PROFESSOR: All right, let's get started. So today we're going to talk about the system called Native Client, which is actually a real world system used by Google. One example, they actually use it in the Chrome browser to allow web applications to run arbitrary native code. It's actually a pretty cool system, and it's also an illustration.

Another isolation or sort of sandboxing or privilege separation technique, that's called software fault isolation, doesn't rely on operating systems to sandbox a process or virtual machines. But instead, it has a very different approach to looking at the particular instructions in a binary, to figure out whether it's going to be safe to run or not.

So before we I guess start looking at the technical details of the system, let's figure out, why do these guys actually want to run native code? So the particular context they are interested in applying their solution to is a web browser, where you can already run JavaScript code and maybe Flash, and maybe a couple of other things. Why are these guys so excited about running raw x86? It seems like a step backwards.

AUDIENCE: Really fast computation.

PROFESSOR: Yeah, that's one huge advantage of native code. Even though it might be unsafe from some perspectives, it's really high performance. And whatever you can do in JavaScript, presumably you could just write the same thing and assemble it, and it'll go at least as fast-- probably much faster. Any other reasons? Yeah?

AUDIENCE: Run existing code?

PROFESSOR: Yeah. So another big thing is maybe not everything is written in JavaScript. So if you have an existing application-- I guess 'legacy' in industry terminology-- if you have some existing code that you really want to run in the web, then this seems like a great solution. Because you could just take an existing library, like some complicated graphics processing engine that's both performance sensitive and lots of complicated stuff you don't want to re-implement, then this seems like a good solution. Anything else-- if you're just like programming a new web app,

should you use Native Client if you don't care about legacy or performance so much? Any other reasons? I guess another-- yeah?

AUDIENCE: You don't have to use JavaScript.

PROFESSOR: Yeah, that's an awesome reason, right? If you don't like JavaScript, then you don't have to use it, right? You can actually use, well, C, if you are so inclined. You could run Python code, you could write Haskell, whatever you think is useful. You could actually support other languages all of a sudden.

So this is a reasonably compelling list of-- motivation for them to run native code in the browser, and it turns out to be reasonably tricky to get right. And we'll look at the technical details, I guess, of how it works in a second. But just to show you guys what this paper is talking about, I just want to show a very simple tutorial demo almost that I got out of their Native Client website.

It's fairly simple as it turns out to just take a C++ or a C program and run in the browser. So just to show you what this looks like, here's basically a demo I mostly sort of took from one of their examples. So you can look at a web page like this index HTML file. And inside of it, you have a bunch of JavaScript code.

And the reason this JavaScript code exists is to sort of interact with the Native Client piece. So the way you can sort of think of this running in the browser is that you have the browser-- well, we'll talk much more about web security later, but roughly you have some sort of a page, web page that contains some JavaScript code. And this runs with the pages privileges. And this can do various things to the web page itself, maybe talk to the network in some circumstances.

But what Native Client allows you to do is to have this Native Client module running sort of to the side in the browser as well. And the JavaScript code can actually interact with the Native Client module and get responses back.

And what you see here in this web page is the little bit of JavaScript code that's necessary in Native Client to interract with the particular NaCl module that we're going to be running. And you can send messages to this module. The way you do it is you take this module object in JavaScript, and you call it postMessage. And you could actually supply a message to send to the Native Client module. And when the Native Client module responds, it'll run this handle message function in JavaScript. And in this particular case, it just pops up and alerts dialog

box in my browser.

So it's a fairly simple interface from the web page side, from the JavaScript side. And the only thing you additional you have to do, is you actually have to declare this Native Client module this way. So you just say embed module with a particular ID. And the sort of most interesting part is this source equals Hello. some NMF barch attribute. And this one just says, well here's the roughly executable file that you need to load and start running in the native side of things.

And this native code actually looks like any other C++ code you might write roughly. So here's the program. The interesting part is roughly this handle message function. So this is a C++ class, and whenever the JavaScript code sends some message to the native code, it'll actually run this function. And it'll check if the message that's being sent is hello. And if so, construct a reply string of some sort and send it back. It's fairly simple stuff. But just to be concrete, let's try to run it and see what happens.

So we can actually build it, and run a little web server that is going serve up this page and Native Client module. So here I can go to this URL and here we go. Right, it's actually loaded in the module. The module seems to have gotten our hello message from JavaScript. It replied back with the string back to JavaScript. And the JavaScript code popped up a dialog box containing that response. So it actually does kind of work.

We can try to see if we could crash Native Client-- hopefully not, but we can take this code and we have this buffer. We could write a bunch of As to it-- I don't know, quite a lot-- and see what happens. So hopefully this shouldn't crash my browser, because Native Client is trying to provide isolation. But let's see what happens.

So we can rebuild it, rerun the web server. And here if you run it, nothing happens. We don't get the message back anymore, so clearly the message didn't get sent back from the Native Client module, because I don't see any popup. We can look at the JavaScript console here, and we can see that the Native Client module tells us NaCl module crash.

So somehow it caught this buffer flow scribbling over some memory, or maybe it jumped to some bad address containing all As. But any case, the Native Client module is actually able to contain this without this arbitrary sort of memory corruption in the module affecting the rest of the browser. So this is roughly just a quick demo of what the system is, and how you use it as an end user or web developer.

So let's look now at some more. So that's all in terms of demos I have to show you. So let's look more now at how a Native Client is going to work, or perhaps even why we need this particular design as opposed to some of the alternatives.

So if your goal, I guess is to sandbox native code, there's a number of alternatives you can make. People actually had these problems before, performance existing legacy code and other languages before Native Client came around. And people just solved them in different ways that maybe weren't as satisfying from a security standpoint or usability standpoint as Native Client. But it is doable.

So let's see. So what could you do if you really want to run native code in a browser? So one option that people did was to trust the developer. And maybe a variant of this approach is to ask the user whether they want to run some piece of code in their browser or not.

So does everybody understand roughly what the plan is, right? Like instead of having that whole Native Client compilation strategy, I could have just built a simple C program, served it up on the browser, and maybe the browser asked, do you want to run this site or not? If I click yes, then it accidentally scribbled over the browser's memory and crashes the browser.

So it's possible, right? It certainly solves all these goals, but what's wrong with it? Well, I guess there's the insecurity part, which is unfortunate. One way to potentially get around this-- and some systems did. Like Microsoft had the system called ActiveX, that basically implemented this plan. You could serve binaries to IE, the browser on your machine. And as long as it came with a certificate from particular developer signed by let's say Microsoft or someone else, then it would actually run this code. What do you guys think about this plan, is this useful? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's right, yeah. So right, you really have to put quite a bit of trust into whoever it is that's signing this, that they will only sign binaries that will not to do something bad. But it's kind of vague what this bad thing is. And presumably they're just writing C code and signing it blindly without doing a huge amount of work. In which case, you might well be susceptible to some problems down the line. What if we ask the user? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh yeah, yeah, yeah. Like the user's goal is to run this thing. And even if the user wants to be cautious, it's actually not clear how should the user decide? Suppose I really want to

understand, should I allow this program to run? All it tells me is well, maybe it's created from Google.com or Microsoft.com, and its file name as foo.exe. I don't know, like what's in there? Even if I dissemble the code, it might be very hard to tell whether it's going to do something bad or not.

So it's actually really hard to decide. So one way to think of Native Client is it's a mechanism by which users can actually get some confidence whether they should say yes or not to running this thing. So in practice, I guess like Paul Young, who was giving his guest lecture last week, and he suggested that you should enable this click to play extension in Chrome.

So every extension, including Native Client, you should rush click on this thing before it starts running. So in some ways, that's ask the user. But then the user actually has some sense, that well, if I click it, I'm still hopefully safe in the sense that Native Client will protect me. So the two aren't necessarily exclusive, but you really want some technical guarantee from Native Client that ensures that even if you say yes, there are something meaningful that isn't enforcing your security. So does this make sense? OK.

So the other I guess approach that people have taken, is to use an OS or hardware sandboxing, or isolation. So this is things we looked at in the last couple of lectures. So maybe you would use Unix, isolation mechanisms. Maybe if you had something more sophisticated, if you're running a free [INAUDIBLE] maybe you would use Capsicum. It seems very well suited to sandboxing some piece of code. Because you can give it very few capabilities, and then it seems great. And Linux has a similar mechanism called Seccomp that we briefly touched on in the last lecture, that could allow you to do similar things.

So it seems like there is already a mechanism for writing code in isolation on your machine. Why are these guys opposed to using this existing solution? Then they are like reinventing the wheel for some reason. So what's going on? Yeah? Oh yeah?

AUDIENCE: Maybe they want to minimize the [INAUDIBLE]?

PROFESSOR: Yeah, so in some sense, maybe they don't want to trust the operating system. So maybe they're here, and they're actually worried about OS bugs. It might be that the previous dekernel, the Linux kernel, has quite a lot of C code written that they don't want to audit for correctness, or maybe can't even audit for correctness, even if they wanted to. And in one of these Capsicum or Seccomp based isolation plan, you probably do trust quite a bit of code in

the kernel to be correct, for the sandbox to actually hold and enforce isolation. Yeah?

AUDIENCE: As you get a lot more ways to use browsers and stuff, like you'd have to deal with having some sort of thing you're doing with it on like iOS and Android, and all these other things, accessing--

PROFESSOR: Yeah, so it's actually another interesting consideration is that normally my OSes have bugs. But actually the OSs are incompatible with each other in some ways, meaning that each OS has it's own-- like right here. Well, there's Unix, there's Capsicum, there's Seccomp, but this is just a Unix variances. There's Mac OS seatbelt, there's Windows something else, and the list just keeps going on and on. So as a result, every platform you will have to use a different isolation mechanism. And the thing that actually bothers them is not so much that they'll have to write different code for Mac and Windows and Linux. But more of that, this impacts how you write the thing inside of the sandbox. Because in Native Client, you actually write a piece of code that runs the same way, or it's the same piece of code the runs on Apple or Windows or Linux systems. And if you use these isolation mechanisms, they actually impose different restrictions on the program being sandboxed. So you'll have to write one program that's going to run inside of a Linux sandbox, another program inside of a Windows sandbox, and so on. So this is actually not acceptable to them. They don't want to deal with these kinds of problems.

So are there other considerations? Yeah?

AUDIENCE: Presumably performance as well. Because if you say Capsicum, you need to fork up a set of [INAUDIBLE], or whatever is running inside the sandbox. With here, they can actually run it in the same [INAUDIBLE].

PROFESSOR: That's true, yeah. So potentially the approach they take, the software fault isolation plan is actually highly performant, and could outperform these sandboxes at the OS level. It turns out that in Native Client, they actually use both their sandbox and the OS sandbox, just for extra precaution for safety. So they don't actually win on performance in their implementation, but they could, right. Yeah?

AUDIENCE: There is some like control aspect to it. Because it can control what happens in the browser, but if they sent one out to the client's machine into their OS, they sort of don't necessarily know what might be happening to it?

PROFESSOR: So I guess maybe one way to think of that is that yeah, the OS might have bugs, or my OS might not do as good of a job at sandboxing it. Or maybe the interface is a little different, so you don't know what the OS is going to expose.

AUDIENCE: So it doesn't like prevent the code from doing some bad things. Like there are a lot of cases of bad things that the code can just do, like maybe you want to statically analyze the distance, but they both sit in a loop and then not allow that as a valid program.

PROFESSOR: So you could, right? So their approach is quite powerful in the sense that you could try to look for various kinds of problems in the code like maybe infinite loops, et cetera. It's hard to decide, kind of like the holding problem, whether it's going to have infinite loops or not. But in principle, you might be able to catch some problems.

I think one actually interesting example that I almost didn't realize this sort of existed before reading this paper, is these guys are worried about hardware bugs as well, that not only are they worried that the operating system might have vulnerabilities that the motions code will exploit. But also that the processor itself has some instructions that will hang it, or it'll reboot your machine. And in principle, your hardware shouldn't have such bug, because the operating system relies on the hardware to trap into the kernel if there's anything bad executing user mode, so that the operating system can take care of it.

But experimentally, it turns out that processors are so complicated that they do have bugs, and these guys actually say, well, we actually found some evidence that this happens. If you have some complicated instruction that the CPU wasn't expecting, the CPU will actually halt instead of trapping to the kernel. This seems bad. But I guess it's not a disastrous, if I'm sort of only running reasonable things on my laptop. But it is bad if you visit some web page and your computer hangs.

So they want basically a stronger level of protection for these Native Client modules than what you would get out of sort of an OS level isolation, even from a hardware [INAUDIBLE] bug standpoint. So they're actually pretty cool. They're like really paranoid about security, including hardware problems. All right, so any questions about all these alternatives, how that works, or why these guys are not excited about that? Makes sense? All right.

So I guess let's try to look at now, how Native Client does actually decide to sandbox processes. Let me pull up these boards. So Native Client takes this different approach that's in general called software fault isolation.

And the plan is actually not rely on the operating system or hardware to check things at runtime, but instead to somehow look at the instructions ahead of time, and decide that there are always going to be safe to execute. So actually look at the binary, and you check all the possible instructions to see whether they're going to be safe instructions or unsafe instructions.

And once you've decided that it's all going to be safe, you can just jump in and start executing. Because you know it's all composed of safe things, so it cannot go wrong.

So we'll talk about exactly what this means. But roughly what they're going to do is, they're going to look at pretty much every instruction in the binary code that is served up to the browser. And they're going to decide that particular instructions are going to be safe or unsafe.

What do they do for safe instructions? Well, they're just going to allow them. What's an example of a safe instruction? What are they thinking of that don't need any extra checks or protections, et cetera?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, so like any kind of ALU operation. Basically you have math, presumably moves between registers and so on. So this doesn't actually affect the security of the system, as far as they're concerned. Because all they really care about is things like memory safety, where what code you execute, and so on. So as long as you are just computing on some stuff, and registers, they don't really care. It's not going to affect the rest of the browser.

So what about unsafe operations? I guess these are things that they might care much more about. So this anything that maybe does a memory access, or maybe some privileged instruction, maybe invoking a system call on this machine. Maybe trying to jump out of the sandbox, who knows what? So for these kinds of instructions, they are going to do one of two things.

If the instruction actually is necessary for the application to do it's job, like accessing memory seems important, then they're going to somehow make sure that they can execute this unsafe instruction safely. Or if you're jumping around for example in the program's address space, [INAUDIBLE] what they're going to do is somehow instrument the instruction.

And what instrumenting the instruction means is that you used to have one unsafe instruction

that sometimes does good things, that the application might actually want to do legitimately. And sometimes the instruction can do bad things. So what instrumenting it means is that you add some extra instructions before it, that it'll check or enforce, but only good things are going to happen with that instruction.

So for example, if you are accessing a memory location, which you could do-- it turns out, they don't do it for performance reasons-- but one way to instrument a memory access would be to put some checks, like an if statement before the instructions, saying, well, is this address you're accessing in range of what's allowed for this module? And if so, do that-- otherwise, exit.

So that would instrument an instruction. And that would turn an unsafe instruction into one that's always safe to execute, because there's always that check in front of it. So does that make sense? This is their sort of big idea for running this enforcing safety without support from that operating system at some level.

And I guess for other unsafe instructions, they don't instrument all of them. Some of them they actually prohibit, if they believe that this is an instruction that's not really necessary during normal operation, and if an application tried to run it, we should just kill the application or not allow it to run in the first place.

Makes sense? Those are their sort of big plan for software fault isolation. And once you've checked the application's binary, and it all passes, and everything seems to be properly instrumented, then you can just run the program and off it goes. And by definition almost, it will not do bad things, if we do all these checks and instrumentation [INAUDIBLE].

And I guess there's one more part to the software fault isolation story, which is that once you've make sure that everything is safe, then it can't actually do anything terribly interesting, other than compute stuff in it's own little memory. So it can't access the network, it can't access the disk, it can't access your browser, display, keyboard, almost anything.

So pretty much every software fault isolation story, there's actually also some trusted service runtime. And this trusted runtime is going to not be subject to these checks on it's instructions. So the trusted runtime could actually do all these potentially unsafe things. But this trusted runtime is written by Google. So hopefully they get a [INAUDIBLE]. And it's going to implement all the functions that Google's Native Client team believes are OK throughout for these modules.

So these are going to be things like maybe allocated memory, maybe creating a thread, maybe communicating with the browser as we saw-- so some sort of a message passing system, et cetera. And the way it's going to be exposed to this isolated module is through certain presumably special jumps or operations that the module is going to perform to transition control into this trusted runtime, but hopefully in a very predictable way to-- question?

AUDIENCE: I'm just curious about, you have to design this application, knowing that it's going to be sent to the NaCl module? Or does it like change the code to [INAUDIBLE] making the [INAUDIBLE] sort of stuff bad?

PROFESSOR: So I think if you are building an application, you do have to be sort of aware that it's going to be running inside of Native Client. So some function calls like malloc or pthread_create, I think they just transparently are placed with calls to their trusted runtime. But if you do anything like opening a file by path name, or anything else that sort of you would expect to do on a Unix machine, that you would probably have to replace with something else.

And you'd probably want to structure your thing to interact at least a little bit with your JavaScript or web page in some way. And that you would have to do by explicitly doing some message passing or RPCs into the JavaScript part. So that you would have to probably change.

So there are some-- it's not like you can run an arbitrary Unix program in there, and you'll just sort of get a shell and you can run commands all of a sudden. Probably if you tried hard, maybe you could create such an environment. But by default, you probably have to make it operate inside of this web page, and then [INAUDIBLE]. Does that make sense? All right. OK.

So that's the overall plan software fault isolation. So I guess let's look at actually what safety means in their case for Native Client. So we talked loosely about this notion of safe instructions, unsafe instructions-- what do they actually care about here?

So as far as I can tell for Native Client, safety basically means two things. One is that there's no disallowed instructions that it can execute. And these disallowed instructions are things like maybe system calls or triggering an interop, which is another mechanism on x86 to jump into the kernel and invoke a system call, and probably other privileged instructions that would allow you to escape out of the sandbox.

We'll look a little bit more later at what instructions could actually let you jump out of a sandbox. And in addition to this no disallowed instructions rule, they also want to make sure that all code and data accesses are in bounds for the module.

So what this means, is that they actually dedicate a particular part of the program's address space-- specifically it goes from zero up to 256 megs in the process. And everything that this untrusted module does has to refer to locations within this chunk of memory in the program.

All right, so just to double check. So why do they want to disallow these instructions? So what if they fail to disallow the instructions? Yeah?

AUDIENCE: The module can manipulate the system.

PROFESSOR: Right, so that's fairly straightforward, exactly, yeah. So it can just directly reboot the machine or open your home directory and enumerate all the files, and do all these things. So that seems like a good thing to do. Why do they care about this plan, like isolating the coded data to only access these low addresses? What goes wrong if they fail to do that? Yeah?

AUDIENCE: Then they can sort of accessing and interrupting [INAUDIBLE] module on the computer.

PROFESSOR: Yeah.

AUDIENCE: We don't care if they ruin their own program and it crashes. That's fine, as long as it's not going to crash the key to [INAUDIBLE].

PROFESSOR: Yeah, so in some sense, that's true. But in their case, they actually run the thing in a separate process. So in theory, this would only crash that extra process. I think that I guess what they're really worried about is that this is a necessary condition to ensure this no disallowed instructions in some way, because there's other stuff, like this trusted runtime in your process. So if what you really care about is not corrupting the rest of your computer, then if the untrusted module can arbitrarily jump into the trusted service runtime and do anything that the trusted service runtime can do, then they could sort of violate this property. So in some ways, this is really like a supporting mechanism for enforcing that.

In principle, this could also be used for lighter weight isolation if you could run this Native Client module inside of the browser process itself, and not start an extra process. But it turns out for performance reasons, they really have to tie down the module to this particular range of

memory, or it has to start at zero anyway. So this means that you can only have one really Native Client untrusted module per process. So you probably are going to start a separate process anyway. Makes sense? Any questions? Yeah?

AUDIENCE: Is there actually a reason why it has to start at zero?

PROFESSOR: Yeah, so it turns out it's more efficient in terms of performance to enforce jump targets, if you know that the legitimate address is a contiguous set of addresses starting at zero, because you can then do it with a single AND mask, where all the high bits are one, and only a couple of low bits are zero-- well, if you're willing-- well--

AUDIENCE: I thought the AND mask was to ensure [INAUDIBLE].

PROFESSOR: Right, so the AND mask ensures alignment, you're right. So why do they start at zero? I think it probably helps them to-- well, I guess they rely on the segmentation hardware. So in principle, maybe they could use the segmentation hardware to shift the region up, in terms of linear space. Or maybe it's just with the application, sort of sees this range. And you can actually place it at different offsets in your virtual address space. It could be-- yeah, so maybe you could actually play tricks with the segmentation hardware to run multiple models in a single address space.

AUDIENCE: But is it because they want to catch a null pointer reception?

PROFESSOR: Yeah, so they want to catch all points of reception. But you could sort of do that. Because the null pointer-- I guess we'll talk about segmentation in a second, but the null pointer actually is relative to the segment in which you are accessing. So if you shift the segment around, then you can map an unused zero page at the beginning of everyone's segment. So yeah, it might be that you could do multiple modules.

I think for-- well, because one reason they probably want to do this is that when they port their stuff to 64 bits, they have a slightly different design. This paper doesn't really talk about it. And I think about 64-bit design, the hardware itself got rid of some of the segmentation hardware they were relying on for efficiency reasons, and they have to do a much more software oriented approach, in which case having a little bit doesn't help them. But in this 32-bit version, I think, yeah, maybe that's true. That's not a deep reason why it has to start at zero. Other questions? All right.

So I guess we sort of roughly understand what the plan is, or what we want to enforce in terms

of safety. So how do we do this? So let's look at some of at least naive approach, and see how could we screw it up, I guess, and then we'll try to fix it afterwards.

So I guess the naive plan that sort of looks like what they do, is to just look for disallowed instructions by just scanning the executable from the start going forward. So how do you detect that instructions? Well, you could just take the program code, and you sort of lay it out in a giant string that goes from zero up to maybe all the way to 56 megabytes, depending on how big your code is, and you start looking. OK, well, maybe there's a nop instruction here. Maybe there's some sort of an add instruction there. Maybe there's a not-- I don't know, some sort of a jump, et cetera. And you just keep looking. And if you find a bad instruction, you say, ah, that's a bad module. And then you discard it. And if you don't see any system call [INAUDIBLE] instructions, then you're going to allow this module to run, module or whatever it is we'll do for the in bounds checks.

So is this going to work? So why not? What are they worried about? Why is so complicated?

AUDIENCE: [INAUDIBLE] the instructions?

PROFESSOR: Yeah, so the weird thing is that x86, which is the sort of platform they are targeting has a variable length instructions. So this means that the particular size of an instruction depends on the first few bytes of that instruction. So you have to actually look at the first byte and say, OK, well, the instruction is going to be this much more. And then maybe you sort of have to look at a couple more bytes, and then decide, OK, well, that's exactly how long it's going to be. But you don't know how ahead of time. So some architectures like Spark, ARM, [INAUDIBLE], have more fixed length instructions. Well, ARM, for example-- ARM is like weird, but it has two instructions lengths. Either everything is two bytes or everything is four bytes. But x86, the instructions could be like one byte or 10 byte or five bytes, anything in between, as well.

I forget actually how big. You can get a pretty long instruction. I think you can get like a 15 byte instruction with x86 if you try hard. It's complicated instructions, though. But as a result, the problem that could show up is that maybe you're scanning linearly and everything seems to be fine. But maybe at runtime, you'll actually jump into the middle of some instruction. Maybe it was a multi-byte instruction, if you interpret it starting from the second byte, it looks like a completely different thing.

So that's just one example of sort of playing around with an assembler. And if you have an

instruction like this, 25 CD 80, 00, 00. And if you interpret it as this five byte instruction, meaning that you look at this byte. And oh yeah, this is going to be a five byte instruction. So you have to look five bytes forward. And then this turns out to be a fairly benign instruction that's an and of the EAX register, with some particular constant that happens to be I think 00, 00, 80, CD. anyway.

So this is one of the safe instructions that Native Client should just allow under the first rule in checking these binary instructions. But if it turns out at runtime the CPU decides this is where it has to start executing the code, then this instruction is actually a four byte instruction, and its actually an int instruction that makes the OX80 interrupt, which is one way to make system calls in Linux. So if you miss this fact, then eventually you're going to allow untrusted module to jump into the kernel and make system calls, which is what you wanted to prevent. Make sense? So how can we prevent it? Like one possibility is like maybe we should try to look at every byte offset. Because at least x86 can only start interpreting instruction on a byte boundary instead of a bit boundary. So you like look at every single possible byte off, and you see what instruction starts there. Is this a reasonable plan to report? Yeah?

AUDIENCE: I mean, what if someone actually is doing an and, and they're never going to jump into that. And now you're just allowing their program.

PROFESSOR: Right, so basically it's prone to eventually false positives. Now if you really wanted to, you could probably contort yourself and change the code a little-- and somehow avoid this. If you knew exactly what the checker was looking for, you could potentially change these instructions. Maybe like end it with one thing first, and then end it with another mask later. But just avoid these suspicious byte patterns. But that just seems pretty awkward to do.

Now it is actually possible that the architecture does involve changing the compiler. So in principle, they do have some component that actually has to compile the code correctly. You can't just take of the shelf GCC and compile a code for Native Client. So in principle it's doable. But I think they're just thinking it's too much hassle, and it's not going to be reliable or high performance, et cetera. Make sense? And plus there's a couple of x86 instructions that are prohibited or should be unsafe and prohibited. But they're like one byte long, so that is going to be pretty damaging to look for or filter out.

OK. So if they can't just assemble and sort of straight up and hope for the best, then they need some other plan for doing this disassembly in a reliable fashion. So what is that client or Native

Client to ensure they don't get tripped up by this variable length encoding? So I guess one way to think about it is, well, so how are we going to solve this reliable disassembly?

So in some sense, if we really do scan forward from the left to the right, and we look for all the possible up codes, if that's the way the code executes, then we're in good shape. So even if there are some weird instruction and has some offset, then the CPU isn't actually going to jump there. It actually executes the same way that we're scanning the instruction screen from left to right.

So the problem with getting the disassembly to be reliable really comes from the fact that there's jumps. Because if we execute linearly from left to right, then the CPU will follow the same rules that our checker is following, and see the same instruction screen. So the problem really comes down to what happens if there's a jump somewhere in the application. Could a jump to some position in the code that we didn't observe in our left to right scan?

So this is what they are sort of going another in the reliable disassembly. And the basic plan is to just check where all the jumps go. It's actually fairly straightforward at some level. They have a bunch of rules we'll look at in a second, but roughly the plan is if you see a jump instruction, then it actually has to be-- well, you have to check that the target was seen before.

So you basically do the left to right scan that we sort of described in our naive approach here. But then if you see any kind of a jump instruction, then you see what is the address to which the jump instruction is pointing to, and you make sure that it's an address that you saw in your left to right disassembly.

So if there's a jump instruction for example that goes to that CD byte, then we're going to flag that jump as invalid because we never saw an instruction starting in the CD byte. We saw a different instruction. But if all the instruction, if all the jumps go to the start of instructions we saw, then we're in good shape. Does that make sense?

So the one problem is that you can't check the targets of every jump in the program, because there might be indirect jumps. For example in x86, you could actually have something like jump to the value of that EAX register. This is great for implementing function pointers. The function pointer somewhere in memory, then you hold to function pointer into some register at one time. And then you jump to whatever address was in the [INAUDIBLE] relocation register.

So how do these guys deal with these indirect jumps? So I have no idea if this actually going to

jump to the CD byte or the 25 byte. What do they do? Yeah?

AUDIENCE: The instrument [INAUDIBLE].

PROFESSOR: Yeah. So this is their basically main trick here, is instrumentation. So whenever they see a jump like this, well actually what the compiler is going to generate is a proof that this jump is actually going to do the right thing. And the way they actually do this is-- they don't actually know-- it's actually kind of hard to put in a proof here that it's one of the addresses that you saw during the left right disassembly. So instead what they do is they want to make sure that all the jumps go to multiples of 32 bytes. And the way they do this is actually change all the jump instructions into something that they pseudo instructions. So they're still that jump to the EAX register.

But they prefix it with an AND instruction that is going to clear the low five bits, E 0 with an EAX register. So that AND instruction clears the low five bits, which means that it forces this value to be a multiple of 32, two to the five. And then you jump to it. So if you look at it during verification time, then you can sort of convince yourself that this instruction pair will only jump to a multiple 32 bytes. And then in order to make sure that this is not going to jump to some weird instruction, you're going to enforce an extra rule, which is that during your disassembly, when you're scanning your instructions from left to right, you're going to ensure that every multiple of 32 bytes is a start of a valid instruction.

So in addition to this instrumentation, you're also are going to check that every multiple of 32 is a valid instruction. What I mean by valid instruction here is an instruction that you see when you disassemble from left to right. Yeah?

AUDIENCE: Why 32?

PROFESSOR: Well, yeah, so why did they choose 32? That seems like some magic number that they pulled out of a hat. Should you choose 1,000 or five? Any comment? OK, should we choose five? So why is five bad?

AUDIENCE: The power of two.

PROFESSOR: Yes, OK, so we're on a power of two. OK, so good point. Because otherwise ensuring something is a multiple of 5 is going to require a couple of instructions here, which is going to lead overhead. How about eight? Is eight good enough, yeah?

AUDIENCE: You can have instructions longer than eight.

PROFESSOR: Yeah, OK, so it has to be at least as long as the longest x86 instruction you want to allow. So if there's a ten byte instruction, everything has to be a multiple of eight, well, you're kind of screwed with a ten byte instruction. There's nowhere to put it. So it has to be at least as long. 32 is pretty good, like the biggest I could find was 15 bytes. So it's probably good enough. Yeah?

AUDIENCE: Can [INAUDIBLE] be longer than [INAUDIBLE] code, they'd have to have a alt instruction at the beginning of the [INAUDIBLE] so you can't just jumpt directly into [INAUDIBLE] and they also need a jump instruction that jumps to another [INAUDIBLE].

PROFESSOR: That's right, yeah. So you have to be able to fit for the entry and exit from the process service runtime, you need to be able to fit some nontrivial amount of code in a single 32 byte slot. We'll see how this works in a second. Or even 31 bytes slot, as you are pointing out, for one hold instruction.

Should it be much larger? Should we make it, I don't know, a thousand byte, well, 10 24?

AUDIENCE: [INAUDIBLE] sparse, because you have to have [INAUDIBLE].

PROFESSOR: Yeah, so if you have a lot of function pointers or lots of sort of indirect jumps, then every time you want to create some place where you're going to jump to, you have to pad it out to the next boundary of whatever this value is. So with 32, maybe that's OK. You're like well, worst case, you waste 31 bytes, because you just need to fast forward to the next boundary. But if it's a multiple of 1024, then all of a sudden you're wasting a kilobyte of memory for an indirect jump. And if you have probably short functions or lots of function pointers, then this might actually be a memory overhead all of a sudden, as well.

Does that make sense roughly? So yeah, I think the 32 is not set in stone. Like Native Client has to have 32 byte blocks, but something on that order probably works. 16 is probably a little short. On the other hand, like 64 could work. 128 maybe is getting a little longish. But I don't think you could derive 32 from first principles. Makes sense? All right.

So let's set a plan for reliable disassembly. And as a result, the compiler has to be a little bit careful when it's compiling your C or C++ code into a Native Client binary. It has to basically follow these rules. So whenever it has a jump, it has to add those extra instruction front. And whatever it's creating a function that it's going to jump to, our instruction is going to jump to, as

we're talking about, it has to pad it out. And it can't just pad it out with zeros, because all those have to valid up codes. So it actually has to pad it out with [INAUDIBLE] just to make the validator happy, just to make sure that every possible instruction is a valid one. And luckily on x86, no op is a single byte, or at least there is a no op-- that's a single byte. There's many no ops on x86. So you can always pad things out to a multiple of whatever this constant is. Make sense?

All right. So what does this guarantee to us? I guess let's make sure that we always see what happens in terms of the instructions that will be executed. So this finally gets us this rule. So we can be sure there's never a system call being issued. What about-- So this is for jumps. What about returns? How do they deal with the returns? Can you return for a function in Native Client? What would happen if you ran into a red hot code? Would that be good? Yeah?

AUDIENCE: [INAUDIBLE] you don't want the-- [INAUDIBLE] modifying your overall [INAUDIBLE].

PROFESSOR: Well, it's true that it pops the stack. But the stack that Native Client modules use, it's actually just some data inside of their section. So you don't actually care-- the Native Client contact doesn't care whether those guys screws up their stacks, overflow their stack, or--

AUDIENCE: Wait, but you could put anything on the stack. And when you [INAUDIBLE] you jump through that--

PROFESSOR: That's true. Yeah, so return is almost like an indirect jump from a particular weird memory location that's at the top of the stack. So I guess one thing they could do for return is maybe prefix it with a similar check, where maybe like pop the top thing of the stack. You check whether it's valid, and then you write, or you somehow do an AND to a memory operand, and that's the top of the stack.

It seems a little fragile, partly because of race conditions. Because for example, if you look at the top location of the stack, you check that it's OK, and then you do a write later, another threat in the same module could modify the thing at the top of the stack. And then you'd be referring to that address.

AUDIENCE: Would this not be the case for the jumps as well?

PROFESSOR: Yeah, so what happens with the jump? Could our race conditions somehow invalidate this check? Yeah?

AUDIENCE: The code is not writable.

PROFESSOR: Well the code is not writable, that's true. So you can't modify the AND. But could another thread modify this jump target in between these two instructions? Yeah?

AUDIENCE: It's in a register, so--

PROFESSOR: Yeah, that's the cool thing. There is basically-- if it modifies it in memory or where ever it loaded into EAX from, sure, you do it before you load, in which case this EX will be bad, but then will clear the bad bits. Or it could modify it after, at which point it's already in the EX, so it doesn't matter that it's modifying the memory location from which the EX register was loaded. And threads don't actually share the register sets. So if another thread modifies the EX register, it will not affect this thread's EX register. So this instruction actually is sort of race-proof in some sense. Other threads can't invalidate this instruction sequence. Make sense? All right.

So here's another interesting question. Could we bypass this AND? I can jump around in this address space all I want. And when I'm disassembling this instruction, this seems like a perfectly fine parallel instruction, an AND and a jump. And the check for static jumps is that, well, it just has to point to some target that was seen. So yeah, we saw an AND. That's one instruction that's valid. We saw a jump, that's another valid instruction.

So when I see a direct jump up here, maybe I jump to some address. I don't know, one, two, three, seven. That's actually OK, even though it's not a multiple of 32. Native Client doesn't generally allow direct jumps to arbitrary addresses, as long as that address is a instruction that we saw during disassembly, as we were just talking about. So could I put in this very nicely checked indirect jump, and then later on in the code jump to the second instruction in that sequence? So then I'll load something into EX, jump here, and then directly jump into this unsandboxed address. That will violate the security, right? Does everyone see that? So how is this avoided?

AUDIENCE: Well, the NaCl [INAUDIBLE] and the jump has to signal instruction.

PROFESSOR: Yeah, so this is why they call this thing a single pseudo instruction. And even though at the x86 level, they're actually distinct instruction, as far as the NaCl validator thinks of it, it's actually a single atomic unit. So as far as this check for whether it's an instruction that you saw before, they think, oh, this is the only instruction that I saw before. Jumping in the middle is like

jumping in the middle of this guy. It's the same thing. So they basically enforce slightly different semantics than what x86 enforces in terms of what an instruction is.

And this might actually mean that you can represent certain instruction sequences in NaCl. So if you actually had legitimate code that looked like this before, this is going to be turned into a single out code in NaCl. But hopefully that's not a problem. Yeah?

AUDIENCE: Presumably they can out that in the trusted code base in the start of the text segment. Because that way, you will always replace those two. Instead of putting those in the binary that you produce anyway, you just jump straight to that jump target that's in the privileged section.

PROFESSOR: In the trusted runtime, you mean or something?

AUDIENCE: No, so in the first 64 K--

PROFESSOR: Yeah, they had these trampolines and yeah--

AUDIENCE: So they can make that one of the trampolines. Except that it doesn't jump out. It jumps back in.

PROFESSOR: Right.

AUDIENCE: So that way, it wouldn't need to be a super instruction. It would just be a single x86 jump into there, which would then jump out again.

PROFESSOR: That's true, yeah. That's another sort of clever solution that you could do, is instead for the deliverance of [INAUDIBLE] so suppose that you didn't want to do this pseudo instruction trick, and you just wanted to have a single instruction replacing the jump EAX. Well, what you can basically come up with is a library of all possible indirect jumps you can ever do. So like well, there is jump EAX, there's jump EBX, and so on. And you would construct a library of these guys for each of them, you would construct the safe check that you'd want to replace them with. So for this one, you'd put an AND in front of it, and so on. And for this one, you'll also put an AND in front of it, and so on.

And then in the compiled binary, every time you want to jump to EAX, what you actually do, is actually jump to a fixed address that corresponds to this helpful piece of code stored somewhere in that low 64 k of the program. And then this guy will do the AND and jump again. The reason that they probably don't do this is performance for this sort of interesting reason.

So that Intel processor, actually most processors these days, have to predict where the branch

goes to keep the pipeline of the processor full at all times. It's one of the really complicated branch predictor that not only decides whether you're going to branch or not, like if statements. But also actually guesses where you're going to jump. So if you see an indirect jump, it actually is going to guess what the address is going to be. And this is actually a guess that's stored per cache line stored in the jump instruction.

So if you have a single place where all the jump EAXs come from, then the CPU is always going to be very confused. Because these jump EAXs seem to be going all over the place, everywhere. Where as if we are really tied to a particular jump instruction, it would be a better prediction. That's just a performance trick that they play. Make sense? All right.

So I guess we roughly understand how it's going to disassemble all of these instructions how it's going to prevent these instructions. So now let's look at the set of rules they are going to enforce through this table in the paper, table one. And it has all these different rules for the validator to follow, or that the binaries have to follow and the validator checks. So I'll go through these rules and just double check that we understand why all these rules are there. So we have these things, C1, all the way to C7. So C1 basically says that once you load the binary memory, then the binary is actually not writable at the page table level. So they set the permission bits for the binary can be non-writable. So why? It should be fairly obvious hopefully.

The reason is that their whole security plan relies on them checking that your binary is correct. So once they've checked it, they want to make sure you can't modify the binary and have it do something illegal that they prohibited. So this is I think reasonably clear then.

So C2 is their plan that basically has to linked at zero at start at 64 K. So this requirement, I think doesn't actually have to do with security so much. I think it's just for convenience, because they just want to have a standard layout for the program. In some sense, it's for simplicity so that they don't have to deal with complicated relocation records, which means that there's fewer things that the validator or a loader might screw up. But basically this is sort of a standardization plan for them, in terms of how to load their executable. Make sense?

All right. So the third requirement is that I guess the indirect jumps use the two instruction. So the two instruction is this thing above. So that, we just talked about, why it needs to ensure this, so that you don't jump to the middle of some instruction, or somehow invoked [INAUDIBLE], et cetera. Make sense? All right.

So what's going on with C4? Basically you have to pad out two page boundary with a halt instruction. Why do they want to pad their binary out with halts? Any ideas? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, I think-- so I have to admit, I don't have a crisp answer for why C4 exists. But I think it's roughly what you are saying, which is that if you-- the code naturally stops at some point. There's some end to it. And the question is, what happens when you just keep executing it, and you get to the end? Then the processor might just keep executing past the end, and execute some extra instructions. Or maybe it wraps around in some weird way. So they just want to make sure that there's no ambiguity about what happens if you keep running and you don't jump, and you just run off the end of the instruction screen. So let's just make sure that the only answer there is if you keep executing, you'll halt, and you'll trap into the runtime, and you'll terminate the module. So it's just sort of about simplicity and safety, [INAUDIBLE]. I don't think there's a concrete attack that I could have [INAUDIBLE].

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. But here's the thing. You can't jump past the end. So the last thing that you could jump to is the last instruction. And by this rule, that instruction must be hold. So you could jump to the hold, but then would be at the hold. And you want to run past the end. So I think it's just sort of cleanliness, as far as I can tell, a guarantee for them to make sure that there's no weirdness in terms of wrap around at the engine. All right? So C5 is basically no instructions that span the 32 byte boundary. So this is sort of a variant of this rule that we talked about before, where every multiple of 32 must be a valid instruction that we solder in disassembly.

So otherwise, we'll jump to the middle of an instruction and have the problem with that sys call that might be hidden there. OK. And then we have C6, which is basically all instructions reachable by disassembly from the start. So this ensures that we see every instruction, and we can check every instruction that could possibly be run at runtime. And I guess C7 is this basically all direct jumps. OK. So this is the example of that jump up there, where you code in the target right away. So it doesn't have a multiple of 32, but it still has to be a valid instruction that we solder in the left to right disassembly. Question?

AUDIENCE: So what's the difference between C5 and C3?

PROFESSOR: Yeah. So I think C5 says that if I have an instruction that's multiple bytes, it cannot stay on that

32 byte boundary. So suppose that I have my instruction stream. And here's an address, 32. Here's address 64. And maybe I have some sort of an AND instruction that sort of spans this boundary. So this is what C5 prohibits. No instructions can span this boundary. Because otherwise, we saw this AND instruction. But because you can always jump 32 byte multiples, you could jump into the middle of that. And who knows what will happen there? So this is what C5 prohibits for that.

And C3 is the counterpart to it on the jump side. So C3 says that whenever you jump, you're going to jump to a multiple of 32 bytes. And C5 says that everything you can find in a multiple of 32 bytes is a safe instruction.

AUDIENCE: I wonder if there's any redundancy because [INAUDIBLE].

PROFESSOR: Yeah, I don't know. I'm not sure. I had a little bit of that feeling when reading this list. It seems like, wow, they're all good rules. But I'm not sure it's the minimal, non-orthonormal set of-- or set of orthonormal rules that you need to enforce this. But any other questions about the set of restrictions they have? All right.

So I guess let's think about this homework question that we sort of assigned. It turns out-- I think there was actually a bug that Native Client had and that's in our sandbox at some point, which is that for some complicated instruction, I believe they didn't have the length encoding correctly, if I remember properly, in which case something bad might happen. I can't remember exactly what the bug was. But suppose that the sandbox validator gets the length of some instruction wrong. What bad things could happen? How would you exploit this?

AUDIENCE: You can hide a sys call or a [INAUDIBLE] potentially.

PROFESSOR: Yeah. So suppose that there's some fancy variant of some AND instruction that you could write down. And maybe the validator gets that wrong and thinks that, oh, this guy is actually six bytes, when in reality, it's five bites long. So you could plop this down, pull this AND down. And then the validator thinks, oh, it's six bites. I'll look six bites further. And I'll check whether this is a valid instruction. And we just have to make sure that whatever is six bytes later looks like a valid instruction. But the CPU itself, when it runs this code, is going to maybe only look five bytes from the AND because that's the real length of the instruction. So if we can use this extra byte to stick a sys call instruction, then we could be in business.

So if we stick that-- remember from that example above on x86, OXCD is how you make an

AND instruction. So if we stick a CD byte at the end of that AND, then maybe we can put something here that looks like an instruction but actually is going to be part of the AND instruction here. And then we can all of a sudden make a sys call and escape the inner sandbox. Makes sense? Any questions? So the validator in Native Client has to be really in sync with what the CPU is doing because it's sort of guessing, well, here's exactly how the CPU is going to interpret every instruction. It has to be right at every level here.

So that's going to be a little bit tricky to do right. But there are actually some other interesting bugs in Native Client that people have found. One, I think, has to do with not properly sanitizing the environment of the CPU when you're jumping into the trusted service runtime. I guess we'll talk about this in a second. But the trusted service runtime is going to basically run with the same sort of set of CPU registers initially that the untrusted module was running with. So if there's something that it forgets to clear or reset or whatnot, then the trusted service runtime might be tricked into doing something that it wasn't meant to do, or the developers didn't want to do it initially.

OK. So let's see. Where are we now? So what we understand now is roughly how we can disassemble all the instructions and how to prevent disallowed instructions. So now let's look at, how do we keep the memory and references for both code and data within the bounds of the module? So for performance reasons, the Native Client guys actually start using some hardware support at this point to make sure this actually doesn't impose much overhead. But before we look at the hardware support they use, does anyone have any suggestions for how we could do it actually without any hardware support? Could we just enforce all the memory accesses going in bounds with the machine we have so far? Yeah.

AUDIENCE: You can instrument the instructions to clear all the higher bits.

PROFESSOR: That's right. Yeah. So actually we see that we have this instruction over here, which every time we, for example, jump somewhere, right now, we clear the low bits. But if we want to keep all the possible code that you're executing within the low 256 Megs, then you could just replace this with a 0. So we end with 0ffffe0. So this clears the low bits and also caps to at most 256 Megs. So this does exactly what you're sort of suggesting and would make sure that whenever you're jumping, you're saying within the low 256 Megs. And the fact that we're doing disassembly, you can also check for all the direct jumps, that they're all in bounds.

And that's actually not so bad. The reason I think they don't do this for code is because in x86

you can very efficiently encode an AND where all the top bits are 1. So this turns out to be, I think, basically a 3 byte instruction for the AND, and a 2 byte instruction for the jump. So the overhead is, like, 3 more bytes. But if you want to have non-1 high bits, then this is actually now a 5 byte instruction all of sudden. So I think they're worried a little bit about the overhead here. Question?

AUDIENCE: Isn't there also the problem that you might have some instructions which increment what version that you're trying to get? So you might say-- your instruction might have a constant offset or something like that.

PROFESSOR: Well, I think, yeah. You would probably prohibit instructions that jump to some complicated formula of an address. So you will only support an instruction that jumps directly to this value. And this value always gets ANDed and--

AUDIENCE: It's more for memory accesses rather than--

PROFESSOR: Yeah. So that's a good point because this is just code. And for memory access, there's lots of weird ways on x86 to refer to a particular memory location. In that case, you basically have to first compute the memory location, then insert an extra and, and then do the access. And I think that's the real reason why they're actually very worried about the performance overheads of this instrumentation. So on x86, at least on the 32-bit, which is what this paper talks about, they actually use some hardware support instead to limit the code and data addresses that the untrusted module can refer to.

So this actually leverages some somewhat esoteric hardware in x86. So let's see what it looks like first before we start figuring out how we're going to use it to sandbox the next module. So this hardware is called segmentation. It's sort of left over from the days before x86 actually even had paging. So the way it works is actually there's a segmentation on x86. They way it works is that whenever a process is running, there's actually a table maintained by the hardware. Let's call it the segment descriptor table.

And what this table has is just a bunch of segments, numbered from 0 up to whatever the size of the table is, kind of like file descriptor in Unix. Except every entry has two values in it, some sort of a base and a length. And the same for every entry, base and length. OK. So what this table does is it tells us that we have a couple of segments. And whenever we refer to a particular segment, what this in some sense means is that we're talking about a chunk of memory that starts at address base and goes for this length from that base upwards. And the

way that this actually helps us to enforce memory bounds is that on x86, every single instruction in one way or another, whenever it's talking about memory, actually talks about memory with respect to a particular segment in this table.

So for example, when you actually do something like move a memory value from a pointer stored in the EAX register into maybe another pointer stored in the EBX register, what you think it does is it figures out, well, what's this address? It knows the memory at this address. And then it figures out, OK, what's this address? And it stores the value there. But in fact on x86, whenever you're talking about memory, there's an implicit-- what's called a segment descriptor, which is kind of like a file descriptor in Unix. It's just an index into this descriptor table. And unless specified otherwise, every opcode has a default segment in it.

So when you're doing a move, this is actually relative to the DS or the data segment register. So it's like a special register in your CPU that's a 16-bit integer, if I remember correctly. And that 16-bit integer points into the descriptor table. And the same here. This is actually the relative to that DS segment selector. Actually, a bunch of these guys are 6 segment selectors on x86. There's a code selector CS, DS, ES, FS, GS, and SS. And the code selector is sort of implicitly used to fetch the instructions. So if your instruction pointer points somewhere, it's actually relative to what the CS segment selector says.

And most data references implicitly use either DS or ES. And then FS and GS are some special things. And SS is always used for the stack operations. If you push and pop, they implicitly come off of this segment selector. It's a fairly baroque machinery, but it turns out to be hugely useful for this particular use case because what happens is if you access some address at maybe some selector DS: some offset or some address here, what the hardware will actually do is translate it into-- well, it'll put this address, and it'll add the table of DS, the base to this guy. And it'll actually take the address modulo the length from the same table.

So whenever you're doing a memory access, it's actually going to have the base of your segment selectors, sort of descriptor table entry, and take your address that you're actually specifying and mod it with the length of the corresponding segment. Does this make sense? It's a little baroque, but that's what [INAUDIBLE].

AUDIENCE: So why isn't this used, for example, for buffer protection?

PROFESSOR: Yeah. It's a good question. So could you use this for protecting against buffer overflows? What's the plan? You could basically set up-- for every buffer that you have, you could put the

buffer's base here. You can put the size of the buffer there.

AUDIENCE: What if you don't need to put it there before you want to write to it? You wouldn't need it there constantly.

PROFESSOR: Yeah. So I think the reason that this isn't used so much for protecting against buffer overflows is that this table has at most 2 to the 16 entries because these descriptors are actually 16 bits long. And in fact, a couple of the bits are used for other stuff. So in fact, I think you can only stick 2 to the 13 entries here. So if you have more than 2 to the 13 variable size array things in your code, then it's probably going to overflow this table. That was actually a little bit weird for the compiler to manipulate this table because the way you actually manipulate it is through system calls.

So you can't actually directly write to this table. You have to issue a system call to the operating system. And the operating system will pull this table into the hardware for you. So I think most compilers just don't bother having this complicated story for managing buffers. Multex actually did this though. So on Multex, you actually kind of have 2 to the 18 distinct segments and 2 to the 18 possible offsets within a segment. And every possible shared library chunk or chunk of memory was a distinct segment. And then they would all be range checked, not maybe at the variable level. But still. Yeah.

AUDIENCE: Presumably, it's also a bit slower if you have to tap into the kernel all the time to--

PROFESSOR: That's right. Yeah. So there's also the overhead. I guess to set this up, there would be some overhead. Or if you create a new buffer on the stack, all of a sudden, you have to call in to this guy and add an extra. So yeah, it is nice machinery. But it's mostly used for coarser grain things because of the overhead of changing it a bit. Makes sense? All right. So how many of these guys actually use now the segmentation machinery? Well, you can sort of guess how it works. I guess by default all these segments in x86 have a base of 0 and a length of 2 to the 32. So you can access the entire range of memory you could possibly want.

So for Native Client, what they do is code in a base of 0 and a length of 256 Megs. And then they point all these six segment selector registers to this entry for a 256 Meg region. So then whenever the hardware does a memory access, it's going to mod it. The offset was 256 Megs. So it'll be restricted to the 256 Meg range of memory that's allowed for the module to [INAUDIBLE] modify. Makes sense? All right. So I guess we sort of roughly understand now

this hardware support and how this works and how you could eventually do the wealth segment selectors.

So if we just implement this plan, is there anything that could go wrong? Could we escape out of the segment selector in the untrusted module? I guess one thing you have to watch out for is that these registers are just like regular registers. And you can actually move values in and out of them. So you have to make sure that the untrusted module doesn't tamper with these registers, the segment selectors, because somewhere in your descriptor table is also the original segment descriptor for your entire process, which has a base of 0 and a length of 2 to the 32.

So if the untrusted module could somehow change the CS or DS or ES or any of these guys to point to this original operating system that covers your entire address space, then you could then do memory references with respect to this segment and get out of this sandbox. So as a result, Native Client has to add some more instructions to this prohibited list. So I think they basically prohibit all instructions that move into a segment selector DS, ES, et cetera, so that once you're in the sandbox, you cannot change the segment that you are referencing things with respect to. Makes sense? Yeah.

AUDIENCE: The segmentation such and such provides [INAUDIBLE].

PROFESSOR: Yeah. So it turns out that on x86, the instructions to change the segment descriptor table are privileged. But changing these indices into the table are completely unprivileged. Yeah. Other questions? Yeah.

AUDIENCE: Can you initialize the table to put all 0 lengths in the unused slots?

PROFESSOR: Well, yeah. So the unused slots-- you could-- yeah. You can set the length of the table to something so there are no unused slots. It turns out that you actually need this extra slot containing 0 and 2 to the 32 because the trusted run time is going to need to run in this segment and access the entire memory range. So you need this entry in there for the trusted runtime to work. Question?

AUDIENCE: [INAUDIBLE] set list? In order to set the table output to some length, do you need to--

PROFESSOR: No, actually. It's pretty cool. It was like something that you don't have any root for. On Linux, there's actually a system called-- I think it's called modify_ldt for local descriptor table. And it lets any process modify its own table. These tables are actually per process. Well, as

everything in x86, it's actually more complicated. There's a global table, and there's a local table. But the local table is for a process you can modify [INAUDIBLE]. Makes sense? All right. OK.

So I guess one thing we could now try to figure out is, how do we jump in and out of the Native Client runtime or out of the sandbox? So what does it mean for us to jump out? So we need to run that trusted code. And the trusted code lives somewhere up above the 256 Meg limit. And in order to jump there, we basically have to undo all these protections that Native Client sets in place. And they basically boil down to changing these selectors. So we already-- I guess our validator isn't going to enforce the same rules for the stuff above 256 Megs. So that's easy enough. But then we need to somehow jump into the trusted runtime and reset these segment selectors to the right values, to this giant segment that covers the entire process address space.

So the way that works on Native Client is through this mechanism they call trampolines and springboards. So all these guys are things that live in the low 64k of the module. And the cool thing about that is that these are going to be sort of chunks of code laying at the lower 64k of that process space. So that means that this untrusted module can actually jump there because it's a valid code address. It's going to be on the 32 byte boundary potentially. And it's going to be within the 256 Meg limit. So you can jump to these trampolines. But the Native Client runtime is going to actually copy these trampolines from somewhere outside. So the Native Client module isn't allowed to supply its own trampoline code. The trampoline code comes from the trusted runtime.

So as a result, it actually contains all these sensitive instructions, like moving DS and CS, et cetera, that the untrusted code itself is not allowed to have. So the way you actually jump out of the sandbox and into the trusted runtime to do something like malop or create a threat is you jump to a trampoline, which lives at a 32 byte offset. So maybe it's an address. Well, who knows? Maybe it's 4,096 plus 32. And it's going to have some instructions to basically undo these segment selectors. So what it's probably going to do is it's going to move some value-- I don't know, maybe 7-- into the DS register and maybe some points to this entry here that's allowed for the entire 2 to the 32 address space.

And then you're going to effectively move CS and then jump somewhere up into the service runtime. And this is basically going to be past 256 Megs. So there's going to be this jump in here that's not regularly allowed. But we're going to be OK with it because it's going to be into

a point in the trusted service runtime that is expecting these jumps. And it's going to perform proper checks afterwards on the arguments and whatever else that is being passed around. And we can actually do this, move DS here, because we know that it's actually safe. The code we're going to jump to isn't going to do anything arbitrary or funny with our untrusted module. Makes sense roughly, what's going on?

So why do these guys bother jumping out of the segments? Like, why not just put the whole thing in the trampoline? It seems like more work on some level. Yeah.

AUDIENCE: We only have 64.

PROFESSOR: Yeah, you don't actually have a lot of space. Well, I guess you have 64k. So that's potentially maybe enough for-- maybe you can move a malop in there. But the problem is not so much the 64k thing but this 32 byte restriction. And it's actually not a restriction on the trusted code, because the trusted code can do whatever it wants here. It's not going to be checked. The problem is that the untrusted code can jump to every 32 byte offset. So every 32 byte offset has to be prepared to be very special in its arguments. So you probably are going to have a hard time writing this code here with every 32 bytes rechecking the arguments and values and so on.

So basically, you have to jump out of the trampoline and into the runtime up here within 32 bytes of code. So then if you jump to the next thing, then, well, something else is going to happen here. But it's not part of this same trusted routine here. Makes sense? OK. So this is how you sort of jump out of the sandbox. To jump back in, you also need to-- you basically need to invert these transformations. You need to sort of set the DS back and CS back and so on. And the tricky thing is that if you're running outside of this 256 Meg limit, if you're running inside of the trusted run time, then you can't really reset these registers yet, because otherwise, you won't then be able to access any of the memory in your space outside.

So what they actually do is they have this thing called a springboard, which is how the trusted runtime from outside the 256 Meg limit is actually going to jump back into the Native Client module. So here it's going to reload the DS register with maybe whatever that limit in the segment descriptor is. Maybe let's say it's 1. And then it's going to reset other things, and then it'll jump to whatever address the trusted runtime wants to return to in the untrusted module. Makes sense? So this is how you sort of jump back in. And the only sort of tricky piece here is that you don't want the untrusted code to jump into the springboard itself. Maybe something

weird will happen. Who knows?

So what they do is they actually put a halt instruction as the first byte of this 32 byte multiple sequence. So if you jump to the beginning of this guy, you'll immediately halt. The trusted runtime is going to jump to one past this byte and be able to execute the springboard. But this is something that only the trusted runtime can do because regularly checked, this is not going to be allowed. Question?

AUDIENCE: Wait. Is it a springboard in the untrusted module?

PROFESSOR: So the springboard is within the 0 to 256 Meg part of the untrusted module. But it actually lives in that 64 bit chunk at the very beginning that doesn't actually come from the binary you download from some website. But it's actually patched into it by the Native Client runtime when it first loads this module into memory.

AUDIENCE: Why not just have it in the runtime?

PROFESSOR: Yeah. So why not have it in the runtime? So what happens if the runtime was allowed to supply the springboard? Is this bad? Yeah.

AUDIENCE: How would it know-- how would it know where to jump back to?

PROFESSOR: Well, I think what this actually is is it actually jumps to something like EAX. So the trusted runtime says, oh, I want to return to this address. It puts it on the EAX register, jumps here. The springboard does this, this, this and then jumps to EAX, wherever the trusted runtime's set up to jump. So what if the module came with its own springboard?

AUDIENCE: Well, you could do it as a natural jump type thing. But it shouldn't know anything about the descriptor table. That's a hardware--

PROFESSOR: Yeah. So actually, this is a really important instruction for sandboxing, the fact that we reload that descriptor to point at one of those limited descriptors up there. It's really important. And if the module was allowed to supply its own springboard, maybe it'll just skip this part and then not restrict itself back to 256 Megs. So once you jump through the springboard, you'll be able to still access the entire process address space. So the springboard is part of the enforcement mechanism. It sort of sets up the boundaries. So this is the reason why I think they don't want the springboard to come from the developer. Now-- yeah, question?

AUDIENCE: Can you put the springboard past 256 megabytes?

PROFESSOR: So I think they don't want to put the springboard past the 256 Megs because then you might have trouble jumping down. So you want to jump to a particular address, but you also want to set up extra registers here. So if you're-- I think this basically has to-- sorry. This has to do with setting that CS code descriptor segment because you want to set the code descriptor segment to this bounded segment, and you want to jump to some particular address at the same time. So I think it's easier for these guys to do it through a springboard because you first sort of jump to here.

Then you can set your CS value. But you can still execute the same code you're still running because you're within the 256 bound. Makes sense? I think basically it has to do with what atomic primitives the hardware provides to you. So basically, you want to set a whole bunch of these DS segment selector registers and the CS register and jump to somewhere at the same time. So this is one way for them to do it. I think that's maybe not as [INAUDIBLE]. Probably, if you tried hard, you could probably come up with some x86 instruction sequence that could do it from outside the bound of the address space in the module. Makes sense? All right. So I guess that's it for Native Client. And I'll see you guys next week. We'll talk about web security as far as I know.