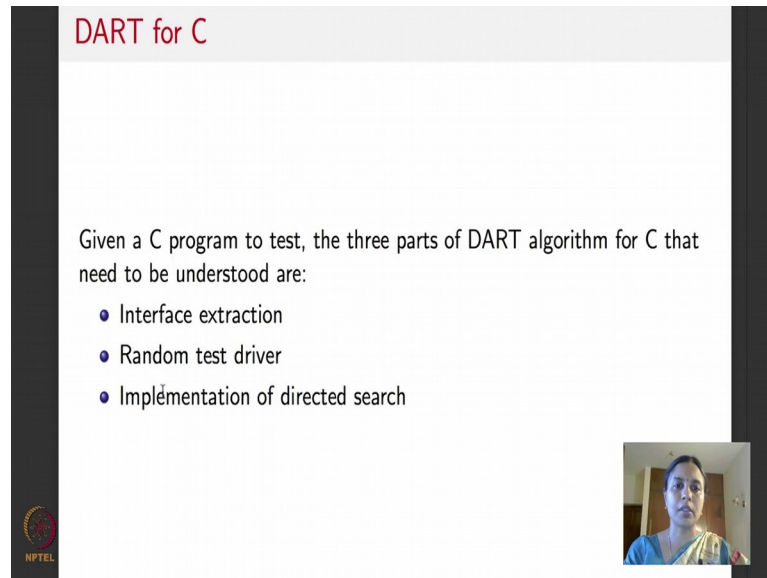


Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 54
DART: Directed Automated Random Testing

(Refer Slide Time: 00:13)



DART for C

Given a C program to test, the three parts of DART algorithm for C that need to be understood are:

- Interface extraction
- Random test driver
- Implementation of directed search

The slide features a dark grey header with the title 'DART for C' in red. The main content is on a white background. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking.

Welcome back, I am going to finish lecturing about DART with this lecture. Just to recap what DART is, DART stands for directed automated random testing, an independent standalone unit testing tool for a C program or Java program and it basically does concolic execution, a mix of concrete and symbolic execution. What are the steps that DART follows? DART begins by generating a random test input for the given C program. It does enough work to figure out how to generate the random test input such that the it is a correct format in test input for the C program, runs the C program on this randomly generated inputs and collects a few pieces of information.

What does it collect? It collects the details about the path that is randomly generated test input takes the C program on, it collects all the constraints that the that were encountered along the path puts them in a stack. Along with that whenever a particular constrain that it is encountering along a path, turns out to be non-linear because it cannot be solved by a constraint solver, DART also keeps the concrete values substitutes the concrete value,

and raises a flag which says that I am not able to symbolically execute this particular path.



After it is generated successfully the path constrain for a randomly generated test vector, DART systematically explores neighboring paths that the program would have taken. By taking one constrain at a time that it encountered on this random test vector, which, it kept these constraints in a stack, if you remember, takes it out of the stack flips the constrain, which means sorry negates the constraint and explore the neighboring program path. Once its explored a particular constraint fully, it goes on to the next up constraint on the program which is below this in the stack, again systematically flips it keeps doing this. When its able to successfully terminate doing this then DART has done what is called a reasonably thorough directed search of a program. But at some point if it encounters an abort statement, then DART has found a bug. So, it will report saying it is found that and stock, but at some point one of the flags which is all linear, all locations definite turned to be set to be 0 by dart; that means, DART is abnormally terminated.

Another option could be there on a non terminating program DART could run forever. Now we understand the algorithm for DART, I will tell you how for a program written in a programming language C, DART will go about interface extraction which it needs to do because it has to figure out what is an input to a program, and how to give the program its input, how does it actually do the random test driver, which generates the random input and how the directed search is implemented.

(Refer Slide Time: 03:02).

AC-controller Example in C

```
int is_room_hot = 0;
int is_door_closed=0;
int ac=0;
/* initially room is not hot, door is open and AC is off */
void ac.controller(int message) {
    if (message == 0) is_room_hot=1;
    if (message == 1) is_room_hot=0;
    if (message == 2) {
        is_door_closed=0;
        ac=0; }
    if (message == 3) {
        is_door_closed=1;
        if (is_room_hot) ac = 1 ; }
    if (is_room_hot && is_door_closed && !ac)
        abort(); /* check correctness */
}
```




So, we will take a small example. Here is a small example of a controller that controls the air conditioning this program is written in C. So, only again as always a program segment is given here. So, is room hot is an integer variable which is initialized to 0, which means the room is not hot, is door closed is another integer variable initialized to 0, which means that the door is open; ac 0 means the AC is off.

So, initially the room is not hot door is not closed or door is open and the ac is off. AC controller takes as an argument or message which is an integer variable. So, message could take values 0 1 2 or 3. If message is 0, then what it means is that it sets room hot to one, which means its starts heating up the room. If message is 1 then its starts cooling the room. If message is 2 then what it does is that its closes the door and puts off the AC. If message is 2 then it opens the door and puts off the AC. If a message is 3 then it closes the door and if the room is still hot it switches on the AC. And if the room is hot the door is closed and the AC is not on, there is an error in the program; because the doors closed AC is not on the room is very hot which means the controller is not working. So, it goes into a state for abort this is like an error state for the AC control.


So, this is the small program which tries to take a message may be through a reader from a user who has AC as part of the home control system, and tries to do some simple command.

(Refer Slide Time: 04:44)

DART for C: Interface extraction




- Apart from the inputs that are initialized, DART first identifies the external interfaces through which the program can obtain inputs via uninitialized memory locations \vec{M}_0 .
- For C, the external interfaces of a program are:
 - its external variables and external functions (reported as "undefined reference" at the time of compilation of the program), and
 - the arguments of a user-specified *top-level function*, which is a function of the program called to start its execution.
- Such external interfaces are obtained by DART using static parsing of the source code.



We will see how DART will test such a program and successfully ensure that it reaches this abort statement here. So, now, DART for C, the first step as we told you is to do interface extraction. How does it do that? Now apart from inputs that are initialized by a program, there are there could be other inputs to a program, DART first works on identifying them. How does it identify? It has to identify the external interfaces through which the program obtains its inputs and these are typically from uninitialized memory locations, because these are inputs that are not initialized. For a program written in C typical interfaces could be its external variables, its external functions or it could be arguments of a user specified top level function, which is the function of the program to call to start it is a execution, it could be any of these kinds. These external interfaces are obtained by DART by statically passing the code of the C program.

(Refer Slide Time: 05:32)



DART for C: Interface extraction

- Inputs to a C program are defined as memory locations which are dynamically initialized at runtime through the static external interface.
- DART for C supports two ways of mapping inputs to memory addresses:
 - Multiple inputs can be mapped to an address m and these are obtained by successively reading m during different successive calls to the top-level function.
 - Same input can be mapped to different addresses in different executions, for example, when the input is provided through an address dynamically allocated with `malloc()`.
- For each external interface, we determine the type of the input that can be passed to the program via that interface.
- In C, a type is defined recursively as either a basic type (int, float, char, enum etc.), a struct type composed of one or more fields of other types, an array of another type or a pointer to another type.

Now, after it passes the code how does it extract the interfaces? The interface extraction happens as follows. Inputs to the C program; how are they defined? The inputs are basically variables in the C program. So, all of them will have memory locations. These memory locations are dynamically initialized, at run time when the program is running through a static external interface the dart creates.



So, how does dart do that? Its suppose two different ways of mapping inputs to memory addresses. First way is multiple inputs can be mapped to one memory address m and all these inputs are obtained by successively reading m during different calls successively to the top level function. This is standard way in which C works or the same input can be mapped to different addresses because program could take different executions. For example, the same input could be provided through an address that is dynamically allocated with a command like `malloc` anything could happen. For each of these kind of external interfaces DART determines the type of the input that can be pass through the program via that interface. So, that it can be generate a random number compatible with that type. For C, type could be a basic type that is supported by C like for example, C supports integer type, floating point type, character enum and so on. It could be a struct type composed of one or more field of other types, it could be an array of another type a pointer, it could be any of the standard types that are supported by C.

(Refer Slide Time: 07:06)

DART for C: Interface extraction

DART distinguishes three kinds of C functions:

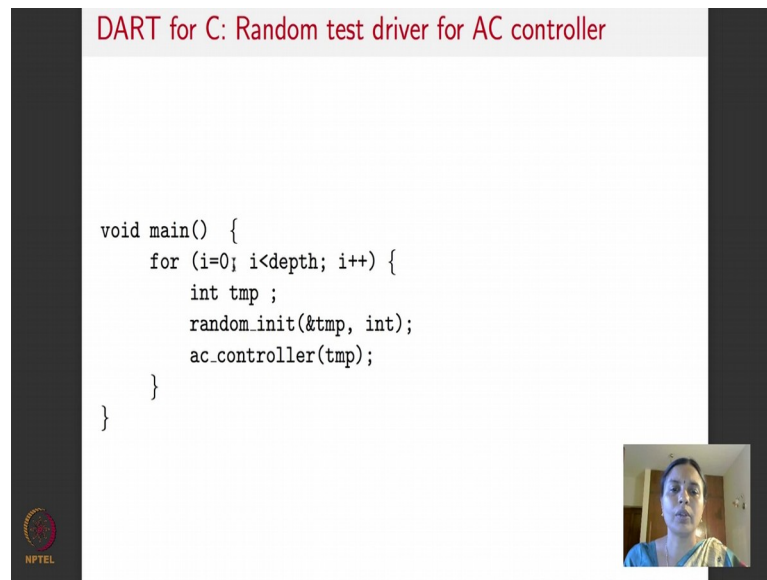
- **Program functions** are functions defined in the program.
- **External functions** are functions controlled by the environment and hence part of the external interface of the program; they can non-deterministically return any value of their specified return type.
- **Library functions** are functions not defined in the program but controlled by the program, and hence considered part of it, e.g. functions defined in the standard C library. These functions are treated as unknown but deterministic black-boxes which DART cannot instrument or analyze.



Now, what happens after this is yeah, so, DART while it does interface extraction it distinguishes three different kinds of functions, there are program functions which are basically functions or procedures that are defined as a part of the program itself, they could be external functions which have functions controlled by the environment. They are part of the external interface of the program, but they are not a part of the co program so, we cannot presume many things about the value that they would return.

They could non deterministically return any value of the compatible type or in a programming language like C you could have several library functions, which have functions not defined in the program, but they are used and controlled by the program. So, they considered a part of it. But because they are library functions DART does not have direct access to the code of these functions. So, dart still treats these functions as unknown functions, but they are deterministic. Deterministic in the sense that I know what a function does, so, it leads like a black box. So, DART really does not go into the function code to instrument it or analyze it. It substitutes a value of the compatible type and moves on. So, for this AC controller program that I showed you in this slide, what are the inputs? Inputs is basically this message which tells you what is the state is that the room should be maintained, in terms of the room being hot or cold, door being open or closed and ac being on or off. So, that is what happens in the external interface here.

(Refer Slide Time: 08:31)



DART for C: Random test driver for AC controller

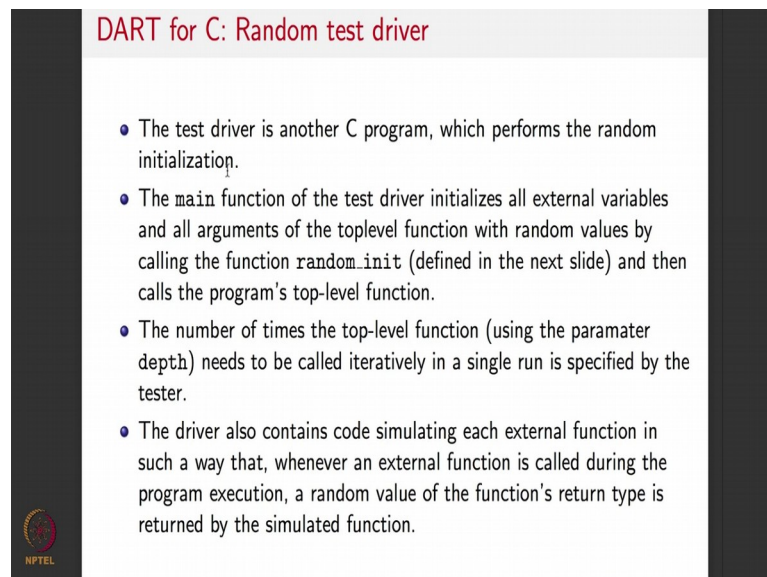
```
void main() {  
    for (i=0; i<depth; i++) {  
        int tmp ;  
        random_init(&tmp, int);  
        ac_controller(tmp);  
    }  
}
```

NPTEL

A small video inset in the bottom right corner shows a person speaking.

So, it says. So, this is the random test driver. So, it calls a function main, and it says for i is equal to 0 to, up to the maximum depth, you have a local variable called tmp and you randomly initialize tmp with this int and call ac controller with this tmp. So, what is this random in it function do? That is what we will try to understand now.

(Refer Slide Time: 08:56)



DART for C: Random test driver

- The test driver is another C program, which performs the random initialization.
- The main function of the test driver initializes all external variables and all arguments of the toplevel function with random values by calling the function `random_init` (defined in the next slide) and then calls the program's top-level function.
- The number of times the top-level function (using the parameter depth) needs to be called iteratively in a single run is specified by the tester.
- The driver also contains code simulating each external function in such a way that, whenever an external function is called during the program execution, a random value of the function's return type is returned by the simulated function.

NPTEL

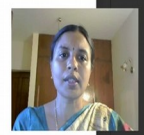

So, the test driver is another C program which performs the random initialization, that is this function random in it. The main function of the test driver initializes all the external variables and all the arguments to the top level function by calling this function random

init. That what, this main function basically calls the random in it and then calls the programs top level function. So, we will see random in it in the next slide. The number of times the top level function is called, that is specified by tester. Like for example, this depth no, how many times to call this random initialization you run it once you run it twice that is up to as, may be its one dart attempted and was not successful. So, you might want to run it again. So, this depth value is determined by the tester. That basically calls the random init functions as many times as it is needed. This driver also contains code for simulating each external function, in such that whenever an external function is called during a program execution, random value of the function return type is returned by the simulated function.

(Refer Slide Time: 10:06)

random_init: Procedure for randomly initializing C variables of any type

```
random_init(m, type) {  
    if (type == pointer to type2) {  
        if (fair coin toss == head) {  
            *m = NULL;  
        }  
        else {  
            *m = malloc(sizeof(type));  
            random_init(*m, type2);  
        }  
    }  
}
```





So, here is how the random code looks like, random init takes an argument m of a particular type. We do not know what a type could be, it could be anything that is supported by c it could be any type. So, I have just mentioned it as type left it like this.

So, now we will see what it is. So, if type is actually a pointer to type 2, then what you do? This is basically a random number generator. So, you toss a fair coin and if it turns out to be head then you say its null otherwise you allot chunk of memory of the size of type whatever that size is, and you randomly initialize that memory to be something of this type 2. I hope this is clear right, it is just the normal random initialization procedure.

(Refer Slide Time: 10:51)

random_init: Procedure for randomly initializing C variables of any type

```
} else if (type == struct) {  
    for all fields f in struct  
        random_init(&(m → f), sizeof(f));  
} else if (type == array[n] of type3) {  
    for (int i=0; i<n; i++)  
        random_init((m+i), type3);  
} else if (type == basic type) {  
    *m = random_bits(sizeof(type));  
}  
}
```




But let say if the type, so, this is the type which is a pointer to another type called type 2. So, you do this randomly allotted memory of the size and initialize it with this one. But if it is a struct type then what do you do for all fields of f in that struct, you randomly initialize each value for the field of the appropriate type. If it is an array type let say of some other type, if it is an array of size n of some other type, then you randomly initialize the value of the array of that same type 3. But if it is a basic type then you generate random bits of that same type. So, this basically tells you that random initialization function does a correct random initialization based on whether the type being a pointer, type could be a struct, type could be an array, type could be one of the basic types. Whatever it is it correctly allots a random input of that size, that type, that is compatible to the correct size.

(Refer Slide Time: 11:49)

DART for C: `random_init`

- `random_init` takes a memory location `m` and the type of value to be stored at `m` as arguments, and initializes `m` randomly depending on its type.
- If `m` stores a value of basic type, its value `*m` is initialized with the auxiliary procedure `random_n_bits` which returns `n` random bits, `n` being its argument.
- If its type is a pointer, location `m` is randomly initialized with either the value `NULL` (with probability 0.5) or with the address of newly allocated memory location, whose value is in turn initialized according to its type following the same recursive rules.
- If type is struct or array, every sub-element is initialized recursively in the same way.

NPTEL



Now, after, what is `random_init` do? It takes a memory location `m` and a type of the value to be stored at `m` as arguments, and initializes `m` randomly depending on its type. That is what is return here. Takes a memory location `m` along with its type it initializes them randomly of this type. If `m` is a value of basic type, then the pointed to `m` is initialized with auxiliary procedure `random bits`, which is this. Generate some bits of a compatible type. If the type is a pointer, then it is a randomly initialized either with an value null or with address of a newly allocated memory location, this is the one, which calls itself again because it is a recursive call and there is fair coin toss. So, the chances here are 50 50. If the type is struct or array every sub element is initialized recursively the same way. That is what is written here. If the type is of struct or the type is a array then you call the same `random_init` to initialize it till you get to the basic type.

(Refer Slide Time: 12:51)

DART for C: Random test driver

- For each external variable or argument to top-level function, DART generates a call to `random_init(&v, sizeof(v))` in the function `main` of the test driver before calling the top-level function.
- If the C program being tested calls an external function, say `return_type some_fun()`, then the test driver generated by DART will include a definition for this function:

```
return_type some_fun() {  
    return_type tmp;  
    random_init(&tmp, return_type);  
    return tmp;  
}
```
- Once the test driver has been generated, it can be combined with the C program being tested to form a self-executable program.





Now, let us look at the go back to the test driver. So, for each external variable argument to top level function, DART generates a call to the random init in the function of the main test driver before calling the top level function, that is the part that we saw till now. If the C program being tested calls an external function, lets say of something like this let say some function call it some name, some fun of some return type, then the test driver generated by DART will also include a definition for that function.

So, it will say this is the return type of this function called some function, inside that the return type there is a variable called tmp which is of type return type, and the random init function can be called to initialize tmp and return it. So, what it basically does is that if there is a particular function, what DART does is basically creates a stub for that function, and returns a type, that is variable that is compatible type to the actual variable that the function would return. So, once this test driver has been generated, now we can combine it with a C program to form an self executable program that is ready for instrumentation.

(Refer Slide Time: 14:03)

Instrumented Program

```
instrumented_program(stack,  $\vec{l}$ ) =  
  // Random initialization of uninitialized i/p parameters in  $\vec{M}_0$   
  for each input  $x$  with  $\vec{l}[x]$  undefined do  
     $\vec{l}[x] = \text{random}()$   
  Initialize memory  $\mathcal{M}$  from  $\vec{M}_0$  and  $\vec{l}$   
  // Set up symbolic memory and prepare execution  
   $\mathcal{S} = [m \mapsto m | m \in \vec{M}_0]$   
   $l = l_0$  // Initial program counter in  $P$   
   $k = 0$  // No. of conditionals executed  
  // Now invoke  $P$  intertwined with symbolic calculations  
   $s = \text{statement\_at}(l, \mathcal{M})$ 
```

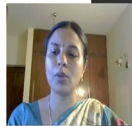



Once it is ready for instrumentation then I will go back for a minutes to the code that I showed you last time which is basically this.

(Refer Slide Time: 14:16)

Instrumented Program, contd.

```
while ( $s \notin \{\text{abort}, \text{halt}\}$ ) do  
  match ( $s$ )  
    case (if ( $e$ ) then goto  $l'$ ) :  
       $b = \text{evaluate\_concrete}(e, \mathcal{M})$   
       $c = \text{evaluate\_symbolic}(e, \mathcal{M}, \mathcal{S})$   
      if  $b$  then  
         $\text{path\_constraint} = \text{path\_constraint} \wedge \langle c \rangle$   
         $\text{stack} = \text{compare\_and\_update\_stack}(1, k, \text{stack})$   
         $l = l'$   
      else  
         $\text{path\_constraint} = \text{path\_constraint} \wedge \neg \langle c \rangle$   
         $\text{stack} = \text{compare\_and\_update\_stack}(0, k, \text{stack})$   
         $l = l + 1$   
         $k = k + 1$   
   $s = \text{statement\_at}(l, \mathcal{M})$  // End of while loop
```

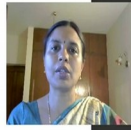



So, it sets up symbolic memory it sets up the stack updates the symbolic memory keeps going updates the stack, and does all, the solving the path constraint and then runs the program.

(Refer Slide Time: 14:20)

Solve path constraint routine

```
solve_path_constraint( $k_{try}$ , path_constraint, stack) =  
  let  $j$  be the smallest number such that  
    for all  $h$  with  $-1 \leq j < h < k_{try}$ ,  $stack[h].done = 1$   
  if  $j = -1$  then  
    return (0, -, -) // Directed search is over  
  else  
    path_constraint[j] =  $\neg(path\_constraint[j])$   
    stack[j].branch =  $\neg stack[j].branch$   
    if (path_constraint[0, ..., j] has a solution  $\vec{I}'$ ) then  
      return(1, stack[0..j],  $\vec{I} + \vec{I}'$ )  
    else  
      solve_path_constraint(j, path_constraint, stack)
```



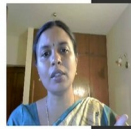

Once its runs the program it can return in it can terminate in any of these three values, it can run forever which is bad.

(Refer Slide Time: 14:24)

DART: Theorem

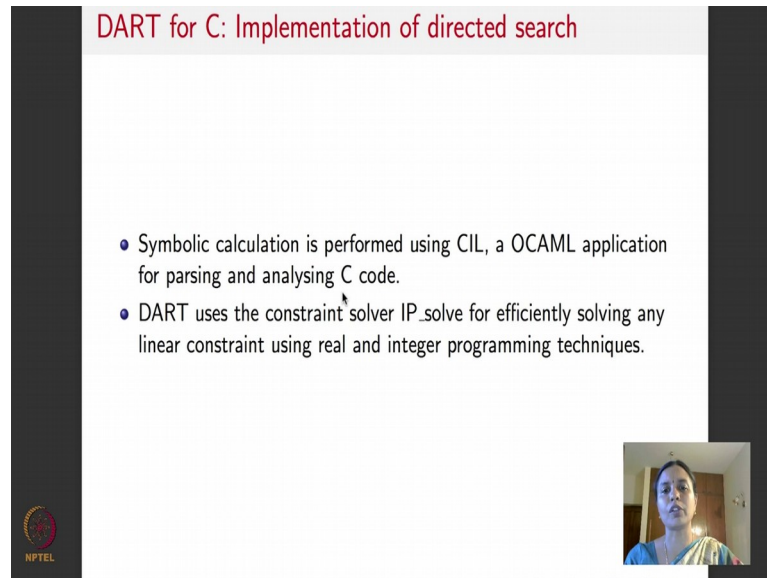
Theorem: Consider a program P (with restricted syntax). The following hold:

- 1 If run_DART prints out "Bug found" for P , then there is some input to P that leads to an abort,
- 2 If run_DART terminates without printing "Bug found", then there is no input that leads to an abort statement in P , and all paths in $Execs(P)$ have been exercised.
- 3 Otherwise, run_DART will run forever.



It can terminate saying I have found a bug which is good because it is able to reach the error statement or it can systematically explore all the program paths in a program right. So, that is what it does for C after random initialization.

(Refer Slide Time: 14:49)



DART for C: Implementation of directed search

- Symbolic calculation is performed using CIL, a OCAML application for parsing and analysing C code.
- DART uses the constraint solver IP_solve for efficiently solving any linear constraint using real and integer programming techniques.

NPTEL

So, how does it do the symbolic updation of C? For C, it so happens that there is this OCAML application called CIL. So, DART uses this for doing parsing and analyzing C code. So, that it can do the symbolic update or the symbolic state, the particular constraint solver that is reported in the DART paper is this solve called IP solve which they use, basically you could use anything any other interpreter that you will like or any other constraint solver that you like.

I will update the correct reference that contains this paper of DART which mainly works for C, and DART is actually now implemented in to a full fledged tool called CUTE as I told you in the last lecture. I will update a reference for CUTE I will also put in a reference for a related open source tool for C called CREST. CUTE is proprietary, but CREST is open source. So, that you could try out symbolic execution if you want and if you know Java if you are interested in symbolic execution for Java there is a tool called JUnit j cute sorry which runs for Java, I will upload a document that contains the references to each of these papers in the announcement section of the course so that you can look at up for further details. If you have any particular doubts specifically about the crest tool feel free to pin me because I have used it in tried at out and I might be able to help you with running and experimenting that tool to do symbolic execution for C programs.

So, I will stop here for now, we are towards the end of the course what I will be doing is I had asked you all feedback for some topics that you would want me to cover. So, in the next few lectures that are available I am going to pick a select subset of those topics especially those that I am familiar with and I will give you introductory modules on each of those.

Thank you.