# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 9 TRANSCRIPTS

**L1 Tracing: Source Level Instrumentation**

>> Hello everyone and welcome back to advanced topics in malware analysis. In this lesson, we're gonna be talking about execution tracing. Our objectives for this lesson are to employ execution tracing to perform dynamic analysis. We're gonna apply static and dynamic binary instrumentation, and we're gonna learn how to create Pintools utilizing the PIN framework.

We're gonna discuss Valgrind, QEMU and other methodologies for performing binary instrumentation. Execution tracing is absolutely the Swiss army knife of dynamic analysis. Tracing a process faithfully records detailed information about that processes execution in order to allow for lossless reconstruction of something that you are interested about in that program's execution.

This has a few benefits. First, that it is lossless, you simply record everything that you need while the program is executing, and then process that trace offline. It's also simple to implement, because you're not doing any processing online, you just simply log, or trace information, and then process it after the fact.

Thus, it's also very low overhead. There is a problem with this approach though. It does require heavy after-the-fact analysis on that execution trace to reconstruct something that you are interested in. For example, you might want to perform control flow tracing. This would log the sequence of all executed statements, so that you could offline reconstruct the observed control flow from the execution of that program.

Similarly, dependence tracing, you could log the sequence of all exercise dependencies, basically printing out the def use chain of every instruction as it executes. This would allow you to trace what the dependence of the data was in that given execution of the program. You could do value tracing, where you print out the values produced by each instruction.

Or memory access tracing to log the sequence of memory references fetched during an execution. These are just some examples, but you can imagine how lossless tracing and then offline processing can be an extremely versatile and powerful tool in dynamic analysis. In fact, you've already performed dynamic tracing before, if you've ever tried to debug some of your old code using printf statements.

Inserting printf into code and then running it to observe its debug output is essentially dynamic tracing. This may seem silly, but consider the following output. In the loop in the loop, true branch in the loop true branch in the loop true branch. Based on only this silly output, you can answer complex questions about this program's execution.

For example, how many elements were in this list? There were four elements in the list. How many of those elements were negative? Only one element in the list was negative. You can tell this just by considering the output from the dynamic tracing. This is called source level instrumentation, and would typically be performed by a compiler plugin in order to automate the process, such as LLVM or GCC plugin.

The compiler would read the source file and parse it into an abstract syntax tree. And then your plugin could annotate that abstract syntax tree with the instrumentation that you wanted to add. The instrumented abstract syntax tree could then be translated into a new source file or compiled directly into a new executable.

Now, when you execute this instrumented program, a trace will be produced and you can see what happened during the execution. Let's look at an example. Imagine you wanted to automatically instrument this for loop, so that it would log every time the for loop executes. To do this, the compiler would parse this source code into an abstract syntax tree representing the for loop.

Your plugin could then add additional instrumentation to the for loop's AST. Now, when the compiler compile this AST into a binary, your added code would execute and you would see the instrumentation in the log. There are some limitations of this type of source level instrumentation. Of course, it requires source code.

This is particularly a problem in this class, because worms and viruses will never provide you with their source code to be easily instrumented. It's also impossible to do on closed source software. And after all, binary analysis is why we're in this class. It's also hard to handle external libraries, because you would have to track down the source code of all those libraries and instrument them as well.

It's also very difficult to handle multi-language programs. Because source code level instrumentation is of course language dependent. You would have to port your plugin to handle Java, C++ or any other programming languages that the program you're trying to instrument is written in. Therefore, we're going to focus the rest of this lesson on binary instrumentation.

**L2 Tracing: Binary Instrumentation**

>> Hello everyone, and welcome back to advanced topics in Malware Analysis. In this lesson, we're gonna be talking about binary instrumentation. And how you can use it to perform dynamic tracing. Tracing via binary instrumentation, is probably the most important binary analysis capability to date. It comes in two flavors.

First, static binary instrumentation, and later we'll talk about dynamic binary instrumentation. Some of the features of this, no source code required. You simply instrument the binary itself, without worrying about what the original source code was. It handles libraries directly, because you can either instrument them when they're running, or statically on the disk.

It also doesn't care what languages the program was written in. Because any binary can be instrumented as a binary. In the worst case, you may have to instrument the JVM, or another interpreter like that. If you want to monitor a Java program, or interpreted language during its execution Static binary instrumentation, is the process of inserting instrumentation into a binary executable file.

Instrumentation will run, the next time that binary gets executed. This is often called binary rewriting. The idea is, given a binary executable, parse its instructions into an intermediate representation. This could be something as simple as the instruction models, that you used in Labs 3 and 4. Or something more advanced that represents the control flow, that may be generated by those instructions.

You then design tracing instrumentation, for that intermediate representation. For example, if you wanted to control floor tracing. Your instrumentation logic would look like for each instruction in the binary. And a print-out that prints that instructions address. Then a lightweight compiler or a binary rewriter. Would take your instrumentation logic and add it to a new executable file, everywhere that you wanted instrumented.

Let's look at an example. Consider the original source code shown on this slide. When we compile it on a 64 bit Linux machine with a little optimization, this is the executable program that gets generated. Now, if we wanted to perform static binary instrumentation, we would need to write some instrumentation logic.

The first part of our instrumentation logic is, what do we want to do on every instruction? So in this case, we want to trace the control flow. So therefore, we want to print the address of every executed instruction. So we make an analysis routine called trace control flow, that takes as input the current instruction.

It then prints that current instructions address. Then, we need to add this instrumentation function, to every instruction in the binary. So there must be some for-loop, that loops through all the instructions in the binary. And adds our instrumentation function, to that instruction. A magic static binary instrumentation tool if one were to exist.

Would then take the original binary as input, and your instrumentation logic. Combine them together, and add your instrumentation code after each instruction in the binary. Producing a new binary with your instrumentation baked in. Let's look at an example of that. First, the magic static binary instrumentation tool, we'd compile your instrumentation routine into a binary itself.

This includes the function which prints the address of each instruction. Then, after each instruction from the original binary, the tool would add a call to your instrumentation function. Recall that the first instruction of our Hello World program, was compared edi to 2. Now that we've instrumented this instruction, we'll see compare edi to 2, followed by a call to our instrumentation routine.

But be careful with control flows. If you try to place your instrumentation after a jump instruction, you'll never get that instrumentation executed. So, when instrumenting control flow instructions, be sure to add the instrumentation ahead of the jump instruction. To look at some more examples, here's the third, fourth, fifth, and sixth instruction, from that Hello World program that we saw before.

We see the Xor instruction followed by a call to our instrumentation. Then the retn instruction, which is preceded by a call to our instrumentation. Then the push and the move instructions from the original binary, both followed by calls to our instrumentation. So what are the good and the bad of static binary instrumentation?

Well, unfortunately it's almost impossible to do accurately. That's why I've been referring to this magical static binary instrumentation tool. The biggest problem is handling pointers in the binary. Because when you start adding code to the binary file, previously computed pointers will then be corrupted. This will leave data in memory corrupted, and likely lead to a crash when you try to execute that program.

Another problem is the original disassembly, must be perfectly correct. If you miss interpret any data as code or code as data. This will cause you to instrument erroneously, an instruction which does not really exist. This also will lead to a crash, when you try to execute this program.

If you're curious, I highly recommend you read these two papers. Which dealt with very advanced ways to try to perform static binary rewriting, in a secure way. There's also been a number of research projects that have come and built off of these two papers, which I recommend you hunt down and read.

Another downside is, static binary rewriting can significantly increase the size of your executable. Because, you are adding five, six, maybe ten instructions, for every one instruction in the original executable. In embedded systems or other platforms where size may be a constraining factor, static binary rewriting may not be an option.

However the good, static binary instrumentation when done correctly, is extremely fast. Because all of the instrumentation logic is baked into the executable. There's literally no overhead, in getting your instrumentation run when the binary itself is executing. But that leads us to the alternative, to static binary instrumentation. And that is, dynamic binary instrumentation.

Static binary instrumentation, requires an original executable. And then generates an instrumented executable, that can be executed with our analysis routines embedded within. That's where the term static instrumentation comes from, that you are instrumenting the binary before runtime. Conversely, dynamic binary instrumentation takes an original binary executable, and an input for executing that program.

It then starts executing the binary with that input from the entry point, and during execution adds instrumentation on the fly. This is why it's called dynamic binary instrumentation, because the instrumentation occurs during runtime. There are some serious advantages to dynamic instrumentation. Number one, we no longer need to recompile, relink, or rewrite the binary.

And number two, we're discovering code at runtime, because we're following it as it executes. This avoids that problem of needing 100% accurate disassembly of the entire binary file. Dynamic instrumentation can also handle dynamically generated code very naturally. And can attach to a running process, making it a lot more versatile than static analysis.

Dynamic instrumentation can also handle dynamically generated code, and attached to running processes. Making it much more versatile than static instrumentation. Let's look at an example. Dynamic binary instrumentation, on our original Hello World. Again, we compile this Hello World programme on 64 bit Linux with a little bit of optimization.

And now, our instrumentation logic remains relatively the same. We'll still want to trace the control flow, and therefore print the address of every instruction as it gets executed. We also have some logic that says, for every instruction that gets executed. And our instrumentation, after that instruction gets executed.

Notice, we get the option now to tell the dynamic binary instrumentation tool, that we want our routine called after the instruction executes. We could also do before the instruction executes, or more advanced tools even allow you to have conditionals. Like If a register value is five, then call my instrumentation.

Now, a magical dynamic instrumentation tool, will take our instrumentation logic as input. Start executing the program from the entry point. And then for each instruction that gets executed, instrument that instruction as we've told the tool to do. Consider the first compared edi to two instruction. Now, when that instruction executes, it will be compared edi.

Followed by our instrumentation logic, that we told the dynamic binary instrumentation tool to insert. There are a few flavours of dynamic binary instrumentation. We're gonna look at platform specific solutions such as Pin. That instrument targets that execute exactly as they are on the host CPU. We'll also talk about a number of emulation based instrumentation systems, such as QEMU and Valgrind.

That instrument, target applications using an intermediate language. There are others and some hybrids of these categories, that we can touch on later in this class.

**L3 PIN**

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson we're gonna be talking about Pin and using it to perform execution tracing. Pin is the undisputed king of x86 dynamic instrumentation. And that is largely because Pin is maintained by Intel themselves. Therefore, it very well supports any popular Intel architecture.

And you can instrument executables on a variety of operating systems, even those that you might not expect to run on Intel architecture. You as an analyst write a Pintool which instruments an application by telling the Pin framework how you want that instrumentation done. Pintools are very powerful because they can insert arbitrary code at arbitrary locations in an executable while it's running.

The instrumentation is added dynamically as the program is executing. And it can even be changed on the fly while the program is still running. Pin handles threads and asynchronous signals automatically within the framework. And this allows Pin to attach to already running processes and start instrumenting them from that point forward.

But let's take a look at how Pin does all of this. You write a Pintool that uses Pin's instrumentation APIs to tell Pin where and how to perform analysis. Internally to Pin, there's a virtual machine and a just in time compiler that fetch instructions from an applications binary.

Combine those instructions with your Pintool's analysis, and then stuffs the whole thing into a code cache. The code then executes inside the code cache at native CPU speeds. What's best is that Pin is freely available for anyone to use. You can download Pin and use the command like this on the screen to launch an instrument an application.

You can even attach Pin to a running process by giving it the -pid flag. Be careful though like Ghidra, Pintools are specific to 32 or 64 bit platforms. You can, of course, share code between the two, but you have to compile them separately and keep an eye out for any incompatibilities.

Pin provides a robust API for your Pintools to interact with. There are basic API's that allow for architecture independent inspection of the execution. Providing common functionalities, like monitoring control flow changes or memory accesses. There's also architecture-specific API's where you can query information about specific opcodes and operands. There's also state change API's where you can get a call back whenever a process is created or ends or an interrupt gets received.

Then there's call-based API's, that are instruction routines and analysis routines that we'll talk about next. An instrumentation routine defines where instrumentation is inserted during the execution. For example, insert a call to my analysis before every instruction. This instrumentation routine will get called the first time any instruction gets executed.

That way you can add your instrumentation to that instruction, and it will be there in the cache for the rest of the time you run that program. As I said Pin is powerful and can actually remove instrumentation from the cache after it's already been inserted. The second type of core routine is an analysis routine.

This is where you define what you want your instrumentation to do once it's been activated. So for example, your analysis routine may print the instructions address whenever it executes or increment a counter. Your analysis routine gets called every time the instruction is executed or based on however your instrumentation routine told Pin to call your analysis routines.

Let's look at an example from the Pin source code. This is the full source code of the itrace Pintool. We can see right at the top is the analysis routine. This is gonna print the instruction pointer of every instruction that gets executed. Following that is the instrumentation routine.

This gets called the first time you execute any instruction. And in this case Pin is telling you what instruction is about to execute for the first time with its INS argument that's being given there. Your Pintool is then telling Pin insert a call before, that's that IPOINT BEFORE argument, insert a call before this instruction gets executed and call my printip function.

And by the way, the first argument to my printip function is the instruction pointer. That's that IARG\_INST\_PTR argument. You can see that the instrumentation routine will tell Pin when to invoke your analysis routine, and the analysis routine does the analysis. There's also a state change analysis routine here, the Fini function that will be called when the process finishes executing.

Finally, the main function of this Pintool opens up the trace file that's gonna get the log of all the instruction pointers logged to it. Initializes the Pin framework, adds your instrumentation function to the Pin framework. Adds the Fini function to the instrumentation framework and then starts the program.

And this is why Pin has reigned as the king. Versatility, Pin can do so many things with this very flexible framework. Look at some of the sample tools in the Pin distribution, in the source tools directory that you download from the Pin website. They do all kinds of things, such as cache simulators, branch predictors, address tracing, syscall tracing, and so on.

Some tools were even developed inside of Intel and have been released such as the opcodemix Pintool that analyzes code generated by different compilers to ensure that it runs optimally on Intel architectures. Commercial companies also write their own Pintools to vet and test their binary programs before shipping them out.

Universities also use Pin for teaching and research like this class. Further, if you get a PhD doing binary analysis like I did, you will very likely spend a lot of time using Pin, like I did. You can go check out my GitHub for some of the Pintools that I've written and some of the powerful things that you can do with the Pin framework.

**L4 Valgrind**

>> Hello everyone and welcome back to advanced topics in malware analysis. In this lesson, we're gonna talk about using Valgrind to perform execution tracing. Valgrind is an extremely powerful dynamic binary instrumentation framework. That although not as performant as Pin, actually works on many different platforms, x86 powerPC, ARM, just to name a few.

When Valgrind was first developed, it won a number of awards for pioneering the space of Dynamic Binary Instrumentation. Its original author Julian Seward from Cambridge University has made it open source for anyone to use and improve. The main downside of Valgrind which has prevented it from overtaking Pin as the king is that overhead is a big problem.

See Valgrind translates one instruction at a time. And so you will see a five to ten times slow down even without any instrumentation added. Since Pin executes everything on the native CPU, it incurs very little slow down with no instrumentation. If you're curious about the in depths of how Valgrind is implemented, you should really read the original publication Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation that was published in 2007.

Valgrind is composed of a dispatcher, a trampoline and a number of instrumentation tools. The dispatcher will begin fetching code from the binary executable based on whatever program counter is currently executing. Note program counter is, of course, just another term for the instruction pointer. If this is the first time the dispatcher has seen that program counter, it will use a basic block decoder to fetch an entire basic block of the binary's code, decode it into an intermediate representation and then that intermediate representation goes through a number of different instrumentation tools.

The instrumentation tools will generate an entirely new basic block with the instrumentation added to the new basic block. That basic block is then compiled and added to a trampoline, which is a type of code cache where the new basic block will then execute. At that point, that new program counter will go back to the dispatcher, who will check if they've already seen that basic block.

If it's an earlier basic block, the earlier basic block can be executed directly from the trampoline. And if any input is needed, that basic block can fetch that input and pass the state to any runtime analysis routines that need to execute. Again, generating a new program counter, which will go back to the dispatcher.

Now let's look at a concrete example of instrumenting Valgrind to do basic block tracing. So essentially a control flow trace where we only log the basic blocks. The first program counter here is gonna be one because that's the first instruction in the program. The dispatcher knows that it has not seen basic block one before.

So it goes to the basic block decoder, which fetches and decodes that basic block and then passes it through the instrumentation. The instrumentation adds a print to log that basic blocks address. The instrumented and recompile basic block gets put into the trampoline and then executed. At that point it will generate output in the tools log so that we can watch the instrumentation tool produce output.

This will now generate the next program counter. So, for this example, imagine we go back around and through the while loop one more time, that will generate the new program counter one, which the dispatcher knows is already in the trampoline. So execution will return to the trampoline, that basic block with the instrumentation will execute one more time producing another one in our instrumentation tools output.

This will generate another program counter, this time five, assuming that we broke out of this while loop. The dispatcher knows that it has not seen basic block starting an address five yet. And so that goes to the basic block decoder. The basic block decoder fetches and decodes and then passes that new basic block through the instrumentation routines.

They add the instrumentation to print that basic blocks number that gets compiled and put into the trampoline. Now block five is gonna execute with our instrumentation. So we will see five output in the tools output. And assuming that's the end of the program, we can now process our output trace and compute what the original value of i would have been to generate this execution.

What would the original value of i been here? It would have been 0.

**L5 QEMU**

>> Hello everyone, and welcome back to advanced topics in malware analysis. In this lesson, we're gonna be learning about using QEMU as a full system execution tracing engine. QEMU is a generic open source machine emulator and hypervisor. As a machine emulator, QEMU you can run operating systems and programs that were compiled for one architecture, for example ARM on a different architecture, for example, your x86 PC.

This is actually how the Android emulator is developed, that allows you to test Android applications on any regular x86 machine. By using dynamic translation, QEMU actually achieve very good performance as being a machine emulator. And we'll talk about how QEMU does this later in this lesson. As a hypervisor, QEMU achieves near native performance by executing the guest code directly on the host CPU.

This functionality has been built into major commercial hypervisors that leverage QEMU for this ability. Of course, as a hypervisor, you're restricting your guest code to being the same architecture as your host CPU. In either case, you can add your analysis routines directly to QEMU'S code base. This allows you to add instrumentation to the entire guest operating system as it executes.

Let's take a look at how QEMU does this. QEMU fetches instructions from a target binary for some architecture. Then decodes those instructions into an intermediate representation known as micro-operations. QEMU comes with a built-in set micro-operations that basically describe how the instructions on the guest architecture behave. These micro-operations are then fed into a tiny code generator as QEMU calls it.

To generate host code that can execute on the host CPU, implementing each one of those micro-operations. That code is put into a code cache. And then code in the code cache is executed on the host CPU. Let's take a look at how QEMU achieves this code translation. QEMU uses as an intermediate representation called tiny code generator operations.

And then it has a set of front ends that are responsible for parsing and decoding instructions of the guest architecture into these tiny code generator operations. If you look at the QEMU source code, there's a directory called target. And this has all of the different guest architectures that QEMU can execute.

This is the What QEMU executes part of QEMU, and it includes architectures such as ARM, MIPS, x86, and so on. Backends and QEMU are implemented inside of the tiny code generator. A QEMU you backend is responsible for taking the tiny code generator operations, and generating host instructions that implement each one of those tiny code generator operations.

This is the Where QEMU executes part of QEMU. And you can see that QEMU supports executing on different architectures such as ARM, x86, MIPS, and so on. Now let's look at an example of QEMU doing this code translation for an x86 Guest to an x86 Host. Consider the x86 Guest code seen here.

The QEMU frontend is going to parse and decode that guests code into tiny code generator operations. Notice that the tiny code generator operations are machine independent intermediate representations of what each of those instructions does. This is where your instrumentation would add additional code in order to perform whatever instrumentation you are interested in.

This would allow your instrumentation to work in an architecture agnostic way. Because you will be operating on only the QEMU tiny code generator operations. Finally, the QEMU backend is responsible for translating the tiny code generator operations into host code that can execute on the host CPU. However, since each of those tiny code generator operations does not match one-to-one with the host CPU instructions.

It will often take multiple host instructions to implement the functionality of a single tiny code generator operation. In this way, you see a large blow up in the number of instructions being executed on the host CPU. For example, shown on the slide is only the host code instructions that implement the ret instruction from the guest.

So you can see that there's a large blow up in the amount of code that actually executes on the host. Because of this large blob in the code, QEMU has to implement a number of tricks to keep things running fast. Like we saw in Valgrind, one of the best tricks to make this happen is a technique called block chaining.

And that is avoiding returning to the dispatcher as much as possible. Because of this blow up in code size, QEMU has to implement a number of tricks to keep things running quickly. Like we saw in Valgrind, returning to the dispatcher from the code cache can be very slow.

Because the dispatcher will need to look up what's the next basic block, if we've already decoded it, or if we need to fetch it from the binary. QEMU solution is to jump directly between basic blocks that are already in the code cache. This avoids returning to the dispatcher as much as possible.

QEMU hatches the jumps at the end of each basic block, to jump to the basic block that would naturally follow it. This allows execution to remain within the code cache as long as possible, until a new basic block that has not been seen before is encountered.

**L6 Offline Trace Reconstruction**

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna talk about different ways of performing offline trace reconstruction. The key word here is lossless. Dynamic analysis is assumed to be lossless, that means no processing of the output during runtime. All analysis and processing takes place offline against the trace that you've created.

A trace must allow offline analysis tools to faithfully recreate the analysis target. So for example, if your analysis target is control flow, you would have generated a control flow trace consisting of a sequence of all the executed statements or instructions. If you're interested in data dependence, you would perform a dependence trace or a sequence of all the exercise dependencies or the def use for each instruction that was executed.

Similarly, value tracing or memory access tracing or any other online tracing you can perform that allows an offline analysis tool to recreate your analysis target. Therefore, tracing is the most fine grained of all possible dynamic analyses, because you must record everything that is pertinent. However, this fine grain tracing makes this very expensive.

Let's look at an example. Consider the sum up function that we have here and the control flow graph off to the side. Let's say you record a trace where n equals 6, and you're trying to reconstruct the control flow of the program. Recording every instruction that executes would blow up very quickly, because you're looking at a space complexity of four bytes times the entire execution length.

Imagine if we converted this to basic block level tracing. You could still reconstruct the control flow because you know that it must hit the beginning of all these basic blocks. However, you haven't really saved yourself too much from the overall tray size. Are there other ideas? Would function level tracing work where, for example, we trace the entry of every function call and the parameters given to that function.

No, unfortunately, this would not be sufficient to reconstruct the control flow. And that's because inside of each function, a function's behavior can be affected by global variables. It would also be impossible to distinguish between nested function calls or just calls to the same function that happened in sequence.

How about predicate tracing, where we reduce a fine grained trace to only the values that were assigned during each predicate. So, for example, an instruction trace of 1, 2, 3, 6 could be shortened to only false, because that is the only predicate encountered at line three, and it returns false.

Let's look at another example. A fine grain trace going from instruction 1, to 2, 3, 4 5, 3, 6. This could be reduced to a predicate trace of just true, false. It seems like we've gained quite a bit, but can this be used to reconstruct the control flow accurately offline?

It turns out the answer is yes, because we can faithfully reconstruct the fine grain trace given just the predicate trace. However, the trace is no longer randomly accessible. You must start from the beginning of the control flow graph and step through each instruction, reassigning the predicate outcomes based on your predicate trace.

However, this costly offline analysis makes your online tracing a lot more simplistic. And that brings us to the end of this lesson. In summary, we've discussed how to employ tracing to perform dynamic analysis. We've talked about static and dynamic binary instrumentation, and how to use the PIN framework as well as Valgrind, QEMU and other frameworks in order to perform dynamic binary instrumentation.