Memory-safe compilation of C: Deputy

- Some systems take the original program and bolt on extra stuff to keep track of all the bounds or pointers
  o Object Bounds methods (Jones & Kelly, Baggy Bounds checking, Softbounds, etc.)
  o Systems where for each struct makes a shadow struct
  o CCured

- Motivation of Deputy
  o Most of the time programmer must be keeping track of the bounds of most of the pointers himself
  o Why bolting on the extra infrastructure to keep track of the bounds rather than using what the programmers already has built-in to track bounds and help the programmer to do it correctly
- Deputy uses Dependent Types
- Key idea: Verify the bounds checking code that programmer already wrote (rather than writing new bounds checking code)
- Example:
  ```c
  void mycopy(int* dest, int* src, int n)
  {
    int i;
    for( i=0 ; i<n ; i++ )
      dest[i] = src[i];
  }
  ```

Using `COUNT` annotation:
```c
void mycopy(int* COUNT(n) dest, int* COUNT(n) src, int n)
{
  int i;
  for( i=0 ; i<n ; i++ )
    dest[i] = src[i];
}
```
o Is the following allowed? Yes

```c
void foo(void)
{
    int A[11], B[7];
    mycopy(A, B, 6);
}
```

o To determine the safety of the above program:
  - Have to make sure that the types of the A, B and 6 matches the method parameter types
  - Type does not mention other variables of the program. Eg. \texttt{dest:ptr(n) int}
do not say anything about \texttt{n}
  - Compiler has to know does “A:ptr(11) int” implies “A:ptr(6) int”?
  - General rule: If,
    
    \[
    \text{p:ptr(n) int n >= m} \\
    \text{p:ptr(m) int}
    \]
    
  - Compiler can analyze the code and outsource the problem of reasoning about integers to a Theorem Prover
  - If the above inference can be made, does the compiler need to insert any kinds of bounds check? No
  - Programming "by contract" is essentially a means of allowing programmers to verify execution of their methods does not corrupt the state of their data structures and so on
    - Specify the requirements of a module/function
    - Check if the requirements are met then module/function behaves correctly
    - Check whatever module/function called meets the requirements
    - So, if every call to the module meets the requirements, the program will never fail
    - So, Deputy checks in the call site if \texttt{mycopy(A, B, 6)} meets the requirement of \texttt{void mycopy(int* COUNT(n) dest, int* COUNT(n) src, int n)}

- Evaluation of the System
  o Effectiveness: Need to do something for the temporal bug. There will not be any kinds of spatial bug.
  o Binary Compatibility: Good. Except extra run-time check which is specified in terms of variable already in the program code. It never modifies the interface of the function. It does not add any extra bounds checking infrastructure like Softbounds.
  o Separate Compilation
  o Compatibility: Good. Don’t need to add any extra variables. Depends on the things that already present in the program.
- Performance: Runtime overhead is ZERO. Every time a call is made, compiler is able to verify that dereference is safe, so there is no need to insert any extra bounds checking code.
- No Code Changes: Introducing annotation does not change the code that much. Code changes is better than rewriting the code.

- Annotation is the part of types in Deputy

- Programmer has to write all the annotation: barrier of the adaptation. Lots of code changes.

- There is 2 kinds of pointer in UNIX/LINUX
  - Kernel pointers: kernel produces (kmalloc) which points to kernel memory
  - User pointers: in read() or write() system call we pass buffer pointer which is user space pointer
  - An troublemaker can pass a pointer to memory which is not allocated or pointer to kernel space memory to read() or write() system calls
  - If there is a structure used in both user and kernel space, then there will be versions like struct foo_kernel and struct foo_user

- Now, to verify,
  
  src:ptr(n) int i:int 0≤i<n
  src[i]: int

- Compiler need to prove that 0≤i<n
  - If compiler failed to prove it in the compile time
    - Insert bounds check: assert(0≤i & i<n)

```c
void mycopy(int* COUNT(n) dest, int* COUNT(n) src, int n) {
    int i;
    for( i=0 ; i<n ; i++ ) {
        assert(0≤i & i<n);
        dest[i] = src[i];
    }
}
```

- Further analysis:
  - Compiler can keep tracks of facts about the run time state of the program
  - Eg. if a statement is: x=5;
  - The fact you know after execution of the statement is {x=5}
- Add new facts to the set of known facts
- Erase old facts with new facts
- So,
  
  ```c
  if(pred)
  {
    Compilers know the fact that predicate is TRUE {pred}
  }else{
    predicate is false {-pred}
  }
  ```

- What kind of facts is TRUE if the expression like below?
  ```c
  p =&i;
  for( i=a ; i<b ; i+=c)  // fact: {a≤i<b}
  {
    *p +=7;
  }
  ```

- Caveat: i can be modified in the loop or aliasing problem (before loop p=&i and *p +=7 in the loop)

- Solution: Compiler has to check that i is not modified in loop and address of i never taken

- Compiler can use a simple rule to ensure that i is not going to be modified in the loop

  ```c
  void mycopy(int* COUNT(n) dest, int* COUNT(n) src, int n)
  {
    int i;
    for( i=0 ; i<n ; i++ )
    {
      assert(0≤i && i<n);  // Theorem Prover can verify that does the fact {0≤i<n} implies assert(0≤i && i<n) always exist.
      If so, we don’t need assert(0≤i && i<n).]
      dest[i] = src[i];
    }
  }
  ```

- Hybrid type checking system
  - Uses bounds checking code the programmer already given
  - Uses programmers bound checking info to prove the assertion is unnecessary
  - Bounds checking is driven by the type checking rules and what the programmer does. So, make type checking rules more tighter
  - If we introduce j in the above function and j is never been used, compiler never do bounds checking for j
Example:

```c
void mycopy(int* dest, int* src, int n)
{
    int i, j = 10;
    for (i = 0; i < n; i++)
    {
        j = i;
        assert(0 <= j && j < n);
        dest[i] = src[j];
    }
}
```

And compiler need to verify,

- src:ptr(n) int
  j:int 0<=j<n
  src[j]:int

So, Compiler asked Theorem Prover to verify whether `{0<=i<n && i=j}` implies `{0<=j && j<n}`. If so, it deletes `assert(0<=j && j<n)`.

Suppose we have a function,

```c
int myprint(int* p, int n)
{
    while (n > 0)
    {
        printf("%d", *p);
        p++;
        n--;
    }
}
```

Here, p:ptr(n) int

If `p++` executed then p no longer points to n items.
Whenever we update variables (parallel assignment: \texttt{p++} and \texttt{n--}), we have to verify

\[
\begin{array}{c}
p: \text{ptr}(n) \text{ int} \\
\hline
n: \text{int} \quad 0 < n \\
(p+1): \text{ptr}(n-1) \text{ int}
\end{array}
\]

- Direction for future research: reduce the annotation burden.