Buffer overflow analysis

Claire Le Goues

15-819O: Program Analysis

Ant and grasshopper

- (Historical comment on the title.)
- What is the particular feature of points-to sets that the paper targets?
- What are the two mechanisms they use to do so?
  - Tradeoffs between them?
- What do you think?

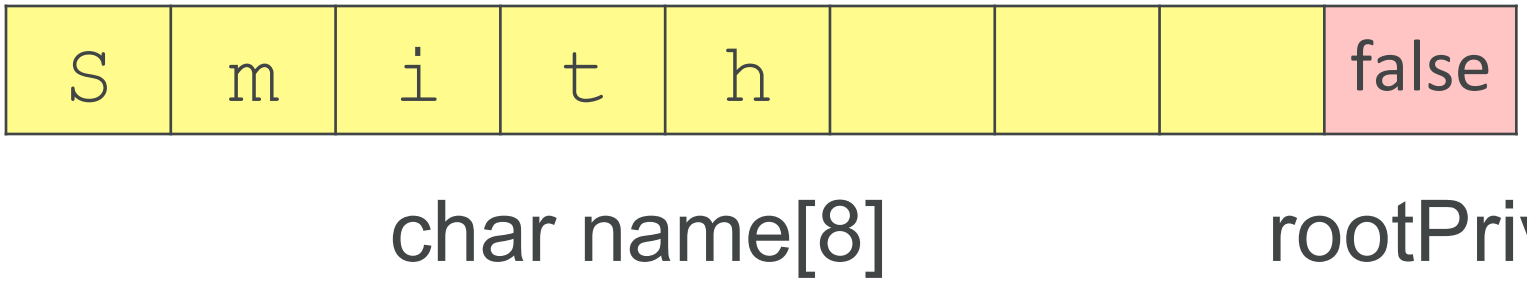
And why are we spending a lecture on them?

WHAT IS A BUFFER OVERFLOW?

Basic Buffer Overflow

```
boolean rootPriv = false;
char name[8];
cin >> name;
```

- When the program reads the name "Smith"

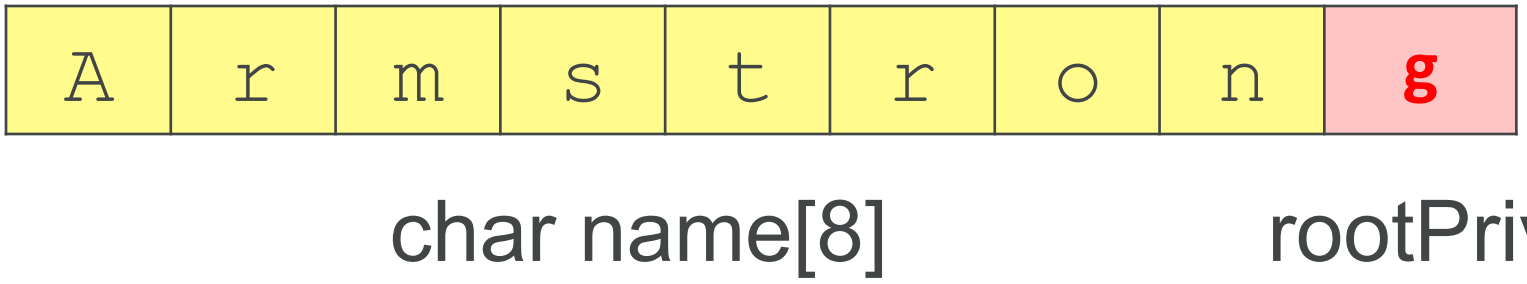


From "teaching buffer overflow"

Basic Buffer Overflow

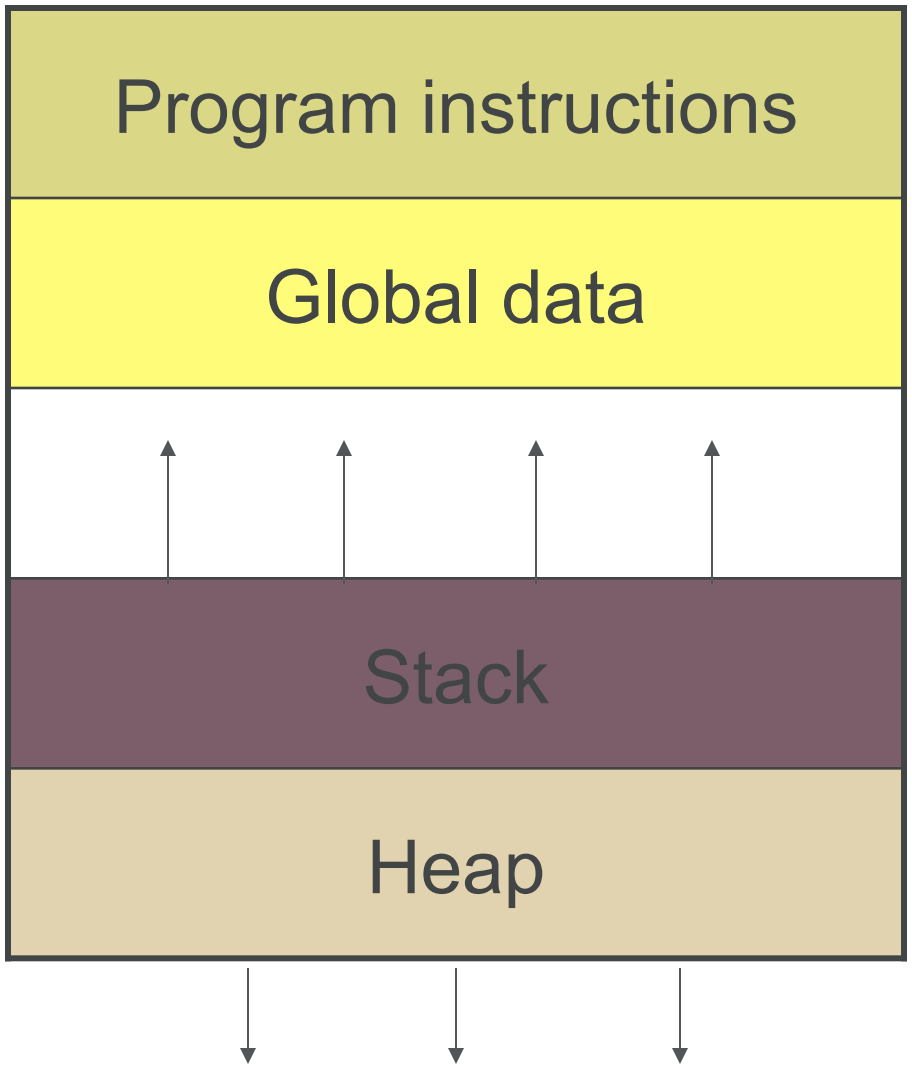
```
boolean rootPriv = false;
char name[8];
cin >> name;
```

- When the program reads the name “Armstrong”



From “teaching buffer overflow”

Program Memory Organization

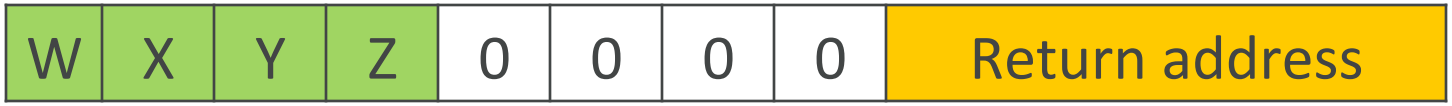


Intel method

Stack Overflow

- A stack overflow exploit occurs when a user enters data that exceeds the memory reserved for the input. The input can change adjacent data or the return address on the stack.
  - Distinct from non-malicious stack overflows.

```
char myStuff[4];
```



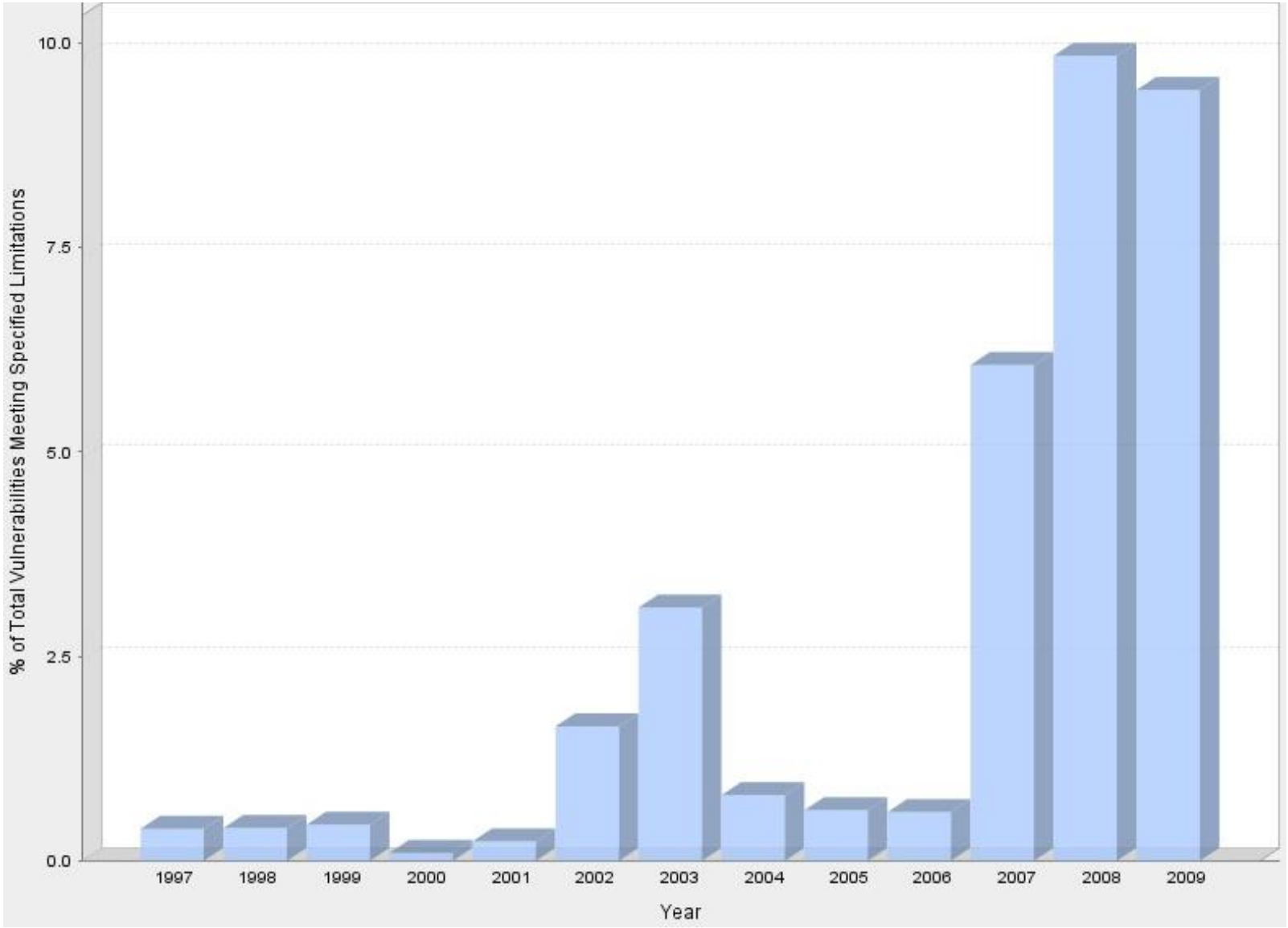
Program Stack

Basic exploit

- Basic exploit: executable attack code is stored on stack, in the buffer containing attacker's string
  - Stack memory usually contains only data, but…
- For the basic exploit, overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Buffer overflows

- Extremely common bug; Often leads to total compromise of host.
- First internet worm: The Morris Worm.
- 10 years later: over 50% of all CERT advisories:
  - 1997: 16 out of 28 CERT advisories.
  - 1998: 9 out of 13 -"-
  - 1999: 6 out of 12 -"-
- Fortunately: exploit requires expertise and patience
- Two steps:
  - Locate buffer overflow within an application.
  - Design an exploit.



source: DHS National Cyber Security Division/US-CERT National Vulnerability Database

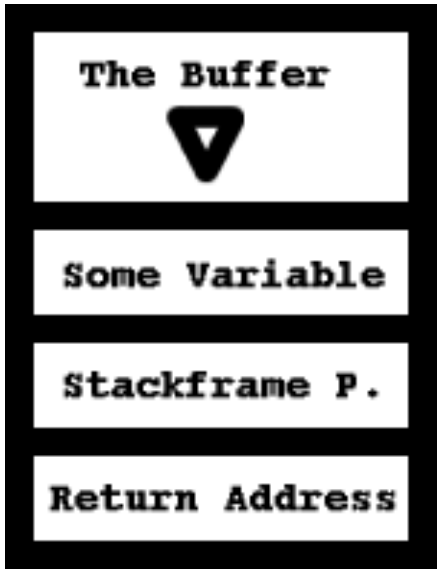
Insecurities

[Chen et al. 2005]

- Of 126 CERT security advisories of 2000-2004:
  - 87 are memory corruption vulnerabilities
  - 73 are in applications providing remote services
    - 13 in HTTP servers, 7 in database services, 6 in remote login services, 4 in mail services, 3 in FTP services
- Most exploits involve illegitimate control transfers
  - Jumps to injected attack code, return-to-libc, etc.
  - Therefore, most defenses focus on control-flow security
  - But exploits can also target configurations, user data and decision-making values
- When a security alert contains the phrase "**`The most severe of these vulnerabilities allows a remote attacker to execute arbitrary code.`**"---probably a buffer overflow.

Stack Basics

Lower memory addresses



High memory addresses

- A stack consists of logical stack frames that are pushed when calling a function and popped when returning.
- When a function is called, the return address, stack frame pointer and the variables are pushed on the stack (in that order).
- So the return address has a higher address than the buffer.
- When we overflow the buffer, the return address will be overwritten.

Stack Buffers

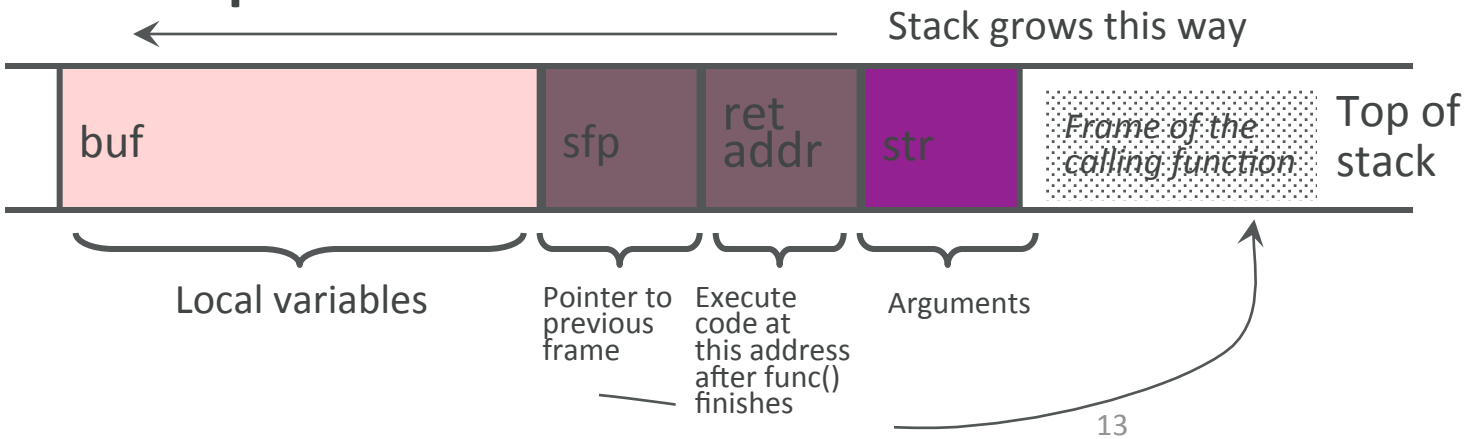
- Consider:

```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame is pushed onto the stack:



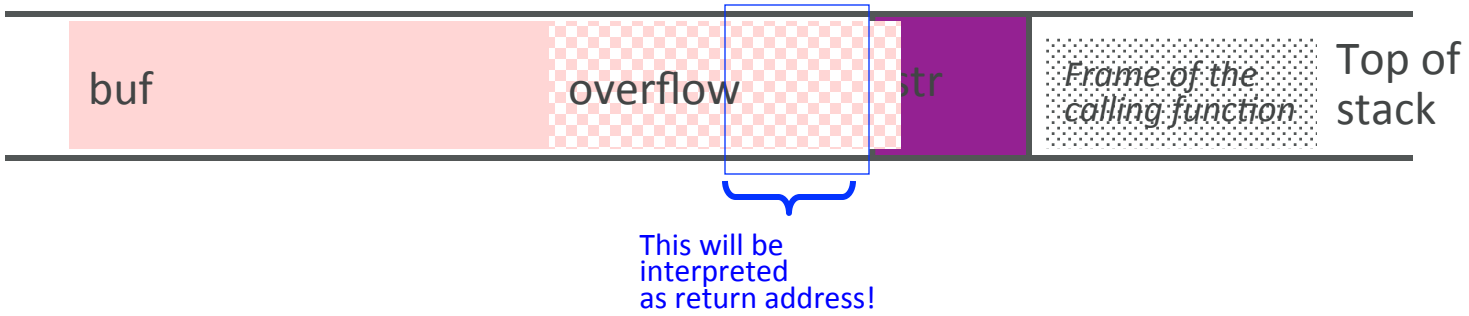
Overfull buffer?

- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
     char buf[126];
     strcpy(buf,str);
}
```

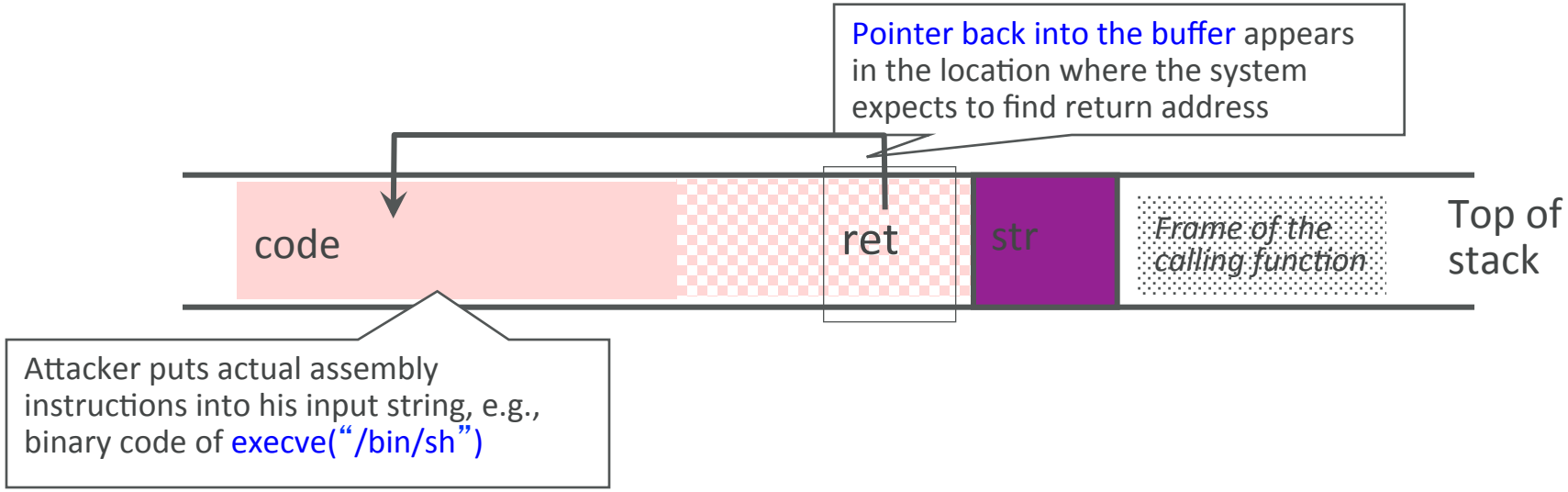
strcpy does not check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations:



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Stacks, again

- Consider the function:

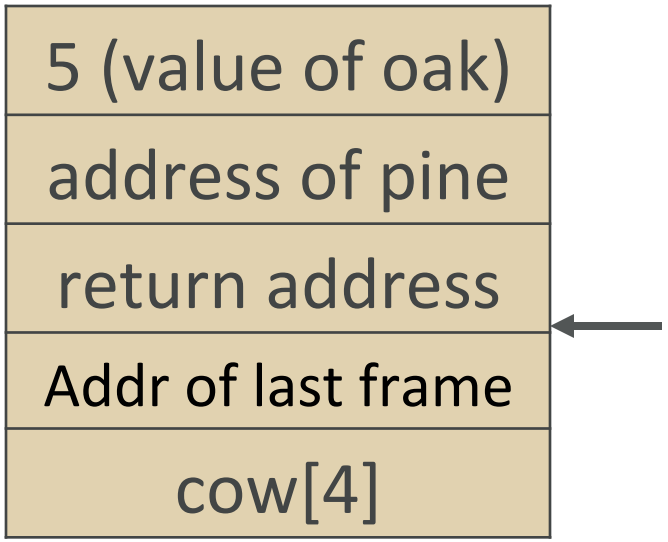
```
void thefunc( float &dog, int cat ){
      char cow[4];
}
```

- …called by the main program:

```
int oak = 5;
float  pine = 7.0;
float *birch = &pine;
thefunc( birch, oak );
```

Stack for Call

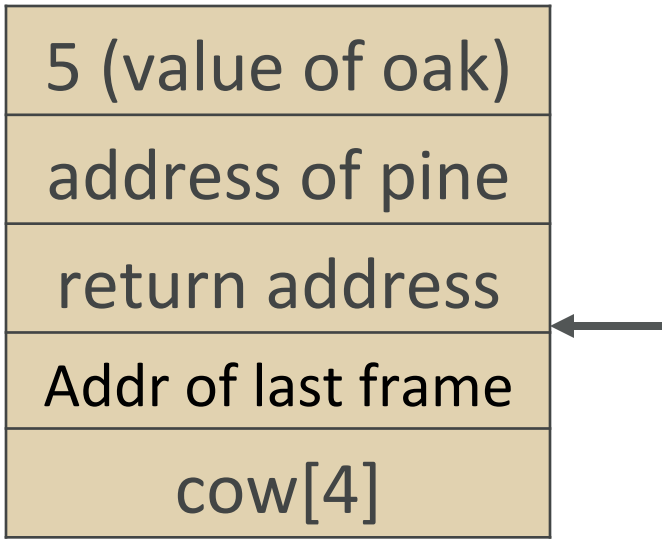
- push oak
- push birch
- push return address
- push frame pointer
- Allocate space for local variable, cow[4]




```
thefunc( birch, oak );
```

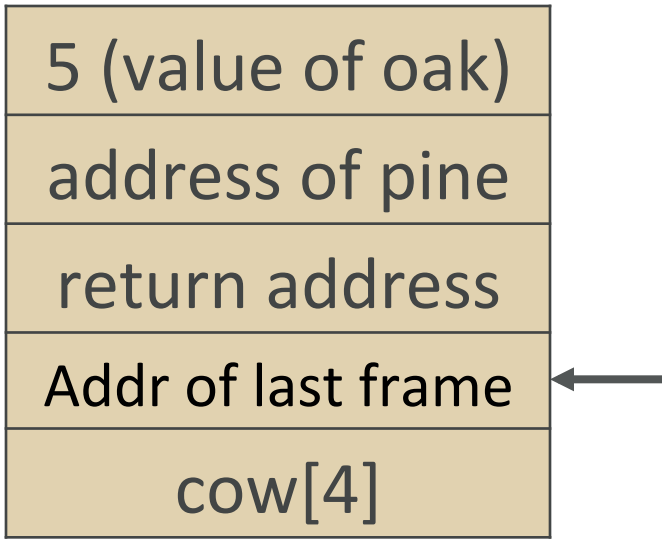
Overflowing Local Variables

- On an Intel processor *(and many others)*, stack is extended to lower addresses
- If you address beyond a local variable, you overwrite the return address.



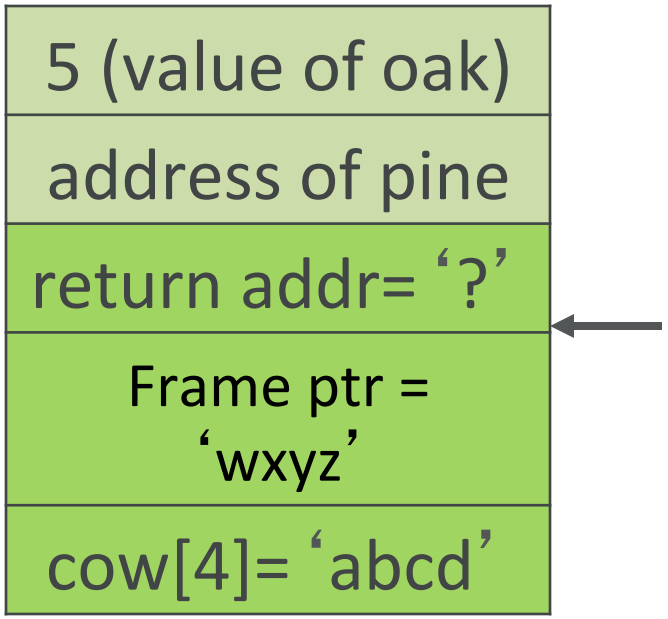
Hacking the Stack

- If a program does not check array bounds, it may be possible to give the program special input that overwrites the return address with a binary value.
- `cow[8]` to `cow[11]` are the return address



Hacking the Stack

- The return address can be changed to the address of a function in the program.
- Function parameters can also be put on the stack

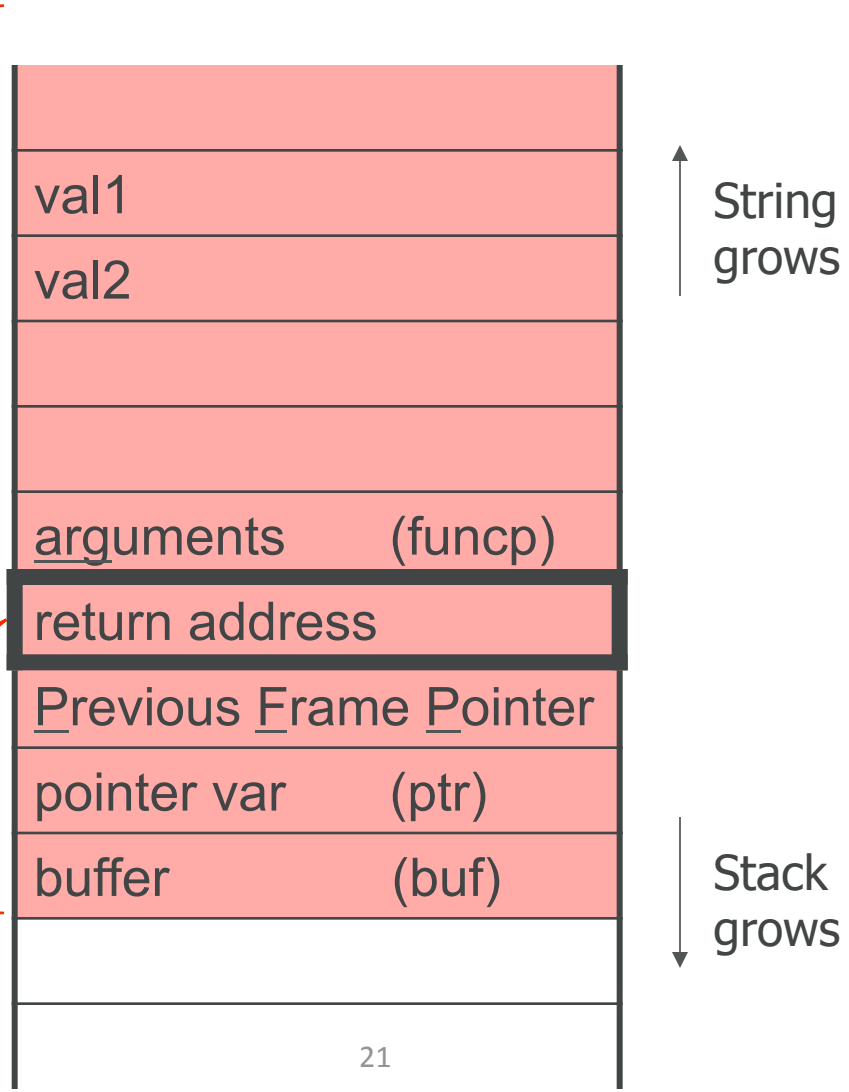


Stack Corruption: General View

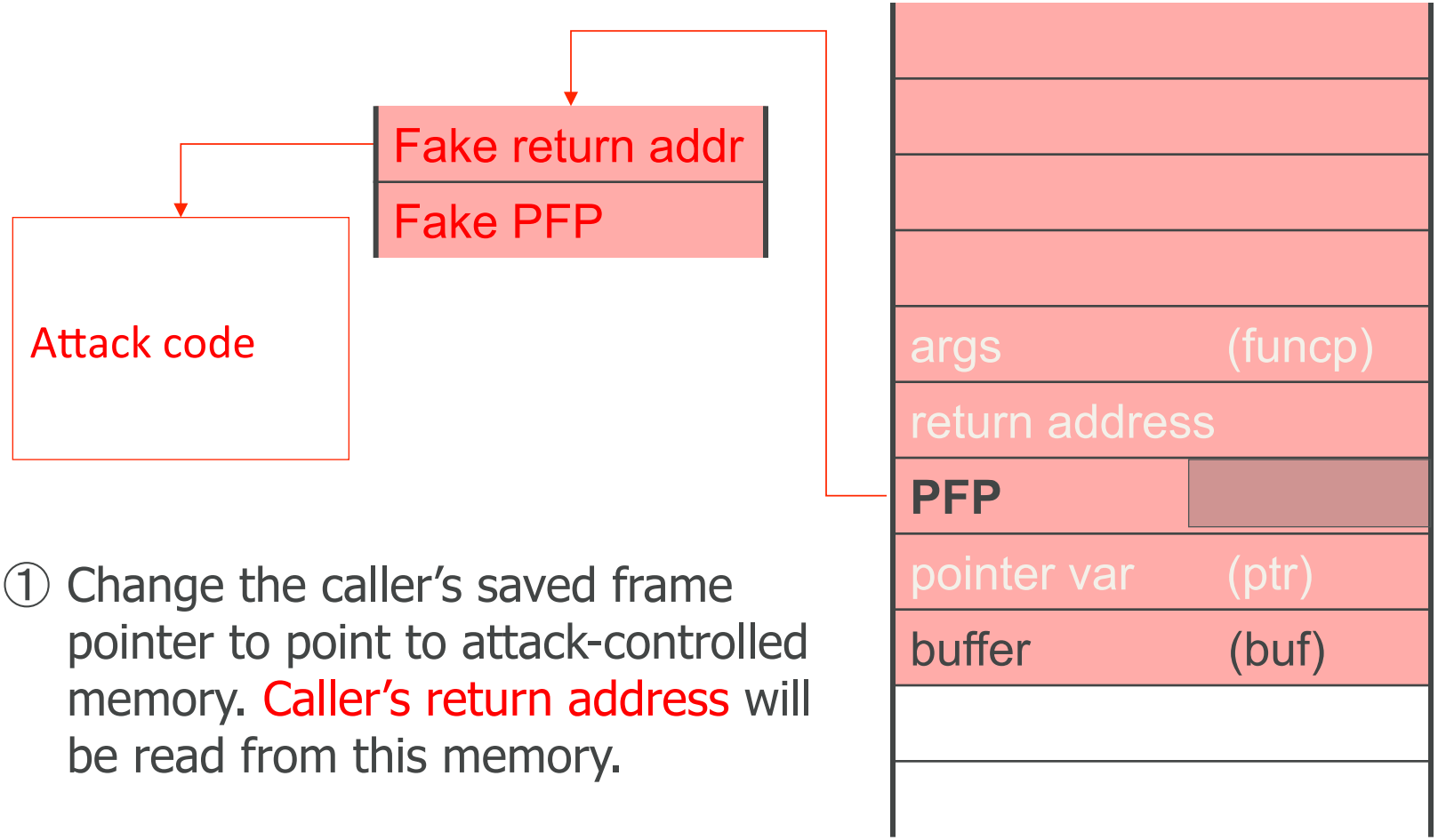
```
int bar (int val1) {
    int  val2;
    foo (a_function_pointer);
}

int foo (void (*funcp)()) {
    char* ptr = point_to_an_array;
    char buf[128];
    gets (buf);
    strncpy(ptr, buf, 8);
    (*funcp)();
}
```

Contaminated memory

Most popular target

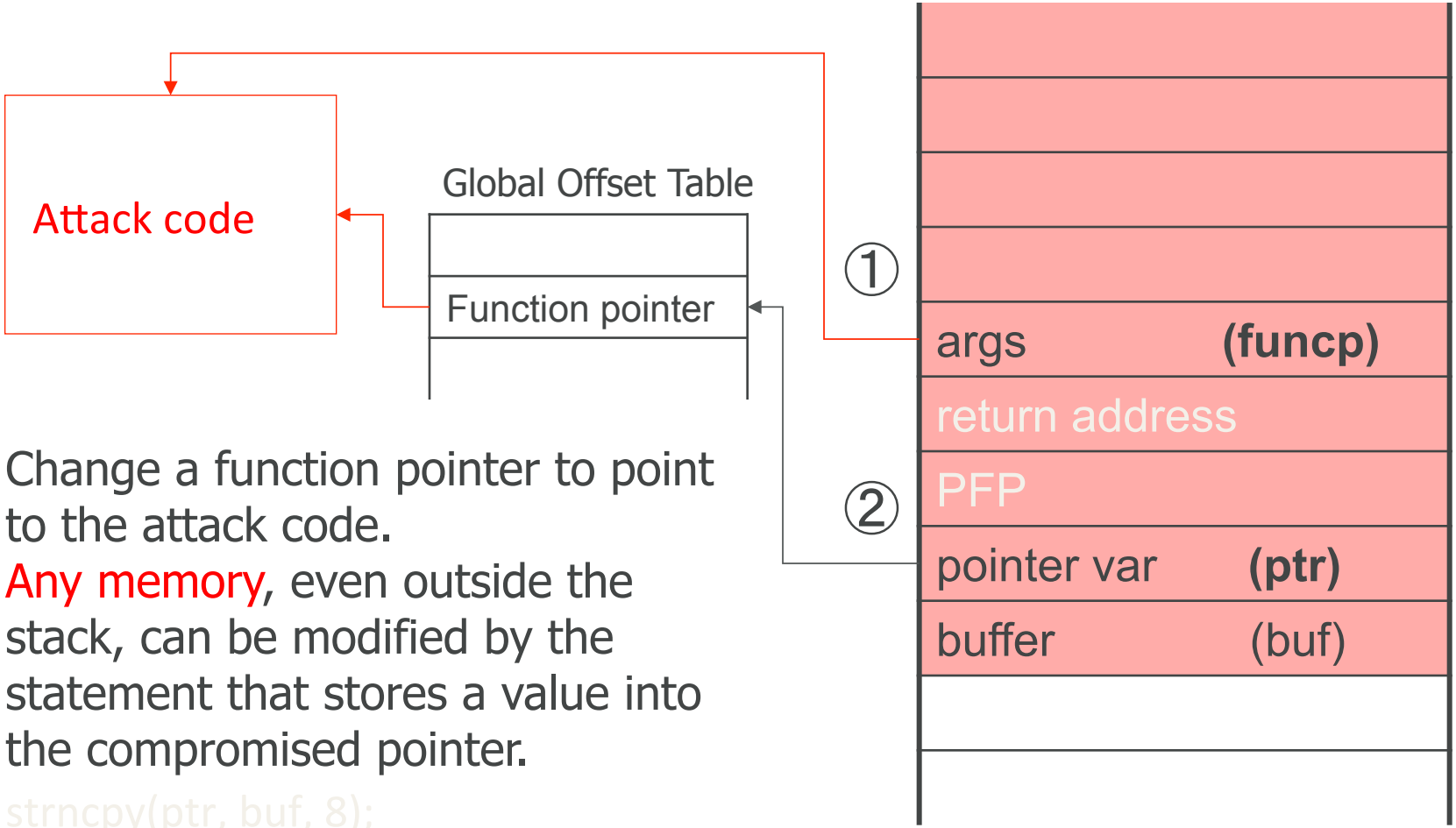


Attack #2: Frame Pointer



① Change the caller's saved frame pointer to point to attack-controlled memory. Caller's return address will be read from this memory.

Attack #3: Pointer Variables



① Change a function pointer to point to the attack code.

② Any memory, even outside the stack, can be modified by the statement that stores a value into the compromised pointer.

```
strncpy(ptr, buf, 8);
*ptr = 0;
```

CAUSES?

Unsafe functions

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Other unsafe I/O Functions

- **gets()**: has no way to limit input length.
- Use precision specifiers with the **scanf()** family of functions (**scanf()**, **fscanf()**, **sscanf()**, etc.). Otherwise they will not do any bounds checking for you.
- **`cin >>`** `char[]` will read more characters than the length of the string.
- The **`cin.get`** and **`cin.getline`** functions allow you to specify a maximum input length.

Off-By-One Overflow

- Home-brewed range-checking string copy:

```
void notSoSafeCopy(char *input) {
     char buffer[512]; int i;
     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

- 1-byte overflow: can't change RET, but can change pointer to previous stack frame

  On little-endian architecture, make it point into buffer
  RET for previous function will be read from buffer!

More on Off-by-one errors

- Off-by-one errors occur when a programmer takes the proper precautions in terms of bounds checking, but forgets that the last index is one less than the size.
- In C strings are terminated by a null character. Programmers often forget to reserve space for the null terminator.
- `char myString[10]` can only hold 9 printable characters, indexed from 0 to 8.

Data Expansion

- The size in bytes of the input might not be what causes the buffer overflow, it might be the input itself.
  - For example, if you're converting a large integer to a string (maybe in binary) make sure the buffer is long enough to hold all possible outputs
  - When converting special characters for web pages (i.e. "`>`" to "`>`") the output can become much larger
  - Unicode is twice the size of ASCII

Any ideas?

STATIC APPROACHES

Coverity, FindBugs, Etc.

SOME OBVIOUS HEURISTICS

Safer Languages

- Several modern languages have built-in protection against stack overflow.
- Java and C# check every array reference to ensure that it is within bounds.
  - Java does not allow stack violations.

Data Execution Prevention

- Most newer processors have a bit in the page table that inhibits instruction fetches from that page.
- Newer operating systems can set data execution prevention for stacks.
- This prevents the program from executing machine language loaded on the stack by an exploit.
- This does **not** prevent programs from overwriting the return address.

Safety Condition

- Let s be some string variable used in the program
- len(s) is the set of possible lengths
  - Why is len(s) not a single integer, but a set?
- alloc(s) is the set of possible values for the number of bytes allocated for s
  - Is it possible to compute len(s) and alloc(s) precisely at compile-time?
- At each point in program execution, want

$$\text{len}(s) \leq \text{alloc}(s)$$

Pointer analysis challenges

Flow-insensitive

```
1. int main() {
2.    int i = 0;
3.    int j = 20;
4.    int *p;
5.    int A[10];
6.    p = &j;
7.    p = &i;
8.    A[ *p ] = 42;
9. }
```

Context-insensitive

```
1.  int *id( int *a ) {
2.      return a;
3.  }
4.  int main() {
5.     int i = 0, j = 20;
6.     int *p, *q;
7.     int A[10];
8.     q = id(&j);
9.     p = id(&i);
10.    A[ *p ] = 42;
11. }
12.
```

TYPE SYSTEMS

BOON

[Wagner et al., 2000]

- Treat C strings as abstract data types
  - Assume that C strings are accessed only through library functions: strcpy, strcat, etc.
  - Pointer arithmetic is greatly simplified (what does this imply for soundness?)
- Characterize each buffer by its allocated size and current length (number of bytes in use)
- For each of these values, statically determine acceptable range at each point of the program
  - Done at compile-time, thus necessarily conservative (what does this imply for completeness?)

Integer Constraints

- Every string operation is associated with a constraint describing its effects

strcpy(dst,src)

strncpy(dst,src,n)

gets(s)

s="Hello!"

s[n]= '\0'

$len(src) \subseteq len(dst)$

$min(len(src),n) \subseteq len(dst)$

$[1,\infty] \subseteq len(s)$

$7 \subseteq len(s),\ 7 \subseteq alloc(s)$

$min(len(s),n+1)) \subseteq len(s)$

and so on

Does this fully capture what strncpy does?

Range of possible values

Constraint Generation Example

[Wagner]

```
char buf[128];                                          128 ⊆ alloc(buf)
while (fgets(buf, 128, stdin)) {                        [1,128] ⊆ len(buf)
    if (!strchr(buf, '\n')) {
        char error[128];                                128 ⊆ alloc(error)
        sprintf(error,"Line too long: %s\n,buf);        len(buf)+16 ⊆ len(error)
        die(error);
    }
...
}
```

Imprecision

- Simplifies pointer arithmetic and pointer aliasing
  - For example, q=p+j is associated with this constraint: alloc(p)-j ⊆ alloc(q), len(p)-j ⊆ len(q)
  - This is unsound (why?)
- Ignores function pointers
- Ignores control flow and order of statements
  - Consequence: every non-trivial strcat() must be flagged as a potential buffer overflow (why?)
- Merges information from all call sites of a function into one variable

Constraint Solving

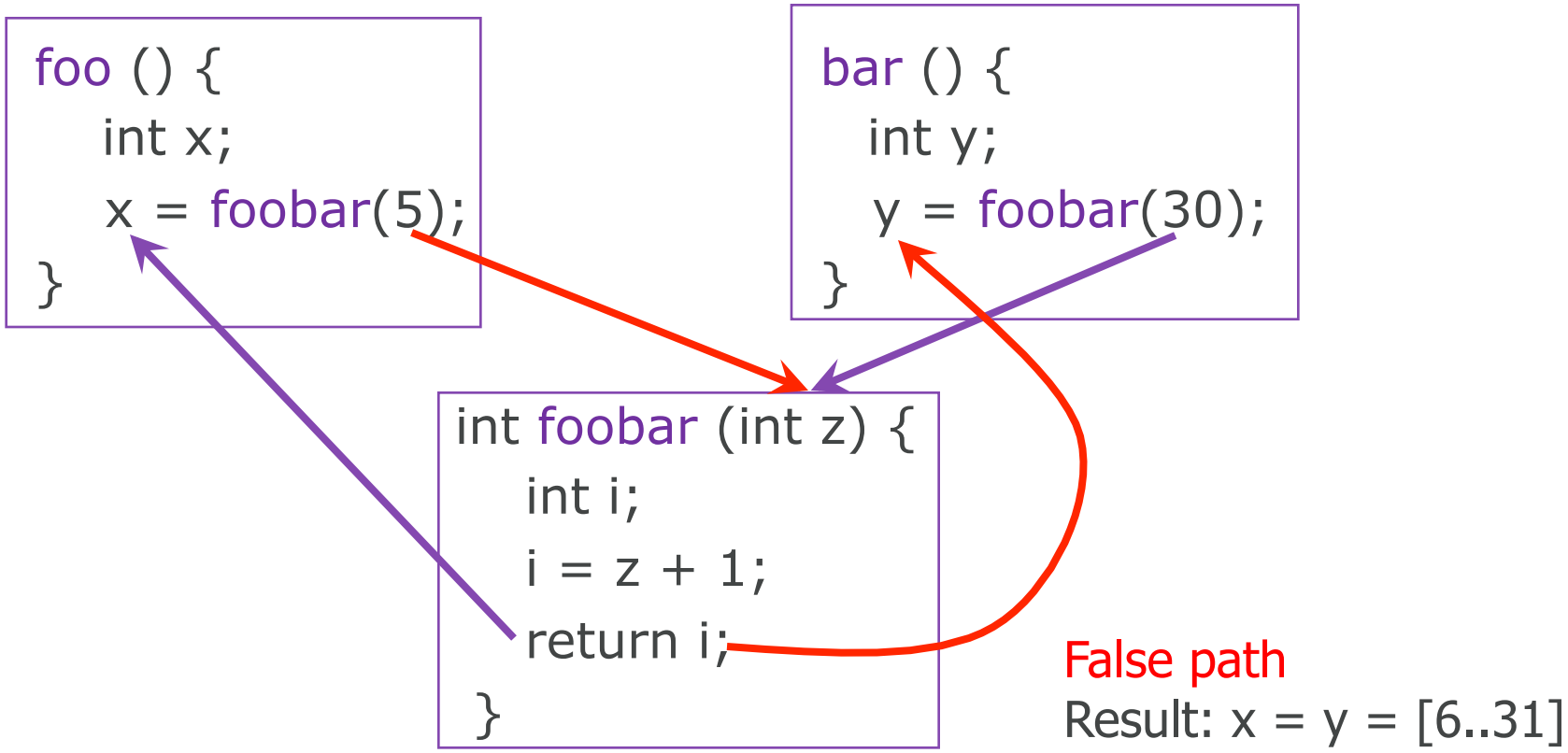
- "Bounding-box" algorithm (see paper)
  - Imprecise, but scalable: sendmail (32K LoC) yields a system with 9,000 variables and 29,000 constraints
- Suppose analysis discovers len(s) is in [a,b] range, and alloc(s) is in [c,d] range at some point
  - If $b \leq c$, then code is "safe"
    - Does not completely rule out buffer overflow (why?)
  - If $a > d$, then buffer overflow always occurs here
  - If ranges overlap, overflow is possible
- Ganapathy et al.: model and solve the constraints as a linear program (see paper)

Practical Results

- Found new vulnerabilities in real systems code
  - Exploitable buffer overflows in nettools and sendmail
- Lots of false positives, but still a dramatic improvement over hand search
  - sendmail: over 700 calls to unsafe string functions, of them 44 flagged as dangerous, 4 are real errors
  - Example of a false alarm:

```
if (sizeof from < strlen(e->e_from.q_paddr)+1) break;
strcpy(from, e->e_from.q_paddr);
```

Context-Insensitivity is Imprecise



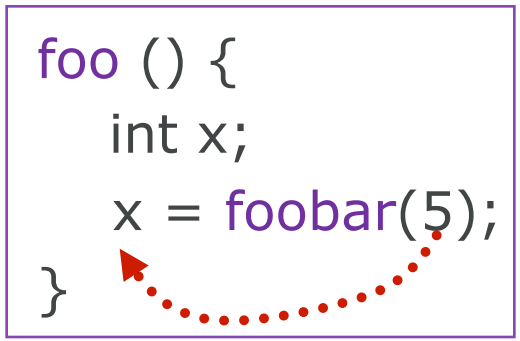
Adding Context Sensitivity

[Ganapathy et al.]

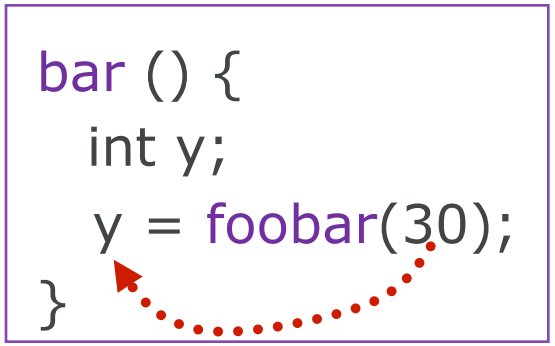
- Make user functions context-sensitive
  - For example, wrappers around library calls
- Inefficient method: constraint inlining
  - ☺ Can separate calling contexts
  - ☹ Large number of constraint variables
  - ☹ Cannot support recursion
- Efficient method: procedure summaries
  - Summarize the called procedure
  - Insert the summary at the callsite in the caller
  - Remove false paths

Context-Sensitive Analysis

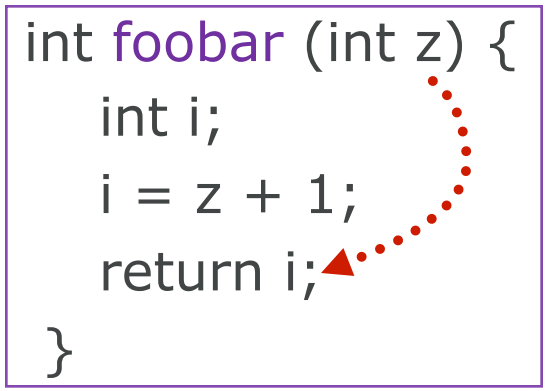
[Ganapathy et al.]



x = 5 + 1



y = 30 + 1



Summary: i = z + 1

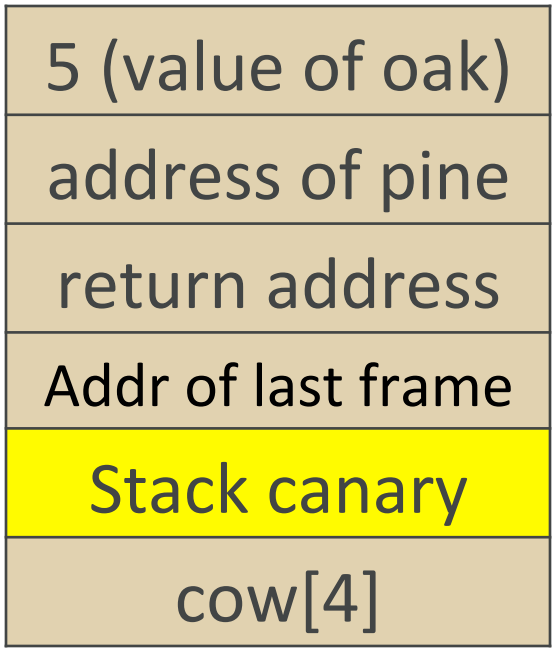
DIVERSION INTO DYNAMIC APPROACHES

Run time checking: StackGuard

- Many many run-time checking techniques …
- Solution: StackGuard (WireX)
  - Run time tests for stack integrity.
  - Enhance the code generator for emitting code to set up and tear down functions
  - Embeds "canaries" in stack frames and verify their integrity prior to function return.

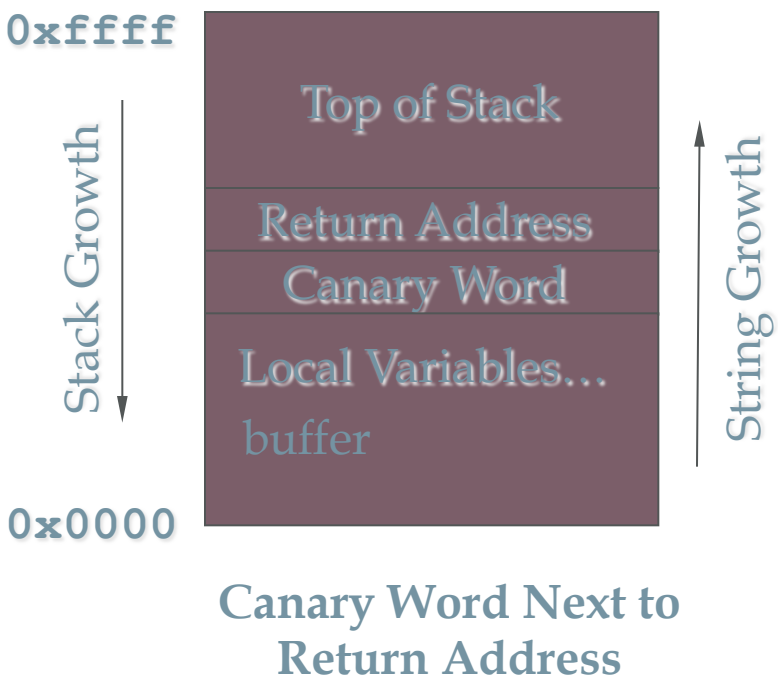
Stack Canaries

- A stack canary is a random number placed on the stack between the user data and the return address.
- Overflowing the local variable and changing the return address will also change the stack canary
- Before returning, the program checks the canary value.



StackGuard

- Prevents changes to active RAs in 2 ways:
  1. By detecting change of the RA before the function returns. (more efficient and portable).
  2. By completely preventing the write to the RA (more secure).



StackGuard - Detecting RA Change Before Return

- Detection done before a function returns.
- A canary word placed next to the RA on the stack.
- When function returns, it first checks to see that the canary word is intact before jumping to the RA pointed word.
- Key: RA is unaltered iff the canary word is unaltered. (How?) – true for buffer overflow attacks.

StackGuard - Detecting RA Change Before Return

- StackGuard implementation - simple patch to gcc 2.7.2.2.
- gcc function_prologue and function_epilogue functions - altered to emit code to place and check canary words.
- *Problem:* Attackers can develop buffer overflows insensitive to StackGuard.
- *Solution:* Randomize the Canary.

Canary Types

- Random canary: (used in Visual Studio 2003)
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.

- Terminator canary:

  Canary = 0 (null), newline, linefeed, EOF

  - String functions will not copy beyond terminator.
  - Hence, attacker cannot use string functions to corrupt stack.

More methods …

- Address obfuscation. (Stony Brook ' 03)
  - Encrypt return address on stack by XORing with random string. Decrypt just before returning from function.
  - Attacker needs decryption key to set return address to desired value.

- PaX ASLR: Randomize location of libc.
  - Attacker cannot jump directly to exec function.

BACK TO TYPE SYSTEMS

CCured

[Necula et al.]

- Goal: make legacy C code type-safe
- Treat C as a mixture of a strongly typed, statically checked language and an "unsafe" language checked at runtime
  - All values belong either to "safe," or "unsafe" world
- Combination of static and dynamic checking
  - Check type safety at compile-time whenever possible
  - When compile-time checking fails, compiler inserts run-time checks in the code
  - Fewer run-time checks ⇒ better performance

Example Dependent Typing Rules

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau}{\Gamma \vdash \lambda x : \tau_2 . e : \Pi x : \tau_2 . \tau} \qquad \frac{\Gamma \vdash e_1 : \Pi x : \tau_2 . \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : [e_2/x]\tau}$$

Note that expressions are now part of types

Have types like “vector 5” and “vector (2 + 3)”

We need type equivalence

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash e : \tau'}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2}{\Gamma \vdash \texttt{vector}\ e_1 \equiv \texttt{vector}\ e_2}$$

Ccured: Safe Pointers

- Either NULL, or a valid address of type T
- Aliases are either safe pointers, or sequence pointers of base type T
- What is legal to do with a safe pointer?
  - Set to NULL
  - Cast from a sequence pointer of base type T
  - Cast to an integer
- What runtime checks are required?
  - Not equal to NULL when dereferenced

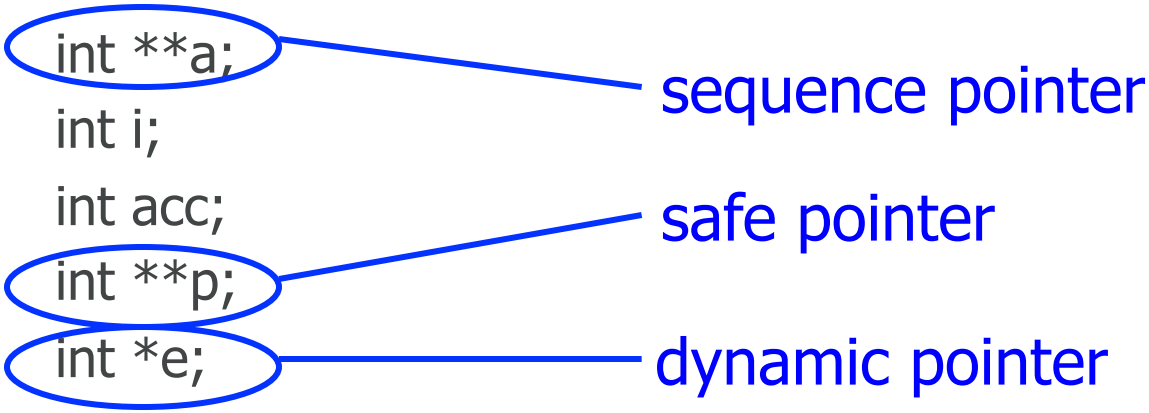
Sequence Pointers

- At runtime, either an integer, or points to a known memory area containing values of type T
- Aliases are safe, or sequence ptrs of base type T
- What is legal to do with a sequence pointer?
  - Perform pointer arithmetic
  - Cast to a safe pointer of base type T
  - Cast to or from an integer
- What runtime checks are required?
  - Points to a valid address when dereferenced
    - Subsumes NULL checking
  - Bounds check when dereferenced or cast to safe ptr

Dynamic Pointers

- At runtime, either an integer, or points to a known memory area containing values of type T
- The memory area to which it points has tags that distinguish integers from pointers
- Aliases are dynamic pointers
- What is legal to do with a dynamic pointer?
  - Perform pointer arithmetic
  - Cast to or from an integer or any dynamic pointer type
- Runtime checks of address validity and bounds
  - Maintain tags when reading & writing to base area

Example


int **a;
int i;
int acc;
int **p;
int *e;
acc=0;
for(i=0; i<100;i++){
p= a + i;
e = *p;
while((int) e % 2 == 0){
e = *(int **) e;}
acc+=((int) e >> 1);
}
sequence pointer
safe pointer
dynamic pointer

Modified Pointer Representation

- Each allocated memory area is called a home (H), with a starting address h and a size
- Valid runtime values for a given type:
  - Integers: ||int|| = N
  - Safe pointers: ||τ ref SAFE|| = { h+i | h ∈ H and 0≤i<size(h) and (h=0 or kind(h)=Typed(τ)) }
  - Sequence pointers: ||τ ref SEQ|| = {<h,n> | h ∈ H and (h=0 or kind(h)=Typed(τ)) }
  - Dynamic pointers: ||DYNAMIC|| = {<h,n> | h ∈ H and (h=0 or kind(h)=Untyped) }

Safe pointers are integers, same as standard C

For sequence and dynamic pointers, must keep track of the address and size of the pointed area for runtime bounds checking

Runtime Memory Safety

- Each memory home (i.e., allocated memory area) has typing constraints
  - Either contains values of type τ, or is untyped
- If a memory address belong to a home, its contents at runtime must satisfy the home's typing constraints
  - $\forall h \in H \setminus \{0\}\ \forall i \in N$

    if $0 \leq i < size(h)$ then

    $(kind(h)=Untyped \Rightarrow Memory[h+i] \in ||DYNAMIC||$ and

    $kind(h)=Typed(\tau) \Rightarrow Memory[h+i] \in ||\tau||)$

Runtime Checks

- Memory accesses
  - If via safe pointer, only check for non-NULL
  - If via sequence or dynamic pointer, also bounds check
- Typecasts
  - From sequence pointers to safe pointers
    - This requires a bounds check!
  - From pointers to integers
  - From integers to sequence or dynamic pointers
    - But the home of the resulting pointer is NULL and it cannot be dereferenced; this breaks C programs that cast pointers into integers and back into pointers

Inferring Pointer Types

- Manual: programmer annotates code
- Better: type inference
  - Analyze the source code to find as many safe and sequence pointers as possible
- This is done by resolving a set of constraints
  - If p is used in pointer arithmetic, p is not safe
  - If p1 is cast to p2
    - Either they are of the same kind, or p1 is a sequence pointer and p2 is a safe pointer
    - Pointed areas must be of same type, unless both are dynamic
  - If p1 points to p2 and p1 is dynamic, then p2 dynamic.
- …etc.

Various CCured Issues

- Converting a pointer to an integer and back to a pointer no longer works
  - Sometimes fixed by forcing the pointer to be dynamic
- Modified pointer representation
  - Not interoperable with libraries that are not recompiled using CCured (use wrappers)
  - Breaks sizeof() on pointer types
- If program stores addresses of stack variables in memory, these variables must be moved to heap
- Garbage collection instead of explicit deallocation

Performance

- Most pointers in benchmark programs were inferred safe, performance penalty under 90%
  - Less than 20% in half the cases
  - Minimal slowdown on I/O-bound applications
    - Linux kernel modules, Apache
  - If all pointers were made dynamic, then 6 to 20 times slower (similar to a pure runtime-checks approach)
  - On the other hand, pure runtime-checks approach does not require access to source code and recompilation
- Various bugs found in test programs
  - Array bounds violations, uninitialized array indices

ANNOTATIONS/CONTRACTS: PROVIDED AND INFERRED

C String Static Verifier

[Dor, Rodeh, Sagiv]

1. "Contracts" specify procedure's pre- and post-conditions, potential side effects
2. Flow-insensitive "points-to" pointer analysis
3. Transform C procedure into a procedure over integers; apply integer analysis to find variable constraints
   - Any potential buffer overflow in the original program violates an "assert" statement in this integer program

Example: strcpy Contract

[Dor, Rodeh, Sagiv]

char* strcpy(char* dst, char *src)

requires string(src) ∧
alloc(dst) > len(src)

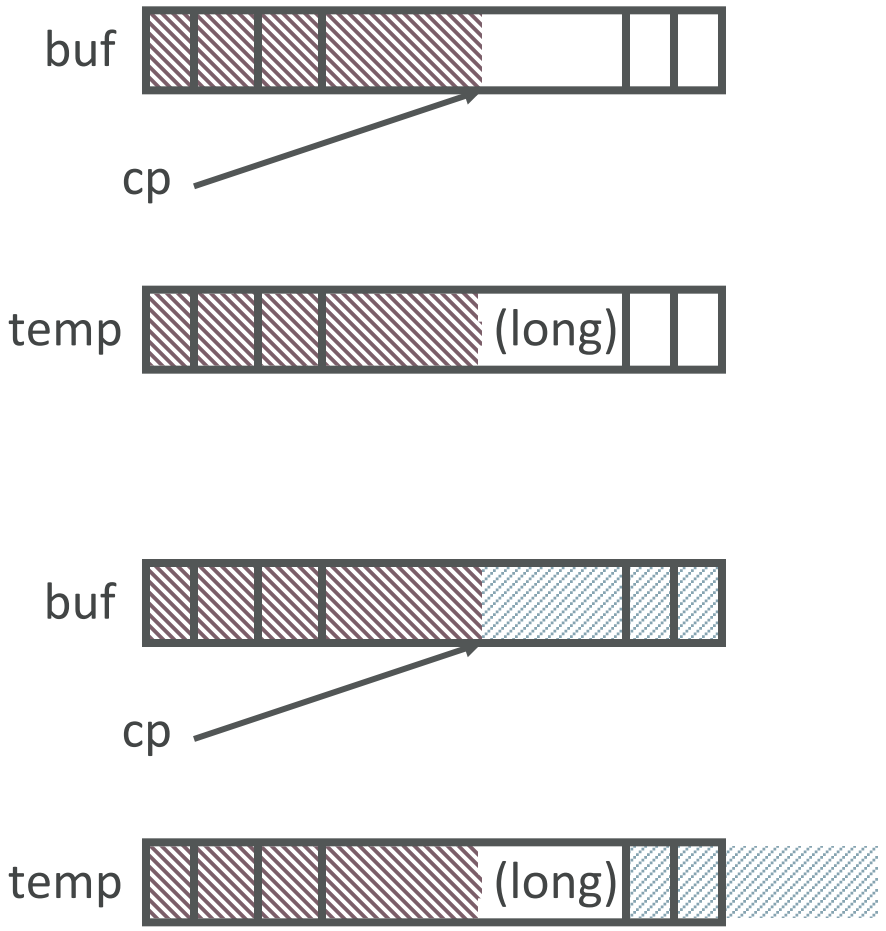
modifies dst.strlen, dst.is_nullt

ensures len(dst) = = pre@len(src) ∧
return = = pre@dst

Example: insert_long()

[Dor, Rodeh, Sagiv]

```
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
   char temp[BUFSIZ];
   int i;
   for (i=0; &buf[i] < cp; ++i){
      temp[i] = buf[i];
      }
    strcpy (&temp[i],"(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}
```



insert_long() Contract

[Dor, Rodeh, Sagiv]

```
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
  char temp[BUFSIZ];
  int i;
  for (i=0; &buf[i] < cp; ++i){
    temp[i] = buf[i];
    }
   strcpy (&temp[i],"(long)");
   strcpy (&temp[i + 6], cp);
   strcpy (buf, temp);
   return cp + 6;
}
```

```
char * insert_long(char *cp)
   requires  string(cp)  ∧
             buf ≤ cp < buf  + BUFSIZ
  modifies  cp.strlen
  ensures
     cp.strlen = = pre[cp.strlen] + 6
     ∧
     return_value = = cp + 6 ;
```

C2IP: C to Integer Program

[Dor, Rodeh, Sagiv]

- Integer variables only
- No function calls
- Non-deterministic
- Constraint variables
- Update statements
- Assert statements

Based on points-to information

  - Any string manipulation error in the original C program is guaranteed to violate an assertion in integer program

Transformations for C Statements

[Dor, Rodeh, Sagiv]

C Construct	IP Statements
`p = Alloc(i);`	$l_p.offset := 0;$ $r_p.aSize := l_i.val;$ $r_p.is\_nullt := false;$
`p = q + i;`	$l_p.offset := l_q.offset + l_i.val;$

`*p = c;`	if $c = 0$ then {
	$r_p.len := l_p.offset;$
	$r_p.is\_nullt := true;$ }
	else
	if $r_p.is\_nullt \wedge l_p.offset = r_p.len$ then
	$l_p.is\_nullt := unknown;$

`c = *p;`	if $r_p.is\_nullt \wedge l_p.offset = r_p.len$ then $l_c.val := 0;$ else $l_c.val := unknown;$
$g(a_1, a_2, \ldots, a_m);$	$mod[g](a_1, a_2, \ldots, a_m);$
`*p == 0`	$r_p.is\_nullt \wedge r_p.len = l_p.offset$
`p > q`	$l_p.offset > l_q.offset$
p.alloc	$r_p.aSize - l_p.offset$
p.offset	$l_p.offset$
p.is_nullt	$r_p.is\_nullt$
p.strlen	$r_p.len - l_p.offset$

For abstract location l,
l.val - potential values stored in the locations represented by l
l.offset - potential values of the pointers represented by l
l.aSize - allocation size
l.is_nullt - null-terminated?
l.len - length of the string

For pointer p,
l_p - its location
r_p - location it points to
(if several possibilities, use nondeterministic assignment)

Correctness Assertions

[Dor, Rodeh, Sagiv]

C Exp.	Generated IP Condition
`*p`	$l_p.offset \geq 0 \wedge$ $((r_p.is\_nullt \wedge l_p.offset \leq r_p.len) \vee$ $(\neg r_p.is\_nullt \wedge l_p.offset < r_p.aSize))$
`p + i`	$l_p.offset + l_i.val \geq 0 \wedge$ $l_p.offset + l_i.val \leq r_p.aSize$

All dereferenced pointers point to valid locations

Results of pointer arithmetic are valid

Example

[Dor, Rodeh, Sagiv]

Assert statement:

```
assert (
5 <= q.alloc &&
(!q.is_nullt || 5 <= q.len) )
```

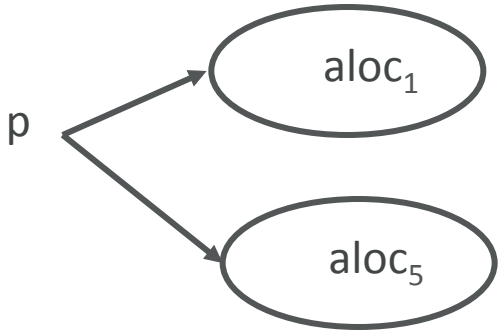
```
p = q + 5;
```

Update statement:

```
p.offset = q.offset + 5;
```

Nondeterminism

[Dor, Rodeh, Sagiv]



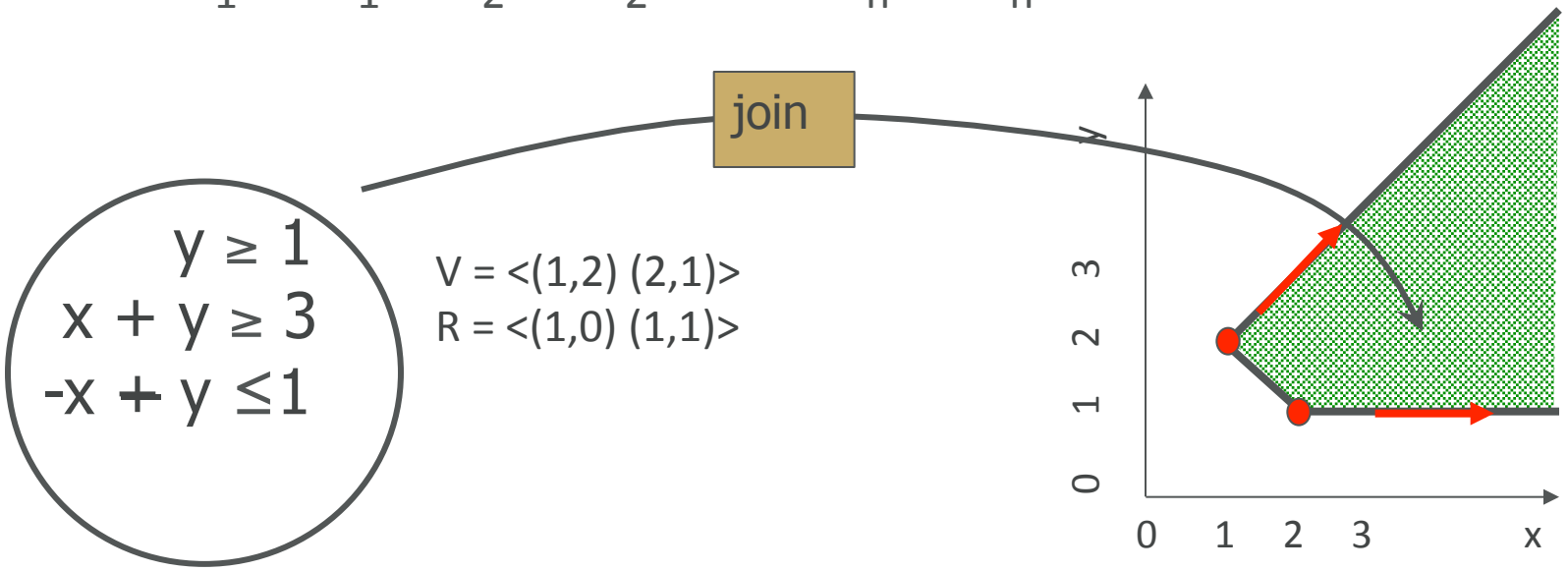
*p = 0;

```
if (...) {
    aloc1.len = p.offset;
    aloc1.is_nullt = true; }
else {
    alloc5.len = p.offset;
    alloc5.is_nullt = true; }
```

Integer Analysis

[Dor, Rodeh, Sagiv]

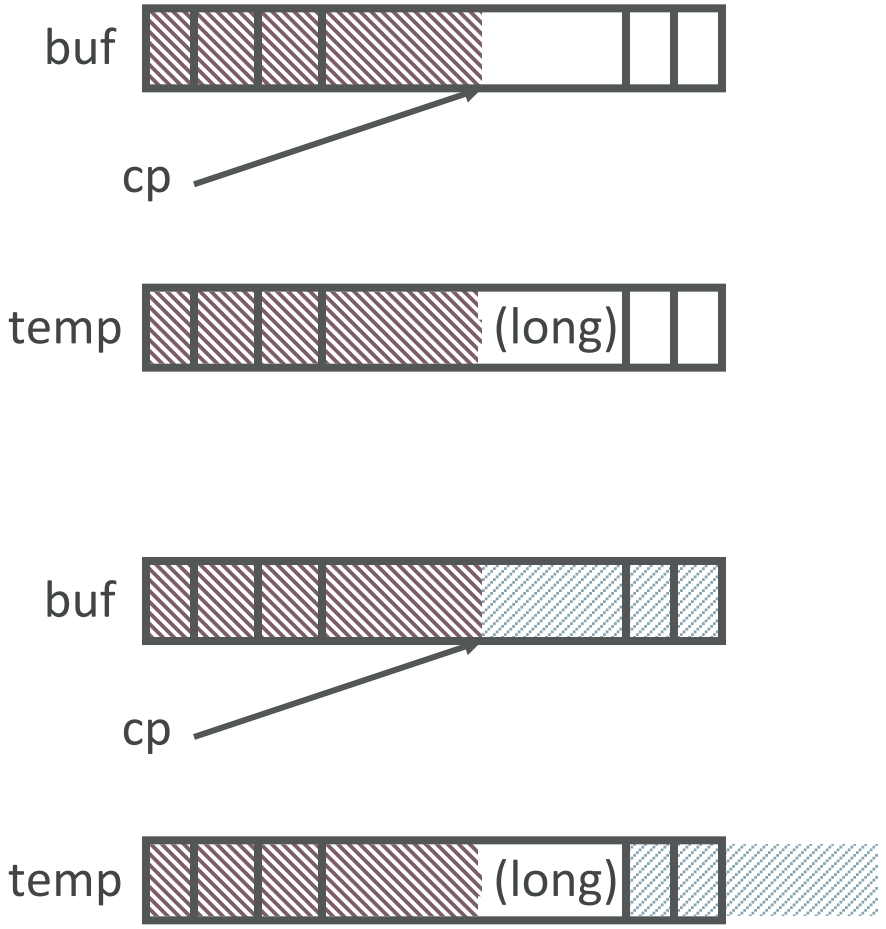
- Interval analysis not enough
  - Loses relationships between variables
- Infer variable constraints using abstract domain of polyhedra [Cousot and Halbwachs, 1978]
  - $a_1 * var_1 + a_2 * var_2 + \ldots + a_n * var_n \leq b$



insert_long() Redux

[Dor, Rodeh, Sagiv]

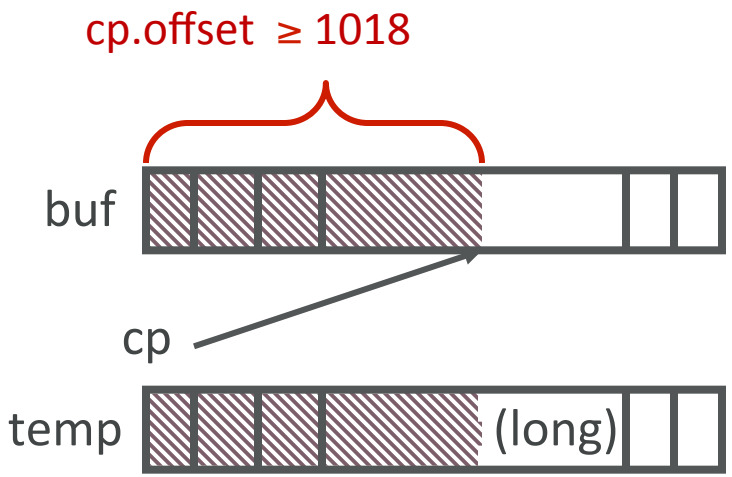
```
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
   char temp[BUFSIZ];
   int i;
   for (i=0; &buf[i] < cp; ++i){
      temp[i] = buf[i];
      }
    strcpy (&temp[i],"(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}
```



Integer Analysis of insert_long()

[Dor, Rodeh, Sagiv]

buf.offset = 0
temp.offset = 0
$0 \le$ cp.offset $= i$
$i \le s_{buf}.len < s_{buf}.msize$
$s_{buf}.msize = 1024$
$s_{temp}.msize = 1024$



assert($0 \le i < s_{temp}.msize - 6$); // strcpy(&temp[i],"(long)");

Potential violation
when cp.offset ≥ 1018

FOR THURSDAY: MODULAR ANNOTATION-BASED OVERFLOW AT MICROSOFT