**ARMANDO SOLAR-LEZAMA:** All right. So good morning, everyone. I'm Armando Solar-Lezama. I'm giving the lecture today on symbolic execution. How many of you here are familiar with what the term is or have heard about it before? We want to get a sense of audience. OK. So let's see. I dropped this machine a little too many times and it takes a while to boot up.

So symbolic execution is really the workhorse of modern program analysis. It's one of those techniques that has really broken out of the research bubble and actually made it into a very large number of high impact applications. For example, today at Microsoft there's a system called SAGE that runs on a lot of important Microsoft code ranging from PowerPoint to Windows to actually find security problems and security vulnerabilities.

There's a lot of that academic projects that have made a lot of real world impact by discovering important bugs in open source software, for example, by relying on symbolic execution. And the beauty of symbolic execution as a technique is that compared to testing, for example, it gives you the ability to reason about how your program is going to behave on a potentially infinite set of possible inputs. It allows you to explore spaces of inputs that would be completely unfeasible and impractical to explore by, say, random testing, or even by having a very large number of testers banging and the code.

On the other hand, compared to more traditional static analysis techniques it has the advantage that when it discovers a problem it can actually produce for you an input and a trace that you can run on your real program and execute that program on that input. And you can actually tell that it is a real bug. And you can actually go and debug it using traditional debugging mechanisms. And this is particularly valuable when you're in an industrial development environment where you probably don't have time to go looking after every little problem in your code.

You really want to be able to tell the difference between real problems versus false positives, for example. So how does it work? So in order to really understand how it works it's useful to start by looking at just normal execution, right? If we think of symbolic execution as a generalization of traditional, plain execution, it makes sense to know what this looks like.

So I'm going to be using this very, very simple program as an illustration for a lot of what I'm going to be talking about today. So what do we have here? Again, it's a very simple piece of code, just a couple of branches and here we have an assertion, assert false. And we want to know could that assertion ever be triggered. Is it possible? Is there some input where that will make that assertion fail?

And in this case because the assertion is just saying, assert false, what I'm really asking is, is there an input that can reach that point in the program? So one of the things I can do is I can try just testing. I can go in and run this code with a concrete input. Right? So let's say that I start with an input where x is 4 and y is 4. And initially t is going to have the value 0 right after I declare it.

So before we go with normal execution, what are some of the important point here? The fact that we need some representation of the state of the program, right? Whether we're doing normal execution or whether we're doing symbolic execution, we need to have some way to characterize the state of the program. And in this case, this is such a simple program that it doesn't use the heap. It doesn't use the stack. There are no function calls.

So the state can be fully characterized by these three variables together with knowledge of where in the program I'm at, right? So if I start executing with 4, 4, and 0, so when I get to this branch, is 4 greater than 4? Clearly not. So then I'm going to be executing t equals y. So now after I do that t is no longer 0. It now has the value 4. Right? So that is now the state of my program. And then I can evaluate this branch. Is it the case that t is less than x? No. Right? So we dodged the bullet. We did not get an assertion failure. There was no problem in this particular execution. Right?

But that doesn't really tell us anything about any other execution. All we know is that under the input x equals 4 and y equals 4, the program is not going to fail. But it tells us nothing about what's going to happen on the input [? 2, 1, ?] for example. Right? And in this input you see that this input is actually going to follow a different path in the execution. This time we're actually going to see that t equals x. We're actually going to set t equals 2x.

So after executing these t will be equal to 2, but is there any problem in this execution? Will there be an assertion failure on this input? Well, so let's see. So if t is 2. And x is 2. Is t less than x? No. So it looks like we dodged a bullet again. Right? So here we have two concrete inputs. And they told us that on these two concrete inputs the program didn't fail. But that really doesn't tell us anything about any other input.

And so the idea with symbolic execution is we want to go beyond these single input executions. And we want to be able to actually reason about the behavior of the program on very large sets of inputs. In some cases, infinite sets of possible inputs. And the basic idea is as follows. So for a program like this, just like before the state of the program is characterized by the value of these three different variables. Right? x, y, and t together with knowing where in the program I'm at.

But now instead of concrete values for x and y what I'm going to have is a symbolic value, just a variable. A variable that allows me to give a name to this value that the user is going to provide at the input. So what that means is that the state of my program is no longer a mapping from variable names to concrete values. It's now a mapping from variable names to these symbolic values. And a symbolic value, you can essentially think of it as a formula.

So in this case the formula for x is just x. And the formula for y is just y. And for t, it's actually the value 0. We know that for every input, doesn't matter what you do. The value of t after the first statement is going to be 0. But now here's where it gets interesting. So we get to this branch right here that says, if x is greater than y, we're going to go in one direction. If it's less than or equal to y, we're going to go in the other direction.

Now do we know anything about x and y? What do we know about them? We know their type, at least. So that's a start. So we know that they're going to be ranging from min int to max int, but that's about all we know about them. And it turns out that this information that we know about them is not sufficient to tell us which direction this branch might go. This branch could go either way.

And so now there are many things and we can do, but what's one possible thing that we could do at this point? Make a wild guess.

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Yeah. We could follow both branches. We could flip a coin and pick one branch and take that. So if we want to follow both branches we have to follow one and then the other one, right? So let's say we start with this branch. Right? So now we are at this branch. So what we know is that if we make it to this branch, in this branch t is now going to have the same value as x. And we don't know what that value is going to be, but we have a name for it. It's this script letter x. Right? So that's the value of t on that branch.

If we were to take the opposite branch then what would happen? The value of t would be something different, right? In that branch, the value of t would be the symbolic value y. So that means that when we get to this point in the program, what is the value of t? Well, maybe it's x. And maybe it's y. We don't know exactly which one it is, but why don't we give it a name? Let's call it t0. And what do we know about t0?

What are the cases where t0 is going to be equal to x?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** That's right. So essentially what we know is that if x is greater than y, then this implies that it's x. And if x is less than or equal to y that implies that it's y, right? And so we have this value that we've defined. We'll call it t0. And it has these logical properties. So at this point in the program we actually have a name for the value of t. It's t0. Right? And so what did we do here? We took both branches of this if statement. And then we computed the symbolic value by looking at under what conditions am I going to take one branch, under what conditions am I going to take another branch? And then looking at what values am I going to be assigning to t on both of those branches?

So now it comes to the point where we have to ask, can t be less than x? Right? So what is the value of t? The value of t is now t0. So what we want to know is, is it possible for t0 to be less than x? Right? Now remember the first branch we hit we were asking a question about x and y. And we knew nothing about x and y. The only thing we knew about x and y was that they were of type int.

But now with t0 we actually know a lot about t0. We know that t0 is going to be equal to x in some cases. And it's going to be equal to y in some cases. And so this now gives us a set of equations that we can solve for. So what we can say is, is it possible to satisfy t0 less than x knowing that t0 satisfies all of these properties? Right?

So, in fact, we can actually express this as a constraint where we say, so is it possible to have t0 less than x? And to have x greater than y implies t0 equals x. And x less than or equal to y imply t0 equal y. Right?

So what we have here is an equation that if that equation has a solution, if it's possible to find a value of t0, and a value of x, and a value of y that satisfies that equation, then we know that those values, when we plug them into our program, when the program executes, it will take this branch. And it will blow up when it hits a assert false. Right?

So what did we do here? So we're executing this program, but instead of keeping our state as a mapping from variable names to values, what we're doing is we're keeping our program as a mapping from variable names to these symbolic values. Essentially, other variable names. And in this case our other variable names are the script x, script y, t0, and on top of that, we have a set of equations that tell us how those values are related. So we have an equation that tells us how t0 is related to x and y in this case.

And solving for that equation allows us to answer the question of whether this branch can be taken or not. Now just looking at the equation, can this branch be taken or not? Right? So it looks like the branch cannot be taken. Why not? Because we're looking for cases where t0 is less than x, which means that if you're in this case, then clearly that's not going to be true. Right? So that means that when x is greater than y, then it cannot happen because t0 will be equal to x. And it cannot be equal to x and less than x at the same time.

And what about in this case? Can it happen in this case? Can t0 be less than x in this case? No, it clearly cannot, right? Because in this case we know that x is less than y. And so if t0 is going to be less than x, then it would also be less than y. But we know that in that case t0 is exactly equal to y. And therefore, again, that case cannot be satisfied. So what we have here is an equation that has no solution. It doesn't matter what values you plug into this equation. You cannot solve it and that tells us that no matter what inputs we pass to this code, it will not go down this branch.

Now notice that when making that argument here I was basically alluding to your intuition about integers, about mathematical integers. In practice we know that machine ints don't quite behave exactly the same way as mathematical ints. And there are some cases where laws that apply to mathematical ints don't actually apply to ints in programs. And so when reasoning about this we have to be very careful that when we're solving these equations, we're keeping in mind that these are not the integers as they were taught to us in elementary school.

These are 32-bit integers that the machine uses. And there are many cases and many instances of bugs that arose because programmers were thinking about their code in terms of mathematical integers, and not realizing that there are things like overflows that can cause the program to behave differently for mathematical inputs.

But the other thing is what I've described here is a purely intuitive argument. I walk you through the process of how to do this by hand, but that's by no means an algorithm. Right?

The beauty of this idea of symbolic execution, however, is that it can be coded into an algorithm. And it can be solved in a mechanical way, which allows you to do this not just for ten line programs, but actually for million line programs. And it allows you to actually take this reasoning, and the same intuitive reasoning that we used in this case to talk about what happens when we execute this program on different inputs. And scale that reasoning to very large programs. Are there any questions so far? Yes?

**AUDIENCE:** What if a [INAUDIBLE] are not supposed to take an input? [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Oh. That's a very good question. Right, so, for example, let's say we have the program that we have here, but instead of these being t equals x, here we will say t equals x minus 1. Right? So now all of a sudden, intuitively you can see that now this program could blow up, right? Because when the program takes this path then t will indeed be less than x. And you will indeed fail here. Right?

So what will happen to a program like this? How will our symbolic state look like? Right? So in this case, so t0, when x is greater than y, what is t0 now going to be equal to? It's not going to be equal to x. It's going to be equal to x minus 1, right? And so that means that, so, this condition now has a satisfying assignment. Right? Now this can fail, but what if you go to the developer and say, hey, this function can blow up whenever x is greater than y.

And the developer looks at this and says, oh, I forgot to tell you. Actually, this function can never be called with parameters where x is greater than y. Right? That the client that calls this function is just a quick function that I wrote for something. And it has this branch for some historical purpose. But actually this function will never get called with x greater than y. You're like, well, now you tell me. Right?

But the way we can think about this is that there is an assumption that x is going to be less than or equal to y, right? This is sometimes referred to as a precondition or a contract for this function. The function is promising to do something, but only if you satisfy this assumption. And if you don't satisfy the assumption, the function says, I don't care what happens. I only promise that I'm not going to fail when this assumption is satisfied. And it's the responsibility of the color to make sure that this condition is never violated, right?

So how would we encode that constraint when we're solving for equations? Well, essentially what we have is we have this set of constraints that tell us whether this branch is feasible. And on top of the constraints that we already have we need to also make sure that the precondition, or the assumptions are satisfied. Right?

And now we want to ask, OK, so can I find an x and a y that satisfy all of these constraints together with these constraint that I have on the input, with these properties that I know that the input must satisfy? And once again you can see that this constraint of x less than or equal to y is the difference between this constraint being satisfiable, and this constraint once again becoming unsatisfiable.

That's a very important issue when dealing with analysis, especially when you want to do this marginally at the level of individual functions at a time. It makes sense to know what the assumptions are that the programmer had in mind when writing this function, because if you don't know what those assumptions were you could say, yeah, here are some inputs where it's going to fail only for the programmer to dismiss myth that by saying, oh, but those inputs are not possible, or those inputs can never happen.

Other questions? All right. So how do we do this in a more mechanical way? So there are two aspects to this problem. Aspect number one is how do you actually come up with these formulas? So in this case it was kind of intuitive how we came up with the formulas. where we were just working through it by hand, but how do you come up with these formulas in a mechanical way?

And aspect number two is once you have the formulas, how do you actually solve them? How can you actually solve these formulas that describe whether your program fails or not? And I'm actually going to start with that second question. Given that we're able to reduce our problem to these formulas that involve integer reasoning that involved in the case of programs generally you care about bit vector reasoning. [INAUDIBLE] programs, a lot of times, you care about arrays. You care about functions.

And you end up with these giant formulas. How in the world do you actually solve them in a mechanical way? And a lot of the technology that we're talking about today, and the reason why we're actually talking about it as a practical tool, have to do with tremendous advances in solvers for logical questions. And in particular, there is a very important class of solvers called satisfiability modulo theory solvers, often abbreviated as SMT. But a lot of people in the community would argue that the name is not a particularly good name, but it's the one that everybody uses and it has stuck.

What you need to know about these SMT solvers is that an SMT solver is an algorithm essentially that given a logical formula will give you one of two things. it will give you either a satisfying assignment to the formula, or it will tell you that the formula is unsatisfiable. And that there is no possible assignment to the variables in that formula that will satisfy these constraints that you defined.

Now in practice, if this sounds a little bit scary and a little bit like magic, it is a little bit scary. A lot of the problems that these SMT solvers have to solve are NP-complete in the best case. All right? the nice ones are NP-complete. The hard ones can get much harrier than that. So how can we have a system that relies as its primary building block on solving NP complete PSPACE-complete problems? And still have something that works in practice?

And part of the answer is that for a lot of these solvers there is a third thing that they can tell you, which is, I don't know. And so part of the beauty of these solvers is that for practical problems, even for very, very large and complicated practical problems, they are still able to do better than simply telling you, I don't know. They are still able to give you either a guarantee that this set of constraints is unsatisfiable or an actual satisfying assignment that tells you exactly what the answer is. Yes?

**AUDIENCE:** [INAUDIBLE] For example, [INAUDIBLE] specification I don't think you said anything about how many bits are used to store an integer. [INAUDIBLE]

**ARMANDO SOLAR-LEZAMA:** That's a very good question. And that really has to do with how you define your constraints, right? So If you look at our simple example from the beginning, in this case, we assume that these were the integers as learned in elementary school. And that we completely decided to ignore overflow errors. If you care about overflow errors, if overflow errors are actually essential to the kind of bugs you're trying to find, this would not be a good way to set up the problem.

What you need is to represent these not so fast integers, but as bit-vectors. And the moment you represent them as bit vectors you have to have a bit width in mind. And this goes back to what this modular theory aspect in the solver means. What this modular theory aspect means is that the solver is actually extensible with different theories.

The most popular theories are the theory of bit-vector which are fixed length bit-vectors. That means that if you're interpreting your formulas in this theory of fixed length bit-vectors you have to fix the length of the bit-vectors. And you have to explicitly specify that these are going to be 32-bit bit-vectors, or 8 bit bit-vectors, or 64-bit bit-vectors.

**AUDIENCE:** So if you wanted to make the the bit symbolic [INAUDIBLE], like this is an x bit, is that--

**ARMANDO SOLAR-LEZAMA:** So there's another theory which is called the theory of arrays. And we'll talk a little bit more about it, where unlike the bit vector theory, which is designed to be for fixed length things the theory of arrays is meant to be for collections where you don't actually know the size a priori.

Now in practice nobody uses the theory of arrays to model integers, for example, because it's too expensive. It becomes way more expensive to reason about when you don't know what the bound is. So generally people use fixed length theory of bit-vectors when reasoning about integers or characters even.

Another very common theory is the theory of actual integer arithmetic, and in particularly linear integer arithmetic. This is a theory that people like a lot because it can be reasoned about very, very efficiently, but it's not particularly good when you're reasoning about programs, because in general you really do care about overflow issues. But it's actually very widely used for many, many things.

The other theory that you're likely to see people using is the theory of uninterpreted functions. So what does it mean, the theory of an uninterpreted function? It means that you have a formula where somewhere in your formula you know that you're calling a function, but you know nothing about that function other than the fact that it is a function, that if you give it the same inputs you get the same outputs in return.

And it turns out this is very, very useful sometimes when trying to reason about things like if you floating point code, modeling, sine, cosines, square roots can be very messy and expensive, but you can say, look, I don't actually care about what the sine function does. I don't care about what its output is. All I know is that if I call the sine function in many different places with the input I will get the same output. And that's enough for me to reason about my code.

And so the most common ones you will see when analyzing real systems are bit-vectors to deal with integers, and logs, and pointers. Actually, pointers are often represented with integer because you're generally not going to be doing complicated bit whittling on pointers. Sometimes you will and then you can't use integers anymore. So OK.

So that's all well and good. That's what an SMT solver can do for you. How does it actually work? What's inside it that makes it work? And SMT solvers actually rely on our ability to solve SAT problems, on our ability to take problems involving just purely Boolean constraints and Boolean variables, and telling us whether there is an assignment to these Boolean variables that is satisfiable or not.

And this is the kind of thing that for many, many years people in undergrad have been taught that actually this is an NP-complete problem. The moment something reduces to SAT you know you shouldn't do it, but it turns out that we actually have some very, very good SAT solvers out there. Probably most of you even built one as part of 6005. Am I right? Or some of you did.

So I'll tell you the basic idea behind how SAT solvers work. And the basic idea is that you take all your constraints on your Boolean variables and you put them into a database. And what is a constraint? Is this too small or can people in the back read this?

**AUDIENCE:** Too small.

**ARMANDO SOLAR-LEZAMA:** Too small? OK. Let's see if we can make this bigger. Is this a little bit better?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** OK. Well, here's what I'll do. I will annotate and I will narrate it as I go. And I'll post the slides later. So people can see what it says. So what we have here in SAT problem is that we have all these variables that represent Boolean unknowns, right? We want to know is it possible for x to be true, and y to be true, and z to be true at the same, for example. Right? And these are our unknowns.

And all the constraints are in conjunctive normal form. What that means is all our constraints are of the form either x1 is true, or x2 is true, or x3 is true, for example. Right? So what we have is we have all our constraints in this form and some of them might say, well, either x1 is true, or x2 is false, or x3 is false. Right? So we have constraints. All our constraints are of this form.

And you probably remember from discrete math that any Boolean formula can be represented in conjunctive normal form. And it has the added benefit that it's actually very, very easy to translate from arbitrary representations of a formula to these conjunctive normal form formula, which means whatever representation you're using to represent Boolean formulas, you can very easily convert it to this format.

So what we have is we have a database with lots of constraints of this form. And what SAT solver is going to do is going to pick one of these variables at random. Let's say it's going to pick x1. And it's going to say, why don't we set x1 to true? I don't know anything about this problem. Might as well try selling it to true. And then what will happen is you'll have some constraints that mention x1 and let's say that you have a constraint that says either x1 is false or x7 is true. Right?

So if you know that x1 is true and you know that either x1 is false or x7 is true, what do you know about x7?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Yeah. It has to be true. Right? Because otherwise this constraint would not be satisfied. And so now you've propagated this assignment from x1 to x7. And let's say now you pick some other random variable. You say, well, what about x5? Why don't we try x5 being true? Right? And now let's say that you have a constraint that says, well, either x7 is false, or x6 is true, or x5 is false. Right?

So I have x5 being true and I have x7 being true. So that means x6 now has to be true. Right? Because otherwise this constraint would be violated. And so from that the system infers, OK. So x6 has to be true. And it keeps at this process essentially trying out assignments. And then looking at all the available clauses, and looking at, hey, are there are other things that are implied by the assignments that I have so far?

And following those implications until one of two things happens. Either you keep following implications and trying random things and eventually you have set a value to every single variable without ever running into a contradiction. And then you're done. Right? You found a satisfying assignment, or what can happen is you run into a contradiction. You run into a place where there was a clause that forced x4 to be true, except there was another clause that forced x4 to be false.

And if there's one rule of Boolean algebra that everybody should know, is that you cannot have a variable be true and be false at the same time. Right? And so what that tells you is you've run into a contradiction. You clearly did something wrong in one of these random assignments that you were trying. So now let's analyze this contradiction. Let's figure out what were the assignments that led to this contradiction.

And based on the assignments that led to that contradiction, let's come up with a new conflict clause that summarizes that contradiction. So in this case, what would happen is that you have x1 being false, and x5 being false. And x9 being false, right? So essentially what this is saying is that based on what I learned from these random assignments I discovered that one of these things has to be true, that it cannot be the case that x1 is true, and x5 is true, and x9 is false. That cannot happen.

And I know that cannot happen because when I tried that things blew up. I ended up with a contradiction. And so what SAT solver is doing is trying random assignments, propagating them through. When it runs into contradictions it's analyzing the set of implications that led to that contradiction. And summarising that in a new constraint that will make sure that it never runs into this contradiction again, that it never runs into this particular problem again. Other questions? OK.

So so far so good. So we can't really think of the SAT solver as just a black box that given a Boolean constraint it can either say, no, this Boolean constraint is unsatisfiable, or it can say, yeah, here's a satisfying assignment to that Boolean constraint. So SMT solvers are built on top of SAT solvers. And what they're able to do is they're able to combine the power of the SAT solver to solve these NP-complete SAT problems with domain specific reasoning to reason about the different theories that are supported.

So to give you an idea of how it works, and this is going to be a fairly high level, but to give you an idea of how it works let's say that you have a formula like this, right? So you say x is greater than 5 and y is less than 5. And either y is greater than x or y is greater than 2. Right? So is that satisfiable? Can we find a satisfying assignment for that?

So what an SMT solver can do is separate out the part of this formula that requires domain reasoning, that requires reasoning in the theory, in this case, of integers. With the part of this formula that is just the Boolean structure. So if you separate the Boolean structure here, essentially what you're saying is that there's some formula, F1 and some formula F2, and either F3 or F4. Right?

And now this is a purely Boolean problem, right? It's just a problem of can I find a satisfying assignment for that? Is there a satisfying assignment for that? And, again, this is just a Boolean formula. Goes to a SAT solver and the SAT solver can say, yeah. I can find a satisfying assignment for this. And I can find a satisfying assignment by making this true, and this true, and this true. Right? It's a satisfying assignment for the Boolean formula.

So now we have a question that we can go and ask the domain specific solver. In this case just a linear arithmetic solver. So we can go to the linear solver and say, hey, so the SAT solver claims that this is a reasonable assignment, that if I can make that assignment work, then my formula will be satisfied.

So I can go and say, well F1 was actually this, and F2 was this, and F3 was this. So I can ask a theory solver, is it possible to get an x and a y such that x is greater than 5, y is less than 5, and y is greater than x? Right, so now this is a question purely about linear arithmetic. There's no Boolean logic involved. And what's the answer? No. Right? And there are traditional methods to solve these kinds of your problems.

You could use the simplex method, for example, to solve systems of linear inequalities. There's lots of methods that you can use to solve systems of linear inequalities. The point is the theory solver knows about all of those. And the theory solver can say, no. This will not work. This is an assignment that will not work. And so the theory solver can now go back to the SAT solver and not just tell the SAT solver, hey, that thing that you did, that didn't work.

But it can also give more of an explanation. So in this case, what you can conclude from the fact that this didn't work is that actually in addition to satisfying this formula you also want to satisfy the fact that I cannot have F1, and F2, and F3, right? My theory solver has told me that these three things are mutually exclusive. I cannot satisfy all three of them together.

And so now that's a piece of information that I can go back to the SAT solver and ask the SAT solver, hey, can you give me a solution that satisfies not only the constraint that you had in the beginning, but also this new constraint that the theory solver discovered? Right? So now is there some other assignment that satisfies now both of these constraints?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Yeah. So there's an assignment where this becomes false. And this becomes true. And that's an assignment that satisfies the constraint on the top. It satisfies the constraint on the bottom. And so once again that's an assignment that leads to a new constraint. So this constraint now goes away. We don't care about it any more. We have a new constraint that we can ask our theory solver, hey, it this possible? And in this case the theory solver says, yeah. That actually is possible. You can make y equal 3 and x equal 6. And it works. Right?

And so now you have an assignment that satisfies the formula in the theory and that satisfies the Boolean structure behind this assignment. And with that the system can come back and tell you, yeah. Here's an assignment that satisfies all your constraints.

And so it's this interaction back and forth between the theory solver and the SAT solver. And really the ability to be able to reason about very, very large and very complicated Boolean formulas. That's what makes symbolic execution possible.

So now that we have that the next question is, so how do we go from a program to a constraint that we can give to an SMT solver? Yes?

**AUDIENCE:** Sorry for going back.

**ARMANDO SOLAR-LEZAMA:** Sure.

**AUDIENCE:** [INAUDIBLE] previously. But could you run me again the whole issue of constructing the SMT statements? Is it an NP-complete or is it not? [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** So the problems that the SMT solvers are solving, those are NP-complete problems in the best of cases. So SAT itself is the canonical NP-complete problem, but a lot of solvers these days even include support for some theories that are outright undecidable. So--

**AUDIENCE:** So how do you approach that in your system?

**ARMANDO SOLAR-LEZAMA:** Well, at the end of the day what you get is you're going to create a constraint from this program. You're going to give it to the SMT solver. And the fact that these are NP-complete problems, or the fact that they're unsatisfiable, what it means is that if you're lucky, you will get an answer in seconds.

And if you're not lucky, then it might take longer than the age of the universe for the thing to give you an answer.

**AUDIENCE:** OK. How often do you run into cases where your system just flat-lines and says, sorry, I just can't figure this out yet? Has that ever happened or is that just--

**ARMANDO SOLAR-LEZAMA:** Yes. Yes, it does happen. And a big part of the engineering of these kind of tools is making sure that this happens as infrequently as possible. And part what makes this work at all is that we're not solving random SAT problems. We're not solving completely random bit-vector problems. We're solving problems that have a certain structure to them that a person was able to look at it and least have some confidence that this worked, right? Build some argument in their head for why this worked.

And so what the solvers are trying to do is essentially exploiting that structure. And taking advantage, for example, the description that I gave you of what the SAT solver is doing internally, that's taking advantage of the fact that, yes. Your problem might have a million Boolean variables, but actually most of those variables are very tightly dependent on the values of each other. So the number of degrees of freedom in the problem is actually much smaller than what the million variables would suggest.

**AUDIENCE:** So you're saying is that this isn't an exam question. This is real life. And someone built this system. It was supposed to work and make sense. So it's probably not going to be one of those wildly bizarre theoretical [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** That's right. And in practice what happens and when you use this tool is the thing is you always do is set timeouts. So generally, what happens is because it's exponential, exponential doesn't mean that you can't do it. Exponential just means that there's a brick wall, that before that brick wall things will work, and in fact, they will work really fast. Right? The exponential works in both ways.

Yes, when you're going out then things are growing very quickly, but when you're going toward smaller problems, or simpler problems things are also getting faster very, very quickly. So in general what that means is that lots of problems finish very, very quickly. And then some problems timeout. And the key is to engineer things in such a way that among the problems that finish quickly are actually problems of practical use. Or problems that will actually point you to security vulnerabilities in your system, will point you to bugs, will point you to a path that you maybe haven't explored before, or inputs that will take you down paths that you hadn't explored before.

**AUDIENCE:** Thanks.

**ARMANDO SOLAR-LEZAMA:** Other questions? All right. So we know how to go from a formula, from a set of constraints, to an answer that will either say, yes, this formula has a solution. And here's a solution, or no, this formula is unsatisfiable. There is no input that satisfies this. So now how do we get a formula from a program?

So one of the things that you have when you're doing symbolic execution is that when you get to a branch and you don't know which direction the branch is going to go. Now there are two possibilities that you can do in that case. One is to do what we did in the early example, which is just to say, I'm going to take both branches at the same time. I'm going to collect what happens in mode's branches, merge at the end.

That is a strategy that is often used when you're trying to get very strong guarantees in general. But it's a strategy that doesn't work too well with modern and SMT solvers. So often people prefer to do one path at a time exploration. And what that means is that you're going to pick a path down your program. And then you're going to create a formula for that path. So you're going to ask, fine me an input that goes down this path and that satisfies my constraint, or that violates my property, that goes out of bounds in my buffer, or that causes a null pointer error.

And then if you can't find one then you try a different path and a different path. And you do these path explorations one at a time. So that's the strategy that we're going to talk about now. It's a little bit easier to describe how to do it. So let's say that we have a problem like this. So, by the way, I switched representations. So I'm not representing the program as a block of code and representing it as a control flow graph. Is everybody here familiar with a control flow graph? Or is anybody here not familiar with a control flow graph? It's just a representation of a program that makes branches more explicit.

So let's pick a path. And so let's say that we care about this path, right, a path that starts at the beginning and takes us all the way down to the point where we are asserting false. And we want to know, is this path feasible? Could the program go down this path? So as we're going down this program we're going to keep two things.

We're going to keep an environment that keeps track of the symbolic values of the different variables. And in addition to that, we're going to keep around an environment for constraints.

And these constraints are essentially going to keep track of all the relationships between these variables as well as any assumptions, whether they were assumptions that were made at the beginning, or assumptions that come from the branches that you are taking.

So in this case, when we start down this path we get to t equals 0, so our state is x, y, and 0. And so far we have no constraints because we didn't have any constraint in the beginning. So now we're going to take this branch and, again, because we've made a decision that we're going to go down the path to your right, then we know that this path will only happen when?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** That's right. So we get our first constraint that says, x is greater than y. Right? So now down here we're looking at t equals y. Now in this case because we're going only one path at a time we don't actually need to introduce a new variable for t necessarily. We can just say, OK. t is equal to y. So that means that t is no longer 0. It's now y. Right?

And then keep going. We get to this point. Now we hit another branch. What's a new assumption that we have to make if we're assuming that we went down this path? Just t less than y, right? And what is t? Right.

So in fact if we look up t, so t has the value y. We look up y. y also has the value of y. So this constraint actually translates to y less than y. So what does this tell us? It tells us that in order to make it to this point, in order to make it to a assert false, all of those things have to hold. Can they hold? Clearly not. Right? y less than y alone is already sufficient for things not to hold.

And so that tells us immediately that this is unsatisfiable. And this is often known as a path condition. This is a condition that has to be true in order for the program to go down that path. And so we know that this path condition cannot be satisfied. And therefore, that it's impossible for the program to take this path. So this path is now completely eliminated. We know that this path cannot be taken. And, in fact, so this constraint we're actually going to just keep them around as the condition itself. All right?

So what about a different path? So now we're trying this path. So what would be the path condition for this? So, again, our symbolic state starts with t equals 0, and x and y equals to just the variables x and y. And now how does the path constraint look like in this case? So by the time we get here how does the path condition look like?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Right. So in this case [INAUDIBLE] this is true and this is false. So in this case it says, OK. x is greater than y. And we are setting t to be equal to x. So then when we get here we have x is less than y. Right? And once again it's very clear that this path condition is unsatisfiable. Right? We cannot have x greater than y and x less than y at the same time. There's no assignment to x that will satisfy both of those constraints.

So what that tells us is, again, that this other path is also unsatisfiable. And now at this point we've actually explored every possible path in our program that could lead us to this condition. So we can actually establish and certify that there is no possible path that will lead to an assertion failure. Yes?

**AUDIENCE:** The way you just presented it, it makes it look as if you would explore every possible branch. I mean, one of the advantages of symbolic execution is that you're trying to prevent [INAUDIBLE] a need of exploring all possible [INAUDIBLE] exponential. So how are you avoiding that over here?

**ARMANDO SOLAR-LEZAMA:** That's a very good question, right? So in this case essentially what you have is you have a trade off between high symbolic and how concrete you want to be. Right? So in this case we are not as symbolic as the first time around when we were visiting both branches at the same time, but in exchange for that our constraints became very, very simple. Right? So the individual path by path constraints are very simple, but you have to do this over, and over, and over again to explore all the different branches.

And there are exponentially-- all the different paths. And there are exponentially many paths in a program. Now there are exponentially many paths, but for every path in general, there's also an exponentially large set of inputs that could go down that path. So this already gives you a big benefit because instead of having to try every possible input you're only trying every possible path. But can you do better?

And this is one of the areas where there's been a lot of experimentation in the area of symbolic execution. When you do path by path reasoning? When do you do all paths at the same time? And one of the things that you saw, for example, in the [? Clee ?] paper is a set of heuristics, and a set of strategies they used to make the search tractable.

For example, one of the things that they do is that they are exploring path by path, but they're not exploring completely blindly. And they are also checking the path conditions after every step. So that, for example, if here instead of just assert false, if this were a very complex program tree, control flow graph. You don't wait until you get to the very end to check whether the path is feasible. The moment you get here you know that this path is unsatisfiable and you never go down this direction. You always go in the other direction.

So pruning the paths early helps cut down a lot on the experiential blow up. And exploring the paths intelligently helps a lot in preventing blow up. A lot of the practical tools that are used today, some of the things that they will do is they will actually start with some random testing to get an initial set of paths. And then they will start looking for paths in the neighborhood of those paths. They will start asking questions like, hey, the random execution went down this branch. What if I flip this branch? What if I flip this branch? What if I flip this branch? What happens in those paths?

Can be particularly useful, for example, if we have a good test suite. And you run your test suite and you find, OK, there is this piece of code that nothing in my test suite exercised. So what you can do is you can take the path that got closest to exercising that, and then ask, hey, can I change this path so that it goes down this direction instead?

And so in general, the moment you try to do all paths simultaneously the constraints start becoming intractable. And it's the kind of thing that you can do if you're doing one function at a time. For example, if you're doing one function at a time then it is generally feasible to explore all the paths in a function together. If you're trying to do larger units, then generally you have to go with path by path exploration. Are there other questions? Yes?

**AUDIENCE:** You referenced how [INAUDIBLE]. How does it do that again? What's the [INAUDIBLE]?

**ARMANDO SOLAR-LEZAMA:** So the most important one really is this idea that for every branch, you check your constraints to check whether that branch can actually go both ways, because if it cannot go both ways then you save a lot just going in this direction of where it can't go. Beyond that I don't remember the specific strategy that they use for searching paths that are more likely to give good results.

But pruning is really, really important. OK. So far though we've been talking mostly about toy code in the sense that it's only integer variables, branches, very simple stuff. Right? What happens when you have a program that is more complicated? And in particular, what happens when you have a program that involves the heap? Right?

So the heap has historically been the bane of all program analysis, analysis that were so clean and so elegant in the days of Fortran, completely blow up when you try to run them on a C program where you're allocating memory left and right. And you have aliasing. And you have all the messiness that comes with dealing with program allocated memory. And with pointers and pointer arithmetic. And this is one of the areas where symbolic execution really shines in the ability to actually reason about these kinds of programs.

So how do we do it? Right, so let's forget now for a moment about branches, and control flow. We have a trivially simple program here. All it's doing is it's allocating some memory. It's zeroing it out. It's getting a new pointer y from the pointer x. It's writing something into y. And then it's checking, hey, is the value stored at pointer y equal to the value stored at pointer x? And just from your basic knowledge of C you could see that, no. Right, that this assertion is actually violated because x got zeroed out and y has 25 in there, but x is pointing to a different location. Right?

So far so good. The way we're going to model the heap and the way the heap is modeled in a lot of these systems is by not thinking of the heap as a heap, but to thinking of the heat the way C likes for you to think of the heap, which is just a giant address base, a giant array where you can put things into.

So what does that mean? It means that we can think of our program as having this very big global array. And we're just going to call it MEM for now. Right? And it's an array that essentially is going to map addresses to values. Right? And what's an address? Well, an address is just a 64-bit value. And what comes after you read something from an address? It depends on how you're modeling memory.

If you're modeling it at the byte level, then what comes out is a byte. If you're modeling it at the word level then what comes out of it is a word. And depending on the kind of bugs that you're interested in, and whether things like memory alignment are an issue for you are not, you're going to model it a little bit differently, but generally memory is just an array from an address to a value. Right?

So an address is just an integer. Right? It's in some sense not that different from the way C thinks I'm an address. It's just an integer. It's just a value. It's just a 64-bit integer, or a 32-bit integer, depending on your machine. It just a value that indexes into that memory. And that you can put things in memory, read them from the memory.

So things like pointer arithmetic just becomes integer arithmetic. In practice there's a little bit of desugaring that has to happen because in C the pointer arithmetic actually knows about the types of the pointers. And things will be incremented proportional to the size, right? So this would actually be x plus 10 times the size of int. Right?

But what's really important is what happens when you're reading and writing from memory. So what used to be just a pointer reference from y to write 25, is now just I'm taking my memory array, and I'm indexing it with y. And I'm writing 25 to that memory location. Right? And this assertion now becomes, well, I am reading from location y in memory. And I am reading from location x in memory. And I am comparing them. And I'm checking whether they are the same or not.

It's a very, very simple reduction to go from program that uses the heap to a program the just uses this giant global array that represents the memory. And now what that means is that in order to reason about programs that manipulate the heap you don't really have to reason about programs that manipulate the heap. As long as you have the ability to reason about arrays, you are good.

Now here's a simple question though. What about the malloc? So one thing you can do is you can say, well, malloc, I can just take the C implementation of malloc and actually implement malloc like that. And keep track of all the pages that I have allocated and keep track of everything that has been freed. And keep a free list, and everything. It turns out for a lot of purposes and for a lot of classes of bugs, you don't need malloc to be that complicated.

In fact, you can get away with a malloc that looks like this, with a malloc that just says, I'm going to keep a counter for the next free memory location. And whenever somebody asks for an address, that address I'm just going to give this position and then increment the position. Right? And then return rv, in this case.

So one of the thing that is malloc is completely ignoring.

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** Yeah. Freeing, right? This malloc says, yeah, forget about freeing. There's no freeing. We're just going to keep walking through our memory allocating further, and further, and further and that will be it. And we don't care about freeing anything. It also doesn't really care about the fact that well, actually, there are regions of memory where you shouldn't be writing. There are special addresses that have special meaning that are reserved for the operating system.

It doesn't model any of the things that actually make writing a malloc function complicated, but at a certain level of abstraction, if you're trying to reason about some complicated code that does pointer manipulation. And you don't care about freeing memory, but you really care about is, am I going to write past the end of some buffer, for example.

Then this malloc might be good enough. And this is actually that happens very, very commonly when you're doing symbolic execution of real code. A very important step is the modeling of your library functions. And how you model your library functions is going to have a huge impact on the one hand on the performance and the scalability of the analysis, but on the other hand, on the precision.

So if you have a Mickey Mouse model of malloc like this, it's going to be very, very fast, but there are going to be certain classes of bugs that you won't be able to catch. Right? So and this model, for example, I'm completely ignoring the allocations. So if I have a bug because somebody is accessing unallocated space. Well, I'm not going to find it with this Mickey Mouse model of malloc. Right?

So it's always a balance between the precision of the analysis versus the efficiency. And the more complicated your models of standard functions like malloc get, the less scalable the analysis is going to be, but for certain classes of bugs you will need those models. And one of the big things in the [? Clee ?] paper was really having reasonable models for all the different libraries in C, all the different libraries that are needed in order to understand what a program is actually doing. So, OK.

So we've reduced the problem of reasoning about the heap to a problem of reasoning about a program with arrays, but I haven't actually told you how to reason about a program with arrays. And it turns out that most SMT solvers support a theory of arrays. And the idea is if a is an array, there's some notation to say, well, take that array and create a new array where location i has been updated to value e. All right?

So if I have array a and I do this update operation, and then I try to read the value k, then the meaning is that the value k is going to be equal to the value k at a if k is different from i. And it's going to be equal to e if k is equal to i, right? That's what updating an array means. That's what it means to take an old array and update it to be a new array.

And the nice thing about this is that if you have a formula that involves the theory of arrays, so, for example, I started with the zero array that is just zeros everywhere. And then I wrote 5 into location i, and 7 into location j. And then I'm reading from k. And then I'm checking whether that's equal to 5 or not. Then that can be expanded by using this definition to something that says, well, if k is equal to i then if k is equal to y, and k is different from j, then, yes, this is going to be equal to 5. And otherwise this is not going to be equal to 5, right?

And in practice SMT solvers don't just expand these into lots of Boolean formulas. They, again, use this back and forth strategy between a SAT solver and an engine that is able to reason about this theory of arrays in order to do it. But what's important is that by relying on this theory of arrays, using the same strategy we saw to generate formulas for integers you can actually generate formulas involving array logic, and involving array updates, involving array axises, involving iteration over arrays as long as you fix your path, these formulas are very easy to generate.

If you don't fix your paths if you want to generate a formula that corresponds to going through all paths, then it's also relatively easy. The Only thing is you have to deal with loops in more of a special way. Yes?

**AUDIENCE:** [INAUDIBLE].

**ARMANDO SOLAR-LEZAMA:** I don't know. So dictionaries and maps are actually very easy to model using uninterpreted functions. And, in fact, the theory of arrays itself, it's just a special case of uninterpreted functions. So more complicated things can be done with uninterpreted functions. In modern SMT solvers there is native support for reasoning about sets and set operations, which can be very, very useful if you're reasoning about a program that involves lots of set computations, for example.

When designing one of these tools the modeling step is really important. And it's not just how you model complicated program features down to your theories. So, for example, things like heaps down to arrays. And also the choice of what theories and the solver you use. And there's a large number of theories and the solver with different trade offs between how efficient they are versus how expressive they are.

And, in general, most of the production tools stick to the theory of bit-vectors and they might use the theory of arrays to model the heap if that is necessary. Generally production tools try to shy away from some of the more sophisticated theories, like the theory of sets just because by virtue being richer they also tend to be less scalable in some cases, unless you're dealing with a program that really requires exactly that kind of reasoning in order to work with.

Are there other questions? Yes?

**AUDIENCE:** [INAUDIBLE] research in symbolic execution, what are people focusing on and where is there room for improvement? [INAUDIBLE] applications.

**ARMANDO SOLAR-LEZAMA:** So one very active area of research is around applications. And looking at models that will allow you to discover new classes of bugs. So, for example, Nikolai, and Franz, and Xi Wang and I had a paper, what was it, last year when we were looking at using symbolic execution to identify coding your program that a compiler might optimize away. Security checks that might get optimized away by a compiler.

So it's very different from the question of will the program go down this path or not, but there is a modeling step to go from this high level conceptual question of, is there a code in my program that can be compiled away to an algorithm based on symbolic execution that will rely on the ability of symbolic execution to easily tell you whether the program can go down a particular path, or whether a particular path is feasible.

So applications is a big area, extending to newer classes of bugs, growing to new and different language features. For example, one of the things that is still fairly hard to model from using symbolic execution are very high level languages, like JavaScript or Python where you have a lot of very dynamic language features, but at the same time they are-- if any technique can work for the symbolic execution, it's definitely very good.

And, in fact, we had some work a couple of years ago using symbolic execution to reason about errors in Python programming assignments, for example. Yes?

**AUDIENCE:** So [INAUDIBLE]. How does [INAUDIBLE]?

**ARMANDO SOLAR-LEZAMA:** It is. So in the case of symbolic execution part of the problem is that your symbolic state, it's very hard to simply say, OK, I executed this instruction, and then this instruction, and then this instruction. The sequence is not there. There was some work a few years ago looking, for example, at very small pieces of code, but very critical, like a concurring data structure in operating system, or lock-free data structure and modeling the interactions between threads by essentially saying, every time there is a variable that could have been overwritten by something else, you replace that value with just a fresh symbolic value that says, I have no idea what this is.

And you generate constraints that relate to those symbolic values to symbolic values in other threads. And this has been used even to reason about things like missing memory fences, for example. And so it is possible, but the complexity grows quite a bit. And it becomes the kind of thing that you cannot no longer do at the scale of Microsoft Word, but you can do at the scale of, say, a concurring data structure, for example.

There had been other work though in the context of concurrency looking at, for example, can I use symbolic execution to reconstruct interleavings based on knowledge of how the program behaved as it was running, for example.

And so this opens a lot of possibilities, having this capability to ask very concrete questions about can my program run down this path. Being able to have symbolic values and ask questions, what values should these things have in order for the program to do something, or in order something to happen is a very powerful capability and there's a lot of applications that have been tried, but this is a fairly new piece of technology as far as technology for analyzing a program goes.