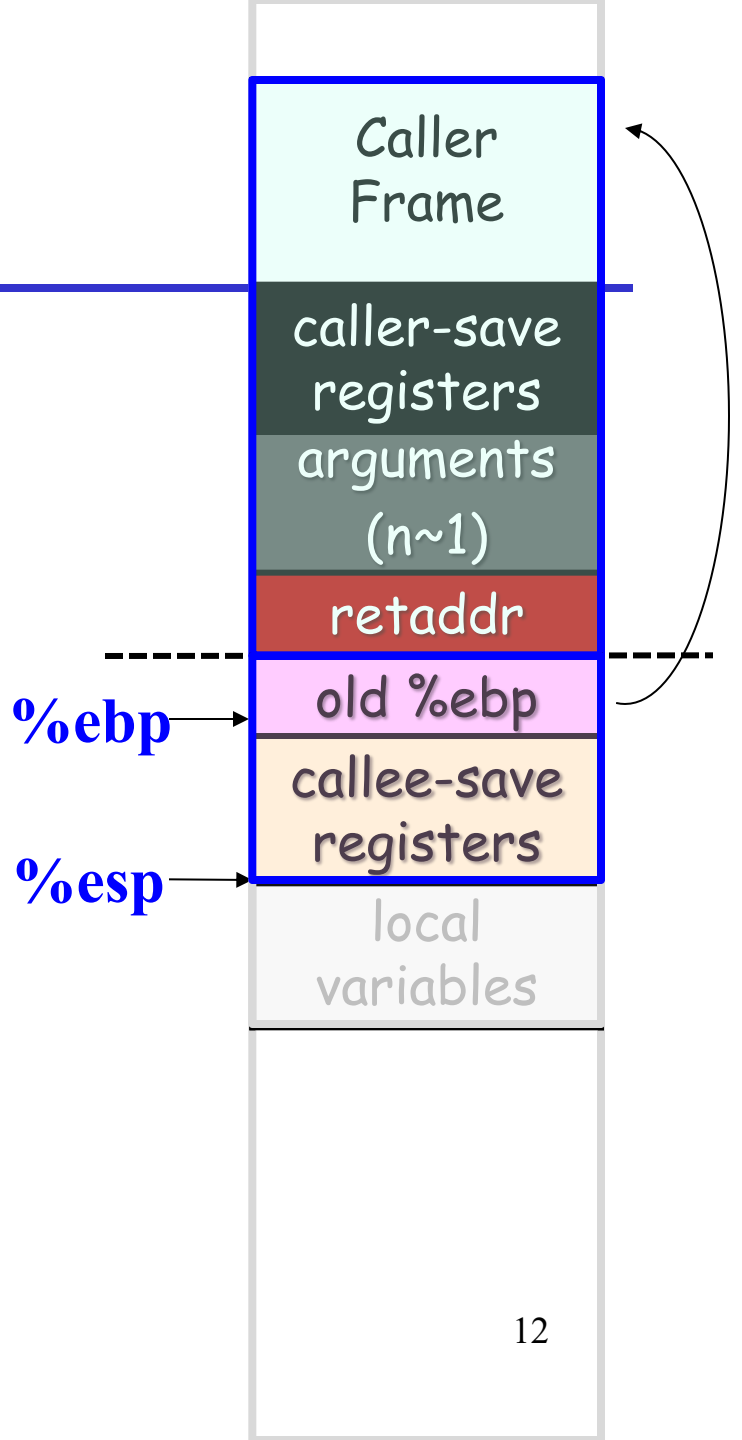


Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable



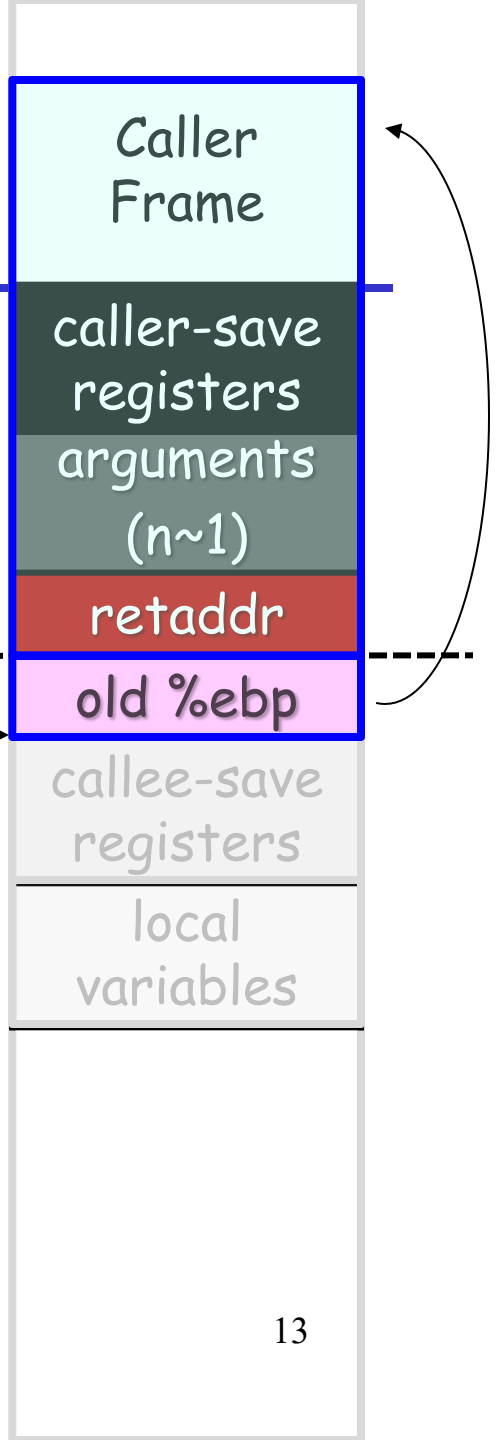
Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers



Put it Together

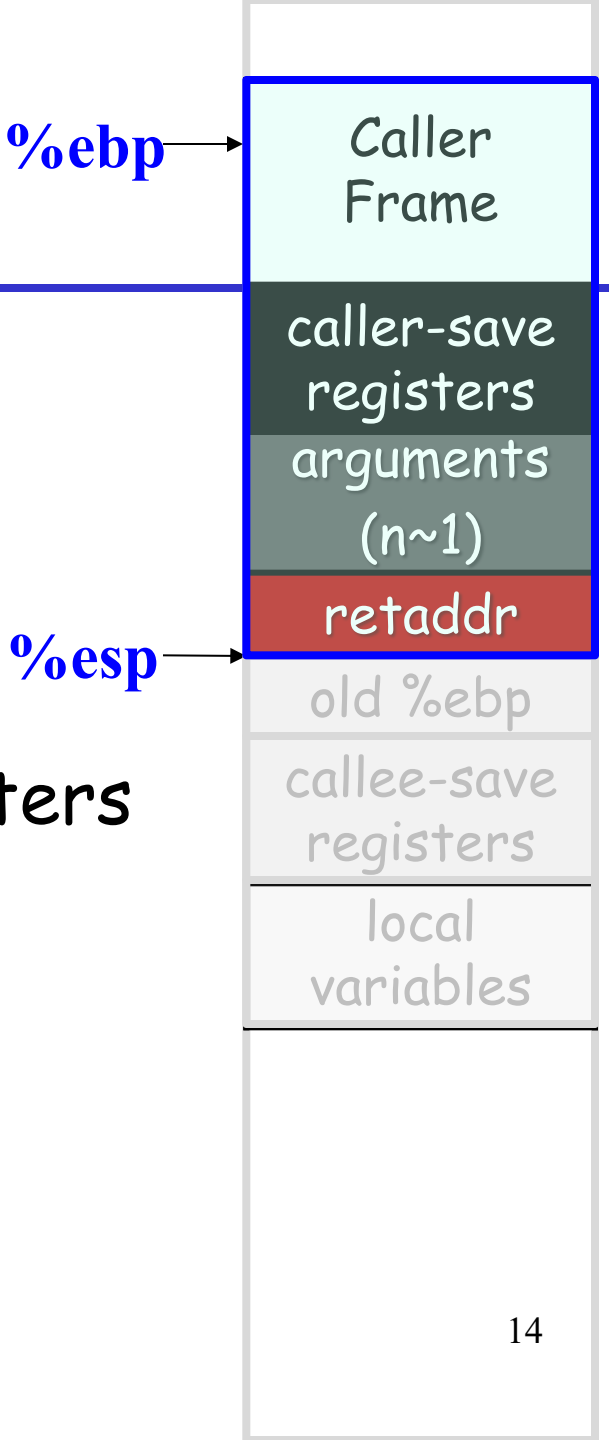
. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp



Put it Together

. . .

n-4. save return value in %eax

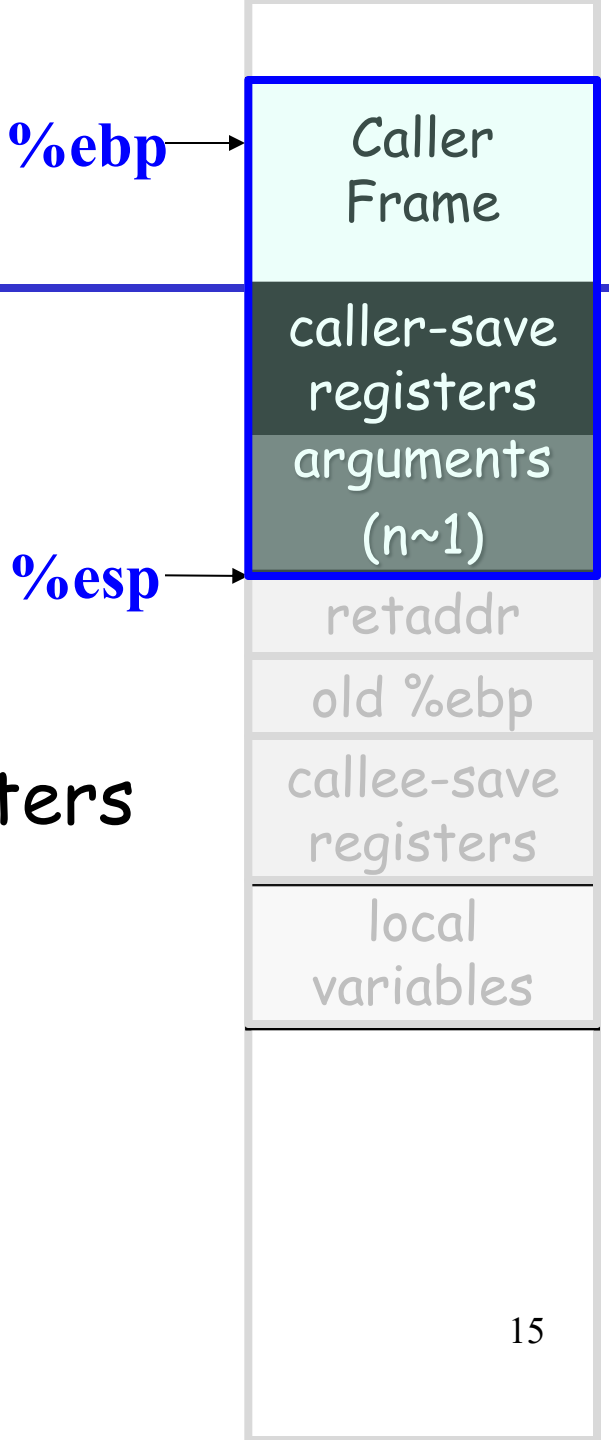
n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp

n. Ret instruction

- pop return address
- Transfer control to caller



Example

```
1 int swap_add(int *xp, int *yp)
2 {
3         int x = *xp;
4         int y = *yp;
5
6         *xp = y;
7         *yp = x;
8         return x + y;
9 }
10
```

Example

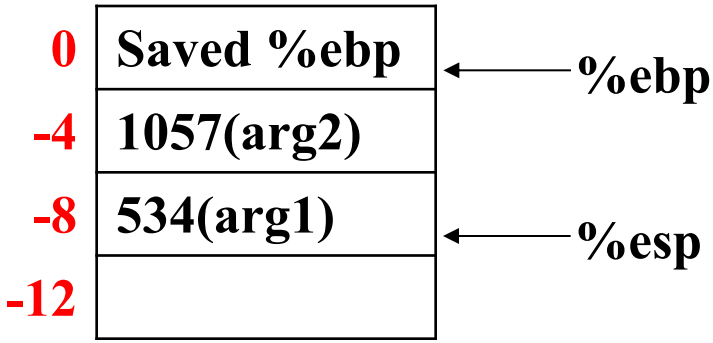
```
11 int caller()
12 {
13        int arg1 = 534;
14        int arg2 = 1057;
15        int sum = swap_add(&arg1, &arg2);
16        int diff = arg1 - arg2;
17
18        return sum * diff;
19 }
```

Example

Before

int sum = swap_add(&arg1, &arg2);

Stack frame for caller



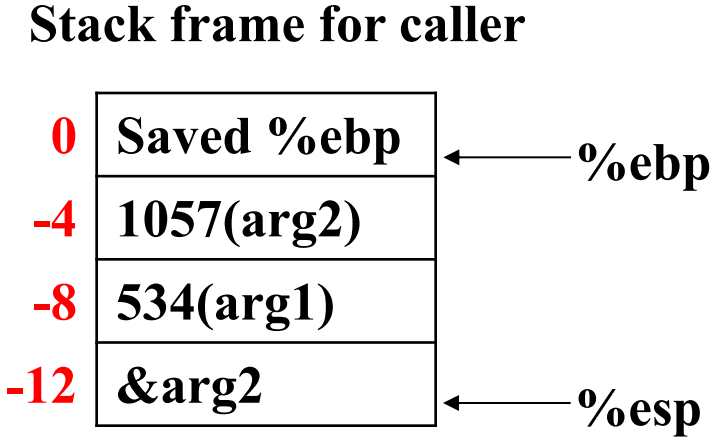
Parameter Passing

1 **leal -4(%ebp),%eax**

Compute &arg2

2 **pushl %eax**

Push &arg2



Parameter Passing

1 **leal -4(%ebp),%eax**

Compute &arg2

2 **pushl %eax**

Push &arg2

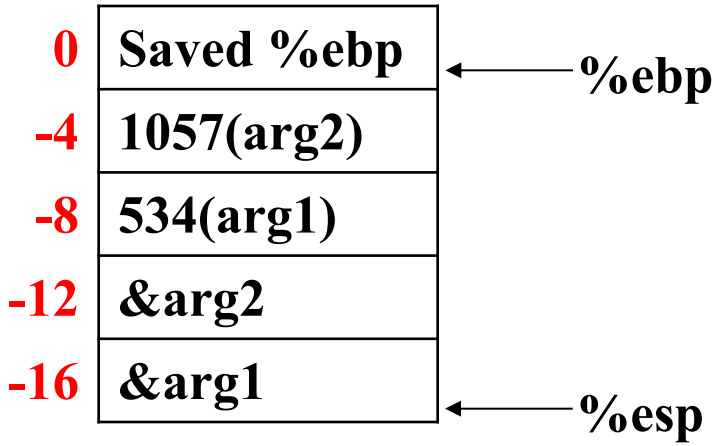
3 **leal -8(%ebp),%eax**

Compute &arg1

4 **pushl %eax**

Push &arg1

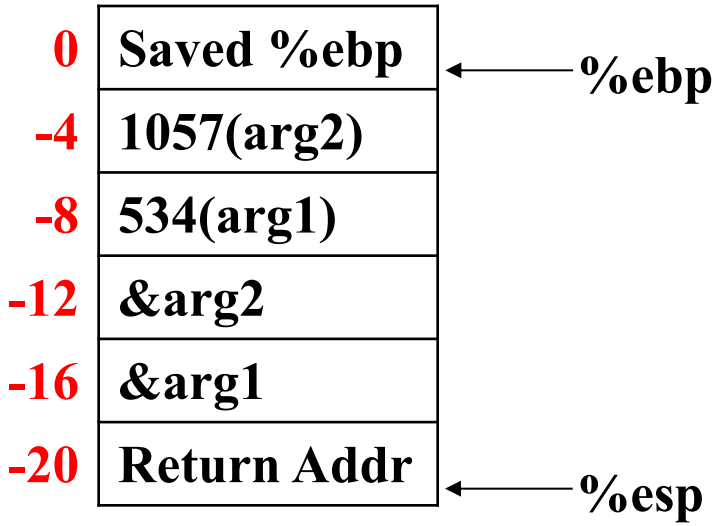
Stack frame for caller



Call Instruction

```
1   leal -4(%ebp),%eax
                    Compute &arg2
2   pushl %eax
                    Push &arg2
3   leal -8(%ebp),%eax
                    Compute &arg1
4   pushl %eax
                    Push &arg1
5    call swap_add
                    Call the swap_add function
```

Stack frame for caller



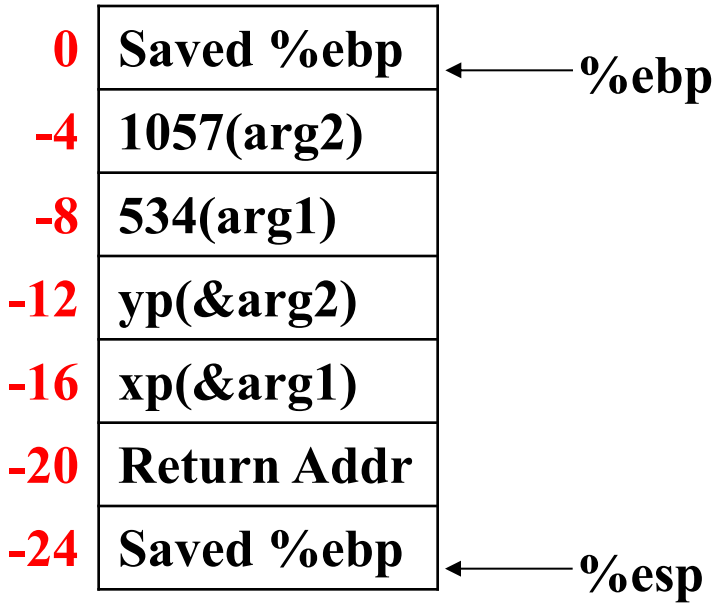
Setup code in swap_add

swap_add:

1 pushl %ebp

***Save old* %ebp**

Stack frame for caller



Setup code in swap_add

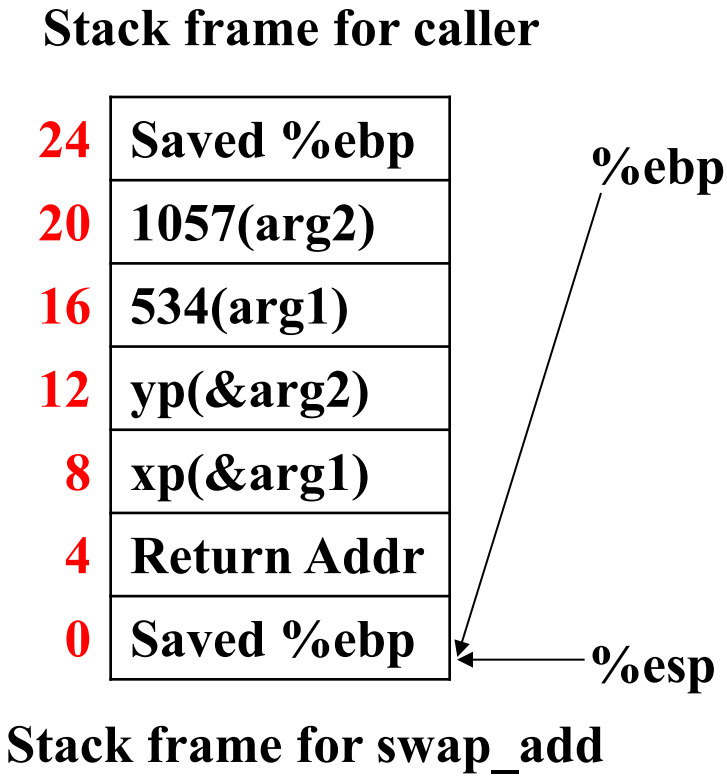
swap_add:

1 pushl %ebp

***Save old* %ebp**

2 movl %esp,%ebp

Set* %ebp *as frame pointer



Setup code in swap_add

swap_add:

1 pushl %ebp

***Save old* %ebp**

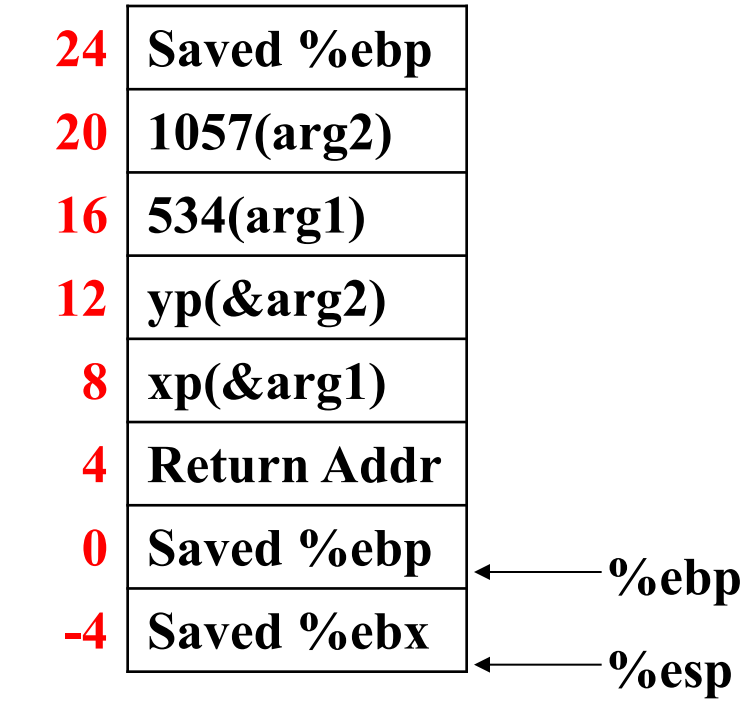
2 movl %esp,%ebp

Set* %ebp *as frame pointer

3 pushl %ebx

Save %ebx

Stack frame for caller

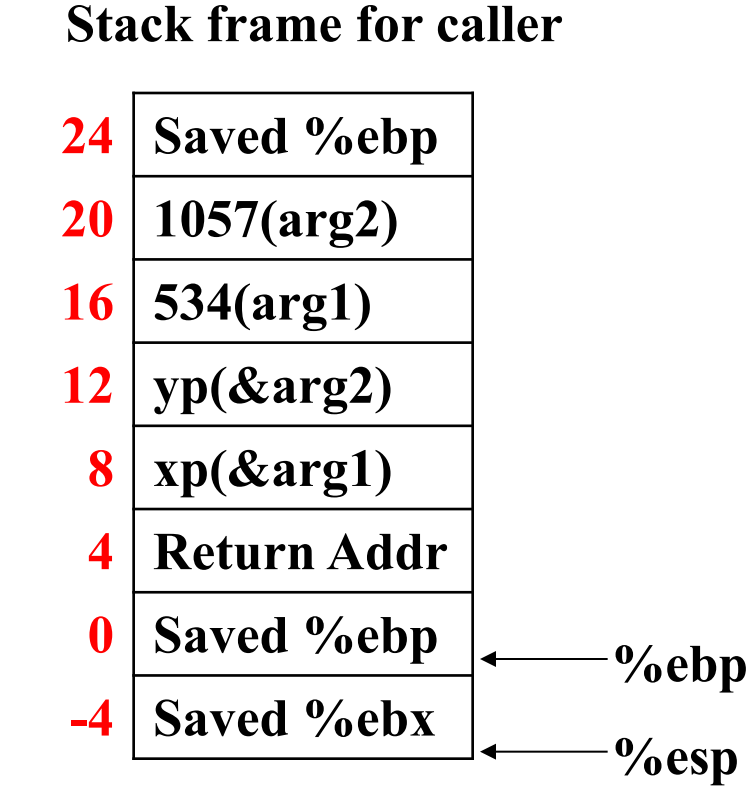


Stack frame for swap_add

Body code in swap_add

5 movl 8(%ebp),%edx

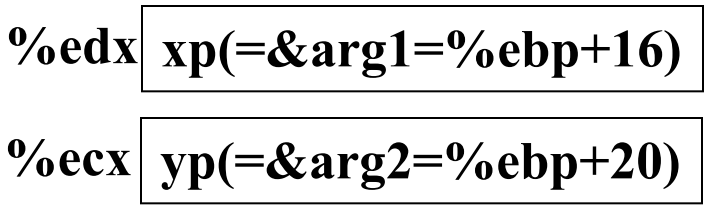
Get xp

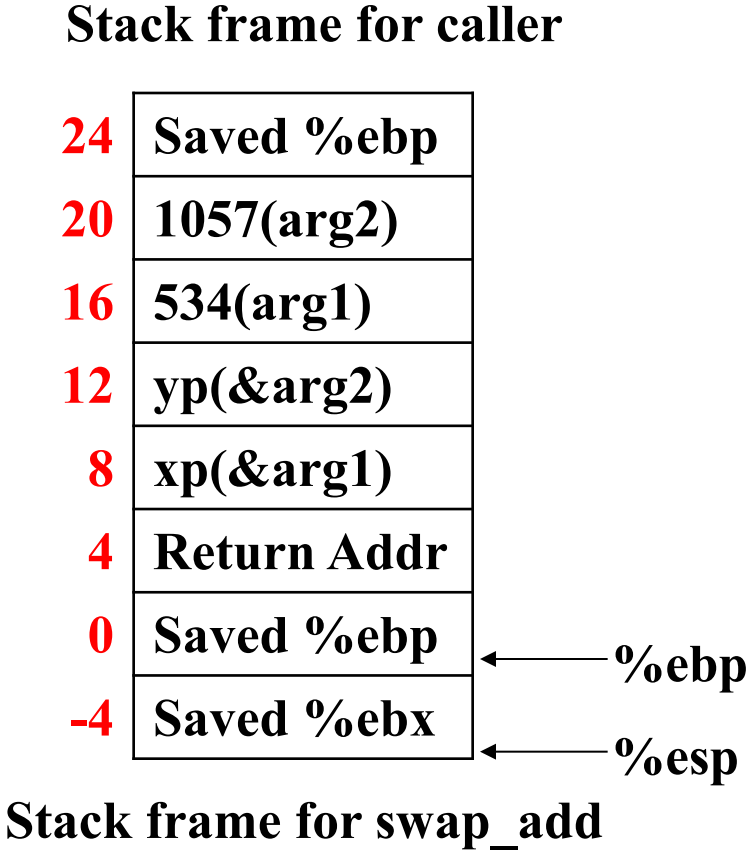


Body code in swap_add

```
5   movl 8(%ebp),%edx
```
Get xp

```
6   movl 12(%ebp),%ecx
```
Get yp



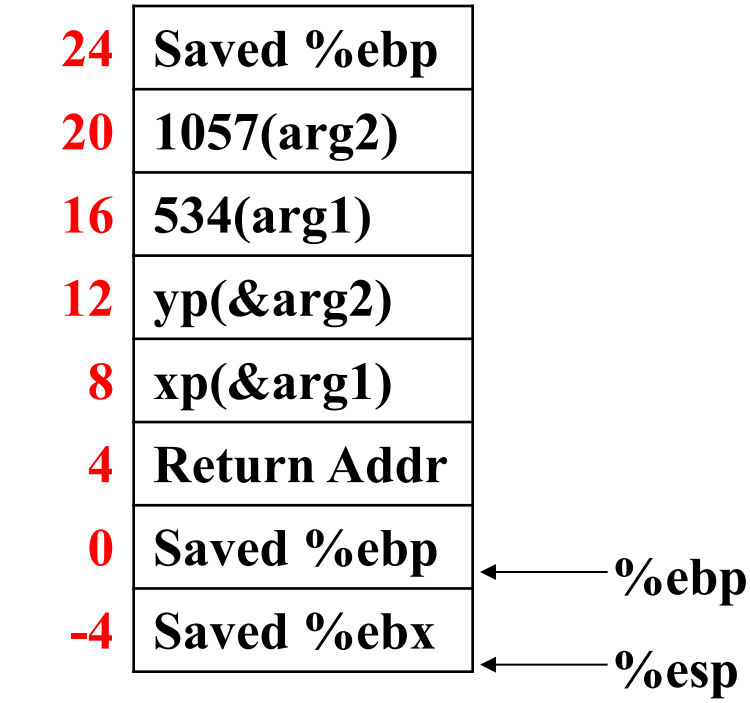


Body code in swap_add

```
5  movl 8(%ebp),%edx
                        Get xp
6  movl 12(%ebp),%ecx
                        Get yp
7  movl (%edx),%ebx
                        Get x
8  movl (%ecx),%eax
                        Get y
```

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1057

Stack frame for caller

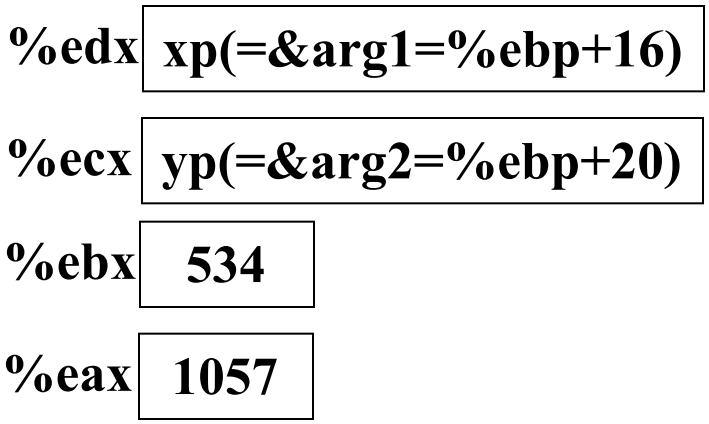


Stack frame for swap_add

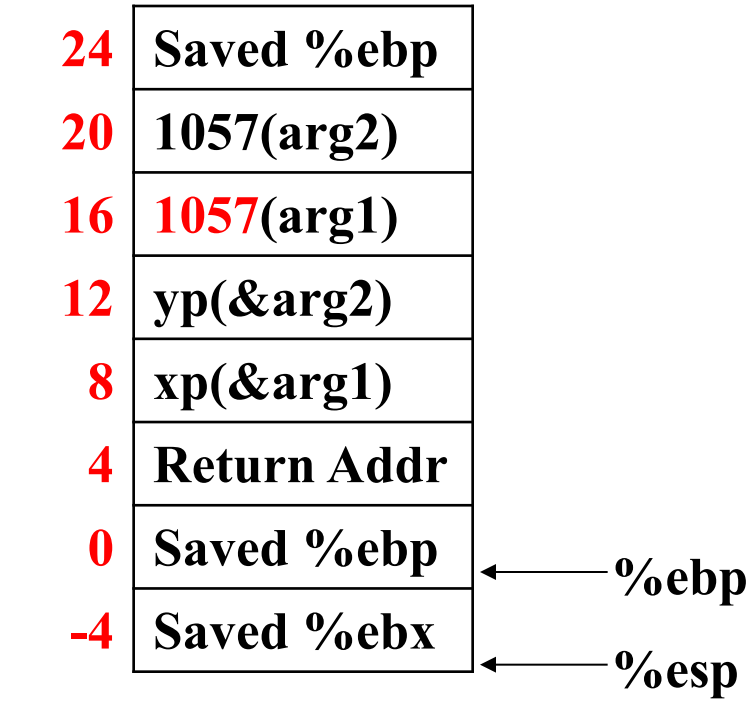
Body code in swap_add

9 movl %eax, (%edx)

Store y at *xp



Stack frame for caller



Stack frame for swap_add

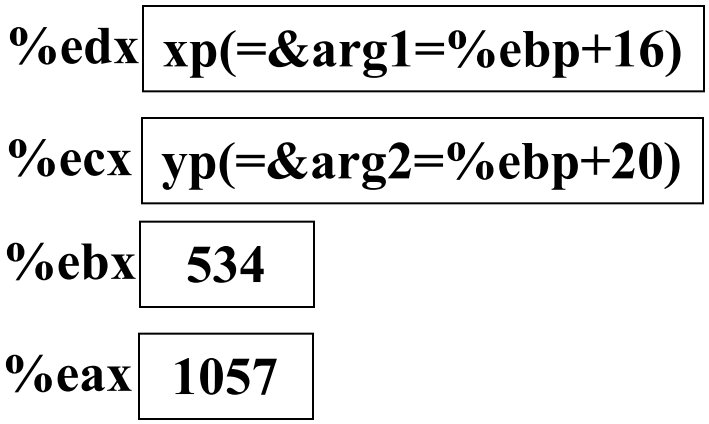
Body code in swap_add

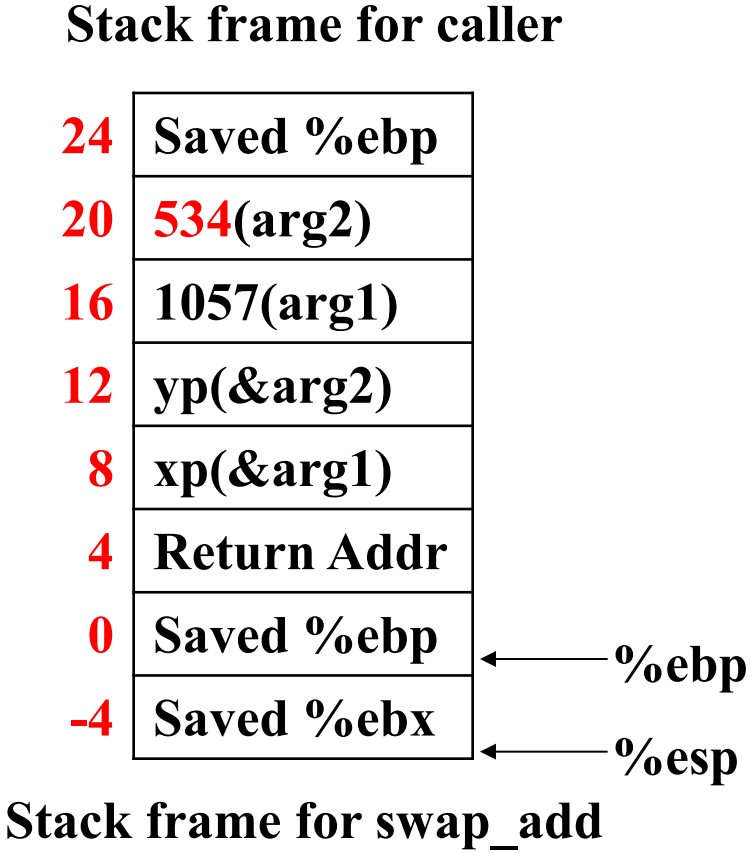
9 movl %eax, (%edx)

Store y at *xp

10 movl %ebx, (%ecx)

Store x at *yp



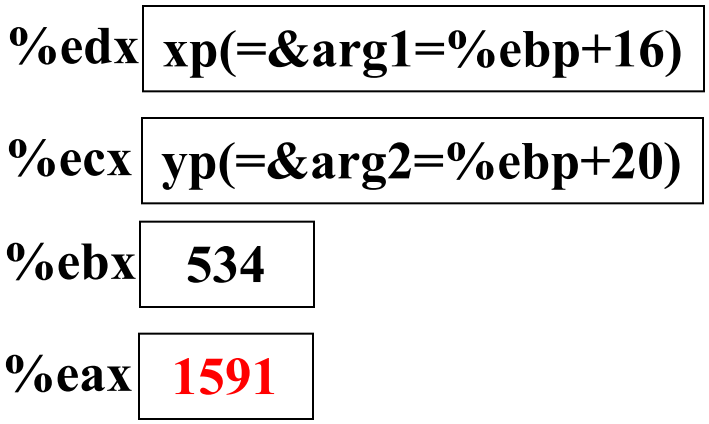


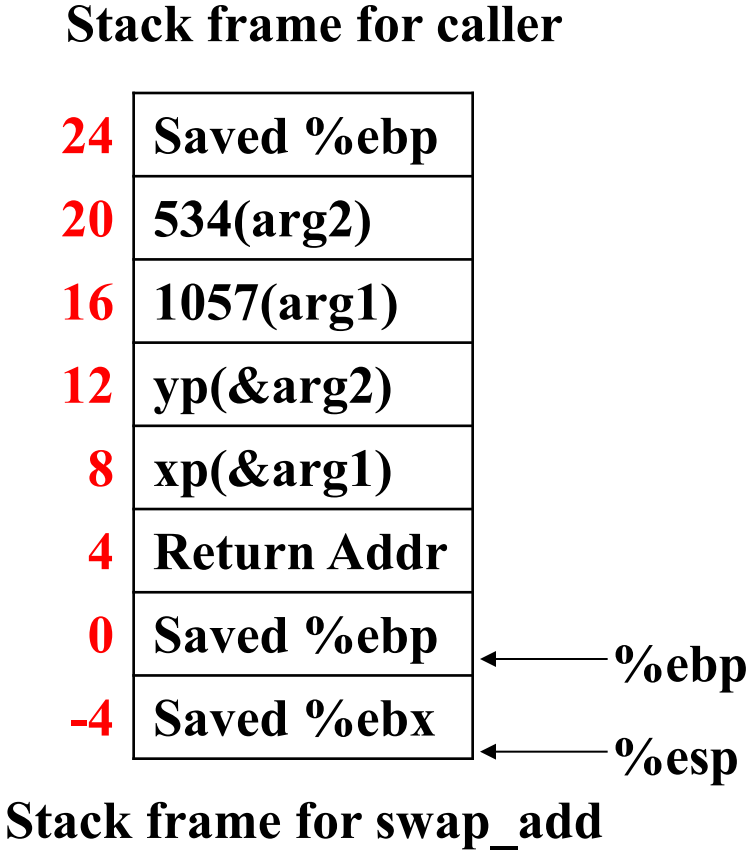
Body code in swap_add

```
9   movl %eax, (%edx)
```
*Store y at *xp*

```
10  movl %ebx, (%ecx)
```
*Store x at *yp*

```
11  addl %ebx,%eax
```
Set return value = x+y

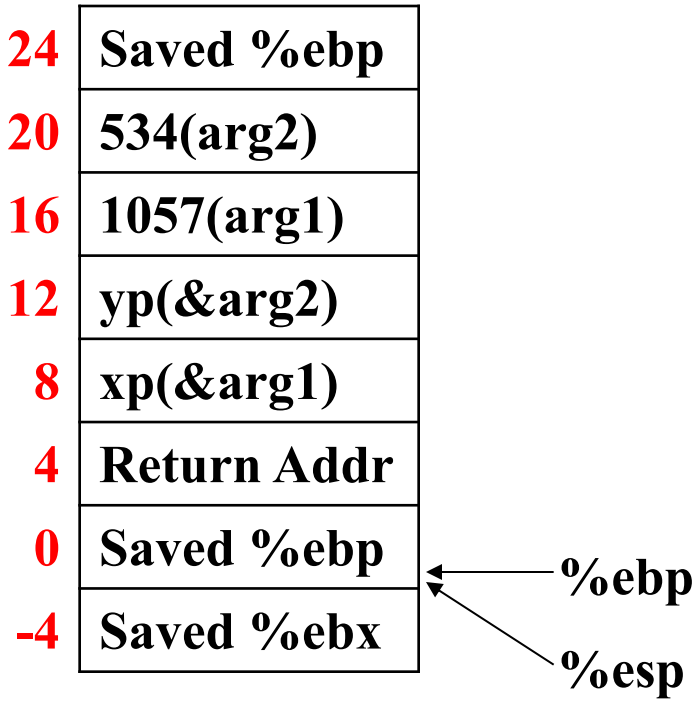




Finishing code in swap_add

12 popl %ebx *Restore* %ebx

Stack frame for caller



Stack frame for swap_add

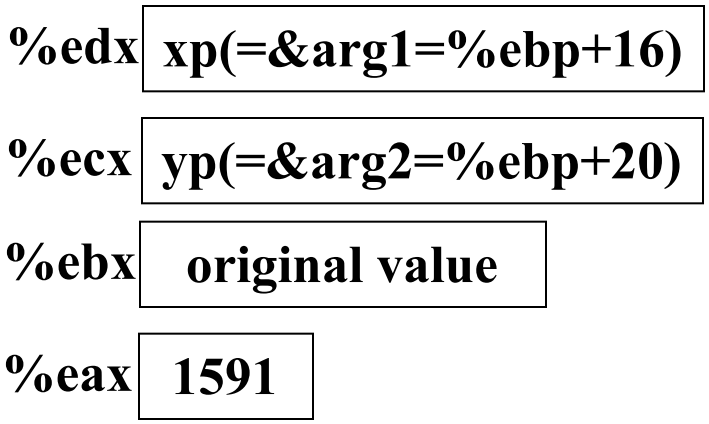
%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

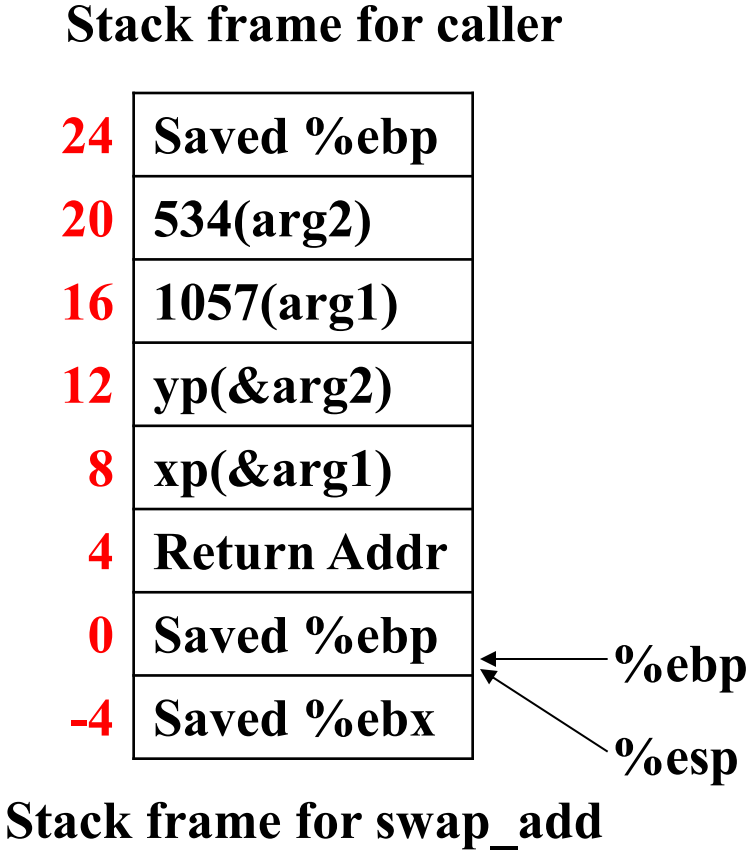
Finishing code in swap_add

12 popl %ebx *Restore* %ebx

13 movl %ebp, %esp

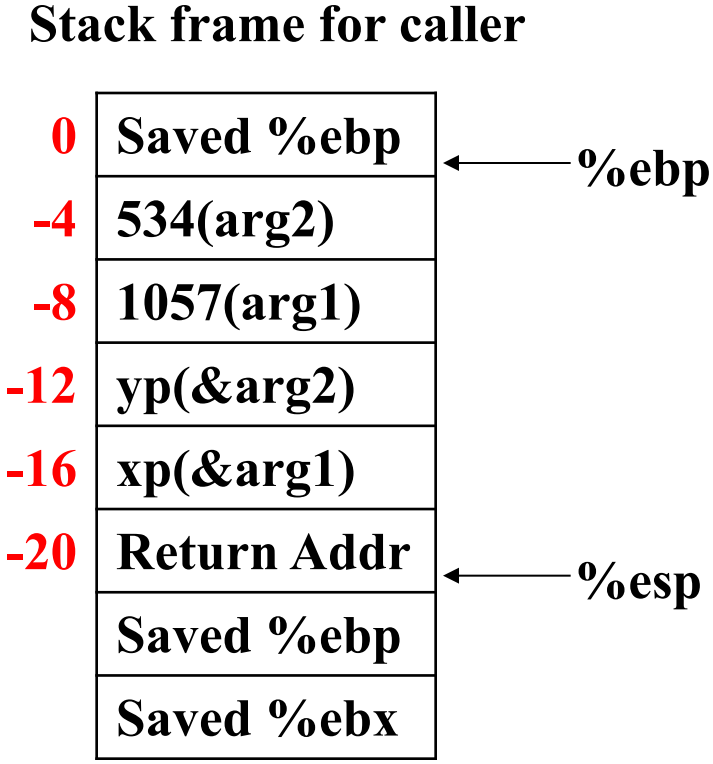
Restore %esp

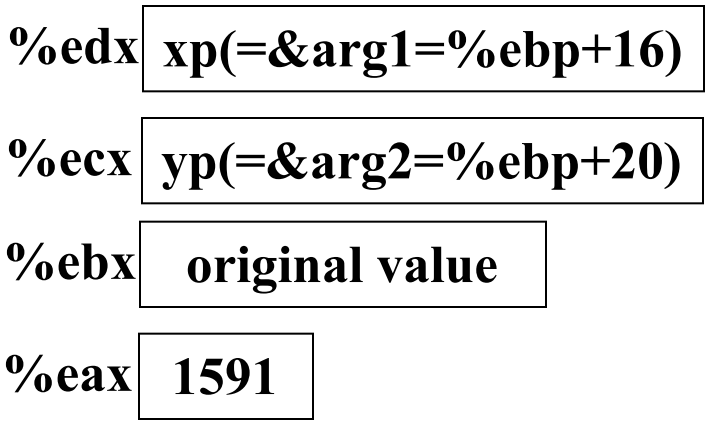




Finishing code in swap_add

```
12  popl %ebx       Restore %ebx
13  movl %ebp, %esp
                    Restore %esp
14  popl %ebp       Restore %ebp
```





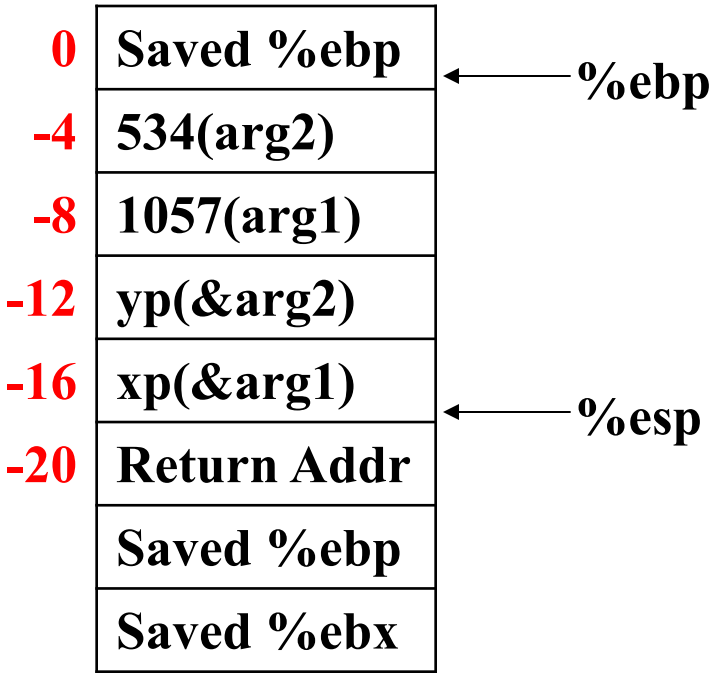
Finishing code in swap_add

```
12  popl %ebx       Restore %ebx
13  movl %ebp, %esp
                    Restore %esp
14  popl %ebp       Restore %ebp
15  ret             Return to caller
```

- **Call by value**

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Stack frame for caller



同学提问

- 关于函数调用过程中的被调者或调用者寄存器保存规则
  - 在栈中保存寄存器的时候只会存数，不存%rbx这种文本（或者键值对映射）
    - pushl %ebp (将ebp的值压栈)
  - 编译器自己知道栈中每个格子是存的什么
  - 后面再pop回去
    - popl %ebp

Buffer Overflow

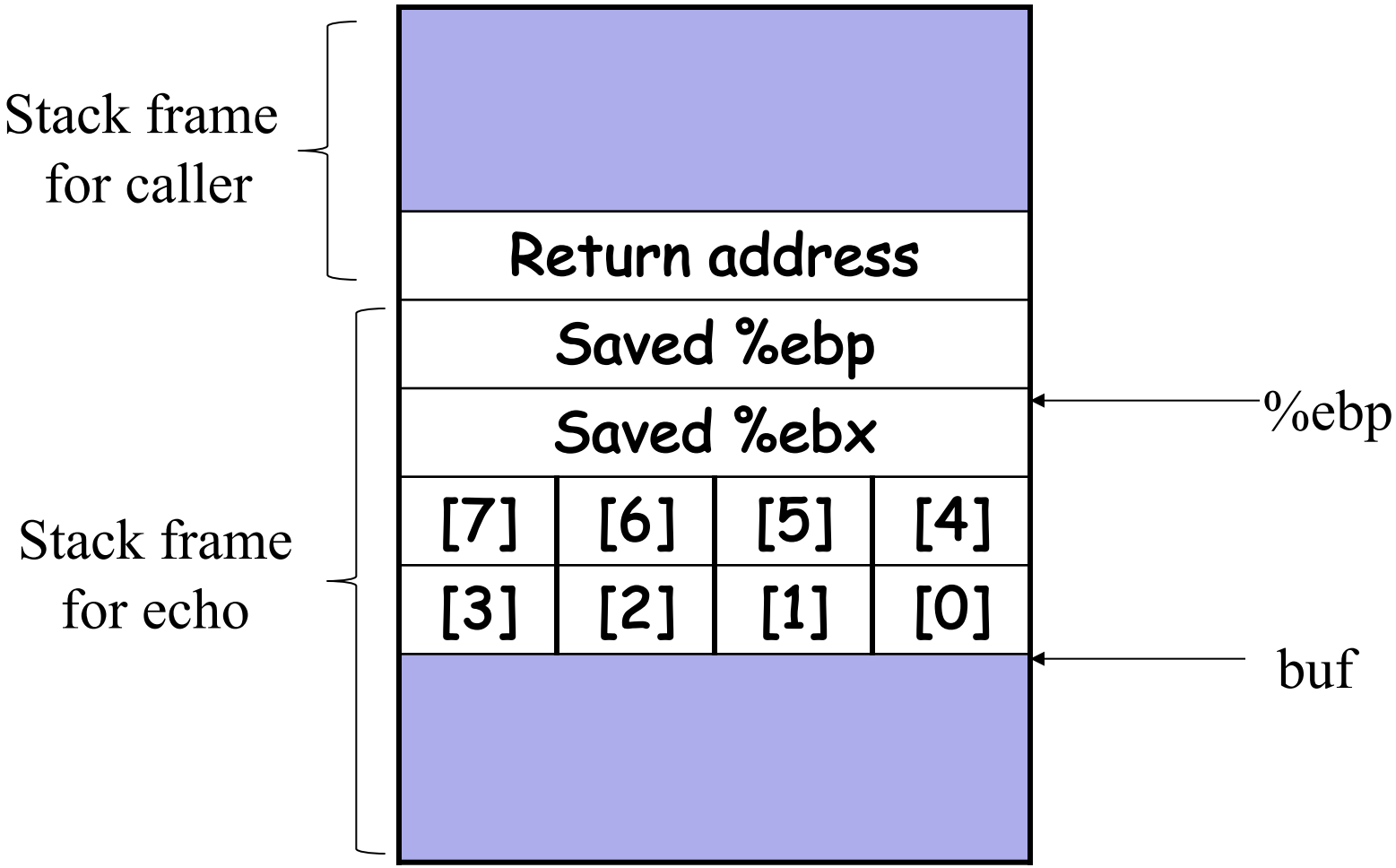
Out-of-Bounds Memory References

```
1  /* Implementation of library function gets() */
2  char *gets(char *s)
3  {
4          int c;
5          char *dest = s;
6          int got_char = 0 ; /Has at least one character been read? */
7          while ((c = getchar()) != '\n' && c != EOF) {
8                  *dest++ = c;        /* No bounds checking */
9                  gotchar = 1;
10         }
11         *dest++ = '\0';                /* Terminate String */
12         if (c == EOF && !gotchar)
13                 return NULL;   /* End of file or error */
14         return s;
15 }
```

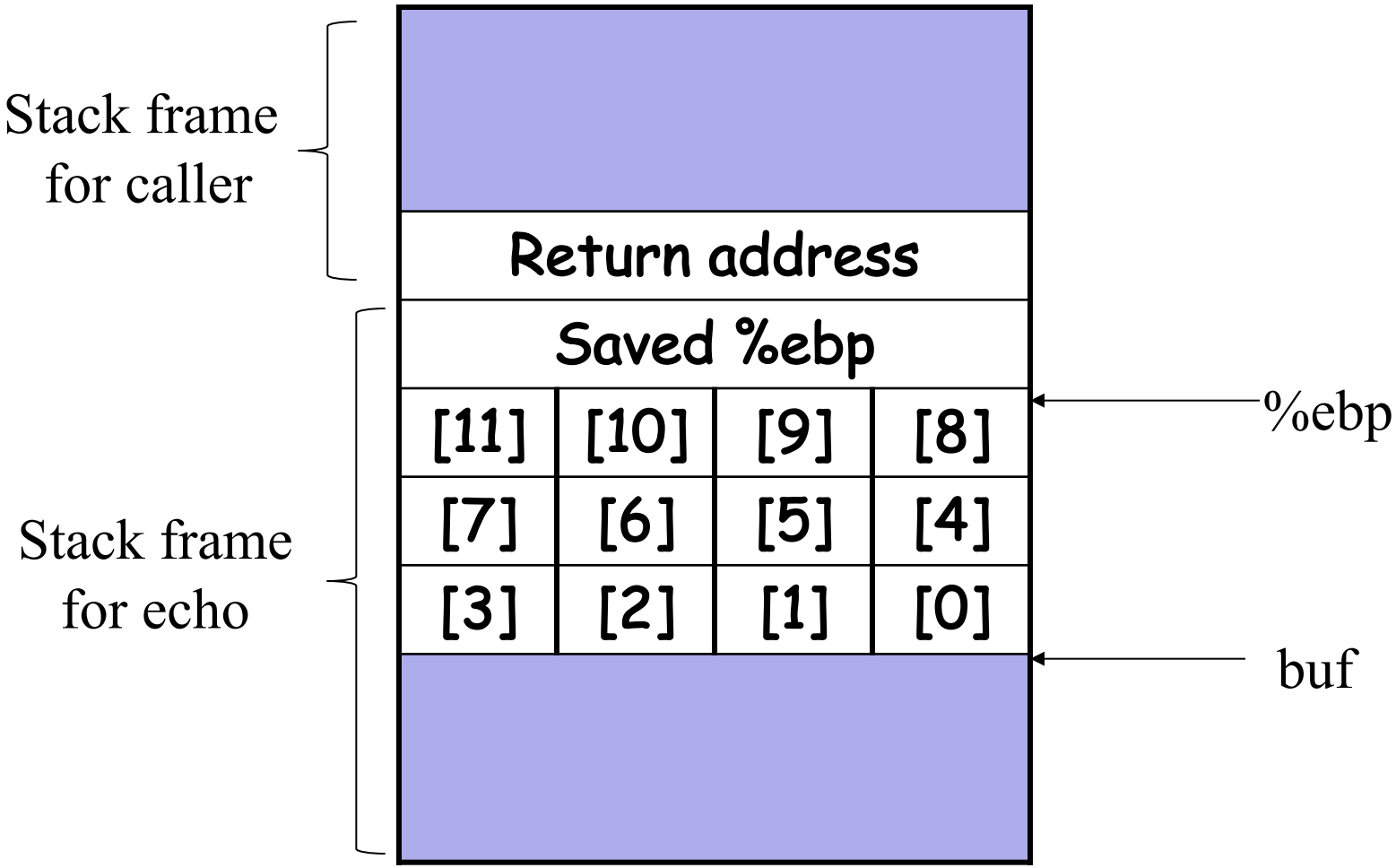
Out-of-Bounds Memory References

```
14 /* Read input line and write it back */
15 void echo()
16 {
17     char buf[8];          /* Way too small ! */
18     gets(buf);
19     puts(buf);
20 }
```

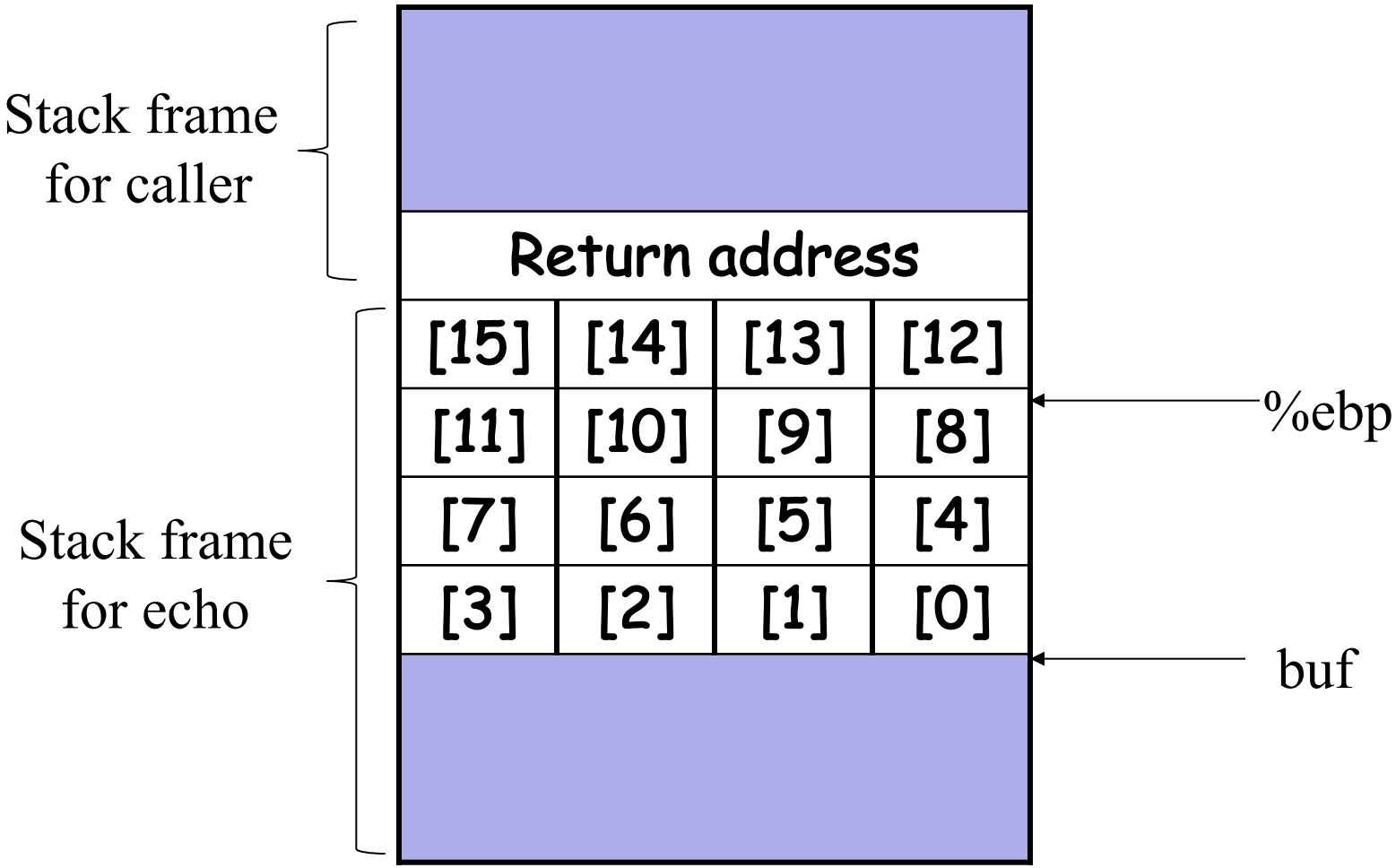
Out-of-Bounds Memory References



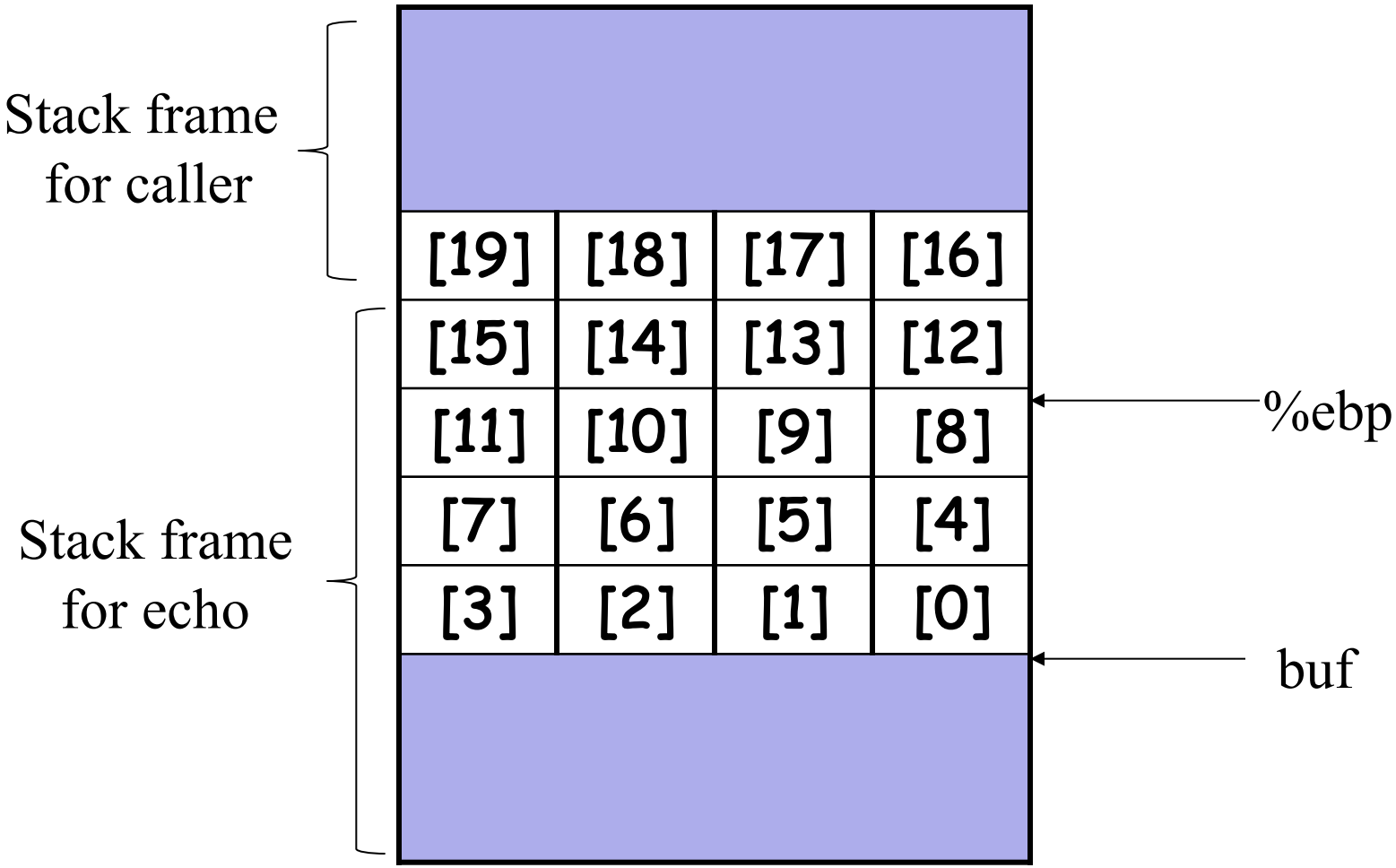
Out-of-Bounds Memory References



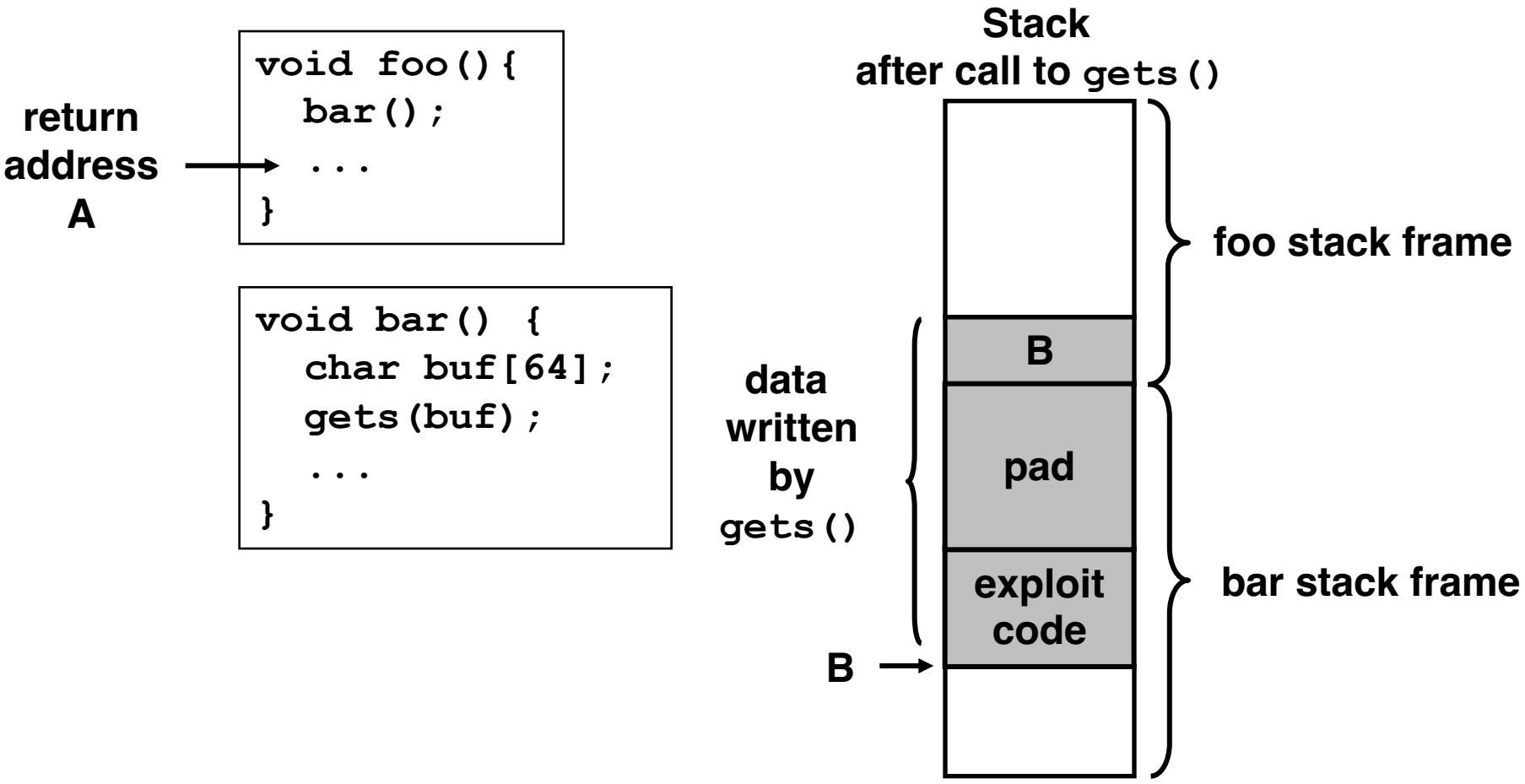
Out-of-Bounds Memory References



Out-of-Bounds Memory References



Malicious Use of Buffer Overflow



The Famous Internet Worm of November 1988

- To gain access to many of the computers across the Internet
  - 4 different ways
  - One was a buffer overflow attack on the fingerd
- Hundreds of machines were effectively paralyzed
- The author of the worm was caught and prosecuted. He was sentenced to
  - 3 years probation
  - 400 hours of community service
  - and a $10,500 fine

Morris Worm

- Robert Tappan Morris
  - born November 8, 1965

他的父亲罗伯特·莫里斯（Robert Morris）为贝尔实验室计算机安全专家

他16岁上初中时，发现UNIX系统漏洞

1983年考入哈佛大学，拿到了文学学士（A.B.）学位

1988年成为康奈尔大学研究生

1988年散布了“莫里斯蠕虫”，造成约6000个系统瘫痪，给这些用户总共带来约200万到6000万美元的损失。

现为麻省理工学院教授 (2006 tenure, 2015 ACM Fellow)

https://www.bilibili.com/video/BV1qk4y197bB?from=search&seid=7720468517381990376

Stack Randomization

```
1  int main() {
2        int local;
3        printf("local at %p\n", &local);
4        return 0;
5  }
```

- Running the code 10,000 times on a Linux (maybe 2.6.16) machine in 32-bit mode
- the addresses ranged from
  - 0xff7fa7e0 to 0xffffd7e0
  - A range of around 2^{23}
- 无法确定目标代码的绝对地址

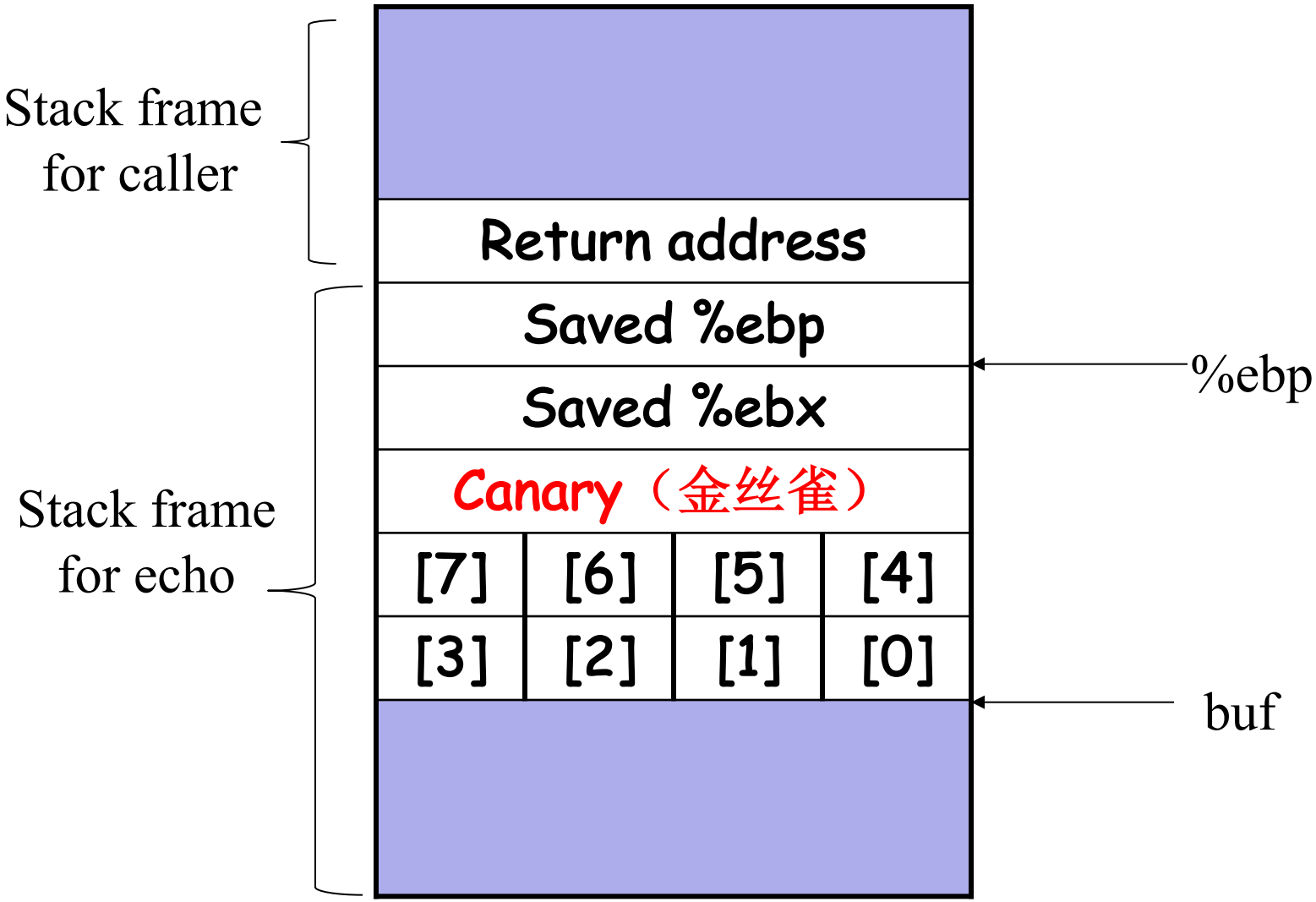
Stack Randomization

- Running in 64-bit mode on the newer machine
- The addresses ranged from
  - 0x7fff00241914 to 0x7fffff98664
  - A range of nearly 2^{32}
- *Address-space layout randomization* (ASLR)
  - each time a program is run
  - different parts of the program are loaded into different regions of memory
    - code, data, heap data, library code, stack

Stack Randomization

- Nop sled
  - a program "slides" through a long sequence of "nop"
- Nop
  - no operation instruction
- Include a "nop sled" before the actual exploit code
  - If insert 256-byte nop sled
  - Need to guess 2^{15} starting addresses (no too much) for 32-bit machine （23-8=15）
  - Still have too many 2^{24} for 64-bit machine

Stack Corruption Detection（栈破坏检测）



Stack Corruption Detection

```
1 echo:
2       pushl   %ebp
3       movl    %esp, %ebp
4       pushl   %ebx
5       subl    $20, %esp
6       movl    %gs:20, %eax        Retrieve canary
7       movl    %eax, -8(%ebp)      Store on stack
8       xorl    %eax, %eax          Zero out register
9       leal    -16(%ebp), %ebx     Compute buf as %ebp-16
10      movl    %ebx, (%esp)        Store buf at top of stack
11      call    gets                Call gets
12      movl    %ebx, (%esp)        Store buf at top of stack
13      call    puts                Call puts
```

Stack Corruption Detection

```
14        movl    -8(%ebp), %eax        Retrieve canary
15        xorl    %gs:20, %eax          Compare to stored value
16        je      .L19                  If =, goto ok
17        call    __stack_chk_fail      Stack corrupted!
18 .L19: ok:
19        addl    $20, %esp             Normal return ...
20        popl    %ebx
21        popl    %ebp
22        ret
```

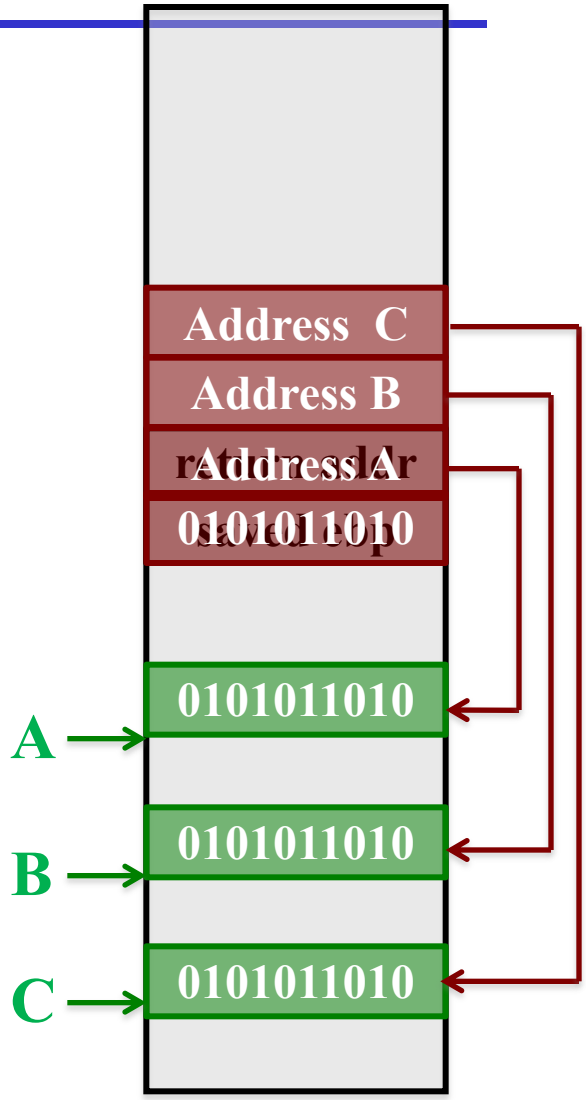
- %gs:20
  - Segmented addressing which appeared in 80286 and seldom used today
  - It is marked as read only

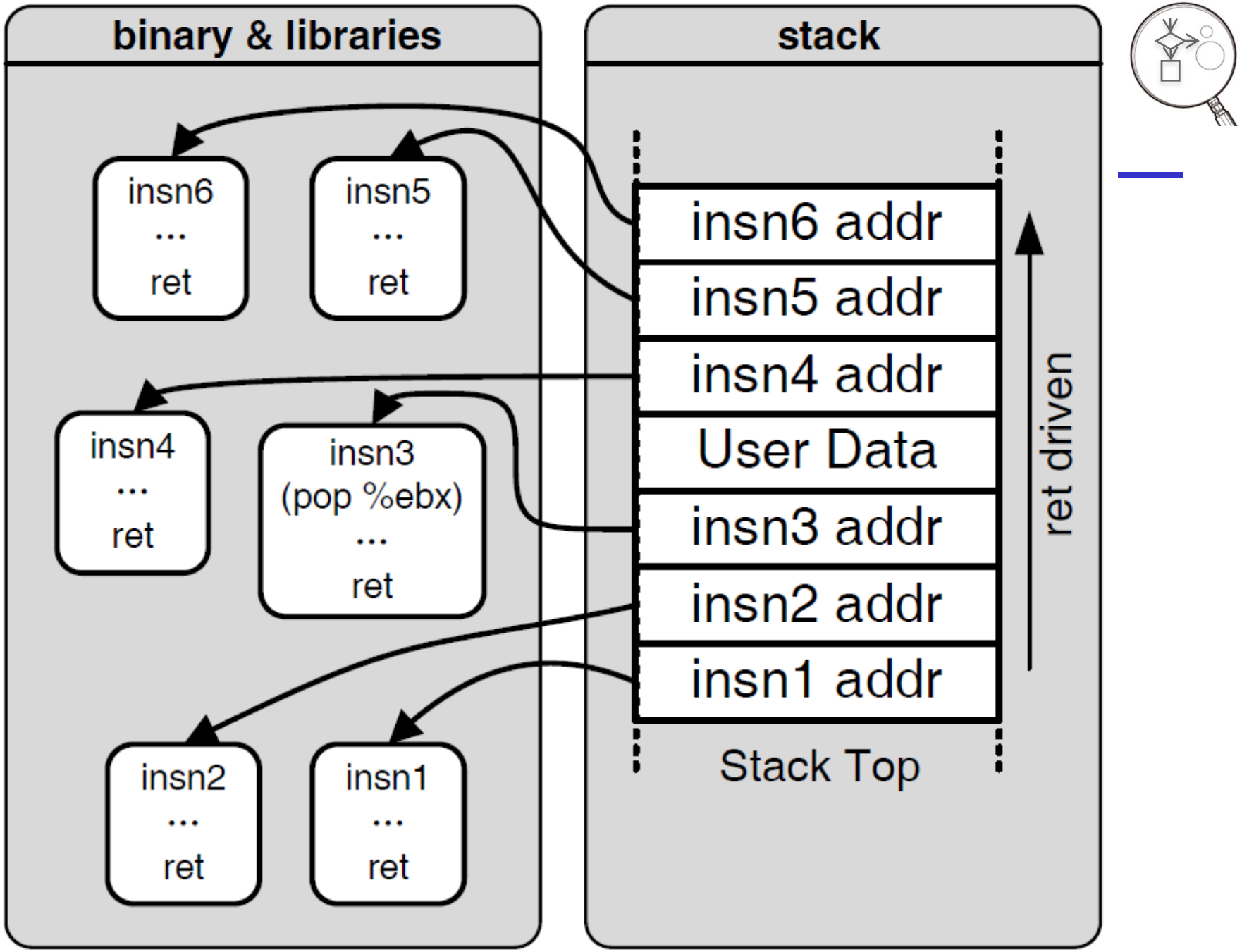
Limiting Executable Code Regions

- Page
  - 4k bytes
  - As a protected unit by OS
  - Should be marked as "readable", "writable" and "executable"
    - 3 bits are required
- Originally Intel merged the "readable" and "executable" into one
  - The exploit code in the stack can be executed
- AMD introduced "NX" in X86-64
  - Now there 3 bits

Code Reuse Attack

- Return-oriented Programming
  - Find code gadgets in existed code base (e.g. libc)
  - Push address of gadgets on the stack
  - Leverage 'ret' to connect code gadgets
  - No code injection
- Solutions
  - Return-less kernels
  - Heuristic means




binary & libraries
stack
insn6
...
ret
insn5
...
ret
insn4
...
ret
insn3
(pop %ebx)
...
ret
insn2
...
ret
insn1
...
ret
insn6 addr
insn5 addr
insn4 addr
User Data
insn3 addr
insn2 addr
insn1 addr
Stack Top
ret driven

主观题 10分

设置

```
1char *getline() {
2  char buf[8];
3  char *result;
4  gets(buf);
5  result = malloc(strlen(buf));
6  strcpy(result, buf);
7  return result;
8 }
```

Getline返回地址0x8048643

%ebp=0xbffffc94, %ebx=0x1,

%esi=0x2, %edi=0x3, 输入字符串

"012345678901234567890123"

作答

```
1 080485c0 <getline>:
2  push %ebp
3  mov %esp, %ebp
4  sub $0x28, %esp
5  mov %ebx, -0xc(%ebp)
6  mov %esi, -0x8(%ebp)
7mov %edi, -0x4(%ebp)
8lea -0x14(%ebp), %esi
9    mov %esi, (%esp)
10  call 804857d <gets>
```

1)画出Line7后栈图；

2)修改你的图，Line10后影响；

3)程序试图返回到什么地址？

4)Getline返回时，哪些寄存器的值被破坏了？

5)除了缓存区溢出，getline的代码还有哪两个错误？

上次讲到这里

Heterogeneous Data Structures & Alignment

Outline

- Struct
- Union
- Alignment

Structures

- Group objects into a single object

```
struct rect {
   int llx;     /* X coordinate of lower-left corner */
   int lly;     /* Y coordinate of lower-left corner */
   int color;   /* Coding of color */
   int width;   /* Width (in pixels) */
   int height; /* Height (in pixels) */
};
```

Structure

- Each object is referenced by name

```
struct rect r;

r.llx = r.lly = 0;
r.color = 0xFF00FF;
r.width = 10;
r.height = 20;
```

Structure

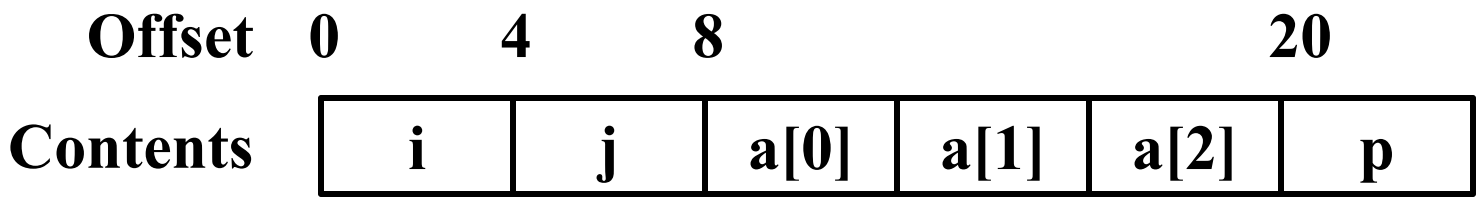
```
int area (struct rect *rp)
{
  return (*rp).width * (*rp).height;
}

void rotate_left (struct rect *rp)
{
  /* Exchange width and height */
  int t = rp->height;
  rp->height = rp->width;
  rp->width = t;
}
```

Structures

- Memory layout
  - All the components are stored in a contiguous region of memory
  - A pointer to a structure is the address of its first byte （最低/最小地址）

Structure

```
struct rec {
  int i;
  int j;
  int a[3];
  int *p;
} *r;
```



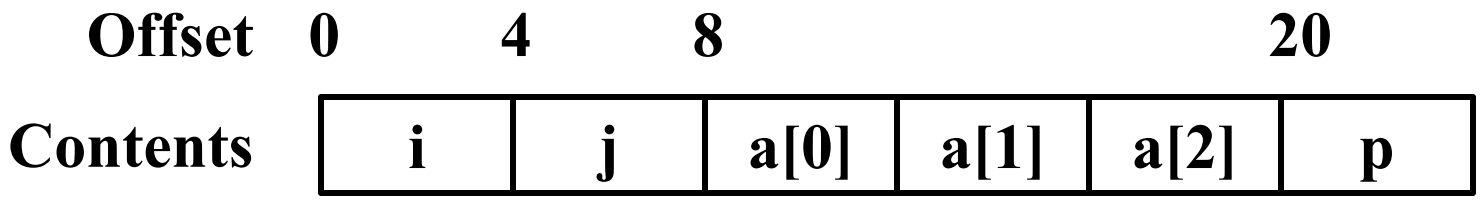
Structure

- References to structure elements
  - Using offsets as displacements

`r->j = r->i` (Copy element r->i to element r->j)

`r is in register %edx.`

```
1 movl     (%edx), %eax      Get r->i
2 movl     %eax,   4(%edx)   Store in r->j
```

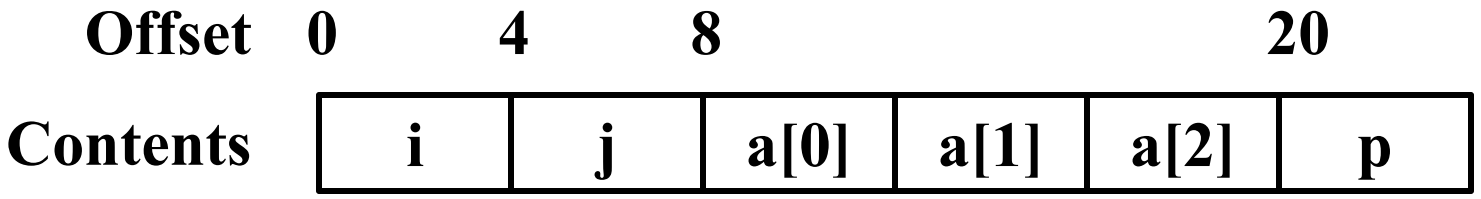


Structure

```
&(r->a[i])
```

r in %eax, i in %edx:

```
1 leal   8(%eax,%edx,4),%ecx  Generate &r->a[i]
```

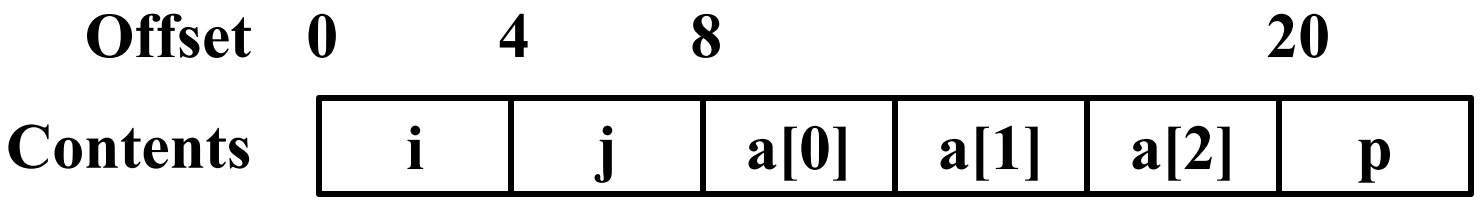


Structure

```
r->p = &r->a[r->i + r->j];
```

r in register %edx:

```
1 movl 4(%edx), %eax          Get r->j
2 addl (%edx),  %eax          Add r->i
3 leal 8(%edx,%eax,4),%eax    Compute &r->a[r->i + r->j]
4 movl %eax, 20(%edx)         Store in r->p
```



Unions

- 两个/多个部分，同一段内存有不同的解读方式
- A single object can be referenced by using different data types
- The syntax of a union declaration is identical to that for structures, but its semantics are very different
- Rather than having the different fields reference different blocks of memory, they all reference the same block

Unions

```
struct S3 {
   char c;
   int i[2];
   double v;
};
union U3 {
   char c;
   int i[2];
   double v;
};
```

The offsets of the fields, as well as the total size of data types S3 and U3, are:

Type	c	i	v	size
S3	0	4	12	20
U3	0	0	0	8

Unions

```
struct NODE {
  struct NODE *left;
  struct NODE *right;
  double data;
};

union NODE {
  struct {
     union NODE *left;
     union NODE *right;
  } internal;
  double data;
};
```

Unions

```
struct NODE {
  int is_leaf;
  union {
     struct {
          struct NODE *left;
          struct NODE *right;
     } internal;
     double data;
  } info;
};
```

Unions

```
1 unsigned float2bit(float f)
2 {
3     union {
4           float f;
5           unsigned u;
6     } temp;
7     temp.f = f;
8     return temp.u;
9 }
```

```
1 movl      8(%ebp),    %eax
```

Unions

```
1 unsigned copy (unsigned u)
2 {
3     return u;
4 }
```

```
1 movl      8(%ebp),    %eax
```

Alignment

- Alignment restrictions
  - The address for some type of object must be a multiple of some value k (typically 2, 4, or 8)
  - Simplify the hardware design of the interface between the processor and the memory system
  - 是否对齐一般不影响正确性
  - 但影响效率（可能两次访问内存，例如1个double）

Alignment

- In IA32
  - hardware will work correctly regardless of the alignment of data
  - Aligned data can improve memory system performance

Alignment

- Linux alignment restriction
  - 1-byte data types are able to have any address
  - 2-byte data types must have an address that is multiple of 2
  - Any larger data types must have an address that is multiple of 4

Alignment

- Windows
  - 长度为K的数据类型，开头地址一定是K的整数倍
  - 空间换时间，效率较高

Alignment

- malloc()
  - Returns a generic pointer that is void *
  - Its alignment requirement is 4
- Linux memalign family
  - 如void *memalign (size_t alignment, size_t size);
  - 分配size大小内存，alignment的整数倍作为首地址，返回这个首地址，其中alignment必须是2的整数倍
  - http://man7.org/linux/man-pages/man3/posix_memalign.3.html

如何实现对齐分配和释放内存？

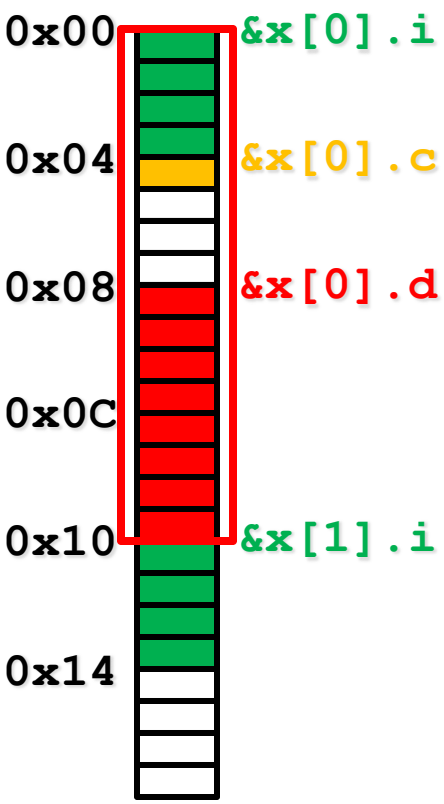
```
void* aligned_malloc(size_t required_bytes, size_t alignment) {
    void* p1; // original block
    void** p2; // aligned block
    int offset = alignment - 1 + sizeof(void*);  //对齐最远偏移alignment-1
    if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
       return NULL;
       }
    p2 = (void**)(((size_t)(p1) + offset) & ~(alignment - 1));
    p2[-1] = p1;
    return p2;
}
void aligned_free(void *p) {
    free(((void**)p)[-1]);
}
```

Alignment

- Structure data type
  - may need to insert gaps in the field allocation
  - may need to add padding to the end of the structure
  - Structure的对齐规则=其最大元素的对齐规则

Simple Example

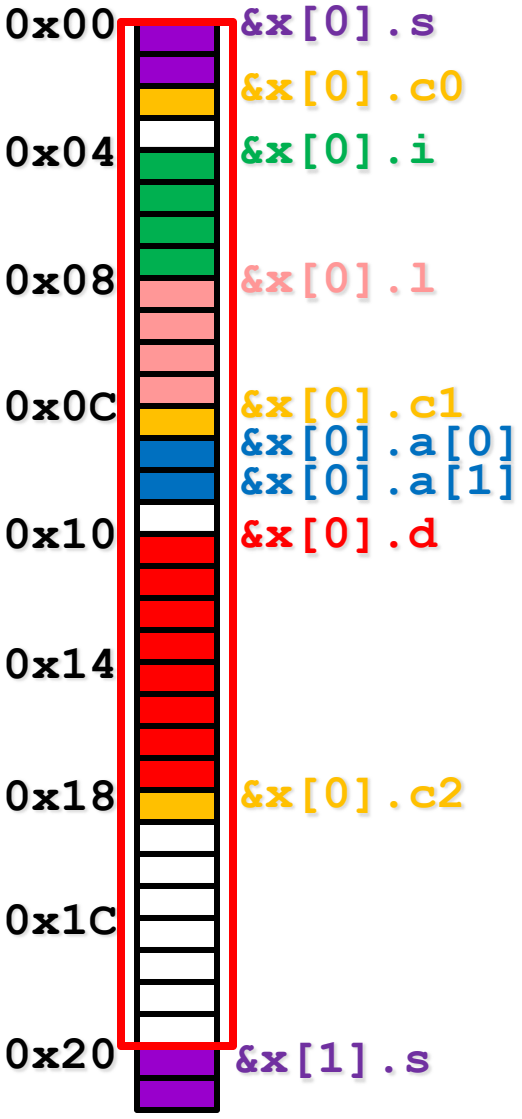
```
struct xxx {
   int i;
   char c;
   double d;
};

struct xxx x[2];
```



Struct整体对齐规则
由其中最大的元素决定
（此例为8字节）

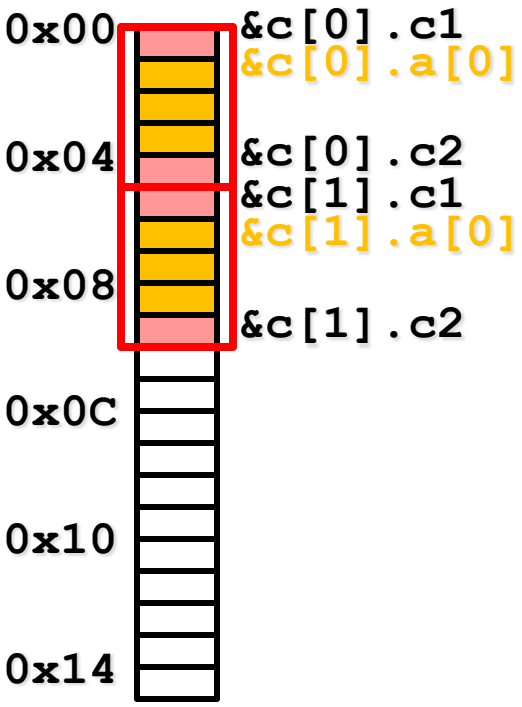
Complex Example

```
struct xxx {
     short s;
     char c0;
     int i;
     long l;
     char c1;
     char a[2];
     double d;
     char c2;
};

struct xxx x[2];
```



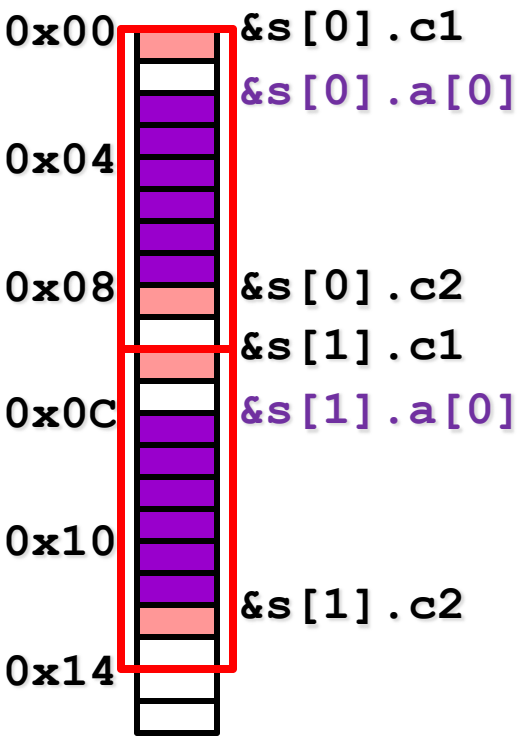
Array

```
struct ccc {
     char c1;
     char a[3];
     char c2;
};

struct ccc c[2];
```



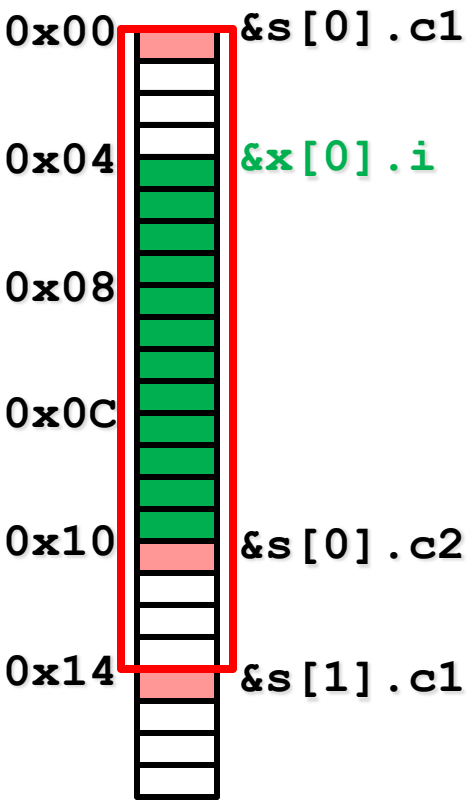
Array

```
struct ccc {
     char c1;
     short a[3];
     char c2;
};

struct sss s[2];
```



Array

```
struct iii {
     char c1;
     int a[3];
     char c2;
};

struct iii i[2];
```



主观题 10分

设置

```
typedef union {
  struct{
    short v;
    short d;
    int s;
  } t1;
  struct{
    int a[2];
    char *p;
  } t2;
}u_type;
```

```
// 32位机环境下
//up@eax, dest@edx
void get(u_type *up, TYPE *dest) {
    *dest = EXPR;
}
```

EXPR分为为以下值时，求TYPE和get函数的汇编代码:

1. up->t1.s
2. up->t1.v
3. &up->t1.d
4. up->t2.a
5. up->t2.a[up->t1.s]
6. *up->t2.p

作答

练习答案

```
typedef union {
  struct{
    short v;
    short d;
    int s;
  } t1;
  struct{
    int a[2];
    char *p;
  } t2;
}u_type;
```

```
// 32位机环境下
//up@eax, dest@edx
void get(u_type *up, TYPE *dest) {
    *dest = EXPR;
}
```

EXPR分为为以下值时，求TYPE和get函数的汇编代码:

1. up->t1.s
2. up->t1.v
3. &up->t1.d
4. up->t2.a
5. up->t2.a[up->t1.s]
6. *up->t2.p

1）int, movl 4(%eax), %eax　　movl %eax, (%edx)

2）short, movw (%eax),%ax　　movw %ax, (%edx)

3）short *, leal 2(%eax), %eax　　movl %eax, (%edx)

4）int *, movl %eax, (%edx)

5）int, movl 4(%eax), %ecx　　movl (%eax, %ecx, 4), %eax　　movl %eax, (%edx)

6）char, movb 8(%eax), %al　　movb al, (%edx)

- 下面的定义声明了一类结构，用来构建二叉树：

```
1    typedef struct ELE *tree_ptr;                    // 表示tree_ptr实际上是ELE*类型
2    struct ELE {
3        tree_ptr left;
4        tree_ptr right;
5        long     val;
6    }
```

对于如下函数原型 long trace(tree_ptr tp); GCC产生了下面的x86-64代码：

```
1 trace:                                              ; tp in %rdi
2        movl    $0, %eax
3        testq   %rdi, %rdi
4        je      .L3
5    .L5
6        movq    16(%rdi), %rax
7        movq    (%rdi), %rdi
8        testq   %rdi, %rdi
9        jne     .L5
10   .L3
11       ret     ;函数返回，返回值一般放在%rax中
```

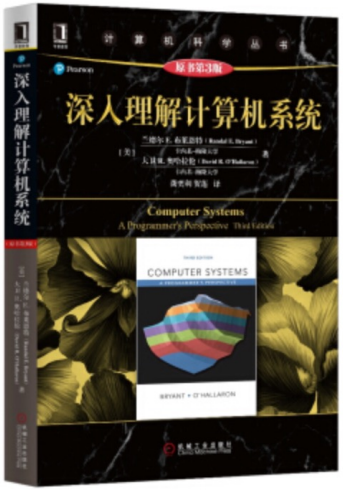
- （1）请写出该函数的最简洁的C语言版本，使用while循环；（2）用自然语言解释该函数的功能。

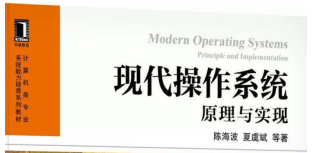
练习答案

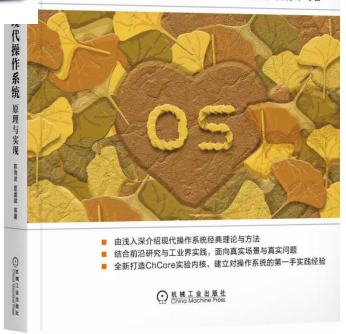
- Long ret = 0;
- While (tp) {
- 　　ret = tp->val;
- 　　tp = tp->left;
- }
- Return ret;
- 求二叉树最左子节点的val

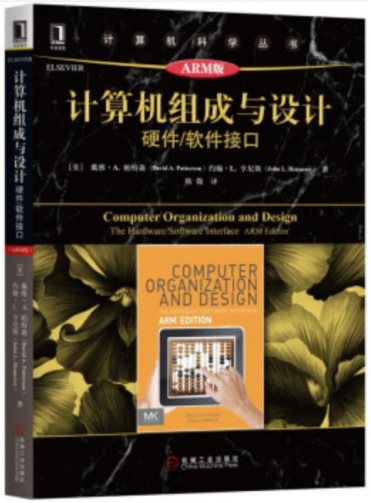
Summary

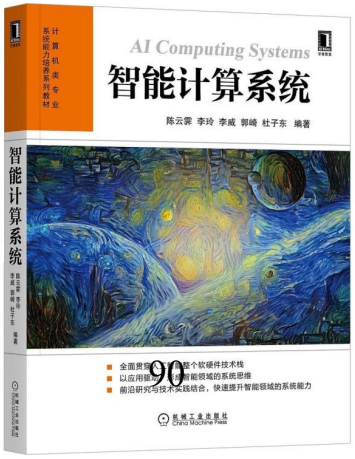
- 系统软件基础
  - 《计算机系统基础II》
- 最活跃的计算机系统技术
  - 《并行与分布式计算》
- 复杂工程
  - 《现代处理器设计》
  - 《操作系统实现》
  - 《编译原理》
  - 《数据库系统实现》
  - 《智能计算系统》







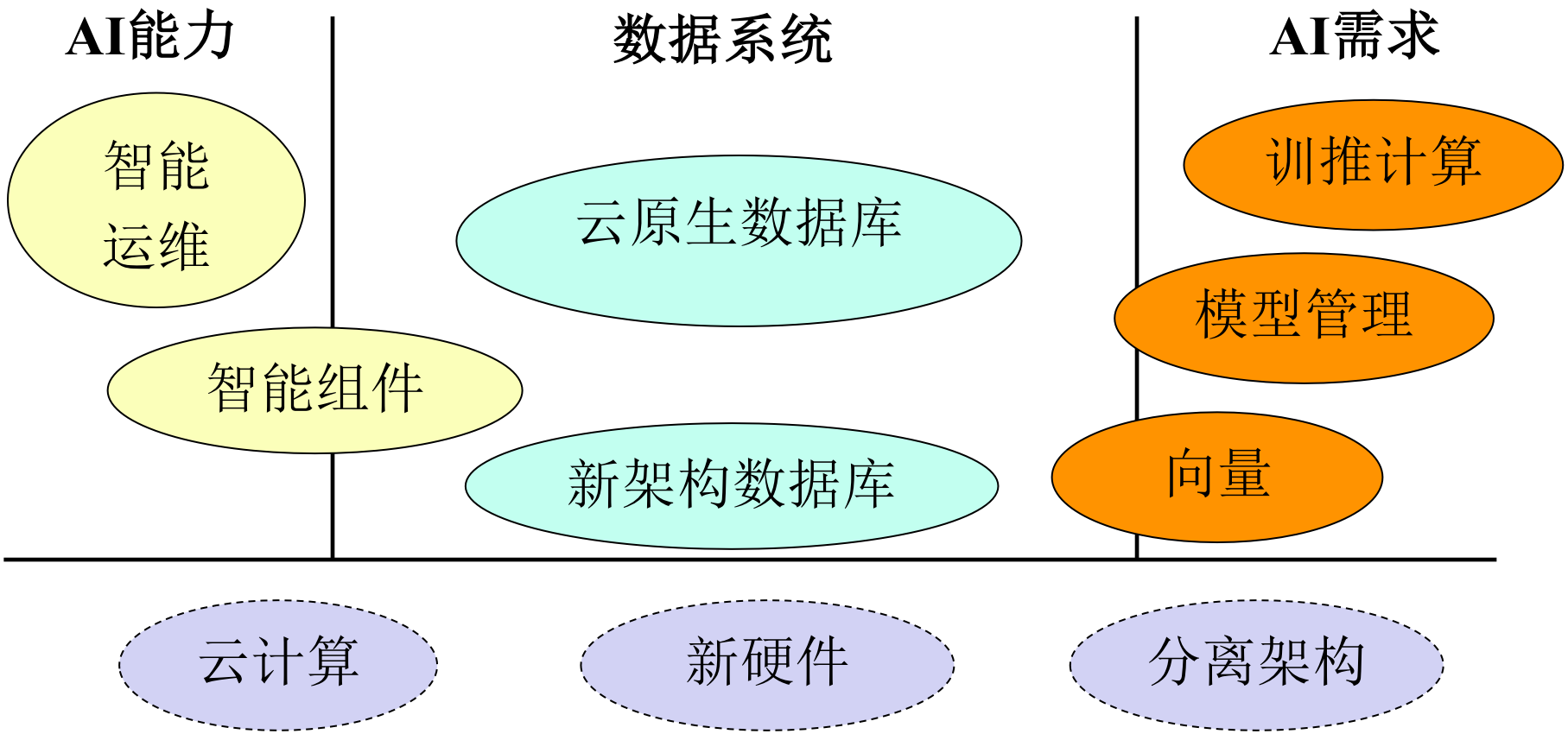




Summary

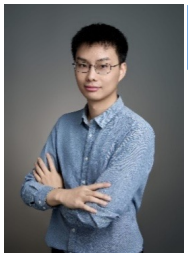
- 经典课程
  - 计算机系统基础/导论
  - 计算机组成原理/系统结构
  - 操作系统
  - 数据库系统
  - 并行计算
  - 分布式系统与云计算
  - 人工智能系统
  - 系统编程/调试/性能优化
- 技术学习资料
- 科研入门*

飞书


AI能力
数据系统
AI需求
智能运维
云原生数据库
训推计算
模型管理
智能组件
新架构数据库
向量
云计算
新硬件
分离架构

新型体系结构

CDSLab-本人



谢旻晖 (Minhui Xie)

研究兴趣：计算机系统

- 存储系统
- 机器学习系统

2015–2019	南京大学（本科） 计算机科学与技术系	南京大學 NANJING UNIVERSITY
2019–2024	清华大学（博士） 计算机科学技术系 导师：陆游游（优青） 舒继武(杰青/长江/IEEE Fellow)	清華大學 Tsinghua University
2024–至今	中国人民大学 计算机科学技术系 讲师/助理教授 柴云鹏院长团队	中國人民大學 RENMIN UNIVERSITY OF CHINA

论文发表

- 发表系统领域CCF-A类会议18篇
- 其中6篇第一作者，3篇为通讯作者

获奖情况

- 2025，北京市青年人才托举工程
- 2025，ACM中国优秀博士学位论文奖
  （计算机全学科仅2位）
- 2025，USENIX ATC (CCF-A) best storage-related paper
- 2025，ACM SIGOPS China (ChinaSys) 优博奖
- 2025，微软铸星学者
- 2024，华为天才少年（顶薪offer）

项目承担

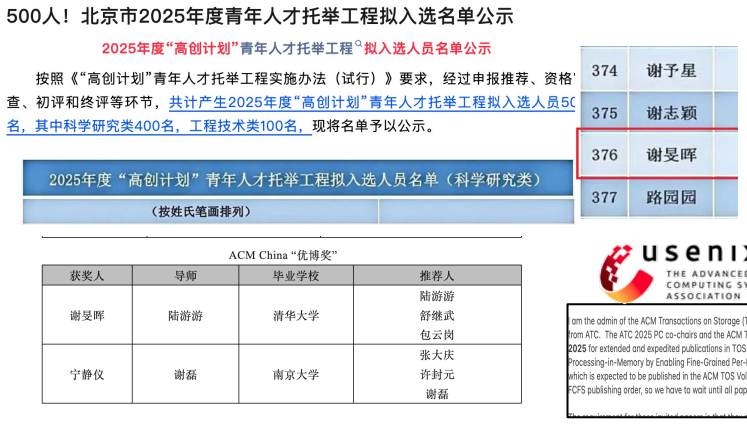
以独立PI身份承担

- 国家自然科学基金青基/北京市自然科学基金
- 快手/腾讯/蚂蚁等企业项目

欢迎联系xieminhui@ruc.edu.cn，进组实习

研究基础 获奖情况

- 2025，北京市青年人才托举工程
- 2025，ACM中国优秀博士学位论文奖（计算机全学科仅2位）
- 2025，USENIX ATC (CCF-A) best storage-related paper
- 2025，ACM SIGOPS China (ChinaSys) 优博奖（计算机系统方向仅3人，顺位第一）
- 2025，微软铸星学者
- 2024，华为天才少年（顶薪offer）
- 2023，阿里云-CCF存储专委优秀论文；计研发封面亮点论文/年度最具影响力论文(全学科仅10篇)


500人！北京市2025年度青年人才托举工程拟入选名单公示

2025年度“高创计划”青年人才托举工程拟入选人员名单公示

按照《“高创计划”青年人才托举工程实施办法（试行）》要求，经过申报推荐、资格
查、初评和终评等环节，共计产生2025年度“高创计划”青年人才托举工程拟入选人员50
名，其中科学研究类400名，工程技术类100名，现将名单予以公示。

2025年度“高创计划”青年人才托举工程拟入选人员名单（科学研究类）

（按姓氏笔画排列）

374	谢予星
375	谢志颖
376	谢旻晖
377	路园园

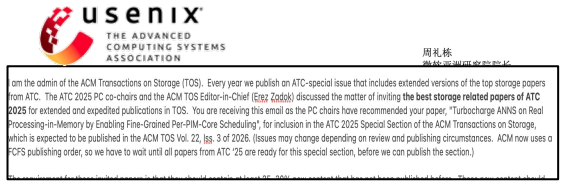
ACM China “优博奖”

获奖人	导师	毕业学校	推荐人
谢旻晖	陆游游	清华大学	陆游游 舒继武 包云岗
宁静仪	谢磊	南京大学	张大庆 许封元 谢磊

署并遵守微软访问学者协议的条款和条件（按微软（中国）有限公司提供的格式）。如有任何问题，敬请告知。

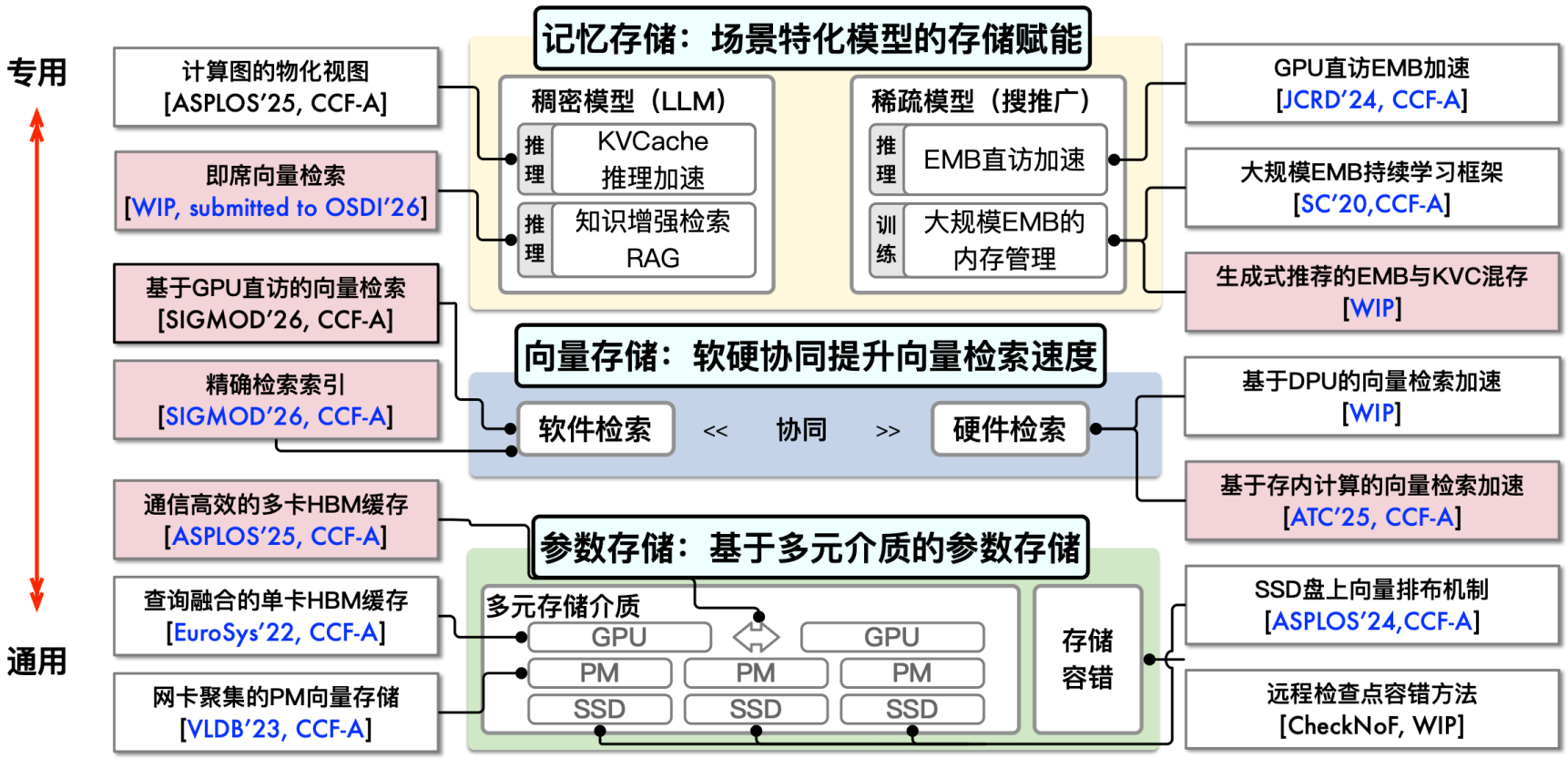
此致，

敬礼



欢迎联系xieminhui@ruc.edu.cn，进组实习

研究基础：长期围绕AI相关的存储开展研究



*蓝色为主要作者（一作/通讯）；黑色为贡献较大的合作论文（二作论文）；粉底为24年8月来人大后的工作

欢迎联系xieminhui@ruc.edu.cn，进组实习

CDSLab

云计算与大数据系统实验室
CLOUD AND BIG DATA SYSTEM LAB

AI能力

数据系统

AI需求

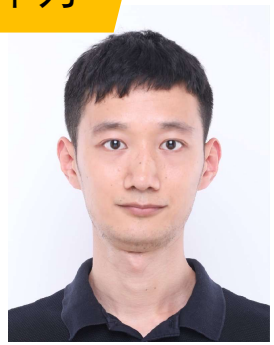
腾讯

智能化
智能向
数据库

柴云鹏

移动

华为



卞昊穹
云原生DB

张延松
新硬件DB

金仓

谢旻晖
AI系统

快手

王晶
AI推理加速

公安部

鄂金龙
分布式AI训推
云边端协同计算

字节

阿里云

新型体系结构

CDSLab eTrip难题

- https://cheese.ruc.edu.cn/spaces/52/tasks