# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 12 TRANSCRIPTS

1. **Symbolic Execution: Introduction**

>> Hello everyone, and welcome to the last lessons of advanced topics and malware analysis. In these lessons, we're gonna talk about symbolic execution. Our learning objectives, we're gonna learn how to employ symbolic execution to create constraints of a program. We're gonna use those constraints to reason about how the program executes.

We're gonna talk about state explosion and even integrating dynamic analysis to perform concolic execution. We'll talk about all that coming up. But first, what is symbolic execution? Symbolic execution, also known as symbolic analysis is a means of analyzing a program to determine what inputs cause each part of the program to execute.

This may sound similar to slicing but instead of concrete values, symbolic execution uses constraints. So diving in a little deeper, symbolic analysis steps through a program. For us a binary and performs abstract interpretation of the statements, that is for us the instructions. By abstract interpretation, I mean that each instruction is essentially performed on paper.

Data values are turned into formulas based on how they're used in the program. Each execution path can then be represented via constraints or formulas on the data that are accessed along that execution path. If you implement a symbolic execution engine, you're essentially implementing an interpreter that follows the program instruction by instruction, assuming symbolic or fake values for the inputs Rather than obtaining actual inputs like a normal program execution would.

As the interpreter goes through the program, it builds expressions in terms of those symbols for the expressions and values in the program. So you can imagine if you take an input as x and you add 5 to it. The interpreter is gonna continue to build those constraints as x plus 5.

The interpreter is gonna generate those constraints in terms of those symbols for the possible outcomes of each branch predicate. So if that x plus 5 is then used in a branch where it says, if x plus 5 is greater than 10, then go down the true branch. If your interpreter wants to go down the true branch, it'll simply add that inequality to its constraints.

A great deep dive on how this all works was published in the paper titled KLEE back in 2008. Surprisingly, the Wikipedia page for symbolic execution is actually very good. I would highly recommend you go take a look at their explanation as well. But you're probably thinking why? Why perform this analysis on programs without any inputs?

Well, you often want to execute a program on purely symbolic inputs. This allows you to build constraints on those input values that drive the execution down each path. Like the example I gave before, your interpreter would step down each execution path and generate some constraint saying, variable x must be 5 in order to execute this path.

The key insight here is that code can actually generate its own test cases. Or put a different way, you can gain code coverage over dynamic analysis by just giving it pretend inputs. There are tonnes of security applications of symbolic analysis. Malware analysis uses symbolic analysis to explore the malwares code even without any input.

Perhaps your command and control server has been taken down. Symbolic analysis let you get that good code coverage without needing the real commands from the server. Vulnerability finding, for example, you could generate constraints saying input A, B, C will cause a buffer overflow. Or even exploit generation, you can generate constraints saying input A, B, C will cause RIP to jump.

An excellent read that you should follow up on is the paper from 1976, which first introduced the idea of using symbolic execution for automated program testing. And then later in 2014, the paper that introduced the idea of automatically generating exploits using symbolic execution

1. **Symbolic Execution: Example**

>> Hello everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, I'm gonna give you an example of running symbolic execution. And we're gonna talk about state explosion. So here's an example, consider this code that we're gonna use to perform symbolic execution. Now, what if I asked you what input would cause this program to execute the fail function?

A symbolic analysis engine will step through the program line by line and create constraints on the values. These constraints can then be queried, to answer questions such as the one I posed. So let's get out our magical symbolic analysis engine, the symbolic analysis engine is gonna begin stepping through the program and essentially performing the operations at each step on paper.

So when it reads in the value y, we will assign some unconstrained variable i0 to represent the value y. When we get to the second statement, we see that y = 2 times y, therefore, i0 should be assigned 2 times the previous value of i0. Now, things get interesting once we get to a predicate.

At this point, the symbolic analysis engine can go one of two paths. If it takes the true path, it will need to account for what must be true to go down the true path, and the same with the false path. So the symbolic analysis engine performs what's called a state split.

At this point, there are two copies of the program state made in the symbolic analysis engine's memory. On one side, the true branch insists that i0 must equal 12. On the other side, the false branch says that i0 must not equal 12. And then the symbolic analysis engine continue stepping through in one state, the false branch, the next thing that executes is the printf("OK").

At this point, we can query those constraints and say, what input values brought us down this path? By simplifying this mathematical equation, the symbolic analysis engine can solve for the value y != 6. Similarly on the other side, we hit the fail function and that's the end of the execution.

So the symbolic analysis engine can solve for y = 6. So to answer that question I posed at the beginning, an input of y = 6 will trigger the fail function to execute. Now, as we're talking about splitting states, you can see that each decision point in a program increases the number of track states, in fact, it doubles it.

So if I give you a program like this, that even has a few predicates as we begin performing symbolic execution through this program, we can see the state splits really start to add up. This is where you see the problem of state explosion. This is encountered whenever you're performing symbolic analysis on a program and the states divide to a point where you can no longer feasibly keep track of them.

In fact, there was a whole series of programs that were designed called the war games suite to trip up symbolic analysis engines. Here's an example, if I wanted to figure out how to execute the win function, the symbolic analysis engine would have to trace back through the states all the way from this complex predicate up to where e is invoked.

e of course, is invoked based on this pointer to a buffer. By following that further, you see that the pointer is incremented and decremented in different cases of the switch. And eventually, you may figure out that there is a pointer at the beginning of a buffer that points into the buffer.

And when the certain character value matches, then the win function gets executed. Yes, this is an extremely complex example and it's difficult for even a human to keep track of. In fact, because of that switch case, there are three decision points for each character in the buffer. You can see the state explosion is going to occur here, because you have 3 to the n states being tracked for an input of size n.

There are many reasons for state explosion but they all boil down to too much input or output data being tracked as symbolic. We can try to limit the amount of symbolic data that our symbolic analysis engine cares about. Perhaps we could say mark no more than 10 bytes as symbolic.

In this case, you may be able to solve for an address or a pointer. If you step it up to 20 to 80 symbolic bytes, maybe you could find a piece of shellcode or a ROP gadget. And if you try to go above 200 symbolic bytes, you're still very limited in how much you can actually compute using symbolic analysis purely.

There's also a problem of too much included states. You want to somewhat limit the symbolic states that get trapped, this is an operation called state pruning. If you're able to identify that you don't care about certain states, then you can prune those from the states being tracked in your symbolic analysis engine.

The other reason for state explosion, too much code to execute through. Tracking every operation will inflate the constraints. There have been some research that has attempted to use a divide and conquer method, where constraints are solved independently and then returned to a major solver that tries to combine them all.

1. **Concolic Execution**

>> Hello everyone and welcome back to advanced topics in malware analysis. In this lesson, we're going to try to combine dynamic analysis with symbolic execution to perform colic execution. So can colic execution can come to the rescue for solving that massive state explosion problem that we talked about in the previous lesson.

Concolic analysis combines concrete and symbolic executions. Get it? Concrete + symbolic = concolic. We use a concrete execution with a concrete input to guide our symbolic execution. And then selectively replace symbolic values with concrete ones. Concrete values, simplify those complex and unmanageable symbolic expressions that leads to state explosion.

Let's look at an example. Here we'll look at some code that uses two reads and two predicates. And maybe I asked you, what value of z will cause the program to execute the fail much? We'll pull out our magical concolic analysis engine this time, which has both a concrete state and a symbolic state.

Again, we'll start executing at the first instruction. In the concrete state, we actually read a real value from the user, so we may input 8. In the symbolic state, we just assign a symbol for the value of y, in this case is i0. Next we step to the z line.

So we read a value for z in the concrete state, maybe I punch in 42. In a symbolic state, I assign a new symbolic value for z, in this case, i1. The next line executes y = 2 times y. In the concrete state, this is done concretely, so 2 times 8, 16.

In the symbolic state, we continue to build up those constraints. So i0 is assigned 2 times i0 but now we get to a predicate. In the concrete state, the predicate is evaluated concretely, y greater than 12. In this case, y is 16 so we take the true branch, easy enough.

In the symbolic state we would do a state split, one state where i0 is greater or equal to 12, and another state where i0 is less than 12. However, this is where you can incorporate some concrete data. Maybe at this point you decide you don't want to go down that path.

You don't want to cause that state explosion. So you can swap in a concrete value for your i0 variable. In this case, we've determined that y is being represented by i0, so we copy over y's value. I0 is now directly equal to 16. So our symbolic state just like our concrete state goes only down the true branch.

At the next line, we check z against 20 and similarly in the symbolic state, we have to do a new state split. Because we've gotten to a predicate based on symbolic data. I1 which was representing z is recorded in the true branch as being greater than 20. And then the false branch as being less than or equal to 20.

Now the branches that went down the false branch, both print OK, that is in the concrete state we see OK get printed. And in the symbolic state we can solve for what value of the remaining not concretized values could have gotten us down this path. That is, i1 less than or equal to 20.

We can no longer solve for i0 because it's been concretized. In the other symbolic state, we continue executing and hit the fail function. This returns to us i1 greater than 20. Again, can't solve for i0 at this point. But this does answer our question, what value of z would cause the program to execute the fail function?

Greater than 20 and we avoided a costly additional state split that could have led to managing 4 different symbolic states at the same time. And we avoided a costly state split that would have made us track an entirely new symbolic state at the same time. But let's rewind.

Be careful when you're considering code coverage and what you want to concretize. Let's imagine instead as we stepped through the execution, and got to that first predicate, when we performed that state split, perhaps we compromised i0 again. Except this time, we actually had input 2 for the value of y.

This would cause the program to concretize i0 to 4. Now what happens when we continue executing? We never hit the fail function. Concretizing too much data cuts off limbs from your symbolic executions exploration. So you need to be careful when considering what values to concretize, because you might just cut off a limb that you meant to go down.

And again, at this point, we cannot solve for the value of i1 because we've not generated any constraints on its value.

1. **Symbolic Execution Tools**

>> Hello everyone and welcome back to Advanced Topics and Malware Analysis. In this lesson, we're gonna talk about all the Symbolic Execution tools and techniques that make this possible. There is a wealth of Symbolic Execution Tools out there. In fact, my lab uses Angr, KLEE, and Triton to perform Symbolic Execution all the time.

S2E is an outstanding Symbolic Execution engine which performs cancolic analysis on entire operating systems. And FuzzBALL is famous for being able to find vulnerabilities using symbolic analysis. But there are many more even beyond the ones named on this slide. The Enabling Technique behind symbolic execution is SAT Solving.

In computational complexity theory, the satisfiability problem is the problem of determining if the variables in a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. The main idea here is given a program and a claim, use a SAT Solver to find whether there exists an execution that violates the claim, basically produce a counterexample.

This was actually the first problem to be shown to be NP-complete back in 1971. But that hasn't stopped researchers from trying to figure out a solution in the meantime. SAT Solving is used in symbolic execution, every time we convert a set of constraints into a Boolean formula. Essentially adding together all of the variable constraints, we then check path condition satisfiability that's SAT Problem, so that we only explore feasible paths.

When execution paths diverge, this adds new constraints on those symbolic variables. When the program terminates, we can use the constraint solver to generate a concrete input, that gets us down that execution path. All symbolic analysis engines have to put time-limits on their constraint solving. This is how they get around that NP-problem.

Because if your solver hasn't returned an answer in, say ten hours, then it's unlikely to return an answer ever. Let's look at an example, consider the program code shown here. If I asked you can the assert statement fail? It would be very complex to tear through this code and figure out what values of the variables would make the assert statement fail.

However, this would be an easy problem for a SAT solver. Simply translate each line of the program into a set of constraints and then hand all those constraints together. To find a case where the assert fails, we negate the constraints inside of the assert. A SAT Solver would then take these constraints as input and try to find a counter example, where they do not hold true.

Interestingly, the constraints here are satisfiable, that means no counter example exists, and this assert statement will always hold. However, let's look at another case, here I've changed the value of z == 5 in the assert statement. Now can that assert statement fail? Well again, convert the program to a set of constraints, and then ask a SAT Solver.

A SAT Solver will find that this is onset and provide you with a counter example. A counterexample is an assignment of the free variables in those constraints, such that the constraints cannot be shown to be true. SAT Solving has actually become very common in many computer science applications.

And z3 has become probably the most popular SAT Solving engine. z3 is a high performance theorem prover developed at Microsoft Research. What makes z3 so powerful is that it handles many different cases common in program solving. Such as real and integer arithmetic, fixed-size bit-vectors, arrays, and so on.

In fact, the code shown on the slide is actually using z3 to solve values for x and y similar to the one on the previous slide. You can see z3 is quite easy to program with a nice Python interface and easy to use API's. If you run this program, z3 will actually find a satisfiability condition and show you the values of y and x that allow this to be satisfiable.

z3 is so powerful, you can even implement a Mini Symbolic Execution Engine entirely In Z3. Z3 can find the input that would cause a crash in the code on the right. This is done by essentially forking at every predicate to explore each side within the z3 engine. Notice the test me function is the one being tested inside of z3.

And symbolic bit-vectors of 32 bits in length are being given as the arguments to test me. So that wraps up this lesson. We've talked about Symbolic Execution, and how you can use it to create constraints and reason about a program's possible execution. We also talked about state explosion and some of the ways to combat that,.

And how to utilize concolic executions to simplify those symbolic constraints. And gotten a little bit into SAT Solving.