# C to Checked C by 3C

**Aravind Machiry**

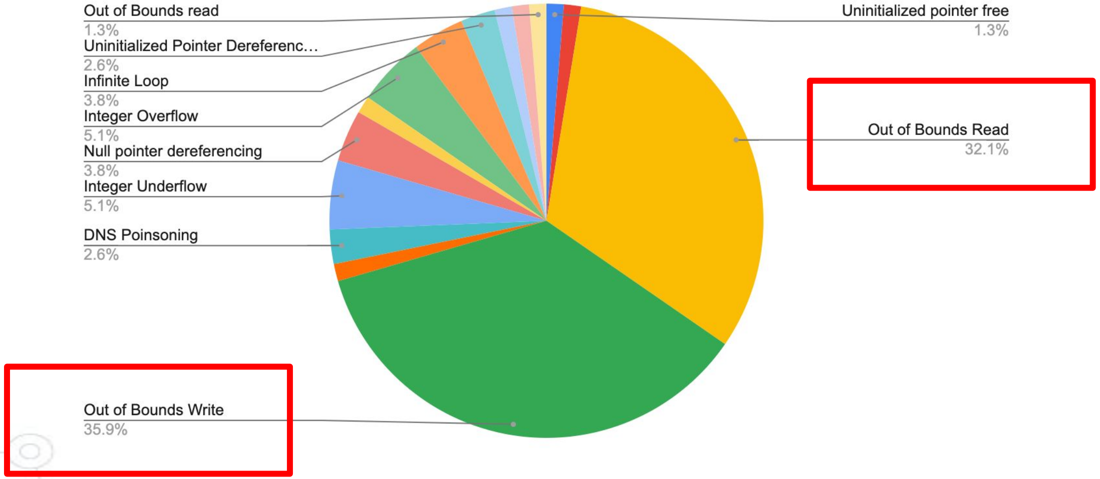
John Kastner , Matt McCutchen , Aaron Eline , Kyle Headley , Michael Hicks

PURDUE UNIVERSITY | Elmore Family School of Electrical and Computer Engineering

amazon (work done prior to starting at Amazon)

# Memory Safety is a Problem (Still!!?)

Vulnerabilities in Embedded Systems (RTOSes) - 68% are spatial violations.



# OpenSSL: CVE-2022-3602 and CVE-2022-3786 (Heap buffer overwrites)

```
diff --git a/crypto/punycode.c b/crypto/punycode.c
index
385b4b1df46a385312c3028c77d17a822200cc70..5e211af6d9
100644 (file)
--- a/crypto/punycode.c
+++ b/crypto/punycode.c
@@ -181,7 +181,7 @@ int ossl_punycode_decode(const c
         n = n + i / (written_out + 1);
         i %= (written_out + 1);

-        if (written_out > max_out)
+        if (written_out >= max_out)
             return 0;

         memmove(pDecoded + i + 1, pDecoded + i,
```

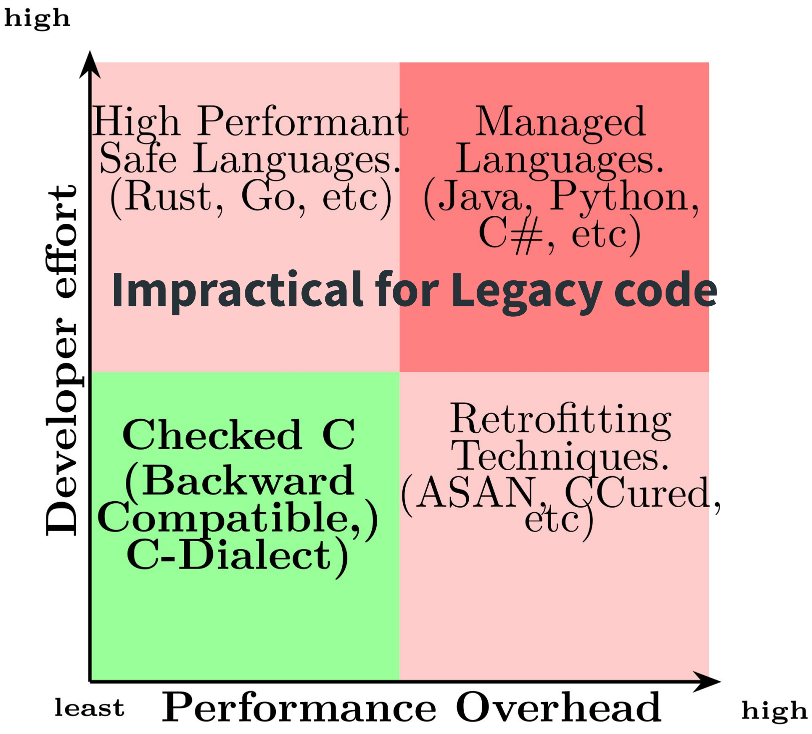
# Heap buffer overread in UEFI driver

Spatial Bugs can be hard to spot.

```
typedef struct {
  UINT8 Len;
  UINT8 Type;
} USB_DESC_HEAD;

void *UsbCreateDesc(UINT8 *Buf, UINTN Len, UINT8 Type){
  UINTN Offset = 0;
  USB_DESC_HEAD* Head = (USB_DESC_HEAD*)Buf;
  while ((Offset < Len) && (Head->Type != Type)) {
    Offset += Head->Len;
    Head    = (USB_DESC_HEAD*)(Buf + Offset);
  }
  DoSomethingUseful(Head);
}
```

# Existing Solutions v/s Checked C

- Checked C
  - Safe-dialect of C.
  - Very Low memory and performance overhead.
  - Fast and backward compatible.



## Checked C Pointer Types

- `_Ptr<T>`
  - Pointer to a singleton object of type T
- `_Array_ptr<T>`
  - Pointer to an array of type T
- `_Nt_array_ptr<T>`
  - Pointer to a null-terminated (ends with ‘\0’) array of type T

**Safety Guarantees:**

Pointers will be checked for safe usage (mostly) at compile time.

May insert run-time bounds checks (**but no extra metadata**)

# _Ptr<T>

Points to *a single memory object* of type T - *no pointer arithmetic or subscript operator (e.g., x[0]) allowed.*

```
struct Data {
  int val;
  long lval;
  ...
};
```

```
_Ptr<struct Data> p = malloc(sizeof(struct Data));
...
printf("val = %d\n", p->val);
...
p++;
```

# _Ptr<T>

Points to *a* *single memory object* of type T - *no pointer arithmetic or subscript operator (e.g., x[0]) allowed.*

```
struct Data {
  int val;
  long lval;
  ...
};
```

```
_Ptr<struct Data> p = malloc(sizeof(struct Data));
...
printf("val = %d\n", p->val);
...
p++;
```

**Compiler ensures through runtime checks that p is not NULL**

# _Ptr<T>

Points to *a <u>single memory object</u>* of type T - *<u>no pointer arithmetic or subscript operator (e.g., x[0]) allowed.</u>*

```
struct Data {
  int val;
  long lval;
  ...
};
```

```
_Ptr<struct Data> p = malloc(sizeof(struct Data));
...
printf("val = %d\n", p->val);
...
p++;
```

**Compiler ensures through runtime checks that p is not NULL**

**error: arithmetic on _Ptr type.**

# _Array_ptr<T>

Points to an array of elements of type T — *permits pointer arithmetic* and subscripting.

- Bounds declared by programmers are checked at runtime when not provable at compile time.

# _Array_ptr<T>

Points to an array of elements of type T — *permits pointer arithmetic and subscripting.*

- Bounds declared by programmers are checked at runtime when not provable at compile time.

```
_Array_ptr<int> p = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

# _Array_ptr<T>

Points to an array of elements of type T — *permits pointer arithmetic* and subscripting.

- Bounds declared by programmers are checked at runtime when not provable at compile time.

```
_Array_ptr<int> p = malloc(sizeof(int) * BUF_LEN);
```

```
int i = p[5];
```

**error: expression has unknown bounds.**

```
= p[5];
  ^~~~
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

→ No dynamic checks inserted as BUF_LEN > 5

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

No dynamic checks inserted as BUF_LEN > 5

```
int j = p[30];
```

**error:** out-of-bounds access.

```
= p[30];
  ^~~~
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
int j = p[30];
int k = p[var];
```

`int i = p[5];` → No dynamic checks inserted as BUF_LEN > 5

`int j = p[30];` → error: out-of-bounds access.

```
= p[30];
  ^~~~
```

`int k = p[var];` → NULL ptr and bounds check inserted if var < BUF_LEN is not provable.

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

No dynamic checks inserted as BUF_LEN > 5

```
int j = p[30];
```

**error:** out-of-bounds access.

```
= p[30];
   ^~~~
```

```
int k = p[var];
```

NULL ptr and bounds check inserted if var < BUF_LEN is not provable.

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);

int l = p[15];
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

→ No dynamic checks inserted as BUF_LEN > 5

```
int j = p[30];
```

→ **error:** out-of-bounds access.

```
= p[30];
  ^~~~
```

```
int k = p[var];
```

→ NULL ptr and bounds check inserted if var < BUF_LEN is not provable.

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);

int l = p[15];
```

→ **error:** out-of-bounds access.

```
= p[15];
  ^~~~
```

Because p's bound are [p, p+15)

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

No dynamic checks inserted as BUF_LEN > 5

```
int j = p[30];
```

**error:** out-of-bounds access.

```
= p[30];
   ^~~~
```

```
int k = p[var];
```

NULL ptr and bounds check inserted if var < BUF_LEN is not provable.

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);

int l = p[15];
```

**error:** out-of-bounds access.

```
= p[15];
   ^~~~
```

Because p’s bound are [p, p+15)

```
_Array_ptr<int> p : count(BUF_LEN + 1) = malloc(sizeof(int) * BUF_LEN);
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : count(bounds_expr) = ...;
```

```
const unsigned int BUF_LEN=30;
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);
```

`int i = p[5];` → No dynamic checks inserted as BUF_LEN > 5

`int j = p[30];` → **error:** out-of-bounds access.

```
= p[30];
  ^~~~
```

`int k = p[var];` → NULL ptr and bounds check inserted if var < BUF_LEN is not provable.

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);
```

`int l = p[15];` → **error:** out-of-bounds access.

```
= p[15];
  ^~~~
```

**Because p's bound are [p, p+15)**

```
_Array_ptr<int> p : count(BUF_LEN + 1) = malloc(sizeof(int) * BUF_LEN);
```

**error:** declared bounds for 'p' are invalid after initialization.

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : bounds(l_bound, u_bound) = ...;
```

```
#define BUF_LEN=30;
_Array_ptr<int> p0 : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

_Array_ptr<int> p1 : bounds(p0, p0 + BUF_LEN/2) = p0;
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : bounds(l_bound, u_bound) = ...;
```

```
#define BUF_LEN=30;
_Array_ptr<int> p0 : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);


_Array_ptr<int> p1 : bounds(p0, p0 + BUF_LEN/2) = p0;

int i = p1[15];
```

# Bounds declaration for _Array_ptr<T>

```
_Array_ptr<int> p : bounds(l_bound, u_bound) = ...;
```

```
#define BUF_LEN=30;
_Array_ptr<int> p0 : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);


_Array_ptr<int> p1 : bounds(p0, p0 + BUF_LEN/2) = p0;

int i = p1[15];
```

**error: out-of-bounds access.**

```
= p1[15];
   ^~~~
```

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = "12345";
```

**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```

**Equivalent to declaring Bounds of p as "p: bounds(p, p+1)"**

```
char c = p[0];
```

**OK.**

```
char c = p[1];
```

**error: out-of-bounds access.**

```
= p[1];
   ^~~~
```

```
if (*p == 'a') {
 if (*(p + 1) == 'b') {
  if (*(p + 3) == 'd') {
```

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```
**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as “p: bounds(p, p+1)”**

```
char c = p[0];
```
**OK.**

```
char c = p[1];
```
**error: out-of-bounds access.**

```
 = p[1];
    ^~~~
```

```
if (*p == 'a') {
 if (*(p + 1) == 'b') {
  if (*(p + 3) == ‘d') {
```
**OK. P’s bound has been widened to [p, p+2).**

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = "12345";
```
**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as "p: bounds(p, p+1)"**

```
char c = p[0];
```
**OK.**

```
char c = p[1];
```
**error: out-of-bounds access.**

```
 = p[1];
   ^~~~
```

```
if (*p == 'a') {
```
**OK. P's bound has been widened to [p, p+2).**

```
 if (*(p + 1) == 'b') {
```
**OK. P's bound has been widened to [p, p+3).**

```
  if (*(p + 3) == 'd') {
```

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```
OK. p0 is pointing to array of 6 chars (including NULL).

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as “p: bounds(p, p+1)”**

```
char c = p[0];
```
OK.

```
char c = p[1];
```
**error: out-of-bounds access.**

```
 = p[1];
     ^~~~
```

```
if (*p == 'a') {
```
**OK. P’s bound has been widened to [p, p+2).**

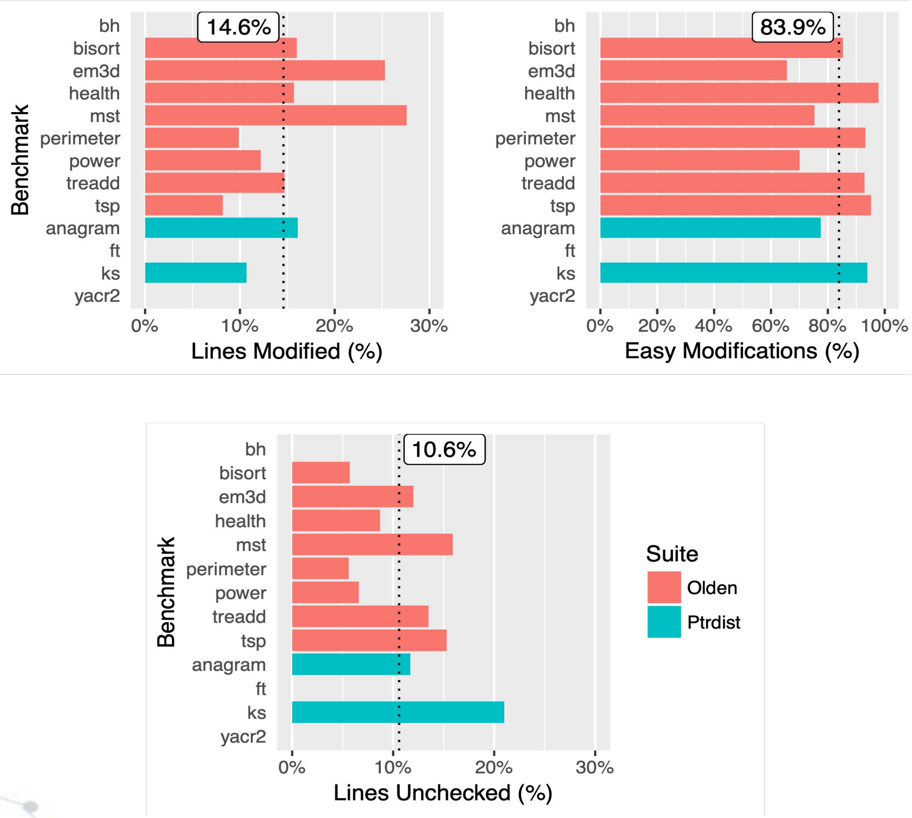
```
 if (*(p + 1) == 'b') {
```
**OK. P’s bound has been widened to [p, p+3).**

```
  if (*(p + 3) == ‘d') {
```
**error: out-of-bounds access.**
**Because, P’s bound is [p, p+3).**

## Other Checked C Features

- `itype`:
  - Interface types enabling passing checked or unchecked types.
- `checked` / `unchecked` scopes:
  - Enable disabling strict checks across code regions.
- `dynamic_bounds_cast`:
  - Force dynamic checks.
- assume_bounds_cast:
  - Force conversion from unchecked pointers to checked pointers.

# Converting C to Checked C



## Converting C to Checked C

# Refactoring the FreeBSD Kernel with Checked C

Junhan Duan,* Yudi Yang,* Jie Zhou, and John Criswell
Department of Computer Science
University of Rochester

It took two undergraduate students approximately three months to refactor the system calls, the network packet (mbuf) utility routines, and parts of the IP and UDP processing code. Our experiments show that using Checked C incurred no performance or code size overheads.

*Index Terms*—memory safety, safe C, FreeBSD

# The Problem

- Conversion of C to Checked C requires quite a bit of effort.
  - 14% lines modified.
- Can we automate the conversion?
  - Rewrite code with Checked types.

# Not all Pointers can be converted to Checked Types

Incompatible Casts:

```
m = (int *)0x7869000;

x = (char *)get_data(..);
```

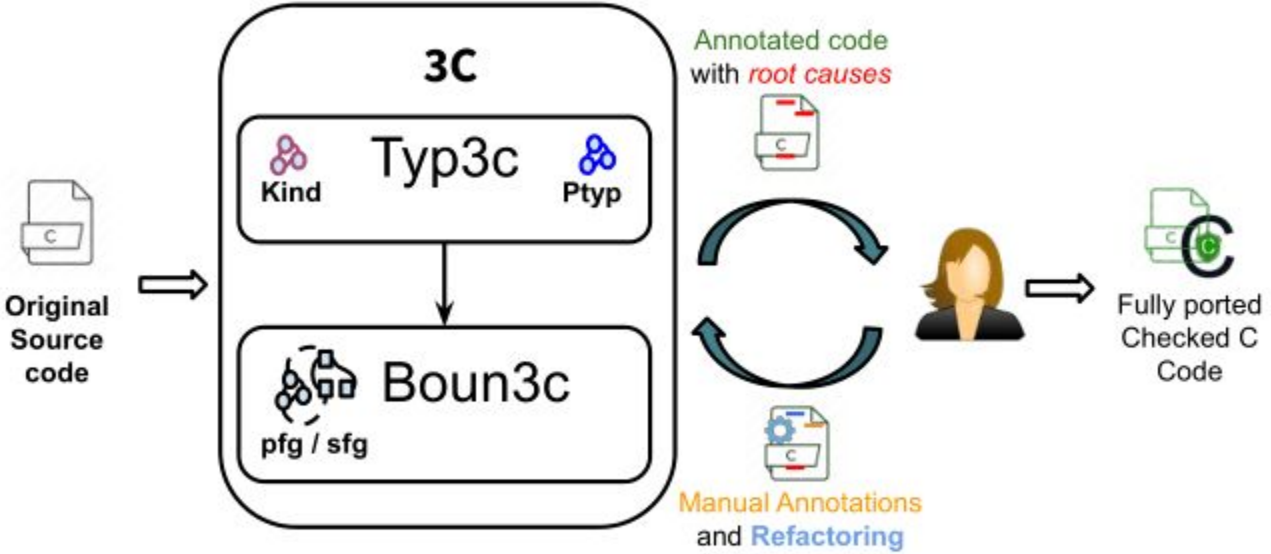
These pointers cannot be converted and should remain as regular pointers (WILD).

# Infeasibility of Automated Conversion

```c
int resize_buf(char **buf, unsigned *sz) {
 char *newbuf = NULL;
 unsigned news = round_up(*sz, 64);
 newbuf = realloc(buf, news);
 *buf = newbuf;
 *sz = news;
 return newbuf != NULL;
}
```

```c
typedef struct {
  array_ptr<char> buf : count(sz);
  unsigned sz;
} SIZEBUF;

int resize_buf(ptr<SIZEBUF> buf) _Checked {
  unsigned news = round_up(buf->sz, 64);
  array_ptr<char> newbuf : count(news) = NULL;
  newbuf = realloc<char>(buf->buf, news);
  buf->buf = newbuf;
  buf->sz = news;
  return newbuf != NULL;
}
// Refactor all callers of resize_buf to
// use the new caller.
```

# Our Approach: 3c



# Infer Checked Types for Pointers

- **typ3c:** Infer Checked base types.
  - Ptr, Array_ptr (arr), Nt_array_ptr (ntarr)

- **boun3c:** Infer bounds association for arr and ntarr types.
  - count(..), byte_count(..), bounds(...)

# typ3c: Infer Checked Base Types for Pointers

- Basic Idea: Type qualifier inference.

```
int *foo(int **p) {
  int *r;
  int *m;
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x;
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

# Type qualifier inference

Step 1: Create qualifier variables for each pointer

```
int *foo(int **p) {
  int *r;
  int *m;
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

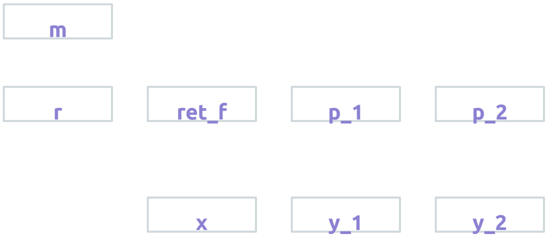
```
bar() {
  int *x;
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

# Type qualifier inference

## Step 1: Create qualifier variables for each pointer

```
  ret_f           p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
      y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

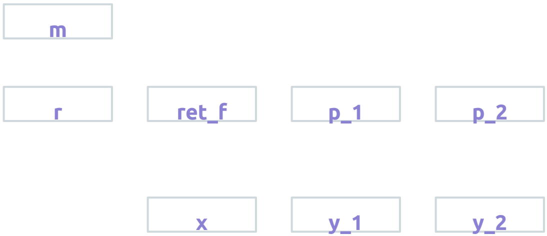


# Type qualifier inference

Step 2: Create constraints based on pointer usage.

```
  ret_f              p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
      y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



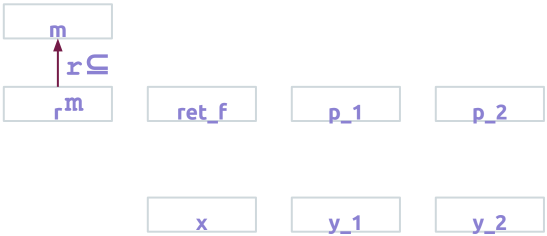
# Type qualifier inference

Step 2: Create constraints based on pointer usage.

```
  ret_f         p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
      y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

**The type of r should be subtype of m.**

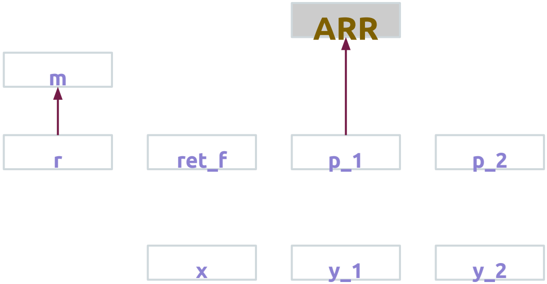


# Type qualifier inference

Step 2: Create constraints based on pointer usage.

```
    ret_f          p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
       y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



# Type qualifier inference

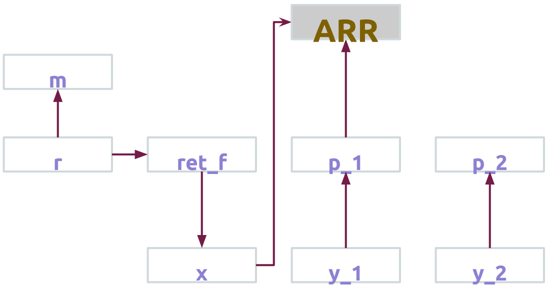
## Step 3: Solve based on typing rules.

**NTARR** ⊆ **ARR** ⊆ **PTR**

Most Specific Type — Most General Type

```
  ret_f          p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
       y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



# Type qualifier inference

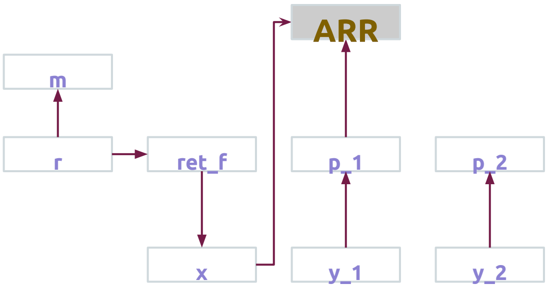
## Step 3: Solve based on typing rules.

**NTARR** ⊆ **ARR** ⊆ **PTR**

Most Specific Type — Most General Type

```
  ret_f       p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```




We want **the greatest solution -- most general type**.

i.e., i ⊆ j will result in Type(i) = Type(j)

# Type qualifier inference

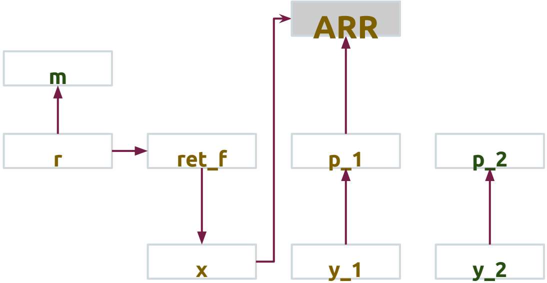
## Step 3: Solve based on typing rules.

NTARR ⊆ ARR ⊆ PTR

Most Specific Type — Most General Type

```
  ret_f
int *foo(int **p) {        p_1 p_2
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
  int **y;    y_1 y_2
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```




**Greatest Solution:**

ret_f, x, r, p_1, y_1 => **ARR**

m, y_2, p_2 => **PTR**

# The Problem of Wildfire.

Regular Qualifier Inference will propagate WILDness unnecessarily across function boundaries.

```
int deref(int *y) { return *y; }

int bar(void) {
  int *p = (int *)5;
  deref(p);
}
```

`y` will be unnecessary made WILD although **it is used safely** in `deref`

# Handling unchecked (WILD) Pointers in typ3c

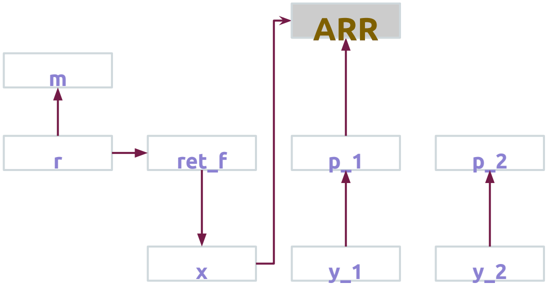
Localizing Wildness:

- A pointer is wild if its used unsafely within the corresponding function.
  - E.g.,
    - Parameter is WILD if used unsafely with in the function.
    - Return type is WILD if the function returns an unsafe type.

# Isolation Across Function Boundaries Does Not Work!

```
  ret_f           p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
    ...
  return r;
}
```

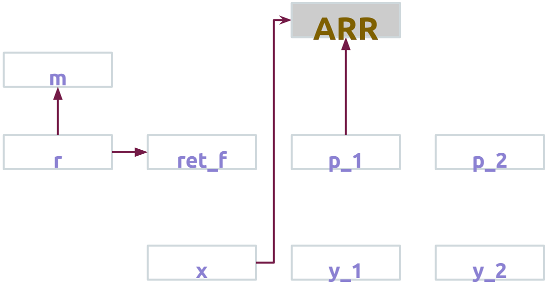
```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



# Isolation Across Function Boundaries Does Not Work!

```
  ret_f         p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

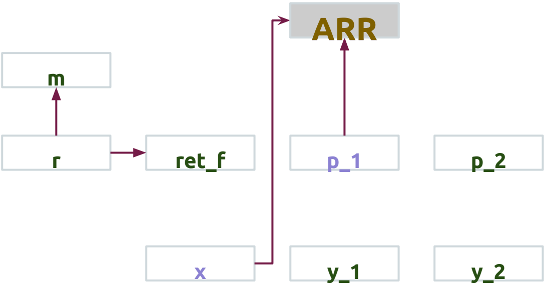
```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



# Isolation Across Function Boundaries Does Not Work!

```
  ret_f              p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```




**Actual:**

x, p_1 => **ARR**

ret_f, r, m, y_1, y_2, p_2 => **PTR**

**Expected:**

ret_f, x, r, p_1, y_1 => **ARR**

m, y_2, p_2 => **PTR**

## Conflicting Requirements!

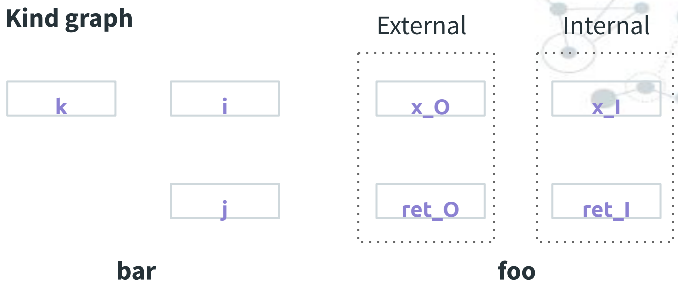
- Need to isolate function boundaries to localize WILDness.
- Should not isolate to infer correct Checked types.

# Our solution: Create two graphs (kind and ptype graph)

- kind graph: Isolate function boundaries.
  - Whether a pointer is checked or wild.
    - `chk` ⊆ `WILD`
- ptype graph: Regular qualifier inference.
  - ptr, arr, ntarr: `NTARR` ⊆ `ARR` ⊆ `PTR`
- Two nodes for pointer parameters and returns.
  - Internal (Inside the function).
  - External (For the callers).

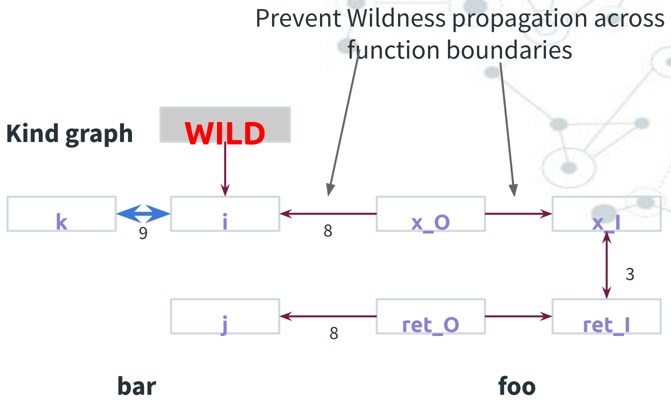
# kind graph

```c
int *foo(int *x) {
  ...
  return x;
}

bar () {
int *i, *j, *k;
i = (int *)0xff86763;
j = foo(i);
k = i;
j[0] = ...
}
```



# kind graph

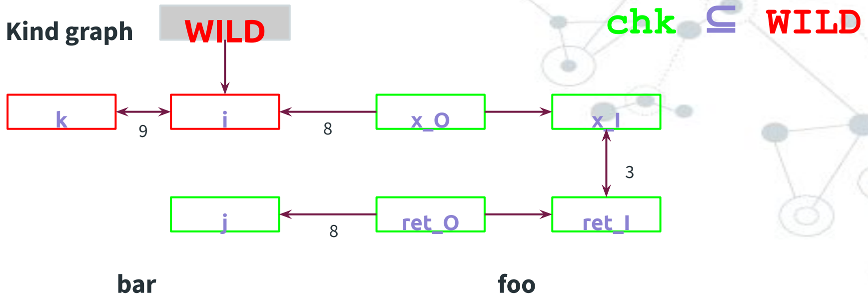
```
1: int *foo(int *x) {
2:   ...
3:   return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```




- Flow sensitive only across function boundaries to avoid propagating wildness.
- **Flow insensitive** within the function.

# kind graph

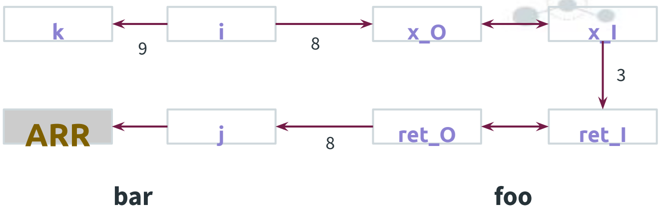
```
1: int *foo(int *x) {
2:  ...
3:  return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```



# ptype graph

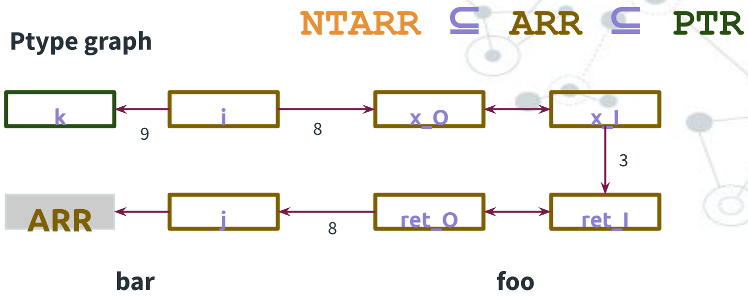
```
1: int *foo(int *x) {
2:   ...
3:   return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```

**Ptype graph**



# ptype graph

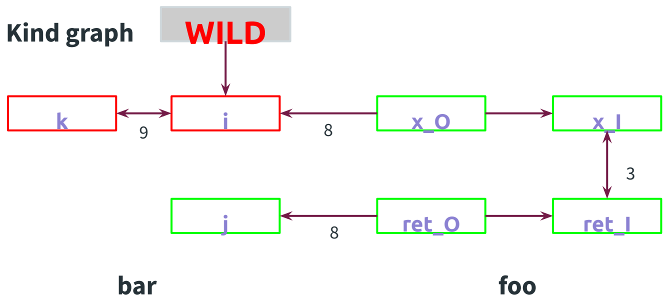
```
1: int *foo(int *x) {
2:   ...
3:   return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```

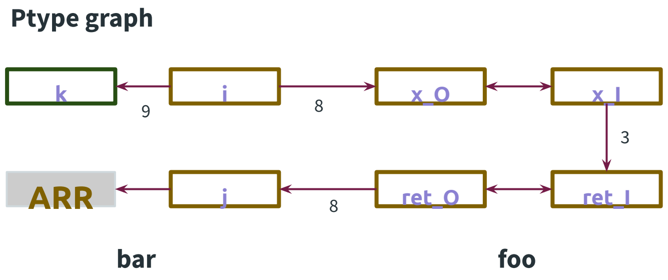


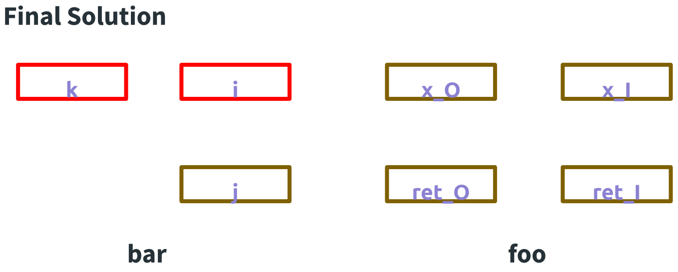
# Merging Kind and Ptype solutions

- if kind(p) = `chk` then checked type is ptype(p).
- if kind(p) = `WILD` then regular pointer.

# Merging Kind and Ptype solutions



## Infer Checked Types for Pointers

- **typ3c:** Infer Checked base types.
  - Ptr, Array_ptr (arr), Nt_array_ptr (ntarr)
- **boun3c:** Infer bounds association for arr and ntarr types.
  - count(..), byte_count(..), bounds(...)

# Inferring Bounds for arr and ntarr pointers

- A variant of correlation analysis [1].

- Associate each pointer (arr and ntarr) with a possible bound, and then propagate that association consistently.

[1] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. Existential Label Flow Inference via CFL Reachability.

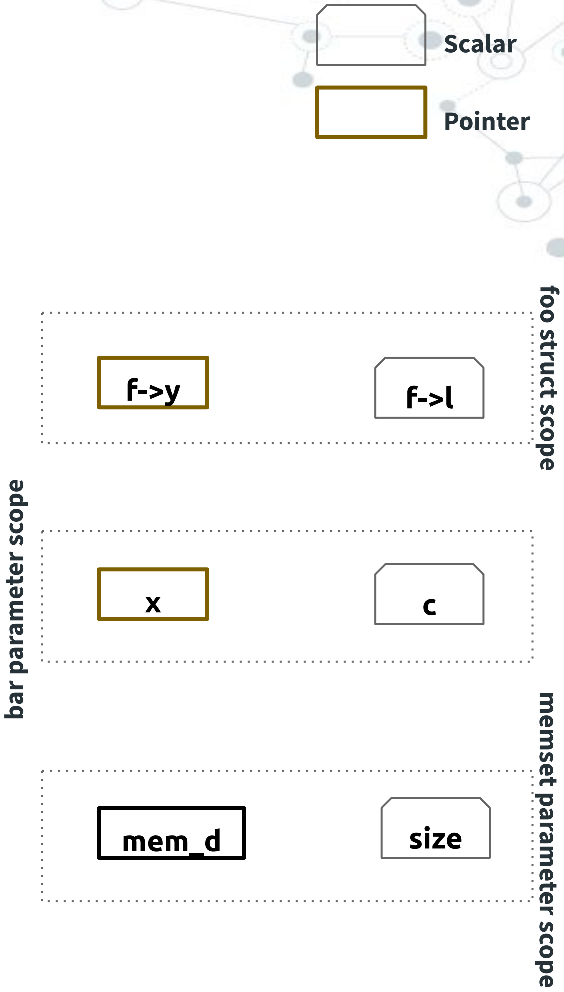
## Boun3c: Bounds Inference

- Keep track of pointer flows and variable flows:
    - pointer-flow-graph (pfg) => A version of ptype graph.
    - scalar-flow-graph (sfg) => Flow of scalar variables.

- Step 1: Figure out seed bounds:
    - Constant array. e.g., `int arr[10]`
        - Bounds => count (10)
    - Allocation using malloc. e.g., `p = (struct foo*) malloc(n*sizeof(struct foo));`
        - Bounds => count(n)
    - others.

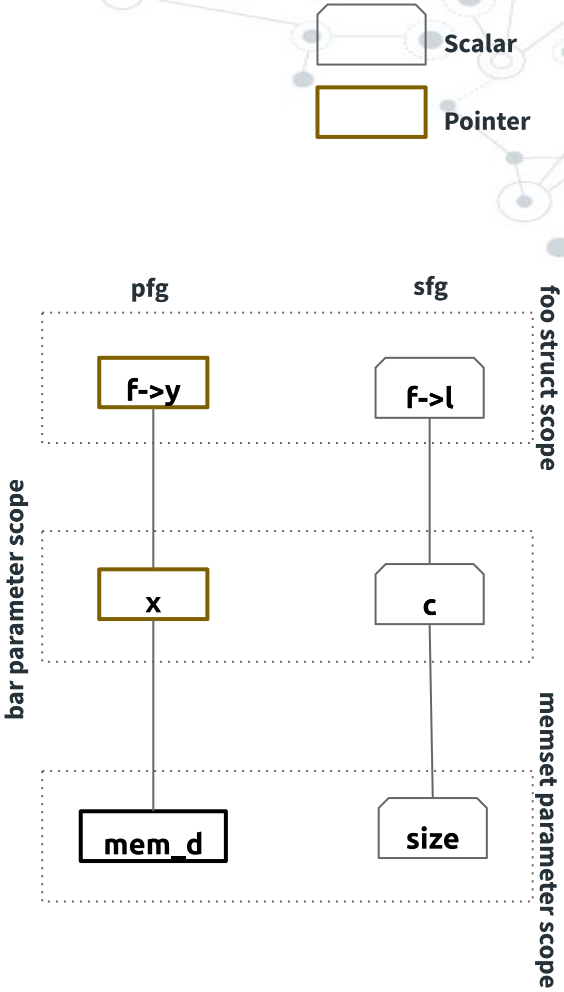
# Boun3c: Example

```
struct foo {
 int *y; ARR
 int l;
};

             ARR
void bar(int *x, int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



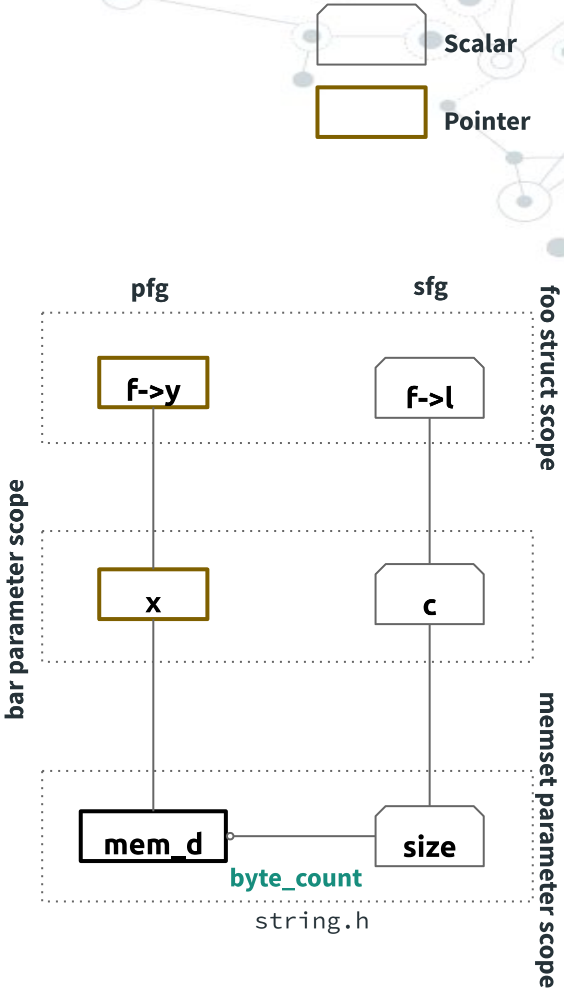
# Boun3c: Example

```
struct foo {
 int *y; ARR
 int l;
};

           ARR
void bar(int *x, int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



# Boun3c: Adding Seed Bounds

```
struct foo {
 int *y; ARR
 int l;
};

            ARR
void bar(int *x, int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



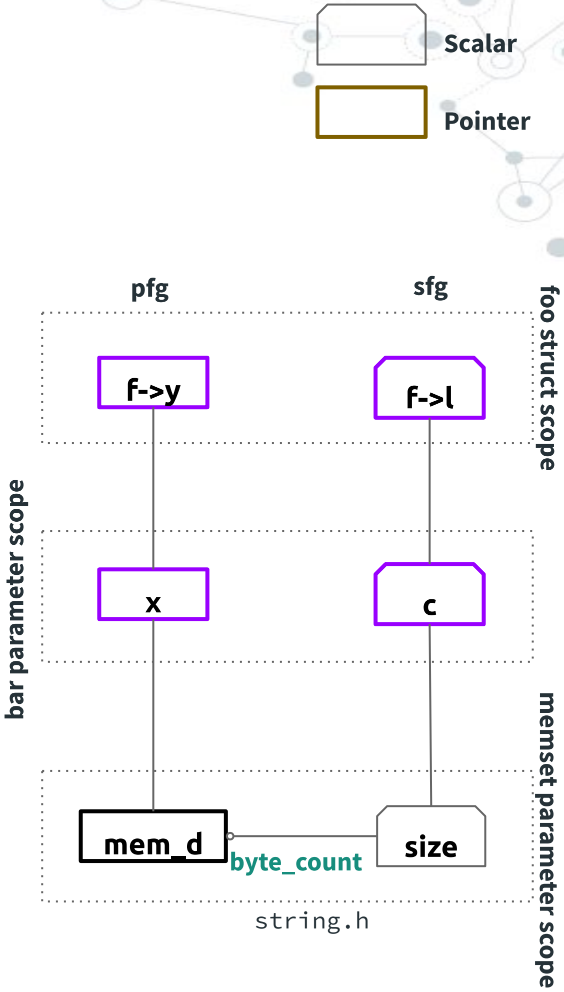
## Boun3c: Bounds Inference

- Step 2: Propagate seed bounds context-sensitively to other pointers via pfg and sfg.

- Identify a common variable to which all incoming pointers' bounds propagate to and pick that variable as the bounds.
    - The variable should be in the same program scope as the pointer.

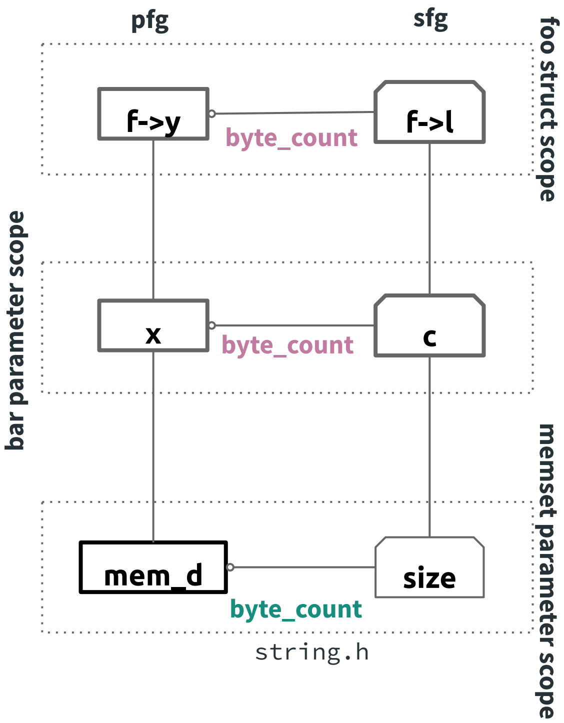
# Boun3c: Propagate Bounds

```
struct foo {
 int *y; ARR
 int l;
};

              ARR
void bar(int *x, int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



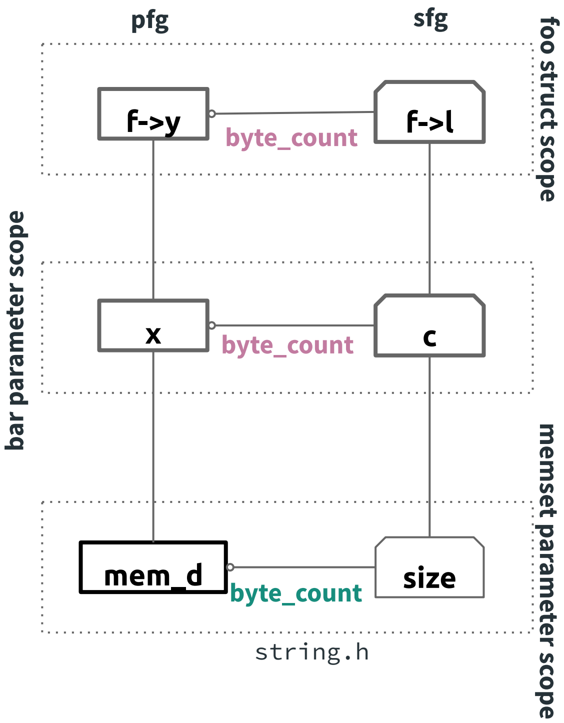
# Boun3c: Find Common Bounds


```
struct foo {
 int *y; ARR
 int l;
};

              ARR
void bar(int *x, int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



# Boun3c: Rewrite and Add Bounds


```
struct foo {
 array_ptr<int> y : byte_count(l);
 int l;
};

void bar(array_ptr<int> x : byte_count(c),
         int c) {
 struct foo f = { x, c };
 memset(x, 1, c);
 x[0] = 0;
}
```



# Boun3c: Need for Context-Sensitivity

Confusion among structure fields.

```
struct obj {
  int *buf;
  unsigned idx;
  unsigned n;
}

func (..) {
 struct obj o1, o2;
 ...
 int *b = malloc(s*sizeof(int));
 o1.buf = b;
 o2.idx = s;
 o1.n = s;
}
```

```
struct obj {
  _Array_ptr<int> buf : count(idx);
  unsigned idx;
  unsigned n;
}
```

# Boun3c: Context Sensitive Bounds Propagation

```
struct obj {
  int *buf;
  unsigned idx;
  unsigned n;
}

func (..) {
 struct obj o1, o2;
 ...
 int *b = malloc(s*sizeof(int));
 o1.buf = b;
 o2.idx = s;
 o1.n = s;
}
```

**Context-Insensitive**

```
struct obj {
  _Array_ptr<int> buf : count(idx);
  unsigned idx;
  unsigned n;
}
```

**Context-Sensitive**

```
struct obj {
  _Array_ptr<int> buf : count(n);
  unsigned idx;
  unsigned n;
}
```

# Boun3c: Heuristics

- Consistent Upper Bound (CUB):

```
if (i<n)
  p[i] = ..
```

count(n)

```
if (j>=k)
  return -1;
p[j] = ..
```

count(k)

```
for (i=0; i< n; i++)
  ...
  arr[i] = ..;
```

count(n)

# Boun3c: Heuristics

- Consistent Upper Bound (CUB):

```
if (i<n)
  p[i] = ..
```

count(n)

```
if (j>=k)
  return -1;
p[j] = ..
```

count(k)

```
for (i=0; i< n; i++)
  ...
  arr[i] = ..;
```

count(n)

- Name Prefix (NPr):

```
struct foo {
 int *p;
 unsigned p_len;
}
```

count(p_len)

# Boun3c: Heuristics

- Consistent Upper Bound (CUB):

```
if (i<n)
  p[i] = ..
```

⇩ count(n)

```
if (j>=k)
  return -1;
p[j] = ..
```

⇩ count(k)

```
for (i=0; i< n; i++)
  ...
  arr[i] = ..;
```

⇩ count(n)

- Name Prefix (NPr):

```
struct foo {
 int *p;
 unsigned p_len;
}
```

⇨ count(p_len)

- Next Parameter (NePa):

```
foo(int *arr, int n)
```

⇨ count(n)

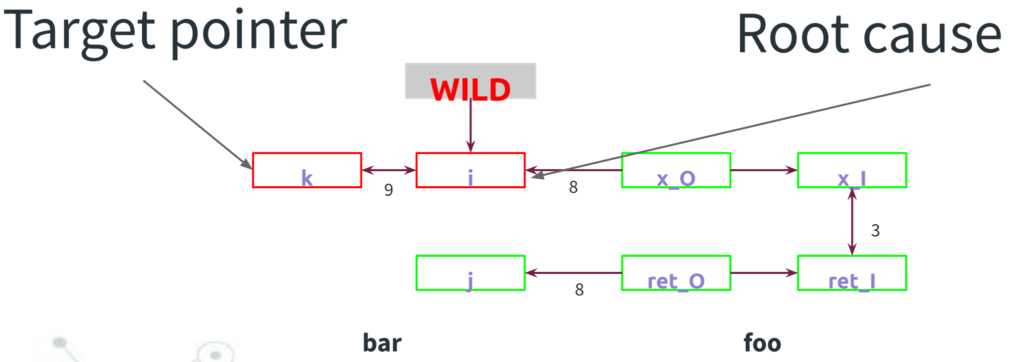
# Consistent Upper Bound (CUB): vsftpd Example

```
struct bin_to_ascii_ret
vsf_ascii_bin_to_ascii(
 const char *p_in,
 ...,
 unsigned int in_len,...) {
  ➔while (indexx < in_len) {
   char the_char = p_in[indexx];
   ...
  }
}
```

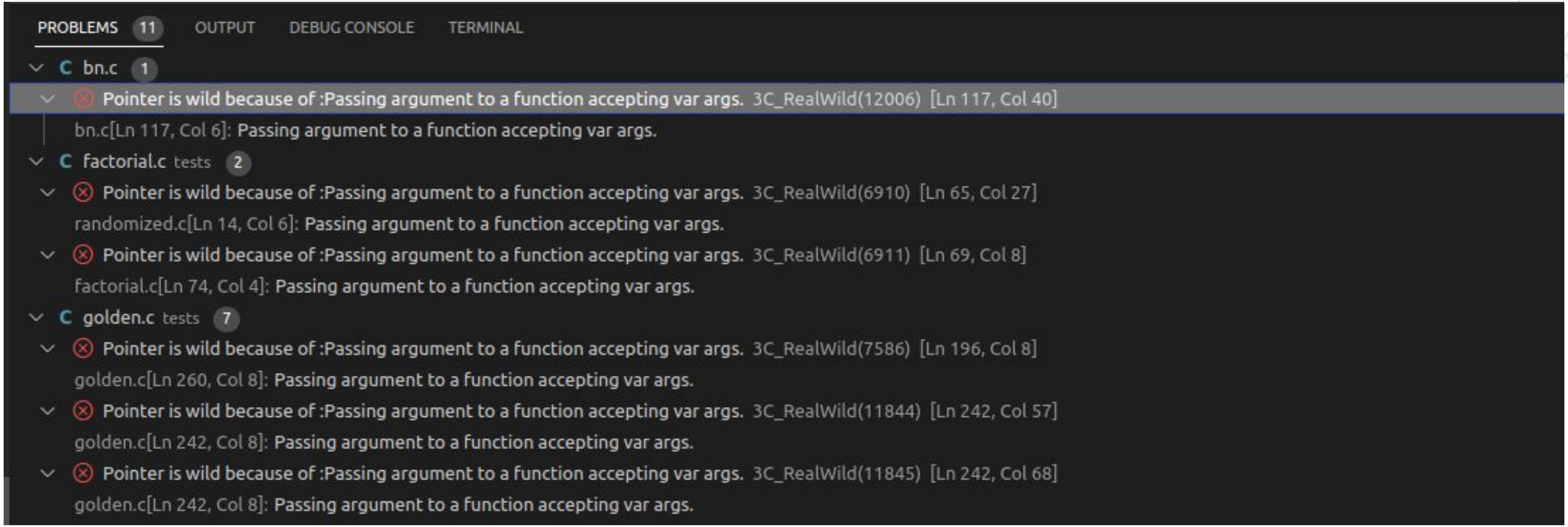
```
struct bin_to_ascii_ret
vsf_ascii_bin_to_ascii(
 (✔CUB)_Array_ptr<const char> p_in : count(in_len),
 ...,
 unsigned int in_len,...)
```

## Interactivity

- For pointers which we are unable to convert:
    - Find the root cause and ask developer.



# Interactivity



# Interactivity

```
char op;
char buf[8192];
```

Make this pointer non-WILD and apply the same observation to all the pointers.

Make ONLY this pointer non-WILD.

```
printf("\nRunning \"golden\" tests (parsed using from_int):\n\n");
```

# Evaluation: Dataset

Programs suggested by Checked C team.

| Program | Category | Size (SLOC) | Total Ptrs (*TP*) | Num Files (`.c` & `.h`) |
|---|---|---|---|---|
| vsftpd | FTP Server | 14.7K | 1,765 | 78 |
| icecast | Media Server | 18.2K | 2,682 | 72 |
| lua | Interpreter | 19.4K | 4,176 | 57 |
| olden | Data-structure benchmark | 10.2K | 832 | 51 |
| parson | Json parser | 2.5K | 686 | 3 |
| ptrdist | Pointer-use benchmark | 9.3K | 920 | 39 |
| zlib | Compression Library | 21.3K | 647 | 25 |
| libtiff | Image Library | 68.2K | 3,478 | 43 |
| libarchive | Archiving library | 146.8K | 10,269 | 149 |
| thttpd | HTTP Server | 7.6K | 829 | 18 |
| tinybignum | Integer Library | 1.4K | 129 | 7 |
| **Total** | | 319.6K | 26,413 | 542 |

# Evaluation: Runtime

Time taken by automated conversion.

< 1 min

| Program | Setup | Constraints Building | Constraints Solving | Bounds Inference | Rewriting | RC Comp | Total Time (s) |
|---|---|---|---|---|---|---|---|
| vsftpd | 1.06 (35.7%) | 0.45 (15.0%) | 0.18 (6.1%) | 0.23 (7.9%) | 0.94 (31.8%) | 0.07 (2.5%) | 2.96 |
| icecast | 6.41 (46.9%) | 1.2 (8.8%) | 1.46 (10.7%) | 1.05 (7.7%) | 2.66 (19.5%) | 0.63 (4.6%) | 13.66 |
| lua | 2.19 (37.7%) | 0.98 (16.9%) | 0.38 (6.6%) | 0.76 (13.0%) | 1.25 (21.5%) | 0.16 (2.8%) | 5.81 |
| olden | 1.57 (63.4%) | 0.23 (9.5%) | 0.22 (9.1%) | 0.1 (4.2%) | 0.25 (10.1%) | 0.06 (2.5%) | 2.47 |
| parson | 0.21 (39.3%) | 0.12 (23.6%) | 0.03 (5.1%) | 0.09 (17.2%) | 0.06 (12.1%) | 0.02 (4.6%) | 0.53 |
| ptrdist | 1.12 (54.2%) | 0.34 (16.3%) | 0.19 (9.3%) | 0.14 (6.6%) | 0.21 (10.4%) | 0.05 (2.5%) | 2.06 |
| zlib | 0.87 (47.3%) | 0.44 (23.9%) | 0.11 (6.2%) | 0.12 (6.5%) | 0.21 (11.7%) | 0.05 (2.5%) | 1.83 |
| libtiff | 3.26 (38.1%) | 1.58 (18.5%) | 0.65 (7.6%) | 0.75 (8.8%) | 1.6 (18.7%) | 0.55 (6.4%) | 8.55 |
| libarchive | 14.63 (33.5%) | 4.02 (9.2%) | 2.91 (6.6%) | 6.2 (14.2%) | 13.94 (31.9%) | 1.31 (3.0%) | 43.74 |
| thttpd | 1.02 (42.2%) | 0.51 (21.1%) | 0.15 (6.1%) | 0.36 (14.7%) | 0.28 (11.6%) | 0.06 (2.6%) | 2.41 |
| tinybignum | 0.24 (52.6%) | 0.07 (14.3%) | 0.06 (12.9%) | 0.02 (4.4%) | 0.06 (12.2%) | 0.01 (3.1%) | 0.46 |
| **Total** | 32.57 (38.5%) | 9.93 (11.8%) | 6.34 (7.5%) | 9.82 (11.6%) | 21.47 (25.4%) | 2.99 (3.5%) | 84.49 |

**Most time spent in compiler frontend**

# Evaluation: Typ3c

**Without separating external and internal**

**Without isolating function boundaries**

| Program | Total Pointers (*TP*) | Checked pointers (*chk*) (% of *TP*) | | | Split of Identified Checked Pointer Types (% of typ3c) | | |
|---|---|---|---|---|---|---|---|
| | | typ3c | $\text{typ3c}^f$ | CCured | ptr | arr | ntarr |
| vsftpd | 1,765 | **1,336 (75.7%)** | 1,226 (69.5%) | 999 (56.6%) | **1,199 (89.7%)** | 44 (3.3%) | 93 (7.0%) |
| icecast | 2,682 | **1,795 (66.9%)** | 1,670 (62.3%) | 1,377 (51.3%) | **1,429 (79.6%)** | 54 (3.0%) | 312 (17.4%) |
| lua | 4,176 | **2,781 (66.6%)** | 2,248 (53.8%) | 1,771 (42.4%) | **2,273 (81.7%)** | 254 (9.1%) | 254 (9.1%) |
| olden | 832 | **721 (86.7%)** | 709 (85.2%) | 709 (85.2%) | **571 (79.2%)** | 130 (18.0%) | 20 (2.8%) |
| parson | 686 | **507 (73.9%)** | 425 (62.0%) | 291 (42.4%) | **405 (79.9%)** | 9 (1.8%) | 93 (18.3%) |
| ptrdist | 920 | **684 (74.3%)** | 652 (70.9%) | 623 (67.7%) | **465 (68.0%)** | 181 (26.5%) | 38 (5.6%) |
| zlib | 647 | **385 (59.5%)** | 375 (58.0%) | 337 (52.1%) | **293 (76.1%)** | 86 (22.3%) | 6 (1.6%) |
| libtiff | 3,478 | **2,111 (60.7%)** | 2,016 (58.0%) | 1,194 (34.3%) | **1,694 (80.2%)** | 177 (8.4%) | 240 (11.4%) |
| libarchive | 10,269 | **6,842 (66.6%)** | 6,190 (60.3%) | 4,924 (48.0%) | **5,532 (80.9%)** | 896 (13.1%) | 414 (6.1%) |
| thttpd | 829 | **634 (76.5%)** | 616 (74.3%) | 449 (54.2%) | **341 (53.8%)** | 57 (9.0%) | 236 (37.2%) |
| tinybignum | 129 | **128 (99.2%)** | 117 (90.7%) | 117 (90.7%) | **110 (85.9%)** | 3 (2.3%) | 15 (11.7%) |
| **Total** | 26,413 | **17,924 (67.9%)** | 16,244 (61.5%) | 12,791 (48.4%) | **14,312 (79.8%)** | 1,891 (10.6%) | 1,721 (9.6%) |

# Evaluation: Conversion Failures

| Program | Total Pointers ($TP$) | Wild Pointers ($wild$) (% of $TP$) | | | |
|---|---|---|---|---|---|
| | | typ3c$_w$ | Root Cause | | |
| | | | Total$_d$ | U$_r$ | Top two Reasons (% of typ3c$_w$) |
| vsftpd | 1,765 | 429 (24.3%) | 218 (12.4%) | 9 (0.5%) | Invalid Cast (64.37%)<br>Default void* type (16.9%) |
| icecast | 2,682 | 887 (33.1%) | 337 (12.6%) | 10 (0.4%) | Source code in non-writable file. (28.97%)<br>Default void* type (24.2%) |
| lua | 4,176 | 1,395 (33.4%) | 308 (7.4%) | 8 (0.2%) | Union field encountered (76.1%)<br>Invalid Cast (10.83%) |
| olden | 832 | 111 (13.3%) | 13 (1.6%) | 6 (0.7%) | Default void* type (56.8%)<br>Assigning from 0 depth pointer to 1 depth pointer. (36.4%) |
| parson | 686 | 179 (26.1%) | 99 (14.4%) | 8 (1.2%) | Inferred conflicting types (62.27%)<br>Invalid Cast (12.85%) |
| ptrdist | 920 | 236 (25.7%) | 23 (2.5%) | 15 (1.6%) | Unsafe call to allocator function. (45.63%)<br>Unchecked pointer in parameter (35.58%) |
| zlib | 647 | 262 (40.5%) | 48 (7.4%) | 6 (0.9%) | Default void* type (51.06%)<br>Invalid Cast (32.22%) |
| libtiff | 3,478 | 1,367 (39.3%) | 337 (9.7%) | 11 (0.3%) | Invalid Cast (52.02%)<br>Union field encountered (13.84%) |
| libarchive | 10,269 | 3,427 (33.4%) | 897 (8.7%) | 10 (0.1%) | Invalid Cast (58.89%)<br>Default void* type (18.75%) |
| thttpd | 829 | 195 (23.5%) | 53 (6.4%) | 7 (0.8%) | Default void* type (51.13%)<br>Source code in non-writable file. (27.57%) |
| tinybignum | 129 | 1 (0.8%) | 0 (0.0%) | 1 (0.8%) | Source code in non-writable file. (100.0%) |
| **Total** | 26,413 | 8,489 (32.1%) | 2,333 (8.8%) | 91 (0.3%) | |

# Evaluation: Boun3c

char* usually do not have explicit bounds association

| Program | Arrays (arr) | | | | | Null-terminated arrays (ntarr) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Require Bounds | Inferred Bounds | | | | Require Bounds | Inferred Bounds | | |
| | | Total | Technique (% of Total) | | | | Total | Technique (% of Total) | |
| | ($RB_a$) | (% of $RB_a$) | Seeded | Flow | Heuristics | ($RB_n$) | (% of $RB_n$) | Seeded | Flow |
| vsftpd | 30 | 26 (86.7%) | 15 (57.7%) | 6 (23.1%) | 5 (19.2%) | 27 | 18 (66.7%) | 17 (94.4%) | 1 (5.6%) |
| icecast | 29 | 20 (69.0%) | 16 (80.0%) | 4 (20.0%) | 0 (0.0%) | 159 | 59 (37.1%) | 48 (81.4%) | 11 (18.6%) |
| lua | 146 | 79 (54.1%) | 61 (77.2%) | 18 (22.8%) | 0 (0.0%) | 99 | 28 (28.3%) | 18 (64.3%) | 10 (35.7%) |
| olden | 91 | 87 (95.6%) | 68 (78.2%) | 19 (21.8%) | 0 (0.0%) | 0 | 0 (0.0%) | 0 (0.0%) | 0 (0.0%) |
| parson | 2 | 2 (100.0%) | 2 (100.0%) | 0 (0.0%) | 0 (0.0%) | 33 | 22 (66.7%) | 12 (54.5%) | 10 (45.5%) |
| ptrdist | 127 | 91 (71.7%) | 53 (58.2%) | 38 (41.8%) | 0 (0.0%) | 11 | 7 (63.6%) | 4 (57.1%) | 3 (42.9%) |
| zlib | 52 | 50 (96.2%) | 37 (74.0%) | 12 (24.0%) | 1 (2.0%) | 1 | 0 (0.0%) | 0 (0.0%) | 0 (0.0%) |
| libtiff | 65 | 62 (95.4%) | 42 (67.7%) | 20 (32.3%) | 0 (0.0%) | 145 | 145 (100.0%) | 144 (99.3%) | 1 (0.7%) |
| libarchive | 449 | 347 (77.3%) | 255 (73.5%) | 83 (23.9%) | 9 (2.6%) | 112 | 40 (35.7%) | 27 (67.5%) | 13 (32.5%) |
| thttpd | 31 | 26 (83.9%) | 19 (73.1%) | 7 (26.9%) | 0 (0.0%) | 127 | 96 (75.6%) | 61 (63.5%) | 35 (36.5%) |
| tinybignum | 2 | 2 (100.0%) | 2 (100.0%) | 0 (0.0%) | 0 (0.0%) | 13 | 13 (100.0%) | 13 (100.0%) | 0 (0.0%) |
| **Total** | 1,024 | **792 (77.3%)** | **570 (72.0%)** | **207 (26.1%)** | 15 (1.9%) | 727 | **428 (58.9%)** | **344 (47.3%)** | **84 (11.6%)** |

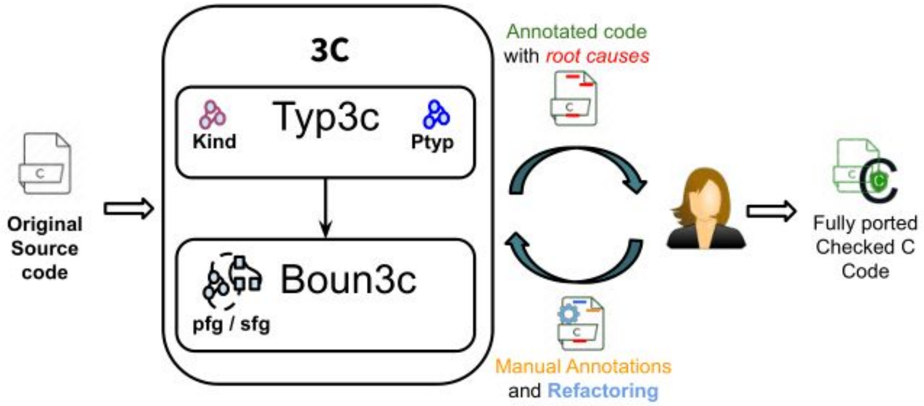
# Evaluation: Effectiveness of Interactive Conversion

| Program | Step | Source Changes (LOC) | | Pointers | | | | Root Causes | |
|---|---|---|---|---|---|---|---|---|---|
| | | Manual | 3C | ptr | arr | ntarr | wild | Num | Avg |
| vsftpd | 3c (Initial) | - | 1760 | 1220 | 46 | 98 | 441 | 304 | 21.4 |
| | Phase 1 | 367 | 1616 | 1261 | 82 | 179 | 290 | 224 | 4.5 |
| | Phase 2 | 1889 | | 1407 | 177 | 240 | 97 | 96 | 1.2 |
| thttpd | 3c (Initial) | - | 704 | 338 | 57 | 236 | 198 | 136 | 29.6 |
| | Phase 1 | 708 | 771 | 392 | 75 | 348 | 58 | 53 | 1.9 |
| | Phase 2 | 1450 | | 398 | 87 | 468 | 15 | 25 | 1.6 |
| icecast | 3c (Initial) | - | 2102 | 1424 | 54 | 312 | 887 | 1142 | 51.2 |
| | Phase 1 | 168 | 5529 | 1667 | 62 | 330 | 636 | 874 | 37.0 |
| | Phase 2 | 2592 | | 1829 | 70 | 523 | 328 | 266 | 4.0 |

**Mostly Annotations**

# Conclusion

Actively Maintained: https://3clsp.github.io/

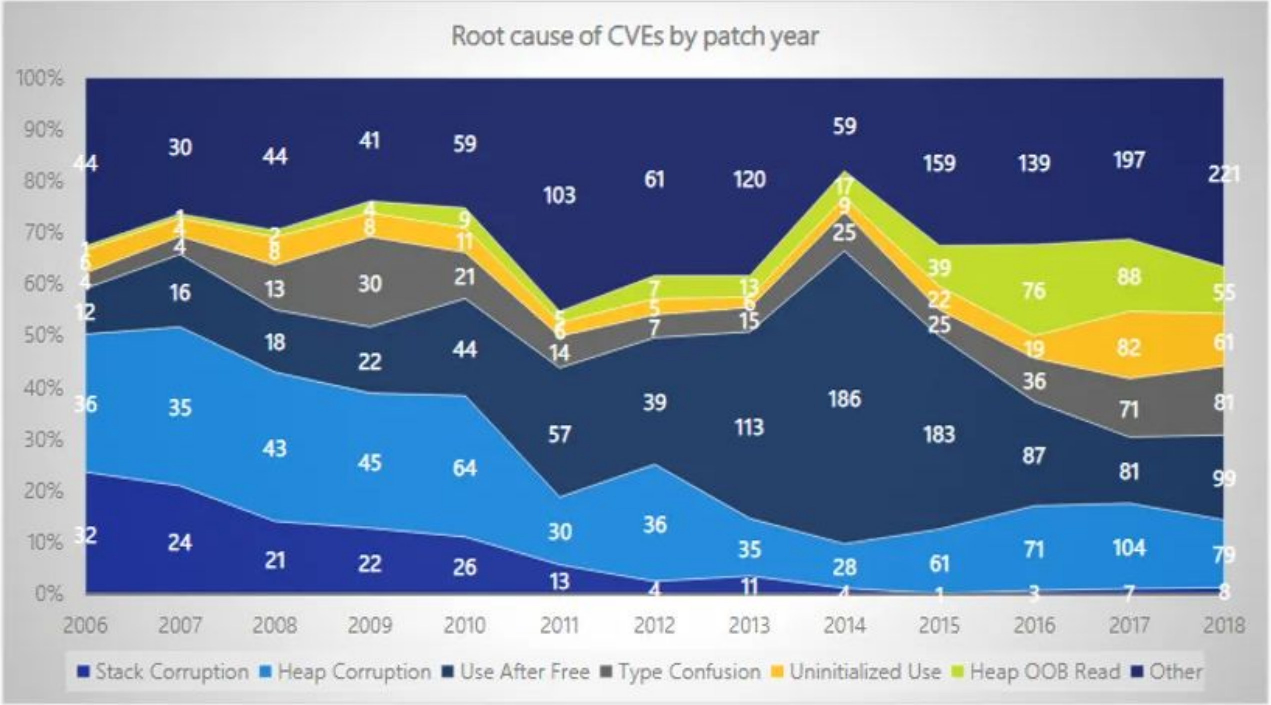



Contact:

Aravind Machiry (amachiry@purdue.edu)

Michael Hicks (mwh@cs.umd.edu)

# Memory Safety is a Problem (Still!!?)




Stack corruptions are essentially dead

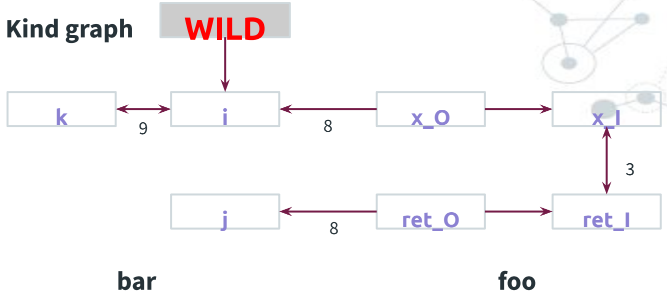
Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

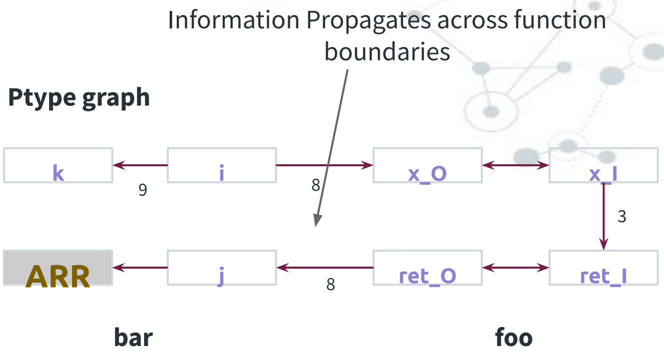
# kind graph

```
1: int *foo(int *x) {
2:  ...
3:  return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```



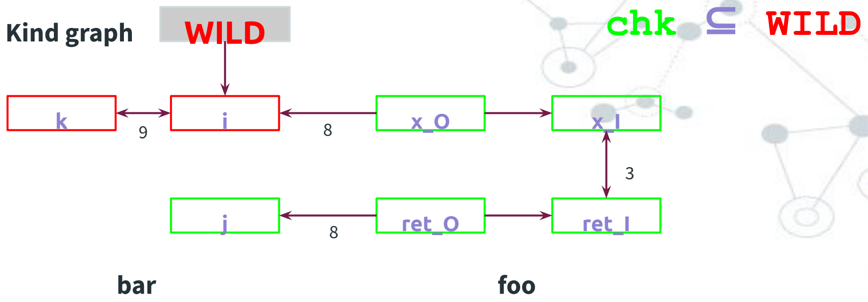
# ptype graph

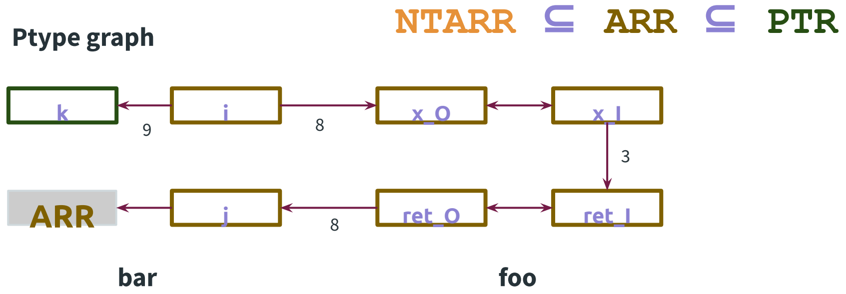
```
1: int *foo(int *x) {
2:   ...
3:   return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```



# kind and ptype graph

```
1: int *foo(int *x) {
2:   ...
3:   return x;
4: }

5: bar () {
6: int *i, *j, *k;
7: i = (int *)0xff86763;
8: j = foo(i);
9: k = i;
10: j[0] = ...
}
```

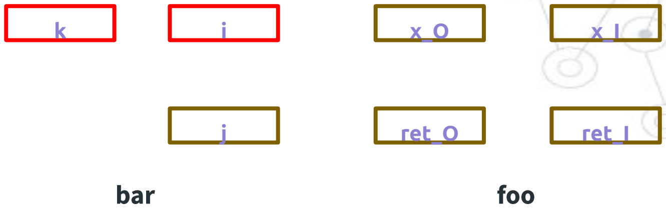


# Rewrite Based on the Final Solution

```
array_ptr<int> *foo(array_ptr<int> x) {
    ...
    return x;
}

bar () {
  int *i, *k;
  array_ptr<int> j = NULL;
  i = (int *)0xff86763;
  j = foo(i);
  k = i;
  j[0] = ...
}
```

**Final Solution**



## Our Approach

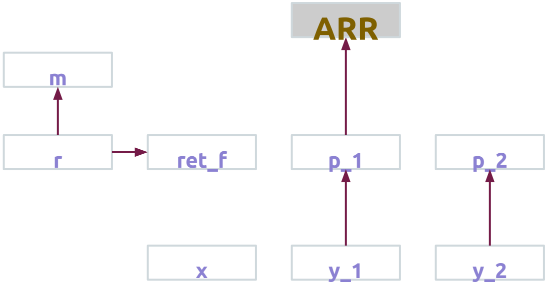
- Automated Conversion of C to Checked C.
    - Infer Checked types for Pointers.
    - Rewrite corresponding pointers with Checked types.

# Type qualifier inference

Step 2: Create constraints based on pointer usage.

```
ret_f        p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

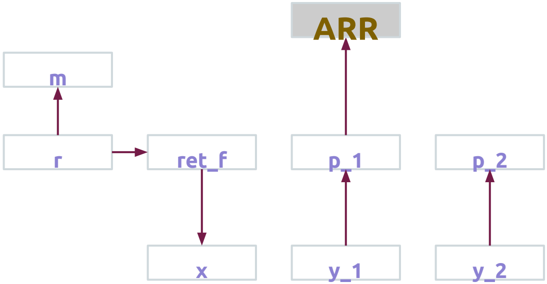


# Type qualifier inference

Step 2: Create constraints based on pointer usage.

```
  ret_f      p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
     y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```

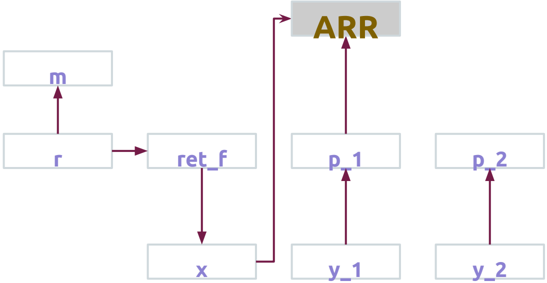


# Type qualifier inference

Step 2: Create constraints based on pointer usage.

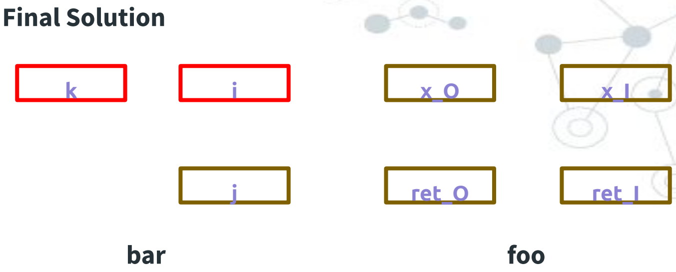
```
  ret_f            p_1 p_2
int *foo(int **p) {
  int *r; r
  int *m; m
  ...
  m = r;
  ...
  if (p[i])
   ...
  return r;
}
```

```
bar() {
  int *x; x
       y_1 y_2
  int **y;
  ...
  x = foo(y);
  ...
  x[i] = 0;
}
```



# Rewrite Based on the Final Solution

```
int *foo(int *x) {
  ...
  return x;
}

bar () {
 int *i, *j, *k;
 i = (int *)0xff86763;
 j = foo(i);
 k = i;
 j[0] = ...
}
```



# Rewrite Based on the Final Solution

```
int *foo(int *x) {
  ...
  return x;
}

bar () {
 int *i, *j, *k;
 i = (int *)0xff86763;
 j = foo(i);
 k = i;
 j[0] = ...
}
```

```
array_ptr<int> *foo(array_ptr<int>
x) {
  ...
  return x;
}

bar () {
 int *i, *k;
 array_ptr<int> j = NULL;
 i = (int *)0xff86763;
 j = foo(i);
 k = i;
 j[0] = ...
}
```

# Rewrite Based on the Final Solution

```
array_ptr<int> *foo(array_ptr<int> x) {
   ...
   return x;
}

bar () {
  int *i, *k;
  array_ptr<int> j = NULL;
  i = (int *)0xff86763;
  j = foo(assume_bounds_cast<array_ptr>(i, unknown));
  k = i;
  j[0] = ...
}
```

- Passing WILD arguments to checked parameters:
  - use assume_bounds_cast.
- If passing checked arguments to WILD parameters:
  - Use itype.

# Bounds-safe Interface for backward compatibility

```
char *strncpy(char *dst, const char *src, size_t len);

void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

**error:** **Passing _Array_ptr<char> to parameter of incompatible type char*.**

# Bounds-safe Interface for backward compatibility

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);

void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);  This is Okay.
 ...
}
```

# Bounds-safe Interface for backward compatibility

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);

void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);  This is Okay.
 ...
}
```

## What about calling from unchecked code?

```
void bar() {
 char *s1 = malloc(10);
 char *s2 = malloc(5);
 ...
 strncpy(s1, s2, 5);
 ...
}
```

# Bounds-safe Interface for backward compatibility

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);

void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

This is Okay.

## What about calling from unchecked code?

```
void bar() {
 char *s1 = malloc(10);
 char *s2 = malloc(5);
 ...
 strncpy(s1, s2, 5);
 ...
}
```

**error:** argument has unknown bounds, bounds expected because the 1st parameter has bounds.

# Bounds-safe Interface for backward compatibility

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);

void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

This is Okay.

Can we have a mechanism working for both checked and unchecked code?

**What about calling from unchecked code?**

```
void bar() {
 char *s1 = malloc(10);
 char *s2 = malloc(5);
 ...
 strncpy(s1, s2, 5);
 ...
}
```

**error: argument has unknown bounds, bounds expected because the 1st parameter has bounds.**

***Yes, by bounds-safe interface.***

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
        const char *src : itype(_Array_ptr<const char>) byte_count(n),
               size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

itype (interoperation type): a special type that can be either checked or unchecked type, depending on the context.

- dst is set to an itype with bounds of n bytes
- src is set to an itype with bounds of n bytes
- return value is set to an itype with bounds of n bytes

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
              const char *src : itype(_Array_ptr<const char>) byte_count(n),
                    size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

itype (interoperation type): a special type that can be either checked or unchecked type, depending on the context.

- dst is set to an itype with bounds of n bytes
- src is set to an itype with bounds of n bytes
- return value is set to an itype with bounds of n bytes

```
void bar() {
  char *s1 = malloc(10);
  char *s2 = malloc(5);
  ...
  strncpy(s1, s2, 5);   Can compile and run without any bounds check.
  ...
}
```

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
        const char *src : itype(_Array_ptr<const char>) byte_count(n),
              size_t n) : itype(_Array_ptr<T>) byte_count(n);


void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
        const char *src : itype(_Array_ptr<const char>) byte_count(n),
               size_t n) : itype(_Array_ptr<T>) byte_count(n);


void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

→ **S1 and S2 are bounds checked at the call-site.**

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
        const char *src : itype(_Array_ptr<const char>) byte_count(n),
                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

```
void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

→ **S1 and S2 are bounds checked at the call-site.**

```
strncpy(s1, s2, 6);
```

# Bounds-safe Interface for strncpy()

```
char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
              const char *src : itype(_Array_ptr<const char>) byte_count(n),
                   size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

```
void foo() {
 _Array_ptr<char> s1 : count(10) = malloc<char>(10);
 _Array_ptr<char> s2 : count(5) = malloc<char>(5);
 ...
 dst = strncpy(s1, s2, 5);
 ...
}
```

→ S1 and S2 are bounds checked at the call-site.

```
strncpy(s1, s2, 6);
```

→ **error:** argument does not meet declared bounds for the 2nd parameter.

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = "12345";
```

**OK. p0 is pointing to array of 6 chars (including NULL).**

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```
**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as “p: bounds(p, p+1)”**

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```
**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as “p: bounds(p, p+1)”**

```
char c = p[0];
```
**OK.**

# _Nt_array_ptr<T>

Points to a null-terminated array of type T (integral and pointer type) — permits pointer arithmetic

```
_Nt_array_ptr<char> p0 : count(5) = "12345";
```
**OK. p0 is pointing to array of 6 chars (including NULL).**

```
_Nt_array_ptr<char> p = "12345";
```
**Equivalent to declaring Bounds of p as "p: bounds(p, p+1)"**

```
char c = p[0];
```
**OK.**

```
char c = p[1];
```
**error: out-of-bounds access.**

```
 = p[1];
     ^~~~
```