Baggy Bound Checking:

Suppose we have to check on the bounds of pointer arithmetic like:
\[ P' = p + i \]

\[ \text{Assert}\ (p \cap p' \gg BNDS [p \gg 4]) == 0 \]

If \( p \) and \( p' \) have sizes \( 2^k \) and both point to same object and all object have size \( 2^k \)

If suppose this is virtual address space and we have the size of each object \( 2^k \) then we can think of the space is divided into even chunks with each space divided into size \( 2^k \).

We can think of a pointer having two parts:

1. Upper 32-k bits will specify which of these chunks points to
2. Lower k bits will point to the offset within the chunk
So if two pointer points to the same memory location then the upper 32-k bits will remain the same and the lower k bits will differ

What does this statement do?

Assert \((p \cap p' \gg BNDS [p \gg 4]) = 0\)

Remainder of the difference is zero

Number of bits they are allowed to differ

Tells the difference between the bits between \(p\) and \(p'\)

So basically this statement shifts the bits to the right.

The main advantage with this is that we can perform fast checks.

The main problem with the Jone's and Kelly approach is that of out of bound pointers, which they took into consideration by padding 1 extra byte. So in this case Baggy Bound Checking will check it by using OOB (Out of bound) pointers i.e. flagging them with a bit and making them an invalid pointer.
Each block is $2^k$ size.
If there are two pointers one points to the upper half and the other points to the lower half then these pointer will differ by their MSBs.

Suppose I have a pointer
\[
p = q + i;
\]
If(!inbounds(p,q))
\[
p = 0!0x800..0 // high bit of p
\]
\[
r = p -l;
\]
\[
*r=0;
\]
So the thing there is that suppose that q points to an object which is in the lower part of the above diagram. Now by this statement p = q +1; p becomes out of bounds which is caught by the buggy bounds checking. So instead of pointing to the lower part of the memory p points to the upper part.

If the programmer dereferences p then it will display segmentation fault.
R has come back in bound and we know where it points to.
Suppose a pointer \( p \) (out of bounds) points to half of the area, which he supposed to point to. So we have to detect the original object it points to.

So, if the pointer is out of bounds then it can be bought in bounds by clearing off the MSB. And how do we figure out which object it points to. Following example can do this:

\[
p = q + i;
k = \text{BNDS}[q>>16];
\text{If(!inbounds(p,q))}
\]

\[
\text{But } p \text{ is within } 2^{(k-1)} \text{ of being in bounds}
\]

\[
p = p - 2^k; \quad \text{// if } p \text{ is too large}
0r
P = p + 2^k; \quad \text{// if } p \text{ is too small}
\]

Set MSB of \( p \)

**Overhead of memory**

If we could map a single byte at a time. How many bytes of memory will be there in the table.

Suppose I allocated 100 kb memory so it is padded by 28kb.

A pointer can point any where in here in this memory. So I want no matter what \( p \) is I want an entry in this BND table for \( p/16 \). So how many distinct values are there.
128,000/16 = 8,192
So if I allocated 128KB of memory I will get 8KB of overhead, if I allocate 64KB of memory I will get 4KB of overhead. So over head is 1/16. Physical overhead of the memory is only 6% in this case.

Different Approach to handle Bounce Tracking
FAT pointers

We store the bounds of every object
We force pointers to be always in bounds so we have to check every pointer arithmetic operation because at that point the pointer can go out of bounds. Rather then storing the bounds on the object it stores the bounds on the pointer, so if the pointer can hover all over the place but what is attached to the pointer is the original bounds of the object it points to. So with pointer arithmetic we do not have to do any work. So for arithmetic it is free but for dereferencing it has to check the bounds. This is called FAT pointers.

If we assign p = q all the info of q gets copied into p.

Language Based Approach

1. For each pointer variable – add a bound variable
2. On pointer assignment – assign corresponding bounds variable
3. On pointer dereferencing – check bounds

Example Program 1:

Void pcopy(int *a, int *b, int n)
{
    int *p = a;
    for(i=0;i<n;i++)
        *p++ = *b++;
}

So for every pointer variable the compiler will add a bound variable.

typedef struct bounds{
    void *lo;
    void *hi;
void pcopy(int *a, bounds a_b, int *b, bounds b_b, int n)
{
    int *p = a;
    bounds p_b = a_b;

    for(i=0;i<n;i++)
    {
        assert(p_b.lo <= p < p_b.hi);
        assert(b_b.lo <= b < b_b.hi);
        *p++ = *b++;
    }
}

Example Program2:

Struct list{
    Struct list *next;
    Int data;
}

void traverse(struct list *l)
{
    while()
    {
        f(l->data);
        l = l->next;
    }
}

This program transforms to :

Struct list{
    Struct list *next;
    bounds next_b; // bound checking for next
    Int data;
}

void traverse(struct list *l, bounds l_b)
{
    while(l)
    {
        assert(l_b.lo<= l < l_b.hi);
        [bound_check(&&->data, l_b)]
This poses a compatibility overhead. This can be overcome by shadow structure.

Struct list_shadow{
    Bounds b;
    Struct list_shadow *next;
};

void traverse(struct list *l, struct list_shadow *b_l)
{
    while(l)
    {
        boundcheck(&l->data,b_l.b);
        b_l = *b_l.next;
        l=l->next;
    }
}

The concept here is if we have a link list in the original program then in the modified program we will also have the link list of the bound variable.

For l

For l_b

void *p malloc(1);
b_p = {p,p+1};

l->next = (struct list *)p;
l_b.next->b = p_b;
l=l->next;
l_b = *b_l.next;