# ECE 6747 Advanced Topics in Malware Analysis

# Module 5: Software Representations

# TRANSCRIPTS

## L1. Software Abstractions

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna start talking about how to automatically abstract software. So you can design algorithms that will automate the process of malware analysis. To do this, we need to start learning all of the building blocks of these algorithms.

We're gonna start with basic blocks and control flow graphs that are gonna allow your program to reason along different paths of execution. We're gonna then utilize those paths to understand dependencies, like where data the malware is using comes from or what systems the malware is trying to infect.

The reason we need these abstractions is because software by its nature is very difficult for humans to analyze and it's even harder for a machine to analyze. So if we're going to build algorithms that can automatically analyze malware samples for us, we have to think of abstractions that we can model our algorithms against.

For example, source code analysis, you'll be faced with programs written in multiple languages, millions of lines of code and tracking down external dependencies. This makes source analysis of programs extremely difficult for both humans and machines. Binaries on the other hand, you have to worry about how the binary is gonna behave across different machines and across different platforms.

You'll often run into a lack of semantic information, such as symbols that have been stripped out of the binary. Worst still, even if you have source code and binaries and test cases, you're still left with just running the binary and seeing how it behaves. This is gonna go back to your old friend GDB who not many people enjoy.

Me perhaps being the exception. So in this lesson, we're gonna cover all of the different ways to represent software so that you can write algorithms to automatically analyze it. We're gonna talk about basic blocks and control flow graphs. And we're gonna use those to build up data flow graphs and program dependence graphs that are gonna allow us to understand where data is coming from and going to in a malware.

We're gonna put those together into even better analysis algorithms that use the super control flow graph and the call graph of all the functions. I'll see you in the next lesson.

## L2. Basic Blocks

>> Hello, everyone. And welcome back to Advanced Topics in Malware Analysis. In this lesson, we're going back to the basics. Basic Blocks that is. The most basic representation of a program is its basic blocks. A basic block is a sequence of consecutive statements. In this case instructions with a single entry and a single exit.

Each block has a unique entry point and exit point. Control flow always enters a basic block at its entry point and exits at its exit point. Multiple basic blocks can enter a single basic block. But it must always occur at the entry point. Likewise, a single basic block can exit to multiple successors.

But it must always exit through the same exit point. There is no possibility of an exit or a halt at any time inside of a basic block. This is under the assumption that no external or asynchronous faults occur. Basic blocks are gonna allow us to build algorithms that can reason about how a program executes.

So we'll make this simplifying assumption that a basic block cannot stop or exit in the middle of a basic block. The entry and exit points of a basic block may coincide when the basic block is only a single statement. Or in the case of binaries are statements are single instructions.

We'll first talk about basic blocks at a source code level so that everyone understands how basic blocks work. Basic blocks are a nice software abstraction. Because they're valid at both the source code and the binary level. We'll start with this simple source code example of a function that computes raising an integer to a power.

The basic blocks lineup as follows. Block 1 covers lines 2, 3, 4, and 5. The entry is at line 1. And the exit is at line 5. You can see this in the source code. Because the only way to start executing this function is at line 1. And once you begin executing at line 1, control flow does not deviate until you get to line 5.

At line 5, you need to break and begin a new basic block. Because you could leave 5 and go to two different blocks. You can either go to block 2, that is line 6, or you could go to block 3, that is line 8. Line 6 and line 8 are single instruction basic blocks.

Because from either of them you flow into block 4, which is on line 9. Line 9 again is a basic block by itself. It has a single entry point, that is line 9. And a single exit, that is exiting line 9. Notice that line 9 is a basic block by itself.

Because even though you can come into line 9 from either block 2, or block 3, it still has only a single entry point and a single exit point. Block 5 includes only line 10. That's only the while statement. At first, this may seem confusing. But think about where you can enter line 10 from.

You can either enter line 10 from line 9, or from line 13 at the end of the while loop. Since a basic block can only have a single entry point that makes line 10 a basic block on its own. The body of the loop however, builds block 6.

That's because once you execute line 11, you definitely are gonna execute line 12. So lines 11 and 12 go in a basic block by themselves. The same thing with block 7, 8, and 9. Each of these are individual statement blocks. Because they all have their own entry point and different exit points.

Now let's talk about basic blocks on the binary level. Binary level basic blocks work the same as source code level basic blocks. This is actually the compiled version of the POW function that we just saw. I've inserted comments, so you can see where the assembly lines up with the source code.

The first bunch of instructions execute the if line from the source code. After the jns instruction, you're executing the power equals negative y line of the source code. The next two instructions perform the else branch where power equals y. Then the next three instructions perform the z equals 1.

Next in the compiled code, we can see that the body of the loop was actually moved above the loop condition. So loc 2B label actually points to the loop body. Where we see the instructions z equals z times x and power equals power minus 1. The loop condition has been moved.

Because as we've seen, compilers tend to prefer post tested loops. So the loop condition is now at loc 42 label. Where we see the while POW not equals zero condition. That's followed by the if y less than zero conditional. And the z equals 1 divided by z statement.

Finally at loc 60, we see the program setting up the return value and returning. Now that we've gone through the assembly language for the compiled version of that POW function. We can divide it up into basic blocks following the same rules of basic blocks that we saw before.

When we start executing at the first instruction of the power function, we're gonna keep executing every instruction along the way until we get to the jns instruction. At this point, execution could either go to loc 1A, or to the instruction immediately below jns. Because we can't have multiple different control variants in a single basic block.

That's where we have to break basic block 1. We pick basic block 2 up with the very next instruction from the mov all the way down to the jmp. Basic block 3 is the following two movs. These have a single entry. That is anyone who jumps to loc 1A and a single exit out of the bottom of the second mov.

Notice that we have to break blocks 3 and 4 into two different blocks. Because you can just jump directly to the label loc 20. Because you cannot enter execution in the middle of a basic block. That forces us to break blocks 3 and 4 into two different blocks.

Block 4 consists of the two movs and the jmp instruction below. The loop body is its own basic block. The same as we saw in the source code. Because you'll execute all of those instructions together whenever you jump to loc 2B. Control flow won't jump away or change until after the sub instruction.

Again, we see that the loop conditional has to be in a block on its own. The reason for this is because execution can enter that block either at the loc 42 label, or by just falling through from the loop body above. The jnz instruction can then either jump to loc 2B that's the loop body.

Or execution can continue at the second compare instruction in block 7. Similarly, block 7 contains the compare instruction and the jns for the if y less than 0. This is because there's two exits after the jns, either into block 8, or down to block 9. Block 8 is some more straight line instructions that perform that division.

And then block 9 stands by itself. Because it has two entries either from block 8, or from block 7. The more you practice with identifying basic blocks in a binary, the more they'll jump out at you. They're actually very straightforward to identify once you get the hang of the routine.

And even more interesting source code basic blocks do not always equal binary basic blocks. Even though basic blocks is a valid abstraction of software at both source and binary level. They're not comparable across both levels. We've actually seen this in the example we've already looked at. Compilers can rearrange the logic of a program, even when no optimization is present.

To show you what I'm talking about, look at the blocks in the source code compared to the blocks in the assembly. Notice that because the body of the loop was moved above the condition by the compiler, blocks 5 and 6 are not the same blocks. Block 5 in the source code is the while condition.

Whereas block 5 in the binary is the loop body. Just keep an eye out for that. While you can compute basic blocks on both source code and binaries, the two are not directly comparable.

## L3. Control Flow Graphs

>> Hello, everyone, and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna be learning how to construct control flow graphs. A control flow graph, or CFG for short, is the most commonly used program representation. A CFG abstracts the paths that might be traversed through a program during its execution.

I say might be traversed because a CFG is only concerned with the branching instruction. And a CFG does not consider any of the data that might drive the program down one path or another. To define a control flow graph, we say that a control flow graph G is defined as a finite set N of nodes and a finite set E of edges.

An edge i,j in the edge set E connects any two nodes Ni and Nj in the node set N. We often write G equals N, E to denote a control flow graph G, constructed with nodes given by the set N and edges given by the set E. In a control flow graph, each basic block of the program becomes a node.

Edges are used to indicate the flow of execution between different basic blocks. So a CFG edge (i, j) connecting basic blocks bi and bj implies that control can be passed from block bi to block bj. We also assume that there exists a node labeled Start in N that has no incoming edge.

The Start node is assigned outgoing edges to all other nodes in the CFG that have no incoming edges. We also assume that there exists a node labeled End in the node set N that has no outgoing edge. The End node is assigned incoming edges from all other nodes that have no outgoing edge.

The Start and End nodes are important to simplify our automated analyses. Basically, an analysis that you design can now assume that if it starts at the Start node and traverses the CFG, eventually it will arrive at the End node. This removes the ambiguity of having multiple entry or exit points within a CFG.

Here's an example CFG from the code that we've been looking at. You can see that the basic blocks have become nodes in the CFG, and the edges represent how the execution would go from one block to another. For example, notice that basic block 1 has edges going to basic block 2 and basic block 3 because, depending on the value of y, execution could flow from basic block 1 into basic block 2 or from basic block 1 into basic block 3.

CFG nodes are typically represented by only their basic block number. So we could simplify the picture of the CFG to look like this. You would then need to reference back to the code or the disassembly, where you got the original basic block numbers from, in order to dig into the code or instructions in each basic block.

But you may be thinking, haven't I seen this before? And the answer is yes. Any good reverse engineering tool like IDA Pro or Ghidra will give you some kind of graph view that represents the basic blocks within a control flow graph This allows you to easily identify how the control or execution flows between the basic blocks while you're reverse engineering.

## L4. Paths

>> Hello, everyone, in this lesson we're gonna learn how to reason along Paths within a control flow graph. Paths are a control flow graph representation that narrows in on only a single sequence of statements. And reasoning along Paths, as they would be traversed during execution, is gonna allow you to design algorithms that can automatically perform binary analysis.

Consider a control flow graph, G, with nodes N and E. A Path consists of k edges from the edge set E where k is greater than 0. So you could say a Path consists of edge 1, edge 2, all the way up to edge k. P denotes a Path of length k through the control flow graph, if the following conditions are true.

Given that node p, node q, node r, and node s are nodes that belong to the node set of the control flow graph, and 0 is less than i is less than k. That means 0, then i, then k in numerical order. So if an edge ei goes from node np to node nq, and another edge, ei plus 1, goes from node r to node s, then node q and node r must be the same node.

Put simply, every node in a Path must be reachable by a single traversal of the Path. That means you can go from the Path's first node to the last node by following the control flow graph. There's no breaks in between edges that make up the Path. We'll go back to the control flow graph that we've seen before as an example of how you can have Complete Paths or Subpaths through a CFG.

Our definition of a Path allows for two different types of valid Paths. A complete Path, which is a valid Path by our previous definition, which includes both the Start and End nodes of the CFG. A Subpath is a valid Path, by our previous definition, which forms a subsequence of a Complete Path.

So you may not have the Start node or you may not have some of the nodes at the end of a complete Path. I'll show you an example of each. If you look at the figure on the side of the slide, you'll see a Complete Path is formed from the Start node to node 1, to node 2, to node 4, to node 5, 6, 5 again, 7, 9, and End.

This is a complete Path because it includes the Start and End node. This is a valid Path because there's no breaks in the control flow graph. Every adjacent edge in the Path follows the previous edge. So there's no skipping over any edges in the control flow graph. To specify this Path unambiguously, we could use the edges themselves to define the Path.

So you could say, P1 is the Path that consists of the edges Start to 1, 1 to 2, 2 to 4, and so on. The set of dashed lines on the graph show a Subpath. So you could say, a Subpath P2 consists of nodes 5, 7, 8, and 9.

An invalid Path would be this Path that I'm showing now. From Path P0, you could say, start at the Start node, then go to node 1, then go to node 2, then go to node 3, then go to node 4, and so on. What makes this not a valid Path is because there is no edge from nodes 2 to 3 in the control flow graph Once you understand Paths, you can reason about the feasibility of a Path.

One of the most important Path analyses that you will code is Path feasibility. This is used in security for questions like, can a malware execute a certain payload, software engineering, and debugging to understand what program states can execute with other program states. And even compilers use infeasible Paths to eliminate dead code.

A Path P through the CFG is considered feasible if there exists at least one test case which, when input to the program, causes every node in P to be traversed. By this definition Subpaths can also be considered a feasible or infeasible Subpath. Don't consider bugs or exploits, in general, when you're computing Path feasibility.

These sort of out-of-bounds executions make it impossible to determine feasibility in a general sense. However, there are techniques that we will talk about which solve localized versions of Path feasibility. Compilers actually implement this frequently in order to do dead code elimination. Let's look at some examples of feasible complete Paths.

P1 and P2, shown here, are feasible and complete Paths. You can think of what the input to this code would need to be in order to execute every node along the Path P1 or along the node Path P2. These inputs may be different, but both Paths are still feasible if and if any input exists.

Two feasible Subpaths, P3 going from Start to node 1, to node 2, to node 4. You can imagine the input required to this code to generate that Subpath. Subpath P4 going from node 5, to 7, to 8, to 9, to End. Similarly, you can consider an input to the program that would allow the execution to go down that Subpath.

Two infeasible paths, P1 in this case, going from Start to 1, to 3, to 4, to 5, to 6, to 5, to 7, to 8, and to 9, then to End. This is infeasible because there's no input to the code which can allow the execution to go to both node 3 and node 8.

The same is true for P2. P2 is infeasible. Notice the Paths can be complete, but also infeasible. That's because the two checks are independent of each other. One thing to watch out for when you're trying to compute the number of Paths in a program, you may run up on an infeasible problem you're trying to solve.

A program with no conditional statements will have exactly one Path. It's just a straight-line program. However, every additional condition statement in your program is gonna increase the number of distinct Paths by at least one. Sometimes you can have a multiplicative effect with every conditional statement that you add.

And this is gonna cause the number of Paths to explode. This is a common problem in program analysis. When you're designing algorithms that reason about all possible Paths through a program, you will often hear researchers talk about hitting Path Explosion. That means that the code simply could not generate all possible Paths because of the multiplicative effect of conditional statements.

But when you have an algorithm that can reason along Paths, it allows you to make statements about the program much easier. Just like basic blocks gave analysis tools away to give structure to sequences of statements, many globally intractable problems can be solved along a single Path. Here's an example.

If I give you just this Path through the control flow graph that we've been seeing throughout the slides, you can reason about what value of Y produced this specific Path? Reasoning about all possible values of Y can be difficult or even impossible. But limiting your analysis to only a single Path allows you to solve it.

In the coming lessons, we're gonna learn how to teach an algorithm to figure out the same thing you just did in your head.

## L5. Dominator Analysis

>> Hello everyone. In this lesson, we're gonna be talking about dominator analysis. When you're building algorithms that can automatically understand binary programs, the most important analysis to implement are dependency analysis. You can look at a path and observe the dependencies between the nodes in the path. We saw an example of this in the previous lesson when we figured out the value of y that triggered this path.

A few other examples, the value of power depends on the value of y. That's a dependency that automated analysis tools could extract if you taught them how. These dependencies can be modeled in a way so that your algorithms can identify them. We're gonna talk about two very important types of dependencies in the coming lessons, control dependencies and data dependencies.

Control dependencies are based off of three key principles. Dominators, post-dominators, and immediate dominators, and immediate post dominators. We're gonna talk about those right now. A Dominator, we say that a node X dominates a node Y if all possible program paths from START to Y must pass through X.

Here's an example. Take the sum up code that we saw before and turn it into a control flow graph. This is our control flow graph and we can check which nodes dominate other nodes. For example, what nodes dominate node 4? Consider, all possible program paths from Start to 4 must pass through which nodes?

You would say the dominator set of node 4 is 1, 2, and 4. Let's think about that. All possible program paths from Start must go through node 1, node 2, and node 4. Not node 3, that's a conditional block that will only get executed sometimes depending on the value of the while loop.

4 is also in its own dominator set because naturally to execute up to and including block 4, you would have to include block 4. The strict dominator set is the dominator set of a node that does not include the node itself. This would be all nodes you have to execute from Start to the node you're checking, exclusive of the node you're checking.

So the dominator set of node 4, if you recall from the previous slide was 1, 2, and 4. Therefore we say the strict dominator set of node 4 is nodes 1 and 2. We also have the concept of the immediate dominator. X is the immediate dominator of Y if X is the last dominator of Y along a path from Start to Y.

So we would say the immediate dominator of node 4 is node 2. Dominators allow for backward reasoning within a program. Dominators will allow algorithms to determine backward control flow. Basically you can ask the question, who needs to execute for block X to execute? If we plot out all of the dominator set for every node in this control flow graph, we see that the dominators of the Start node is the empty set because no one needs to execute for the Start node to execute.

The dominators of block 1 are just block 1. Similarly, the nodes build up in the dominator sets based on who needs to execute for that node to execute. Notice that Start and End are not true nodes. As we discussed before, they're just pseudo nodes that we add to make our analysis a little easier.

So they're not in any of the dominator sets themselves. An interesting side effect here is that the dominator set of End, the End node are all blocks that get executed for any input. Take a minute to think about why that is. An equally important concept is the concept of post-dominators.

We say X post dominates Y, if every possible program path from Y to End has to pass through X. There's also similar definitions of strict post-dominator and immediate post-dominator. If we think about what is the post-dominator set of node 3, we can ask, on every possible program path from node 3 to the End node, what other nodes must the execution pass through?

The post-dominator set of node 3 is nodes 2, 3, and 4. Again, the node itself is considered in the post-dominator set but not the strict post-dominator set. And the immediate post-dominator is the node you would immediately step to on a path to the End node. Consider block 1, what is the post-dominator set for block 1?

Post-dominators allow algorithms to perform forward reasoning. You can ask questions like, if block X executes, then who else must also execute? Notice, the post-dominators set of Start includes all of the nodes that would execute for any input. Also interesting, the post-dominators set of Start equals the dominator set of End.

Consider carefully why this is the case. Another interesting side effect of dominators and post-dominators is it allows you to identify back edges in a control flow graph. A back edge is any edge in a control flow graph whose head dominates its tail. This is very useful when trying to locate loops in a large control flow graph.

A closed loop back edge is an edge whose head dominates the tail and posts-dominates the tail. This means you have a closed loop which enters and exits through the same node. You see this often with while loops. What would be different if you added a break statement into block 3?

Consider this, and we'll continue to talk about dependency analysis in the next lesson.

## L6. Control Dependencies

>> Welcome back everyone. In this lesson we're gonna put together those concepts of dominators and post -dominators to define control dependence. Control dependence probably one of the most important applications of dominators and post-dominators will allow us to understand what nodes control the execution of other nodes. We say a node Y is control dependent on a node X if and only if X directly determines whether Y executes.

In general, statements inside of each branch of a predicate are control dependent on their predicate. This is just a simplification, and you'll often have to check two criteria in order to be 100% certain of the control dependence of a node. When checking control dependence of a node, you need to check the following two criteria.

One, that X is not strictly post-dominated by Y. And two, there exists a path from X to Y, such that every node on that path other than X is post-dominated by Y. For example, if you have a control flow graph that has this node X and the node Y in it, and there is a path to end.

You can check the control dependence of Y by checking the following two criteria. First, check that X is not strictly post-dominated by Y. That means some edge exists from X to end, that does not pass through Y, or the case that X equals Y. But this extra edge allows X to not be strictly post-dominated by Y.

The other condition is that there's no way to escape executing Y, once X decides to execute towards Y. Put more scientifically, every node on the path from X to Y is post-dominated by Y. That means there's no escape route in the middle of the path from X to Y, that would avoid executing Y.

If these two conditions hold true, then Y is control-dependent on X. And that means that there is a predicate or a decision being made in X that chooses if Y is gonna execute. Now we'll consider the control dependence of the example CFG we worked with before. Remember the two checks.

X must not be strictly post-dominated by Y and there exists a path from X to Y such that every node on that path other than X is post-dominated by Y. Consider the control dependence of block 3. Let's check those two conditions. If we say, X is node 2 and Y is node 3, then indeed 3 is not in the strict post-dominator set of 2, and the edge from 2 to 3 includes only 3, and 3 is indeed in the post-dominator set of 3.

Therefore you'd say the control dependence of block 3 is block 2 or 3 is control-dependent on 2. Why is 3 not control-dependent on 1? Well, let's work out the checks. If X is 1 and Y is 3, 3 is not in the strict post-dominator set of 1. So our first condition actually holds.

But consider the edge from 1 to 3. That includes nodes 2 and 3. Now we need to check the post-dominator set of each of those nodes along the path. 3 is not in the post-dominator set of 2, 3 is in the post-dominator set of 3. But remember our second condition, on the path from X to Y, every node on the path other than X must be post-dominated by Y.

Since 2 is not post-dominated by 3, 1 cannot be in the control dependent set of block 3. The more examples of control dependence that you work out, the more you'll understand it. It's similar to math problems where the more practice you have, the better you get at identifying it.

Consider the control dependence of block 2, block 2 is tricky. Block 2 is actually control dependent on itself. If we check out the math, everything works out. Consider the path from 2 to 3 to 2 again and then to 4. If we say X is 2, and Y is the second iteration of 2, the strict post-dominator set of 2 is 4.

Now let's check our conditions. 2 prime is not in the strict post-dominator set of 2, so our first condition holds. The path from 2 back to 2 passes through 3 and 2 again. Let's check those post-dominator sets. 2 is in the post-dominator set of 3 and 2 is also in the post-dominator set of 2 therefore, 2 is control dependent on itself.

You often see this for loop conditions that will execute again every time the loop body is chosen to execute.. It may seem confusing at first, but what you're doing is unrolling the loop body. In fact, loop unrolling is a binary analysis term for the more formal term loop induction that you hear all the time in algorithm analysis.

This is a very interesting point where algorithm analysis meets binary analysis. Basically, one is building analyses from the top down and binary analysis building the analyses from the bottom up. This often blows the minds of new binary analysis students, because they never realised that the to actually meet in the middle.

Control dependence is not syntactically explicit. Don't assume that just because you see a condition statement, that means everything is gonna be controlled dependent on it. Consider block 4. What's the control dependence of block 4? Let's check our conditions. Recall our two conditions, and we're gonna say that X is 2 and Y is 4.

The strict post-dominator set of 2 is 5, so if we check our 2 conditions 4 is not in the strict post-dominator set of 2. So our first condition holds. But when we look at the path from 2 to 4 and includes 3 and 4, but 4 is not in the post-dominator set of 3.

Therefore, the control dependence does not hold. 4 is not control dependent on 2. Similarly, we can check the control dependents for 3 and 4, X = 3, Y = 4. You check your two conditions, and it turns out that 4 is control dependent on 3. This is an interesting result, because formally you can prove that the execution of block 4 only depends on block 3, and does not depend on block 2, even though they're in the same while loop.

Control dependence can be very tricky. Always consider your two conditions. Considering those can one statement be control dependent on two different predicates? Consider this code. Here's the control flow graph for the code. You can see that the foo function gets called depending on which predicate actually returns true.

If we work out the control dependence of these blocks, you can check if 3 is control dependent on block 2. Considering our two checks, you'll find that indeed 3 is control dependent on 2. But wait there's more. If you check if 3 is control dependent on 1 again you will find that 3 is control dependent on 1.

Therefore it is possible for a single statement to be control dependent on two different predicates. This is more obvious when you look at this code once it's been compiled in assembly language. You didn't think we were done with assembly language, did you? You can see that in order to get to the call to foo, you have to jump through to conditional branches.

And this sort of hints at the fact that you can be controlled dependent on multiple predicates.

## L7. Data Dependencies

>> Hello everyone and welcome back to advanced topics and malware analysis. In this lesson, we're gonna learn about data dependence. Data dependence can be checked by two conditions. First, is there a variable V defined at a statement Y? And used at a statement x. The second condition is that, there exists a path of nonzero length from Y to X along which V is never redefined.

You can think of this simply as, there's a piece of data, that one instruction defines, that gets used by another instruction later on, without any other instructions in between the two, redefining that value. Notice, that I'm talking about things in terms of instructions or statements. And that's because data dependence is calculated, per statement.

You very rarely aggregate data dependence results, to a basic block level. Control dependence, basic blocks. Data dependence, instructions. Here's an example of data dependence. Consider the source code for the power function, where we're going to raise one number to another. You can see on the slide, the source code and the control flow graph generated for that source code.

Now, let's keep in mind our two conditions for checking data dependence. We can look at a statement, say line 12. And we can check, what variables line 12 uses and where they're most recently defined. In this case, we see that line 12 is gonna use the value of power.

The value of power could have come from line 6, but it also could have come from line 8, depending on the control flow through the program. If you check your two conditions, that is there is a variable V, in this case power, defined at line 6 and used at line 12, that condition holds.

And there exists a path of nonzero length from 6 to 12, along which power is not redefined. And that's true. Therefore, there is a data dependence between lines 12 and 6. Similarly, the two checks work for line 8. Notice, there's one more data dependency here, and that is line 12 back onto itself.

This is actually two iterations of line 12. When you execute through the loop, line 12 will redefine power. And then on the next trip through the loop, you will use the value of power that was defined at line 12. Keep these in mind, you can say that the data dependence of line 12 is 6, 8 and 12.

But notice, we are not data dependent on the value Y, why would that be? That's because, just like control dependence, data dependence only covers direct dependencies, not transitive dependencies. So, while line 6 and 8 both rely on the value of Y, that dependence is for lines 6 and 8, it does not get passed on transitively to line 12.

Let's try another example, line 15. Line 15 uses the value of Z, in your mind workout, where the last place Z was defined, such that it's being used at line 15. The answer is 9 and 11. So, you would say, the data dependence of line 15 is statements 9 and 11.

Let's try line 14. Line 14 is a little trickier, because it depends on the value of the argument given to this function. When you implement data dependence, as you will in the lab, you can pretty much count arguments as their own thing, or just point the data dependence to the start block of each function.

The two data dependence checks, work exactly the same, if you're talking about binaries. Consider the compiled code, for that power function. You can see that on your screen now, if you pick a statement, say neg eax, that corresponds to the original source line, power equals minus Y or -Y.

And you can check, what values that statement uses, where they were last defined. And the second condition that, there exists a path of nonzero length, from where it is defined to where it is used, but it is not redefined along that path. In this case, there's a register dependence for the value that's stored in eax, to the previous statement.

Therefore, we can say the data dependence of the instruction at line .text 13, is the instruction at .text 10. Or more simply, instruction at address 13, is data dependent on the instruction at address 10. Remember, no transitive dependence. The variables that are used by statement at address 10, are not grouped into the data dependencies, for the instruction at address 13.

Let's try another one, the move instruction at address 10. In this case, we have a few things to account for, first, there's a memory read for rbp plus var\_Y. We can see that, that has last been set by the instruction at address 7. But we're also dependent, on the value of the rbp register.

That comes from the instruction at address 1. You may be asking, what about the value of var\_Y? Well, as you know from using IDA or Ghidra, the value of var\_Y, var\_X and the other local variables, are just placeholders for constant offsets. Your tool, IDA or Ghidra, is inserting these labels to make your life easier and more human readable.

In reality, in the code, they're just the hard coded value minus 14. Therefore, you could say, the data dependence of the instruction at address ten, is the instruction at address 7 and the instruction at address 1. Let's try another, how about this jump instruction? Be careful of implicit data flows.

Remember, this is a conditional jump instruction. Therefore, it's data dependent on the value in the flags register. If you're coding your own data dependents algorithm, like you will in the lab, you'll need to make a choice if you wanna mark data dependents, for the entire flags register. Or for individual flags in the register.

Of course, individual flags would be more accurate, but it's going to require much more complex coding on your part. One last example, how about the push instruction? What is it data dependent on? Again, this is a tricky case, because it's dependent on values from outside of this function.

Therefore, you can mark all of its data dependence on the start node. And then later on, go back and patch function data dependencies together, if you need to. Although, this is very rarely done, because data dependence within a function tends to be good enough for most cases. And finally, the move instruction at address 4, what's the data dependence of the move instruction at address 4?

This one has both, a local data dependence within the function, that is on the value in the rbp register. But also, on start because of the value in the edi register that it is reading. The value in edi is an argument to this function.. Tracking global memory reads, is gonna be very tricky, and we're gonna talk about it in depth in the coming slides.

Consider this call instruction, what is this call instruction data dependent on? In fact, the call instruction here is not data dependent on anything. It's an unconditional jump to the address, RegQueryValueExA. That's a hard coded address, or at least the binary knows where that function is at run time, therefore, there's no direct data dependence on the call instruction.

If you wanted to optimize your code, you could sometimes mark call instructions, as data dependent on the arguments, being given to that function call. This requires a little bit more tricky code, but you'll get a more accurate data dependence graph. What about the test instruction? Again, don't forget about your implicit data flows.

The call to this function, probably redefined rax or eax, if they returned a value from the function as, RegQueryValueExA does. Therefore, if you wanna more complete data dependence graph, you could mark this test instruction, as data dependent on the call instruction, which most recently redefined eax. Another example, what about moving a constant offset into a register?.

Again, the offset is constant here, the label off 429014 is just a way of saying, this is a human readable representation of the offset itself. Therefore, this instruction has no data dependence. Handling global data, can be very different between different static analysis tools. Because, tracking global dependence, especially across aliased memory locations, can be very trick, and often an unsolvable problem.

One of your options is to track global data globally. That is, make a first pass over the binary, to identify global pointers and then track those as you through each function. Another approach, try to track all data globally. Basically, don't have any start or end nodes on your functions and try to tie everything together.

This can easily result in path explosion. And the third option, and probably the most likely option, is to mark global dependencies to any start node for their containing function. And then, try to patch them together later if you need to. For this instruction, you could say, that the data dependence is on the value of the eax register.

That one's easy enough. But, what data is being read out of memory, being pointed to by the eax register, is probably out of scope if that's a global pointer.

## L8. Def-Use Chains

>> Welcome back everyone. Now we're gonna learn a useful trick for computing the data dependence, using Def-Use chains. You can model the data dependence of a program using a structure called a Def-Use or DU chain. DU chains link the definition of a variable to the next use of that variable.

The pro of using DU chains is that they're very fast to collect data dependence, and very easy to program since you can use just a table data structure. One of the cons is that you have to compute them and store every variable's Def-Use. You can't omit unused variables because you don't know at the time that you're computing the DU chain if the variable is not used.

Consider the source code on the side, and the DU chains being modeled along with it. We can say that the first line defines the x variable and the y variable. Line 5 uses the y variable. Line 6 defines the power variable and uses the y variable. By computing the DU chain along this program, you basically pre-compute most of the complexity of data dependence.

Now consider the program along with its control flow graph. If we remember our two checks for computing data dependence, we can use the DU chain to very easily look up the data dependence of any block. So if you want to compute the data dependence of block 16, you can directly look at what variables are used in block 16, that would be the Z variable.

And then consider the paths through the control flow graph and where the Z variable was most recently redefined. Consider this path through the control flow graph. Here we see the most recent redefinition of z is at line 9. Therefore we could say, line 16 is data dependent on line 9.

This is directly obvious from the DU chain if we check it along every block that gets executed in this path. Now consider a different path. If we go through the loop this time, then the most recent redefinition of the z variable is on line 11. Therefore, we can say that line 16 is data dependent on line 9 and on line 11.

Finally, consider this path through the control flow graph. The most recent redefinition of the z variable is on line 15. Therefore, we can say that line 16 is data dependent on lines 9, 11, and 15. Because there's paths through the control flow graph along which the variables being used at line 16 are most recently redefined at lines 9, 11 and 15.

DU chains are really convenient because they work the same way on binaries as we just saw on source code. You can pre-compute the def and the use of each instruction. And then when you step through the control flow graph, you can always check who has the most recent instruction to define any variable that you're using.

This works the same for registers and for memory. Be careful, when computing data dependence on binaries. Daily dependence in terms of source lines is sometimes easy to see just from the source. That's because our brains naturally follow control flow when we see it indented in source code. However, if we compile the simple source code, it gets a lot trickier to follow the data dependence.

Consider the if statement at line 4 is dependent on the argument in line 1. The x+1 at line 7 is dependent on the initialization of x at line 3. And the return statement is dependent on both paths of the if, lines 5 and 7, since either one is possible for redefining x.

When we compile this code, it's very easy to lose track of that data dependence. A lot of times, a new binary analyst might try to linearly parse the Def-Use chain without following the CFG. This would incorrectly lead to the assumption that the add esi, 1 is data dependent on the move esi, 2.

And the mov eax, esi is only data dependent on add esi, 1. Both of those statements are incorrect and come from just a linear parse of the DU chain. Remember, you need to follow the control flow graph, even when computing data dependence on binaries. The Def-Use chain is just a simple way of modeling the definitions and the uses of each instruction.

Let's consider this example. We've now broken that code into a control flow graph. And we can check the data dependence of each instruction by following the control flow graph and checking the def and use in the DU chain. Consider block 4. We could say block 4 is data dependent on block 1 because that's where the esi is most recently redefined.

What's the data dependence of statement 7? Looking at esi, which is the variable that statement 7 uses, we can see that statement 7 is data dependent on both statements 4 and 5. Because that's how the control flow graph could arrive at statement 7.

## L9. Value Tracking

>> Welcome back, everyone. In this lesson, we're gonna learn how to use value tracking to compute the def use of each instruction and keep more accurate track of the stack. As we saw before, def use chains are extremely helpful for computing data dependence. Unfortunately, the stack and other memory aliases cause problems for def use chains.

And it can be very tricky to keep pushes and pops straight when you're just looking at the def use chains of instructions. Consider the sequence of instructions shown here. The pushes and the pops all define and use rsp as you would expect. Unfortunately, figuring out which pop corresponds to which push is very difficult.

And the DU chain alone will not help you with that, even if you're following the CFG, as you should. The solution, create a shadow stack that models the stack and tracks who pushed what values. I'll show you how this works. If we consider the final instruction in the sequence, and we wanna know the data dependence for this instruction, we look at the use list in the DU chain.

We see that there's a register read to the rsp register, so we look for the most recent definition of rsp. That's the previous pop instruction, which is correct. Then we look for the memory read, who's the last person to define the value at the memory address pointed to by rsp.

Even following the CFG, in this case, it's still just straight line code, you get the wrong instruction. Because it's not in fact the most recent definition of the memory at rsp. In fact, it's the very first push in this entire sequence that defined the value that's being popped in the last instruction of this sequence.

The solution is to use a shadow stack. If we follow the instructions in this listing, we can see that as we walk through the control flow graph, we can build our shadow stack to keep track of pushes and pops. Every time we reach a push instruction, you first push the address of that push instruction to your shadow stack.

Then you can compute the data dependence of that push instruction. Look at what is used by that push instruction and then mark that as the data dependence. In this case, we don't actually know who previously defined rax and rsp. But if you had the rest of this program, you would probably know that right away.

When you get to the next push instruction, again, first, push the address of the push instruction to the shadow stack. Next, you can compute the data dependence of that push instruction. We look at who defined the rsp register and the rbx register previously. In this case, rsp came from .text:00 and the prior rbx value came from somewhere above us in the code.

When you get to a pop instruction, you first compute the data dependence by checking the top of the shadow stack. This gives you who the last person to write to the top of the stack was. You can now see directly who your data depended on for the value that's popped from the stack.

You then pop the most recent value off the top of the shadow stack and continue stepping through the code. Again, when you reach a push instruction, first, push the address of that push instruction to the shadow stack. Then compute the data dependence of the push instruction. Here we see rsp comes from .text:02 and rdx comes from somewhere above us in the code.

When you get to a pop instruction, first check the top of the shadow stack. That will allow you to compute the data independence of the pop instruction, then pop that value off of the shadow stack. Get to another pop instruction, again, check the top of the shadow stack.

By the end of the program, your shadow stack will probably be empty because everything you've observed being pushed will have a corresponding pop. Be careful, computing data dependence is hard in general. On binaries in specific, aliasing is the main kryptonite for the data dependence superhero. An alias of a variable can actually be referred to by multiple pointers.

If you think of this in terms of C code, you use aliases all the time. You give pointers to other variables and redefine that variable via the pointers. Assembly also loves aliasing. For example, you may have a value that's pointed to by one register, and then assigned to it from another register.

This is the case even if the original code did not have any aliasing. And that's because assembly tends to allocate registers to do whatever it needs at the moment. Keeping track of the value of variables statically can get more and more complex the more in-depth you try to implement.

Eventually, your algorithm may break down to a technique called static value tracking. During static value tracking, you will keep essentially a shadow copy of the registers and memory locations that you're interested in tracking. You may have to assume some initial values at the beginning of a function or a program to start your value tracking with.

Then, as you step through the code, you can parse the instructions and update the tracked values as you go. For example, if you get to the push rbp instruction, you'll decrement rsp correspondingly. If you get to the mov rsp, rbp, you'll do the mov correspondingly on your shadow versions of the registers.

Another push will update the register accordingly. And now, if you're interested in tracking these values in memory, you can add additional registers or memory to your shadow copies. If we perform the subtract, you'll subtract 16 correspondingly from your shadow copy. Now, when you get to the compare instruction that we cared about, you can directly see where rbp is pointing to, because you've got it stored in your shadow version.

Hopefully, you've actually tracked this register that you care about ahead of time. If not, you may need to reimplement your solution to track that variable and then rerun your static analysis. Finally, when you get to the end of your program, you can check what the actual value of the registers you're interested in are.

In this case, if you remember from the previous slide, we could not figure out what value was being moved into rax. Using static value tracking, you can keep track of these. I'll caution you, though, don't write a static value tracking tool by yourself from scratch. There are many good implementations of this, especially in symbolic analysis tools that we'll talk about at the end of this class.

## L10. Program Dependence Graphs

>> Hello everyone and welcome back. In this final lesson, we're going to talk about program dependence graphs. Program dependence graph is the second most widely used program representation. Because it combines the information of both data dependence and control dependence. If you're curious, I'd highly recommend you go read the following research papers, which although they date back a while, can show how powerful program dependence graphs are to program analysis.

Consider the code we've been seeing throughout this week. The program dependence graph is a graph of nodes and, N, and edges of the edge set Ed, and a different edge set Ec. N is a finite set of nodes which represent each statement in the program. And possibly super-nodes that make up the basic blocks.

The edge set Ed is a set of edges (i, j) That represent that a node nj is data dependent on a node ni. Recall the two checks we had for determining the data dependence of a node. Similarly, the edge set is Ec is a set of edges (i,j) that represent that a super node nj Is control-dependent on another super-node, ni.

Recall the two checks for control dependence of basic blocks. The program dependence graph is an excellent abstraction for keeping track of all the dependencies which influence the execution of every statement. Just keep in mind control dependence affects basic blocks, while data dependence affects individual statements within the basic blocks.

Call graphs are another abstraction that you will often see when performing program analysis. In the simplest case, nodes in a call graph represent functions. And an edge represents that there's a function call from one function into another. Consider the source code shown on the side of your slide.

And the call graph that can be generated from that source code below. You often use a call graph to stitch together control flow graphs or programme dependence graphs that are computed within each function to perform higher level intraprocedural analysis. Super control flow graphs are an inter-procedural control flow graph that gets stitched together at call instructions.

You add an edge for every call, and a return edge for every return instruction. These are rarely used, however, during static analysis due to path explosion. Recall if we have trouble enumerating all the paths within a single function, it's going to be nearly impossible to enumerate all the paths within an entire program.

They are sometimes useful during dynamic analysis due to the ambiguity of function calls. For example, a jump to very far away code, or fetching a return address and jumping to it. Interprocedural analysis is a very deep topic, especially when applied to static program analysis. There's an excellent lesson on interprocedural analysis taught at Harvard, which I'd recommend you take a look at If you're curious.

Interprocedural analysis can be very precise when you do it statically. Because it allows you to reason about an entire program instead of just locally to a single function. But path explosion makes global reasoning nearly impossible when performing static analysis. For this reason, we only selectively cover interprocedural analysis as it applies to dynamic analysis.

In general, you don't see much interprocedural analysis being done statically. As we'll see, dynamic analysis will actually avoid many of the problems we faced when doing static analysis. But dynamic analysis introduces its own corner cases. Because you're only visibility is that direct execution path that you're observing dynamically.

Basically, everything is global until you realize that it's actually local just to that path. And with that, we can wrap up this lesson. We've learned how to describe software in terms of abstractions and representations. We've talked about basic blocks and how to break large programs down into bite sized pieces.

How to reconstruct control flow graphs, and compute the dependencies observed over those control flow graphs. We've talked a little bit more about interprocedural analysis, and how to bring it all together to answer complex questions about programs.