Static Model Construction (Contd.)

When we have 3 func()

```plaintext
func1()
{
}

func2()
{
}

func3()
{
}
```

Solution?? Push down automaton
This is Context-Sensitive Intrusion Detection System

But this also leads to ambiguity

Since there is an ‘if’ in main(), we need to track all states the system can be. This could lead to states explosion! Also system becomes very slow.

Efficient Context Sensitive IDS

We are accounting both function calls and system calls. Make a function call trace in the application memory. Everytime a function call is made, a log is maintained and every such log is at a known location to the monitor. So everytime a function is made, the monitor checks its location in the log and replays the series of function and decide if the call is allowed. It then purges the log. The technique helps in batching all the function calls and crosses the boundary to check with the monitor only when it has to: say when an actual syscall is made.

Trick: modify the program to enforce efficient enforcement.
Note: the log is in the application memory. A shell code can erase the log as well or modify it to suit its needs (to tell the monitor the path from one system call to another)

Can we have a model that can rigidly enforce a control flow that would be exactly same as a unattacked executable?
(Sequences of system call, arguments, etc)
1. Have a model that accepts strings that the actual program would accept only
Paradox: if we have a model that can replicate the exact sequence of system calls, then why need the program itself

Memory safe compilation of C:
A way to prevent a buffer overflow even if the programmer screws up .c with buffer overflow -> special compiler -> a .o with no buffer overflow
we shall have a program that aborts when there are buffer overflows:
java and python dont have buffer overflow purely because they dont have pointers

Two main approaches:

1. Jones & Kelly method
   
   **GOAL:** is that no pointer ever goes out of the bounds.
   
   **METHOD:**
   - track bounds of all objects
   - check all potentially dangerous pointer operations

   **Pointer operations:**
   - dereference \(*p\)
   - increment / decrement \(q = p + n\)
   \(q = p\)

   \(p[5]\) can be broken into above two fundamental ops

   e.g  func(p); func(q);
   \(p = malloc();\)
   \(free(p);\)
   \(p = &x;\)
   \((\text{cast *})p;\)
   Let’s say \(p\) is a safe pointer
   \(q = p+5\)  doesn't guarantee \(q\) is a safe pointer
   \(q = p\)  guarantees \(q\) is a safe pointer

   **Store extra bounds tracking Data structures inside the program and use them to keep track whether pointer are growing out of bounds**

   **Possible operators on this data structure:**

   \[
   \text{add (p, n)} \quad // \text{remember new object of size n at location p}
   \]

   \[
   (\text{lo, hi)} = \text{lookup(p)} \quad // \text{tells me the low address and high address of a pointer}
   \]

   **example:**

   ```c
   char * duplicate (char * p, int n) {
       char * q = (char *) malloc (n);
       int i;
       for(i =0; i < n; i++)
           *q++ = *p++;
       return q-n ;
   }
   ```
Lets say bounds of p was already added to the bounds table. Compiler might insert extra code to ensure nothing goes wrong.

So:

```c
char * duplicate (char * p, int n) {
    char * q = (char *) malloc (n);
    add(q, n);
    int i, lo, hi;
    for(i =0; i < n; i++) {
        (lo, hi) = lookup(p);
        assert(lo <= p <= hi);
        (lo, hi) = lookup(q);
        assert(lo <= q <= hi);
        *q = *p;
        (lo, hi) = lookup(p);
        assert(lo <= p +1 <= hi);
        p = p + 1;
        (lo, hi) = lookup(q);
        assert(lo <= q +1 <= hi);
        q = q + 1;
        (lo, hi) = lookup(q);
        assert(lo <= q - n <= hi);
    }
    return q-n;
}
```