Static Defenses against Memory Corruption

Vitaly Shmatikov

Reading Assignment

- Wagner et al. "A first step towards automated detection of buffer overrun vulnerabilities" (NDSS 2000).
- Ganapathy et al. "Buffer overrun detection using linear programming and static analysis" (CCS 2003).
- Dor, Rodeh, Sagiv. "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C" (PLDI 2003).

Static Analysis

- Goal: catch buffer overflow bugs by analyzing the source code of the program
  - Typically at compile-time, but also binary analysis
- Static analysis is necessarily imprecise
  - Soundness: finds all instances of buffer overflow
    - Problem: false positives (good code erroneously flagged)
  - Completeness: every reported problem is indeed an instance of buffer overflow
    - Problem: false negatives (misses some buffer overflows)
  - No technique is both sound and complete (why?)
  - Maybe don't need either…

Static vs. Dynamic

- Both static and dynamic approaches have their advantages and disadvantages (what are they?)
- Hybrid approaches (example: CCured)
  - Try to verify absence of memory errors statically, then insert runtime checks where static verification failed
- Performance and usability are always important
  - Does source code need to be modified?
  - Does source code need to be recompiled?
  - How is backward compatibility (if any) achieved?
    - Rewriting binaries vs. special runtime environment

BOON

[Wagner et al.]

- 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?)

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

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

Integer Constraints

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

strcpy(dst,src)	$len(src) \subseteq len(dst)$
strncpy(dst,src,n)	$min(len(src),n) \subseteq len(dst)$
gets(s)	$[1,\infty] \subseteq len(s)$
s="Hello!"	$7 \subseteq len(s),\ 7 \subseteq alloc(s)$
s[n]='\0'	$min(len(s),n+1)) \subseteq len(s)$
	and so on

Does this fully capture what strncpy does?

Range of possible values

Constraint Generation Example

[Wagner]

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

$128 \subseteq alloc(buf)$

$[1,128] \subseteq len(buf)$

$128 \subseteq alloc(error)$

$len(buf)+16 \subseteq len(error)$

Imprecision

- Simplifies pointer arithmetic and pointer aliasing
  - For example, q=p+j is associated with this constraint: alloc(p)-j $\subseteq$ alloc(q), len(p)-j $\subseteq$ 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)

BOON: 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

```
foo () {
    int x;
    x = foobar(5);
}
```

```
bar () {
    int y;
    y = foobar(30);
}
```

```
int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}
```

False path

Result: x = y = [6..31]

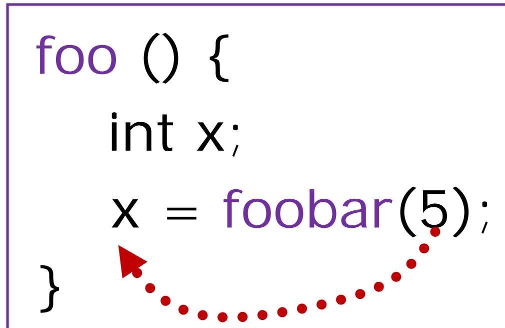
Adding Context Sensitivity

[Ganapathy et al.]

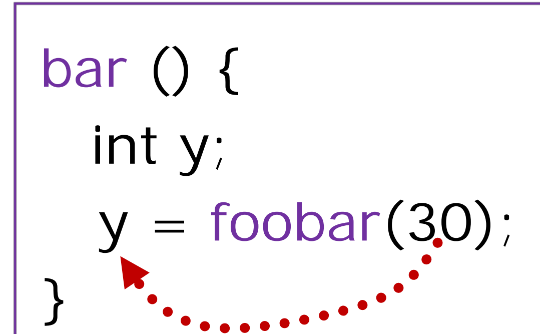
- Make user functions context-sensitive
  - For example, wrappers around library calls
- Inefficient method: constraint inlining
  - ☺ Can separate calling contexts
  - ☹ Large number of constraint variables
  - ☹ Cannot support recursion
- Efficient method: procedure summaries
  - Summarize the called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

Context-Sensitive Analysis

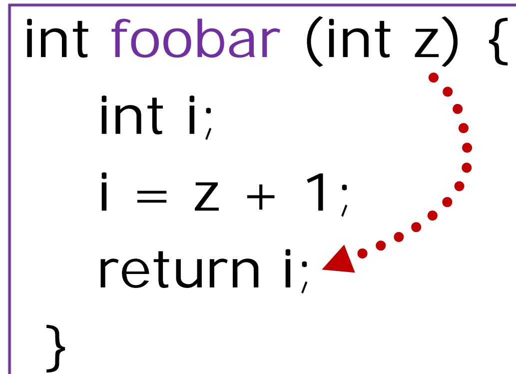
[Ganapathy et al.]



x = 5 + 1



y = 30 + 1



Summary: i = z + 1

No False Paths

[Ganapathy et al.]

```
foo () {
    int x;
    x = foobar(5);
}
```

```
bar () {
    int y;
    y = foobar(30);
}
```

Jump functions

```
int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}
```

Constraints

x = [6..6]

y = [31..31]

i = [6..31]

Computing Procedure Summaries

[Ganapathy et al.]

- If function produces only difference constraints, reduces to an all-pairs shortest-path problem
- Otherwise, Fourier-Motzkin variable elimination
- Tradeoff between precision and efficiency
  - Constraint inlining: rename local variables of the called function at each callsite
    - Precise, but a huge number of variables and constraints
  - Procedure summaries: merge variables across callsites
    - For example, constraint for i in the foobar example

Off-by-one Bug in sendmail-8.9.3

- orderq() reads a file from the queue directory, copies its name into d->d_name and w->w_name
  - As long as 21 bytes, including the '\0' terminator
- runqueue() calls dowork(w->w_name+2,...), dowork() stores its first argument into e->e_id
- queuename() concatenates "qf" and e->e_id, copies the result into 20-byte dfname buffer

19 bytes

21 bytes

- Wagner et al.: a pointer to a structure of type T can point to all structures of type T
  - Finds the bug, but do you see any issues?
- Ganapathy et al.: precise points-to analysis

CSSV

[Dor, Rodeh, Sagiv]

- Goal: sound static detection of buffer overflows
  - What does this mean?
- Separate analysis for each procedure
- "Contracts" specify procedure's pre- and post-conditions, potential side effects
  - Analysis only meaningful if contracts are correct
- Flow-insensitive "points-to" pointer analysis
- Transform C 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 $\quad$ string(src) $\wedge$ alloc(dst) > len(src)

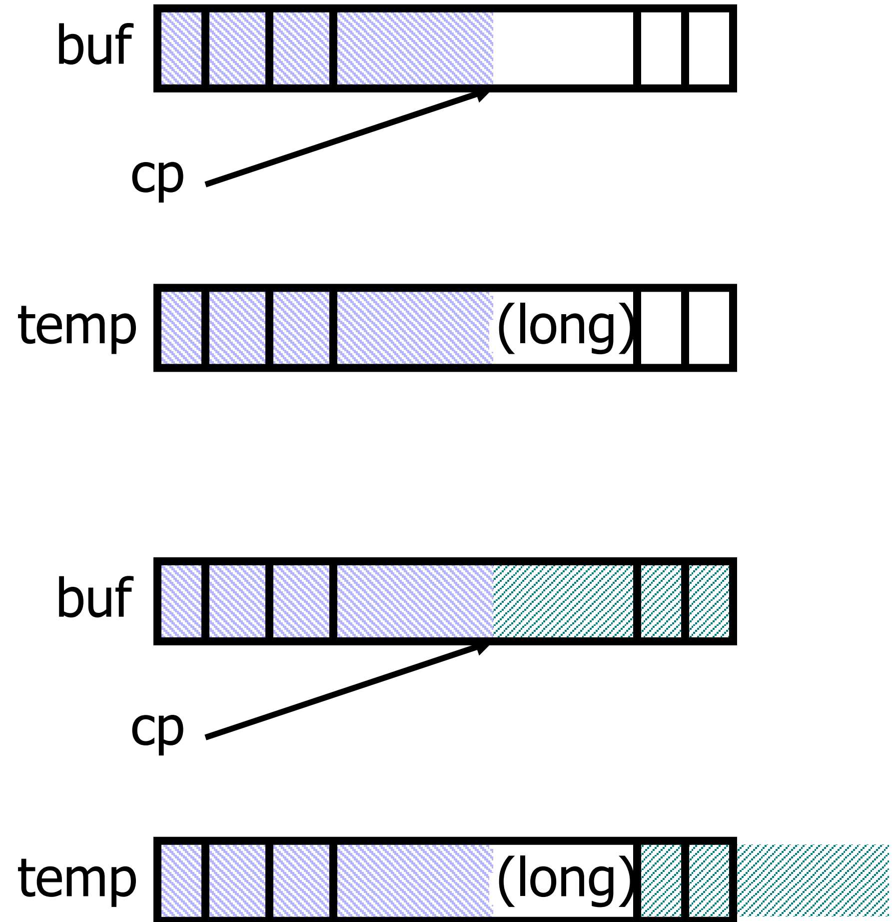
modifies $\quad$ dst.strlen, dst.is_nullt

ensures $\quad$ len(dst) = = pre@len(src) $\wedge$ 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 ;
```

Pointer Analysis

[Dor, Rodeh, Sagiv]

- Goal: compute points-to relation
  - This is highly nontrivial for C programs (see paper)
  - Pointer arithmetic, typeless memory locations, etc.
- Abstract interpretation of memory accesses
  - For each allocation, keep base and size in bytes
  - Map each variable to their abstract locations
  - We'll see something similar in CCured
- Sound approximation of may-point-to
  - For each pointer, set of abstract locations it can point to
  - More conservative than actual points-to relation

C2IP: C to Integer Program

[Dor, Rodeh, Sagiv]

- Integer variables only
- No function calls
- Non-deterministic
- Constraint variables
- Update statements
- Assert statements
  - Any string manipulation error in the original C program is guaranteed to violate an assertion in integer program

(Constraint variables, Update statements, Assert statements:) Based on points-to information

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) )
```

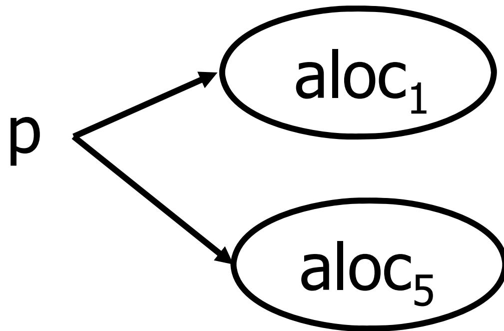
```
p = q + 5;
```

Update statement:

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

Nondeterminism

[Dor, Rodeh, Sagiv]



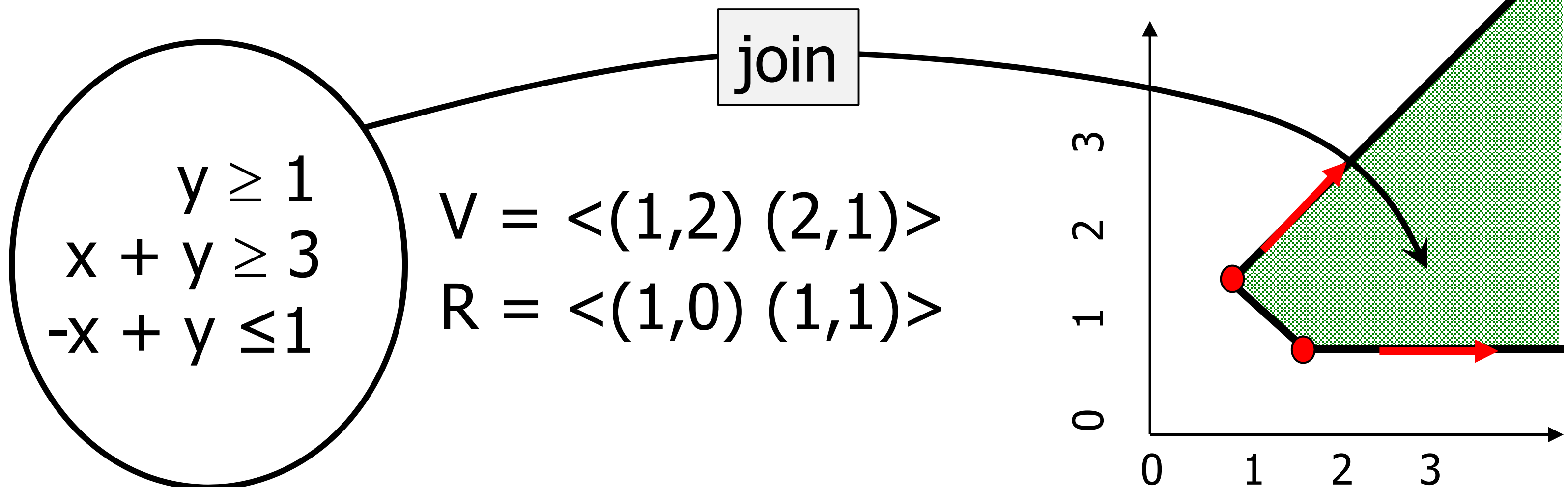
*p = 0;

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

Integer Analysis

[Dor, Rodeh, Sagiv]

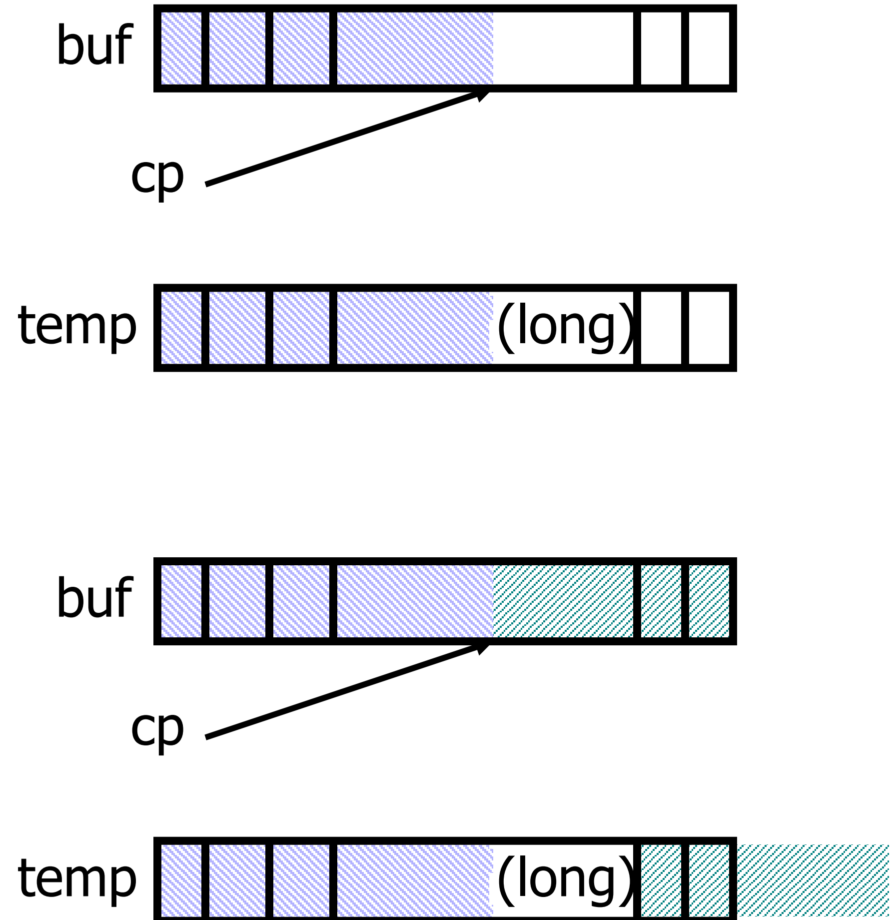
- Interval analysis not enough
  - Loses relationships between variables
- Infer variable constraints using abstract domain of polyhedra [Cousot and Halbwachs, 1978]
  - $a_1 * var_1 + a_2 * var_2 + \ldots + a_n * var_n \leq b$



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]

$buf.offset = 0$

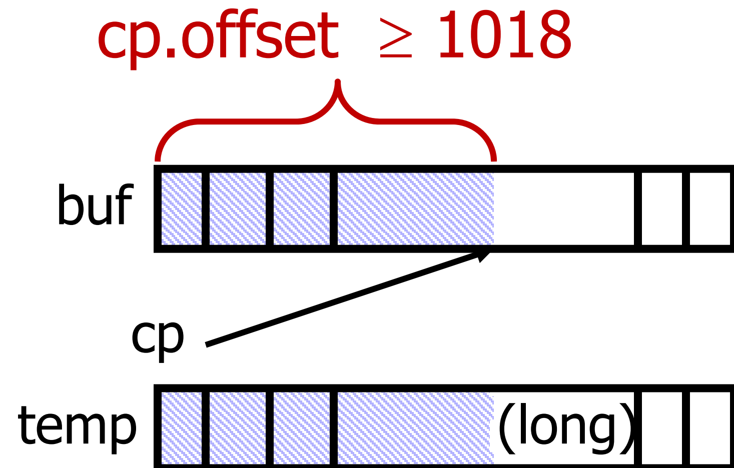
$temp.offset = 0$

$0 \leq cp.offset = i$

$i \leq s_{buf}.len < s_{buf}.msize$

$s_{buf}.msize = 1024$

$s_{temp}.msize = 1024$



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

Potential violation
when $cp.offset \geq 1018$

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 $\Rightarrow$ better performance

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: $||\text{int}|| = N$
  - Safe pointers: $||\tau \text{ ref SAFE}|| = \{ h+i \mid h \in H \text{ and } 0 \le i < \text{size}(h) \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Typed}(\tau)) \}$
  - Sequence pointers: $||\tau \text{ ref SEQ}|| = \{\langle h,n \rangle \mid h \in H \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Typed}(\tau)) \}$
  - Dynamic pointers: $||\text{DYNAMIC}|| = \{\langle h,n \rangle \mid h \in H \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Untyped}) \}$

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
  - $\forall h \in H \backslash \{0\}\ \forall i \in N$
    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
- See the CCured paper for more details

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

Other Static Analysis Tools

- Coverity
- PREfix and PREfast (from Microsoft)
- PolySpace
- Cyclone dialect of C
- Many, many others
  - For example, see http://spinroot.com/static/