# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 11 TRANSCRIPTS

**L1 Slicing: Program Dependent Graphs**

>> Hello everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna be talking about Program Slicing and Program Dependency Graphs. The learning objectives for this lesson are to understand how to use dynamic analysis to locate the set of statements that compose a program slice.

To do this, we're gonna construct a program dependency graph. And we're gonna assemble dynamic slices using that program dependency graph. But first, what is slicing? A slice of variable v at statement s is the set of all statements involved in computing the value of v at statement s.

In fact, slicing is not a new technique, and has been around the field of program analysis since the 80s. So for example, if I asked you what is the slice of the variable i at line 5 in the sumUp program that we've been seeing. You would compute all of the statements in the sumUp program that are involved in computing the value of i at statement 5.

To do this, we would consider the control dependence of statement five and the data dependence of i. We can see that the value of i will be data dependent on line 3, also on line 5, and initially all the way back up on line 1. The data dependence on line 1 is introduced by line 4.

Line 4 is introduced because that's the control dependence that gets you to line 5. Computing this slice is actually a backward traversal of the program dependence graph. Remember program dependence graphs from the previous program analysis slide sets. A program dependence graph is a finite set N of nodes which represent the statements in a program, possibly with basic block super nodes.

There's a finite set Ed of edges from nodes i to j representing that node nj is data dependent on node ni. And a finite set Ec of edges ij representing that a super node nj is control dependent on a node ni. This is the same definition of program dependence graphs that we introduced before in this class when doing static analysis.

Computing a slice is simply a backward traversal of this program dependence graph from your slice criterion. That is a variable at a point in the program. Consider the slice(i @3.1), 3.1 meaning basic block 3, instruction 1. Our slice criterion (i @ 3.1) is where we begin our backward traversal of the program dependence graph.

This brings in the statements 3.1, 2.1, 1.2 and the start node. Remember the start node is there to handle dependencies that came from outside of the program. Now we can consider how to compute a slice statically. First build your program dependence graph. So you need to build your control dependence and your data dependence and merge those edges into your program dependence graph.

Here we have our edges of data dependence. Recall the two checks on a statement which define the data dependence. There exists a variable v that is defined at y and used at x, and there exists a path of nonzero length from y to x along which v is not redefined.

Similarly, you add the edges for the control dependence between the super nodes. Remember, y is control dependent on x if and only if x directly determines whether y executes. And we had those same two checks that we did previously. Now, given a slice criterion that is the starting point you can do a slice by backward traversing that program dependence graph.

And just collecting all of the backward reachable nodes in the program dependence graph. Now try one for yourself. What's the slice of the sum variable at line 4.1? It's actually the entire program. This is because 4.1 introduces a data dependence on 1.1 and on 3.2. 3.2 introduces data dependences on 3.1 and on 1.2.

Now try one for yourself. What is the slice of the sum variable at line 4.1? It's actually the entire program. If you follow the data dependence and control dependence arrows backwards, you will find that every node in this graph is actually reachable. But I know what you're thinking.

I thought we were done with static analysis. We are, we are, don't worry. Static slices are extremely imprecise. And this is because static analysis does not have dynamic control flow information. And static analysis can be very difficult to keep things straight when you're talking about data dependence. These two things lead to imprecision in your program dependence graph, which is only amplified into imprecision in your slice.

It is because of this imprecision that slicing is generally only performed during dynamic analysis. Static analysis using slicing tends to be very inaccurate.

**L2 Dynamic Slicing**

>> Hello everyone and welcome back to advanced topics in malware analysis. In this lesson, we're gonna start introducing dynamic program slicing. So, let's consider the differences between computing a slice statically versus computing a slice dynamically. A static slice starts at program criterion such as sum @ 4.1 and performs a backward reachability on the program dependent's graph.

If you were to compute this slice dynamically, you would start with an execution trace. That execution trace would collect all of the data and control flow during the program's execution with a given input. On an execution trace of n = 0, your execution would cover lines 1.1, 1.2, 2.1, and 4.1.

n being 0 means the while loop body will never execute. This makes the dynamic size much more accurate because you can directly follow the control flow and the data flow in the execution trace. A DynamicSlice of sum at line 4.1 is only data dependent on the value of sum at 1.1, and so that's your entire dynamic slice.

As we've seen before with dynamic analysis, it concretizes many of the assumptions that restrict static analysis. Dynamic slicing was actually introduced back in 1988. Dynamic slicing essentially makes use of all possible information about a particular execution of a program making it one of the most powerful dynamic analyses that you can do.

Dynamic slices are computed by constructing a Dynamic Program Dependence Graph, where each node represents an executed statement. And an edge is present in between two nodes if there exists a data or control dependence. A dynamic slice criterion is actually a triple of a variable, an execution point, and also the input that allowed you to execute the program dynamically to generate that Dynamic Program Dependence Graph.

The set of statements that is reachable in the Dynamic Program Dependence Graph from the criterion constitutes the dynamic slice. Dynamic slices are smaller, more precise, and often much more helpful to a user than static slices. Especially during malware analysis, where static analysis can be thwarted by many different malware tricks.

Computing data dependence dynamically is significantly easier than computing data dependence statically. The main reason for this is we no longer have to worry about aliasing. When you're tracing a program, a linear sweep over the trace will allow you to collect all data dependences. We now only need to compute a dynamic Def/Use chain.

This makes data dependence far easier to compute dynamically. Control dependence however can be a bit trickier to compute dynamically. Recall that we need to find the predicate instance that a statement is control dependent on. You might initially think that a simple backward traversal of the control flow graph that you collect dynamically would find the closest predicate.

This unfortunately is not the case. Recall our execution trace from before. If you did a linear sweep backwards from 4.1, you would mistakenly say that 4.1 is control dependent on 2.1. This is not the case, so we will need to define a new notion of dynamic control dependence.

That is a predicate that was observed during a specific execution of the program which to the best of our knowledge controlled if we executed that statement or not. We'll talk more about this in the next lessons.

**L3 Dynamic Data Dependence**

>> Hey everyone, and welcome back to advanced topics in malware analysis. In this lesson, we're gonna talk about dynamic data dependence. Dynamic data dependence can be traced using something called def use chains. You essentially instrument the executable to log all of the defines and uses of variables at runtime.

This is nice because unlike static analysis, at runtime, all memory locations will be resolvable because they will have concrete values that you can read at runtime. Look at the code example on this slide. We've instrumented each instruction to print out the def and the use of variables at runtime.

So when we execute the push RBP instruction, we will print out the address of this instruction, and that it defines RSP, and the memory location pointed to by rsp-4. It also uses RSP and the RBP registers, the same thing with the move instruction. We'll print out the address 01, and the define where it defines RBP, and the use where it uses RSP.

If we carry on to the next move, you see that it defines rbp+var\_X. At runtime RBP will have a concrete value, so we can simply print out the resulting address ofrbp+var\_X in the define list. And it uses the RBP register of course to calculate the final address and the EDI register where it reads the data from.

You can continue in this fashion, printing out at runtime, the concrete registers and memory locations that are defined and used by each instruction. Then when you execute the program with this instrumentation, you will get a trace that traces all of the defines and uses for each instruction. A backward linear sweep over this trace recovers all your data dependencies.

This is much simpler as you may note than static data dependence where a backward linear sweep was absolutely not possible. However, dynamic analysis has encoded the control flow into this trace because you are printing things out as they get executed. Therefore, you know the concrete control flow when you're looking at the trace.

Let's look at an example. What is the data dependence of the instruction at address 1? If we sweep back in the trace, we see that it uses rsp, and rsp was defined at the instruction at address 0. Let's try another one. The instruction at address 7 uses rbp and esi.

If we sweep back for who defined those, we see that rbp was last defined at instruction one, and esi actually came from outside of this function. So we could say start, like usual. And one more example, the instruction at address 10 is data dependent on whatever instruction previously defined rbp and the memory address 460004.

If we look back for who defined those two values, we see the memory address was defined by the instruction at address 7. And rbp was defined by the instruction at address 1, much simpler than static data dependence, wouldn't you agree? But how about control dependence? Control dependence is actually much more challenging to extract from a dynamic trace.

A linear scan will not work for control dependence, that's because an execution trace has no notion of the closing curly bracket. For an example, consider the two execution traces below. Would you say that the printf at line 4.1 is or is not control dependent on the while loop at 2.1?

You can't know from just this execution trace, because all you saw executed was the while and then the print. It maybe that the print is inside the loop, it maybe that the loop ends before the print. This is why a linear sweep will not work for reconstructing control dependence dynamically.

So, can we just fall back to our previous idea of static control dependence? Well, maybe we can, a dynamic control dependence algorithm, where you reconstruct control dependence from an offline trace analysis. First, assume there are no recursive functions, then assume you've already built a dictionary of control dependence for each instruction.

You could do this using our previous static control dependence. And then if you discover new code at runtime, you could update this dictionary by running a static control dependence algorithm on the newly discovered code. So imagine you have this dictionary CD, that goes from an instruction to any instruction i in the set of static control dependencies for the instruction i.

Once you have this dictionary, collect a dynamic control flow trace, this is simply logging the control flow or every instruction that gets executed. Then to reconstruct the control dependence for any instruction j in that trace, you traverse backwards starting from j in the trace, and find the closest instruction x, such that x is in the control dependence set for j.

j is therefore dynamically control dependent on x, and I'll explain why in the next slide. Consider j being control dependent dynamically on x, this is very problematic in the presence of recursion. Notice that each instruction in the trace can only be dynamically control dependent on a single instruction.

This is different from static control dependence where you could have a static control dependence on multiple instructions. But this is because in a concrete execution trace, every instruction can only come from a single execution path. And therefore, you only observe the control dependence of the previous instruction that your control dependent on.

Essentially, the control dependence right before you is the one who determined if you executed. So you're probably thinking, okay, data dependence control dependence, are we done yet? No, unfortunately not. The previously mentioned algorithms are basically offline graph reconstruction. This requires offline traversals of extremely long data structures and traces that have come from your data tracing and your control flow tracing.

This is gonna require very slow offline processing, and huge amounts of memory and disk space to store these giant traces. So, can we do better? Yes, we definitely can do better. There are efficient online algorithms to compute data dependence and control dependence online while the programme is executing, and then the output is simply the actual dependence.

So there's no need for a secondary pass over your trace to compute the dynamic dependencies

**L4 Dynamic Data Dependence Detection**

>> Hello everyone and welcome back to advanced topics and malware analysis. In this lesson, we're going to be talking about computing data dependence dynamically. In order to efficiently compute data dependence during the execution of a program, we're going to use what's called a Full Shadow of the registers and memory.

The basic idea here is to store metadata in a structure, which models every real data location. This allows us to compute online defuse tracking, specifically for each data location that an instruction can define. Keep a shadow copy to mark the last instruction which defined it. This can be implemented via a dictionary or similar data structure, which maps data locations that are registers or memory to the last instruction, which was observed defining them.

Consider when we statically computed the def use chains of instructions such as this push rbp. Push rbp is going to define the rsp register and the memory pointed to by rsp- 4. This instruction will also use the rsp register and the rbp register. If we were to change this to dynamic data dependence we would start by checking who was the last instruction to define the data locations that this instruction uses.

Recall this instruction uses the rsp and the rbp registers. Therefore, we can check our shadow registers for who the last instruction to define the rsp register and the rbp register are. We print those two instructions out, because that's the data dependence of this instruction. After this push instruction executes, we should update the shadow registers to reflect any data that was defined by this push rbp.

We can update our shadow registers to show that this instruction most recently defined the rsp register and the shadow memory to show that this instruction updated the memory pointed to by rsp. Be careful of the differences between updating shadows before or after instruction executes. For example, consider the push instruction that we just saw.

Since we updated the shadow memory after the instruction executed, we only need to update the shadow memory with the memory pointed to by rsp. Had we done this update before the instruction executed, we would need to update the shadow with rsp- 4. You also need to consider the granularity of the data that you want to shadow.

For example, do you want to shadow each flag individually or the flags register as a single unit? Or would you like to shadow memory at the granularity of 1 byte, 4 bytes, or 8 bytes. The less granular you shadow the less accurate your data dependence will be, because you will have to lose some information.

But it may not always be possible to implement the most granular solution for data dependence tracking. Let's take a look at some more instructions and how we would track the data dependence dynamically using shadow registers and memory. We've already seen the push instruction. Now consider the move instruction.

The move instruction uses the rsp value. Therefore, we can immediately print out that this instruction is data dependent on the most recent instruction to define the rsp register. After this move executes, it will define the rbp register. So we need to update the shadows to reflect that the rbp was most recently defined by this move instruction.

Similarly, consider these moves to memory. We can output the data dependence based on what data locations these moves use. And after the moves execute, we should update the shadow memory to reflect that those bytes of memory were most recently written by this move instruction. Consider the compare instruction, its data dependence will come from the data locations used, ie each of its operands.

In this case, we use the rbp register and the memory pointed to by rbp + var\_Y. After the compare instruction, we again need to update the shadow registers. In this case to reflect that this compare instruction most recently wrote to the rflags register. Again, you may choose to shadow flags at an individual flag granularity.

Consider the jump instruction the jns instruction actually does not need to update any shadows because it doesn't define any new data locations. It does use the flags register. So we should output that it is data dependent on the most recent instruction to define the flags register. The following move and regs eax instructions operate as you would expect, continuing to check what data locations are used by each instruction.

And then updating the shadow registers for the locations that this instruction defines. At the very end, notice that the unconditional jmp instruction has no data dependence and updates no shadows. Be careful of your space constraints, efficient dynamic data dependence must be done carefully. Because you will quickly run out of RAM space trying to shadow every single byte.

You cannot possibly pre-allocate 4 bytes of metadata for every byte of RAM. There's going to be implementation trade offs everywhere. Trading time for space in your algorithm to track the shadows and how you pack or unpack data to be more or less granular. For more information on this topic, you should read the paper, how to shadow every byte of memory used by a program, written by Julian Seward.

Yes, that's right, the valgrind author, he knows quite a bit about this topic. Shadow data is also used to implement what's called taint tracking. This is a technique often used in program analysis, especially binary analysis to mark inputs as tainted, that is one bit of shadow metadata. And then watch everywhere that taint propagates through the data flows during execution.

This is often used to detect sensitive data leaks from a program

**L5 Regions**

>> Hello, everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna be talking about how to compute online control dependence using a technique called regions. We can perform a similar online tracking of dynamic control dependence. Recall our definition of control dependence. Y is gonna be control-dependent on X if and only if, number one, X is not strictly post-dominated by Y, and two, there exists a path from X to Y such that every node in the path other than X and Y is post-dominated by Y.

We can then watch for these conditions at runtime and determine control dependence from the final execution trace. This has a number of benefits. First, we may not have to require ahead-of-time static control dependence analysis. Because if we can watch for these two conditions at runtime, they will produce the control dependence of the executed statements.

And two, this does not require logging the huge dynamic control flow trace, and instead only outputs the control dependence of the instructions. This does come with a serious downside, though, and that is that the algorithm is not nearly as straightforward as a simple offline computation of control dependence.

In order to monitor the execution for our control dependence conditions, we need to introduce a new concept called an execution region. A region is a set of statements that are executed between a predicate instance and its immediate post-dominator. We can then say that each statement instance xi is dynamically control-dependent on the predicate instance leading xi's nearest enclosing region.

Thanks to our definition of immediate post-dominator, this gives the very favorable conditions that regions can either be nested in one another or completely disjoint, but regions can never overlap. If you want full proofs of these two conditions, I recommend you read the paper Efficient Online Detection of Dynamic Control Dependence.

Let's look at an example of regions. Consider the program shown here that consists of a for loop, an if condition in the for loop, and a statement following the for loop. A region is a set of executed statements between a predicate instance and its immediate post-dominator. We need to compute the immediate post-dominators for the program shown below.

The immediate post-dominator for statement one is six, and the immediate post-dominator for statement two is four. Now consider an execution trace. We can now draw regions between every predicate instance and its immediate post-dominator. We first draw the outermost region from statement 1, 1 to statement 6, 1. Similarly, there is a region from the predicate 2, 1 to statement 4, 1.

There is a final region going from statement 1, 2 to 6, 1. We can say that each statement instance xi is dynamically control-dependent on the predicate instance leading xi's nearest enclosing region. Regions are either nested or disjoint, as we can see from this example, but they cannot overlap.

To understand those two properties of regions, let's look at them and consider our rules for control dependence that we laid out previously. Remember that each statement instance xi is dynamically control-dependent on the predicate instance leading xi's nearest enclosing region. Consider a proof that shows this property to hold.

The proof is let the predicate instance be pj and assume xi is not control-dependent on pj. Follow along in the diagram on the right side of the slide. If xi is not control-dependent on pj, therefore either, one, no path exists from pj to exit that does not pass through xi.

This would indicate that xi is a post-dominator of pj, contradicting the condition that xi is in the region delimited by pj and its immediate post-dominator. Essentially, if pj must go through xi to get to exit, then xi is an immediate post-dominator of pj. And therefore it can't be in the region which the immediate post-dominator of pj is enclosing.

Or there is a node xk between pj and xi such that xk has a path to the exit that does not pass through xi. Since pj's immediate post-dominator is also a post-dominator of xk, xk and pj's post-dominator form a smaller region that includes xi, contradicting that pj leads the enclosing region of xi.

Essentially, there's another predicate somewhere around xk, which would allow you to escape and get to the exit node. That would mean that there's a smaller region inside of there that xi is control-dependent on. That would be a contradiction saying that pj leads the enclosing region of xi. By contradiction, we have proven both possible cases, making the proof hold that pj will be the predicate which xi is control-dependent on when xi is in the nearest enclosing region.

The second property, regions are either nested or disjoint, but can never overlap. This one's a little more straightforward to see just based on the graphic on the right side of the slide. Assume, for contradiction, that there are two regions (x, y) and (m, n) that actually overlap. Thus, m must reside in (x, y) and y must reside in (m, n), as shown in the diagram.

This implies that there is a path P from m to exit that does not pass through y. This implies that there is a path P from m to exit that does not pass through y. Therefore, the path from x to m and P constitute a path from x to exit that does not pass through y, contradicting the fact that y is a post-dominator of x.

If y was not a post-dominator of x, then x and y would not be our region. That right there gives us our contradiction. And that's the proof that two regions cannot overlap. Interestingly, detecting regions at runtime is as simple as following a LIFO pattern. Because regions cannot overlap, as you enter a region, that region is the next one you will exit.

The implication of this is that the current sequence of nested regions for a current execution point can be maintained by a stack, called a control dependence stack. Each region is nested within the region immediately preceding it on the stack. The enclosing region for the current execution point is always the top of the stack.

Therefore, the execution point is always control-dependent on the predicate that leads the top entry on the stack. An entry is pushed onto the control dependent stack if a branching instruction is executed. The current entry is popped from the stack when the immediate post-dominator of that branch is executed.

That would denote the end of the current region.

**L6 Algorithms**

>> Hello everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna look at pseudocode algorithms to implement the region's method for tracking online control dependence. Now, idealistically this algorithm would follow what we described in the previous lesson. Every branch instruction would cause a push to the control dependent stack.

We would push that branch instruction along with its immediate post-dominator. Every merge, that is considering a control flow graph and where two branches join each other. We would pop elements off of the control dependent stack until we reached a predicate which we were not the immediate post-dominator for.

And if at any time we wanted to get the current control dependence, we would only need to look at what predicate was on the top of the control dependent stack. Consider this algorithm with this example code. If we start with a control dependent stack that is empty at the beginning of this code, we can mark the branching and the merging points based on the control flow graph.

The first instruction leads to a branch, and so we push the first instruction and its immediate post-dominator. Continuing our execution, we get to another branching instruction, so we push that instruction and it's immediate post-dominator. Continuing to execute straight line statements does not affect the control dependent stack. Finally, we reach a merge point where two branches of a control dependence graph come back together.

At this point, we keep popping items off of the control dependence stack until we reach one who this is not the immediate post-dominator for or the stack becomes empty. Since 5 is the immediate post-dominator for p2 at 1 and p1 at 1, we will pop both of those elements off of the control dependence stack.

That is to say, the enclosing region from predicate 1 ends at statement 5, and the enclosing region starting at predicate 2 ends at statement 5. Statement 5 itself is also a predicate. And so after doing our merge, we now must consider it as a branching instruction. That leads to pushing 5 and its immediate post-dominator onto the stack.

Continuing to execute, we see statement 6. It is also a branching instruction. And so we push statement 6 and it's immediate post-dominator onto the stack. Continuing to execute straight line code does not affect our control dependent stack. We get back to statement 6. Again, it is a branch.

So we push statement 6 the second iteration onto the control dependent stack along with its immediate post-dominator. Continuing to execute, we see that we reach a merge point. Again, continue to pop elements off of the control dependent stack until you reach one where you, the current instruction, is not the immediate post-dominator.

Again, reaching the n node, this is also a merge point. And so we continue to pop elements off of the control dependent stack. You should find that your control dependent stack is empty by the time you finish your execution. Control dependent stacks even handle recursion, which is rare in terms of dynamic analysis algorithms.

Control dependent stacks also handle recursion which is rare considering most dynamic analysis algorithms do not. Annotating control dependent stack entries with the calling context makes them recursion proof. Consider the code shown on this slide. It is recursive in that it calls f minus 1 within f function. Now consider the control dependent stack.

As we step through the execution trace, we simply annotate the calling context, in this case cc1 for the first call to function f. When we get to the predicate at line 1 of the second call to function f, we can still push that instance onto the control dependent stack, along with its immediate post-dominator.

The same thing for the third call to the f function. As we step forward, we simply pack elements off of the control dependent stack, within the same calling context. I've been calling this algorithm idealistic, because it does face some challenges in practice. The most obvious challenge you face, is that you have to know the immediate post-dominators ahead of time.

You could do this one of multiple ways, either first collecting a control flow trace, computing the immediate post-dominators and then updating your control dependence the next time you execute your dynamic analysis. Or you could compute these offline using a tool such as IDA Pro or Ghidra. And then feed the immediate post-dominator knowledge into your dynamic analysis tool.

The second challenge in practice is if you only have execution traces, it's very difficult to identify what is a branch until you have actually executed both paths. In general, implementations use a hybrid of offline and online analyses to compute control dependence. Similarly, merge points are not immediately obvious until you have seen two paths merge into the same instruction.

Consider the following execution trace. When executed with 0, 0 for x and y respectively, you produce the following execution trace, which does not execute the body of the while loop. It is not clear from this point if the IF statement is inside the while loop or not. It's not until you execute the program with x=1, y=1 that you observe the body of the while loop once and then again, the while predicate followed by the IF predicate.

**L7 Dynamic Slice Concepts**

>> Hello, everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna combine the dynamic control dependence and dynamic data dependence topics that we've been discussing, and wrap up the idea of dynamic slice computation. As a wrap-up, we have talked about dynamic slices, slice criterions, and the differences between static and dynamic slicing.

We've seen two approaches for computing a dynamic slice. Offline dynamic slicing algorithms that are based on inefficient tracing algorithms that trace the control flow and data flow and then compute dependence graphs offline. And two online algorithms that are more efficient in that they're able to detect dependencies online.

However, control dependence can be quite complex and somewhat limited when detecting it online. In fact, this form of slicing is called backward slicing. We are finding all of the instructions, which have executed previously, which got us to the value we have at the slice criterion. Consider the slice of i @ 3.1.

The backward slice here will consist of the start node 1.2, 2.1, and 3.1. All of the approaches we've discussed so far have been for backward slicing, that is, traversing the dependence graphs backward from a slice criterion. There is an orthogonal concept called forward slices. A forward slice of a program with respect to a program point p and a variable v consists of all statements and predicates in the program that may be affected by the value of the v at p.

Given a slice criterion, again, the starting point of the slice, a forward slice is computed by traversing the set of forward-reachable nodes in the program dependence graph. Looking at an example of forward slicing, we can consider a program dependence graph as discussed previously, and traverse the forward-reachable nodes from the slice criterion.

Consider a forward static slice i @ 3.1. Following the edges of the program dependence graph forward brings us to a forward slice, including statement 2.1, 3.1, 3.2, 4.1, and the END node. Notice that these are all the statements which could be affected by the value of i @ 3.1 in the future execution.

Try this one, the forward slice of the sum variable @ 3.2. This includes 3.2, 4.1, and the END node. Forward slicing has a lot of applications in cybersecurity. Slicing in general is by far the most common employed dynamic analysis. But if I asked a question, what program components might be affected by this particular execution?

That's a job for forward slicing. Slicing is very useful in determining which statements in a program can be affected by changes to a value v at a given statement. I'll give you two examples. As software is being maintained, modifications to parts of the program can lead to unforeseen side effects.

When part of that program is changed, these effects ripple through the program. These are essentially changes in the control and data dependence of the program. Forward slicing can expose these effects. And differential analysis of programs before and after being patched can expose differences in how a patch affected the program's execution.

Another example, more relevant to this class. When analyzing malware, we don't want to waste our time focusing on every single instruction. Instead, we want to focus on only the most important payloads in the malware. When malware receives input or checks for an environment condition, that value will only affect certain payloads.

Forward slicing from that input point can allow you to foresee what values will be important later in that execution. And finally, the concept of chopping. Given a source criterion and a target criterion, chopping determines what statements transmit the effects of S to T. Simply put, a chop is the intersection of a forward and a backward slice.

Computing these two is as straightforward as you might think. Step one, compute a backward slice from the target criterion. This will give you all prior statements involved in computing the value of T variable at T statement. The next step is to compute a forward slice from the source criterion S.

This will give you all future statements affected by the value of S variable at S statement. You then compute the intersection of the two graphs. What this gives you is the transmission of data and control from the source criterion to the target criterion. This is very useful for answering questions such as, if the malware receives this command and transmits the credit card information, what statements inside the malware consist of that specific payload?

And that brings us to our lesson summary. In this lesson, we've learned how to locate the set of statements that compose a program slice. We have talked about different algorithms to build dynamic program dependence graphs. And assemble dynamic slices using program dependency graphs to answer complex questions about data and control flow all in one algorithm.

And assemble dynamic slices using a program dependence graph to answer complex questions about a malware's execution.