PROFESSOR: All right, let's get started. Thank you for showing up to this very special pre-Thanksgiving lecture. I'm glad you guys have such devotion to security, I'm sure that you will be rewarded on the job market at some point. Feel free to list me as a recommendation. So today we're going to talk about taint tracking, and in particular we're going to look at a system called TaintDroid that looks at how to do this type of information flow analysis in the context of Android smartphones.

And so the basic problem the paper deals with is this fact that apps can exfiltrate data. So the basic idea is that your phone contains a lot of sensitive information, right. It contains your contacts list and your phone number and your email and all that kind of stuff. So if the operating system or the phone itself isn't careful, then a malicious app might be able to take some of that information and send it back to its home server, and that server can use it for all types of nefarious things as we'll talk about later.

The high-level solution that the TaintDroid paper suggests is that we should basically track the sensitive data as it flows through the system, and essentially, we need to stop it from going over the network. In other words, we need to stop it from being passed as an argument to networking system calls. And so presumably, if we can do that, then we can essentially stop the leak right at the moment that it's about to happen.

So you might think to yourself, so why are traditional Android permissions insufficient to stop these types of data exfiltrations? And the reason is that these permissions don't really have the appropriate grammar to talk about the type of attack that we're trying to prevent. So a lot of times these Android permissions, they deal with these things like can an application read or write to a particular device. But we're talking about something at a sort of different level of semantics. We're saying even if an application has been granted the authority to read or write a particular device, like the network, for example, it still might not be good to allow that application to read or write certain sensitive data over that device to which it has permissions.

In other words, using these traditional Android security policies, it is difficult to speak about specific types of data. It's much easier to talk about whether an application accesses a device or not. So you might think, all right, so that's kind of a bummer, but maybe we can solve this problem by-- we have this alternate solution, so we'll call this solution star. So maybe we can just never install applications that can do reads of sensitive data and also have network access.

At first glance, that seems to solve the problem. Because if it can't do both of these things, it either can't get to the sensitive data in the first place, or it can, but it can't send it anywhere. So does anyone have any ideas where this probably isn't going to work out very well? Everyone's already thinking about turkey. I can see in your eyes.

The main reason why this is probably a bad idea is that this is going to break a lot of legitimate applications. So you could imagine that there are a lot of programs, like maybe email clients or things like that, that should actually have the ability, perhaps, to read some data that's sensitive and also send information over the network. So if we just say that we're going to prevent this sort of and type of activity, then you're actually going to make a lot of things that work right now fail. So users are not going to like that.

There's also a problem here is that even if we did implement this solution, it's not going to stop a bunch of different side channel mechanisms for data leakage. So for example, we've looked in previous classes about how the browser cache, for example, can leak information about whether a particular site has been visited or not. And so even if we have a security policy like this, maybe we don't capture all kinds of side channels. We'll talk about some other side channels a little later in the lecture.

Another thing that this wouldn't stop is app collusion. So two apps can actually collaborate to break the security system. So for example, what if there's one app that doesn't have access network, but it can talk to a second application, which does. So maybe it can use Android's IPC mechanisms to pass the sensitive data to an application that does have network permissions, and that second app can actually upload that information to the server.

And even if the apps aren't colluding, then there may be some type of trickery that an application can engage in to trick some other applications into accidentally revealing sensitive data. So maybe there's some type of weakness in the way that the email program is written, and so perhaps that email program accepts too many random messages from other things that are living on the system. So perhaps we could craft a special intent that's somehow going to trick your Gmail application, for example, into emailing something to someone outside of the phone. At a high level, this approach doesn't really work very well.

One important thing to think about is OK, so it seems like we're very worried about the sensitive data leaving the phone. So what does Android malware actually do in practice. Are there any kinds of real world attacks that we're going to be preventing by all this taint tracking type stuff. And the answer is yes. So increasingly, malware is becoming a bigger problem for these mobile phones. So one thing it might do is it might use your location or maybe your IMEI for ads. So similarly to malware, it's actually going to look and see where you are physically located in the world and then maybe it will that oh, you're located near the MIT campus, therefore you must be a hungry student so hey, why don't you go to my food truck that happens to be located right where you are.

IMEI is kind of like this-- you can think of it as an integer that's like a per device uniquefier. So this could be used perhaps to track you in ways that you don't want to be tracked, in different locations, so on and so forth. So there's actually malware in the wild that does things like that. Another thing that malware might try to do is steal your credentials. So for example, it might try to take your phone number, or it might try to take your contact list, it might try to upload those things to a remote server. Maybe that's useful for trying to impersonate you, for example, in a message that's going to be used for spam later on. There's malware out there that does things like this today.

Perhaps most horrifyingly, at least for me, malware might be able to turn your phone into a bot. This, of course, is a problem that our parents did not have to deal with. Modern phones are so powerful that they can actually be used to send out spam messages themselves. So there's actually a pretty nasty piece of malware that's going around right now that seems to be targeting some corporate environments that's doing precisely this. So it gets to your phone and just starts sending out stuff.

AUDIENCE: So this type of malware, is it malware that subverts the Android OS, or is it just a typical app? If it's a typical app, it seems that it should be able--

PROFESSOR: Yeah. That's a good question. There's both types of malware out there. As it turns out, it's actually fairly easy to get users to click on things. So I'll give you an example. This isn't necessarily indicative of malware, more about the sad state of humanity. There'll be a popular game out there, let's say Angry Birds, for example. You go to the App Store and you type in

Angry Birds, I want to get Angry Birds. So hopefully the first hit that you get is the actual Angry Birds.

But then the second hit will be something like Angry Birdss, with two S's, for example. And a lot of people will go there, and maybe it's cheaper than the regular version, and they go there. It's going to present that thing that says, do you allow this application to do this, this, and this. The person is going say, yeah, because I got to get my Angry Birds, yeah, sure. Boom, then that person could be owned. So in practice you see now where it exploits both types of vectors. But you're exactly right that if you assume that the Android security model is correct, then the malware sort has to depend on users being foolish or naive and giving it network access, for example, when your tic-tac-toe game shouldn't really have network access.

Yes, so you can actually have your phone get turned into a bot. This is horrible for multiple reasons, not only because your phone is a bot but also because maybe you're paying for data for all those emails that are getting sent from your phone. Maybe your battery's getting ground down because you phone's just sitting around constantly sending ads about whenever, free trips to Bermuda or whatever.

There are actually malicious applications out there that will use your private information for bad. And the particularly bad thing about this bot here is that it can actually look at your contact list and some spam on your behalf to people that you know and make the likelihood of the victim clicking on something in that email much, much higher.

One thing to note, and this kind of getting back to the discussion we just had, so preventing this data exfiltration is very nice, right. But in and of itself, preventing that exfiltration doesn't stop the hack in the first place. So there's actually mechanisms that we actually should look at to prevent your machine from getting owned in the first place or to educate users about what they should and should not click on. So just doing this taint tracking isn't a full solution for preventing your machine from getting compromised.

How is TaintDroid in particular going to work? Let's see. So as I mentioned before, TaintDroid is going to track all of your sensitive information as it propagates through the system. So TaintDroid distinguishes between what they call information sources and information sinks. So these sources are things that generate sensitive data. So you might think of this as things like sensors. So for example, GPS, accelerometer, things like that. This could be your contact list database, this could be things like the IMEI, basically anything that might help to tie you, a particular user, to your actual phone.

So these are the things that generate the taint. And then you can think of these sinks as being the places where we don't want tainted data to go. And so in the case of TaintDroid, the particular sink that we're concerned about is the network. As we'll talk about later, you can generalize information flow to more scenarios than TaintDroid specifically covers. So you can imagine there might be other sinks in a more general purpose system. But for TaintDroid, they're literally caring about the network as the sink for information.

So in TaintDroid, they're going to use a 32-bit bitvector to represent taint. And so what this basically means is that you can have, at most, 32 distinct taint sources. So each sensitive data value will have a one in a particular position if it has been tainted by some particular source of taint. That's like, has it been derived from your GPS data, for example. Has it been derived from something from your contacts list, and so on and so forth.

One interesting thing is that 32 sources of taint is actually not that big, right. And so an interesting question is, is that big enough for this particular system and is it big enough in general for these information flow systems. So in a particular case of TaintDroid, 32 possible sources of taint seems to be somewhat reasonable, because it's actually looking at a fairly constrained information flow problem.

So it's saying given all the sensors you have on your phone, given all of these sensitive databases, and things like that, 32 seems roughly the right order of magnitude in terms of storing these taint flags. And as we'll see in the implementation of this system, 32 is actually very convenient, too, because what else is 32 bits? Well, an integer. So you can actually do some very efficient representations of these taint flags in the way that they actually build this.

As we'll discuss a little bit later, though, if you want to expose information flow to programmers in a more generic way, so for example, if you want programmers be able to specify their own sources of taint and their own types of sink, then 32 bits probably isn't enough. In systems like that you actually have to think about including more complex runtime support for a larger label space. So does that all make sense?

OK so roughly speaking, when you look at the way that a taint flows through the system, at a high level, it basically goes from the right hand side of a statement to the left hand side. So as a very simple example, if you had some statement, like you declare an integer variable that's going to get your latitude, and then a high level you call gps.getLat(), then essentially this thing here is going to generate a value that has some taint that's associated with it.

Some particular flag will be set that indicates that hey, this value I'm returning comes from a sensitive source. So the taint will come from here on the right hand side and go over here to the left hand side, and now that is actually tainted. So that's sort of what it looks like from the perspective of the human developer who writes source code. However, the Dalvik VM actually uses this register-based format at the lower level to actually build programs, and that's actually the way that these taint semantics are implemented in reality.

This is what's explained in table one of the papers, so they have this big list of classes of opcodes, and they describe how taint sort of flows for those types of opcodes. So for example, you might imagine that you have an operation that looks kind of like a move, and so it mentions a destination and a source. So in Dalvik, to register a base virtual machines, so you can think of these as being registers on this sort of abstract computation engine. And so essentially what happens here is that, like I said, taint goes from the right hand side to the left hand side. So in this case, when the Dalvik interpreter executes this instruction here, it's going to look at the taint label, this, and it's going to assign it over here.

Then you might imagine you have another instruction that's like a binary operation. So think of this as something like addition, for example. So here you'll have a single destination, but then you'll have two sources. And what will happen in this case is that when Dalvik interpreter encounters this instruction, it'll take the taints of both of these, construct a union of those, and then assign that union to be the taint tag over here.

Does that all make sense? It's fairly straightforward. So the table breaks down all the different types of instructions that you'll see, but to a first approximation, these are the most common ways that taint propagates through the system. Now there are actually some interesting special cases that they mention in the paper. So one of those special cases involves arrays.

Let's say that you have some code that's going to declare a character, and you get the value for the character somehow, doesn't really matter. And then let's say the program declares some array, we'll call it upper(). And it's basically going to have uppercase versions of letters. And so one very common thing to do in code is to index into an array like this using, for example, maybe just C directly, because as we all know, Kernighan and Ritchie teach us that basically characters are integers, so hooray for that. So you can imagine that you have some code that says something like the upper case version of this character here is going to be whatever is at a particular index in this table here, in the index that table by c like this. So there's a question of what taint should this receive.

It seems pretty straightforward what should happen in these cases, but in this case, it seems like we have multiple things that are going on. We've got this array here that may have some type of taint, we've got this character c here that may have some type of taint. What Dalvik decides to do in this case is a little bit similar to what it does in the case of this binary op here. So it's essentially going to say that this character over here is going to get the union of the taint of c and also of the array.

And the intuition behind that is that to generate this character, we somehow had to know something about this array here. We had to know something about this index here. So therefore I guess it makes sense that this thing should be as sensitive as both of these things combined.

AUDIENCE: Can you explain again move op and binary op, what exactly it means, like the union of a taint.

PROFESSOR: Yes, so imagine that-- let's look at the move op here. So imagine that this source operation here just had-- actually, let me get more concrete. So each variable, as I'll described in a second what a variable is, has this integer, essentially, that has a bunch of bits that are set according to what taint it has. So imagine each one of these values flying around has this associated integer flying around that has some bits set.

So let's say that this source had two bits set, corresponding to the fact that it had been tainted by two things, it doesn't really matter. So what the interpreter will do is it will look at this source thing, it'll look at the associated integer, and it'll say aha. I should take that integer has those two bits set and then essentially make that integer the taint tag for this.

So that's sort of a simple case, right. The more complicated case, like what does the union actually look like. So imagine that we've got these two things here and we've got source 0, source 1. And so I'm going to show you here, these are the tainted bits for this particular--

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Yeah, so imagine that you have this is the taint for this one. And imagine that the taint for this one is this. So what's the taint going to look like for dest? You basically take all of the bits that are saying either one of those and then assign that to that throwback to this one.

AUDIENCE: All right, thanks.

PROFESSOR: Yeah, no problem. And so one reasons, so once again I should emphasize this, so since we can represent all the possible taints in this 32 bits, as we were just discussing, doing this operation here, it's just bitwise operations. So this actually really cuts down on the overhead from implementing these taint bits. If you had to express a larger universe of taints then you might be in trouble, because you might not be able to use these very efficient bitwise operations to do things. Any other questions about that? OK.

So the way that arrays work is a little bit like that binary op like I mentioned. So this is going to get the union of the taint of this and that. And so one design decision that they made in TaintDroid is that they associate a single taint tab with each array. So in other words, they're not going to try to taint all the individual elements in there. So basically what's going to end up happening is that this is going to save them storage space, right, because for each array they declare, they'll just have a single through route to the entity that sort of floats around that array and represents all the taint that belongs to that array.

There is one question about why is it safe to not have a finer grain system for taint. Because it seems like an array is a collection of data, so why shouldn't we have a bunch of labels flying around for each thing that's in that array? And so the answer to that is that by only associating one taint tag with the array and making it the union of all the things that's inside, that actually is going to overestimate taint. So in other words, if you have an array that has two items in it, and that array is tainted with the union of all of those things, well, that's probably a little bit-- it's conservative. Because it may be that if something only accesses this, maybe it didn't learn anything about the taint that was over here.

But by being conservative, hopefully we will always be correct. In other words, if we underestimate the amount of taint that something had, then we might accidentally disclose something that we didn't want to actually disclose. But if we overestimate, then in the worst case, maybe we prevent something from going outside of the phone that should actually OK, but we're going to be err on the side of safety. Does that all makes sense?

Another instance of-- a sort of special case taint propagation that they mention are things like native methods. And so native methods might exist inside of the v in itself, so for example, the Dalvik VM exposes some function like a System.arraycopy(), so we can pass in anything through this, and internal to the VM, this is implemented in C or C++ code for reasons of speed. That's one type of example of a native method you might have.

Another thing you might have, a type of native method is what they call JNI expose methods. So the native interface essentially allows Java code to call into code that is not Java, that's implemented using x86 or ARM or something like that. There's a whole calling convention that's exposed here to allow those two types of stacks to interoperate. And so the problem with these native code methods, from the perspective of tracking taint, is that this native code is not being executed directly by the Dalvik interpreter. In fact, it is often not even Java code, maybe C or C++ code. So that means that once execution flow goes into one of these native methods, TaintDroid can't do any of this taint propagation that it's doing for code that lives in the Java world.

So that seems a little bit problematic because these things are kind of like black boxes. You want to make sure that when these methods return, we can actually somehow represent the new taint that was created by the execution of those methods. And so the way that the authors solve this issue is, they essentially result to manual analysis. So they basically say, there are not a whole lot of these types of methods here. So for example, the Dalvik VM only exposes a certain number of functions like Systems.arraycopy(), so we as human developers can look through this relatively small number of calls and essentially figure out what the taint relationship should be.

So for example, they can look at something like array copy and say, OK, based on what we know the semantics of this operation are, we know that we should taint the return values from this function in a certain way given the input values to this function. And so how well does this scale? Well, if there are in fact only a small number of things exposed by, for example, the VM in native code, this actually works OK. Because if you assume that the Dalvik VM interface doesn't change very often, then it's actually not too burdensome to look at these things, view the documentation, and figure out how taint's going to spread.

This may or may not be more troublesome. They give some empirical data that suggests that a lot of applications are not, in fact, including code alongside of them that's actually going to execute in C or C++. So they argued that empirically, this is not going to be a big problem. They also argue that for certain types of method signatures, you can actually automate the way in which these taint calculations are done. So they say that, for example, if only integers or strings are pass in to some of these native functions here, then we can just do the standard thing of tagging the output value with the union of all things the taints of the input. So in practice, it seems like this isn't probably going to be too big of a problem here. AUDIENCE: But why couldn't you just scan-- whatever scans your code [INAUDIBLE]?

PROFESSOR: Oh yeah, so in practice, what do they do. So they know that whenever the interpreter is going to execute something like this, then when the return value comes back, they do have special case code that's going to automagically say return values of System.arraycopy() should have this taint assigned to it.

AUDIENCE: Right, so what's the manual part of it?

PROFESSOR: Oh, the manual part of it is figuring out what that policy should be in the first place. So in other words, if you just look at off the shelf Taint or off the shelf Android, this is going to do something for you, but it's not going to automatically assign Taint in the right way. So someone looks at this and figures out what that policy is. Make sense? Any other questions?

It doesn't look like this is going to be a big problem in practice, although you can imagine that, for example, if there was this increasing amount of applications that define these native outcalls, then we could be in a little bit of a problem. All right.

So another type of data that we have to worry about assigning taint to, IPC messages. And so IPC messages are essentially treated like arrays. So each one of these messages is going to be associated with a single taint that is the union of the taint of all the constituent parts. Once again, this helps with efficiency because we only have to store one taint tag for each one of these messages. And in the worst case, this is conservative, it overestimates taint. But that should never result in a security leak. At worst, it should only result in something that should have been able to go over the network not being able to go on the network.

This is how things work when you're constructing the message, so that message gets the union of all the taint of its components. Then when you're reading it, what you receive in the message-- so extracted data gets the taint of the message itself, which makes sense. So that's how IPC messages are treated.

Another resource you might worry about is how a file's handled. So once again each file gets a single taint tag, and that tag is essentially stored alongside the file in its metadata on stable stores like the SD card or whatever. So this is basically the same conservative scheme that we've seen before. So the basic idea is that the application accesses some sensitive data like, for example, your GPS location, maybe it's going to write that data to a file.

So TaintDroid updates that file's taint tag with the GPS flag, maybe the application closes down, later on some other application comes out, it reads that file. When it comes into the VM, into the application, TaintDroid will look and see that it has that flag marked, and so any data that's derived from reading that file will also have that GPS flag set. So pretty straightforward, I think.

So what kind of things do we have to taint in terms of Java State. So there's basically five types of Java objects that need taint flags. And so the first kind of thing is local variables that live in a method. So we can imagine back over here, this is a local variable, char c, for example. So we have to assign taint flags to those things. You can also imagine that method arguments need to have taint flags. Both of these things here, these live in a stack. So a TaintDroid has to keep track of assigning flags and whatnot for those types of things. Also we need to assign flags to object instance fields. And so this is like, imagine that I have some object called c, it's a circle so of course the proper thing to do is I want to look at its radius. Here's a field here. And so we have to associate taint information for each one of these fields here.

Java also allows you to have a static class field, and so you need taint information for those. This is saying something like, for example, maybe the circle that some property, OK, we'll assign some taint information there. Then arrays, as we've already discussed before, we'll assign one piece of taint information per that entire array. And so the basic idea for how we're going to store these taint flags at the implementation level, is that we're going to try to basically store the taint flags for a variable near the variable itself.

The basic idea here is we've got, for example, let's say some integer variable, and we want to store some taint state with that. We want to try to keep that state as close to the variable as possible for reasons of making the cache work efficiently at the processor level. So if we were to store taint very far away from that variable, that can be problematic because probably, the interpreter is going to look at the memory value for the actual Java variable. It's going to want to very quickly thereafter, or even before that, look and see what the taint information is. Because if you look at these operations here, the same places in the code where the interpreter's looking at the values, it's also looking at taint.

Basically by storing these things close to each other, you try to make the cache behavior better. And the way that they do this is actually pretty straightforward. So if you look at what they do for method arguments and local variables that live on a stack, they essentially allocate the taint flags right next to where the variables are allocated. So let's say that we have our favorite thing in this class, a stack diagram, which you'll probably hate after you get out of here.

So you've got some local variable 0 on the stack, and then what TaintDroid will do is it will store the taint tag for that variable right next to where that local variable is in memory. So similarly, if you had another local variable here, then you would see its taint tag right down here. So on and so forth. Pretty straightforward. So hopefully you get these things in the same cache line, that's going to make the accesses very cheap. Yeah?

AUDIENCE: I was just wondering, how can you have a single flag for an entire array and a different flag for every property of an object. What if one of the methods of the object can access data which is stored in its properties. That would like-- know what I mean?

PROFESSOR: Let's see. So you're asking as a policy reason, why?

AUDIENCE: As a policy reason, right.

PROFESSOR: So I think some of this they do for implementation efficiency reasons. I think that for the case-- so they have some other rules, too. For example, they say that they don't say a length of the data array, is actually going to leak information, so they don't propagate taint for that. So some of it is just for reasons of efficiency. I think that in principle, that there's nothing that stops you from saying, take every element in the array and, when you do some particular access on it, then you just say the thing on the left hand side's going to get the taint, only that items.

It's not completely clear that's the right thing to do, though, because presumably in getting that thing into the array in the first place, the thing that did that had to know something about the array in some way. So I think it's a combination of both policy reasons-- they think that by being overly conservative, you shouldn't allow any data leaks that you want to prevent. And also I think that it kind of does intuitively make sense that accessing an array, you should have to know something about that array. And when you have to know something about something, that typically means that you want to get tainted by. Any other questions?

OK, so this is the basic scheme that they use for essentially storing all of this information close to each other. So you can imagine that for class fields and for object fields, you do a similar thing. So in the declaration of the class, you've got some slot memory for a particular instance variable, and then right next to that slot you have the taint information for that particular variable. So I think that's all pretty reasonable.

That's kind of a high level overview of how TaintDroid works, so if you get all this, then the basic idea behind TaintDroid is actually pretty simple. So at system initialization time or whatever, TaintDroid looks at all these sources of potentially tainted information, and essentially assigns a flag to each one of these things. So things like your GPS, your camera, and so on and so forth. As the program executes, it's going to pull out sensitive information from these sensitive sources, and then as that kind of thing happens, the interpreter is going to look at all these types of op codes here and basically follow those policy rules in the table on the paper, and figure out how to propagate taint through the system.

So the most interesting part is what happens if data attempts to exfiltrate itself. So essentially, TaintDroid can sit at the network interfaces and they can see everything that tries to go over the network interface. We actually look at the taint tags there and we can say if data that's trying to leave the network has one or more taint flags, then we will say no. That data will not be allowed to go in the network.

Now what happens at that point is actually kind of application-dependent. You could imagine that TaintDroid shows an alert to the user which says hey, somebody's trying to send your location over the network. You could imagine that maybe TaintDroid has some policies that are built in which, for example, maybe it allows that network flow to go out, but it zeros out all that sensitive data, so on and so forth. That's from a certain perspective, a little bit orthogonal to the core contribution of the paper, which is to find those data exfiltrations in the first place.

In the evaluation section of the paper, they discuss some of the things that they found. They do find that Android applications will try to exfiltrate data in ways that were not exposed to the user. So for example, they will try to use your location for advertisements, they will send your phone number and things like this to remote servers. Once again, it's important to note that these applications, typically they weren't breaking the Android security model in the sense that the user had allowed these applications with access to the network, for example. Or they had allowed these applications to have access to things like a contact list.

However, the applications did not exposed to the user in the EULA, in the End User License Agreement, that hey, I'm going to take your phone number and actually send it to some server in Silk Road 8 or whatever. That's actually misleading and deceptive, because most users, if they'd actually seen that in the EULA and they'd known that was happening, they might have at least had a second thought about whether they want to install this application or not.

AUDIENCE: Is it reasonable to guess that even if they put it in the EULA, that that's not really worth it because people never read those.

PROFESSOR: Yes, it is, in fact, quite reasonable to assume that. So even well trained computer scientists like myself do not always check out the EULA because it's like, you gotta have Flappy Birds or what are you going to do. I think what is useful, though, and this is kind of spiritually unsatisfying but useful in practice, is that if it is put in the EULA, then maybe there will be some virtuous individuals who do actually read the EULA.

AUDIENCE: And they could tell you like--

PROFESSOR: That's right, that's right.

AUDIENCE: --don't do that one.

PROFESSOR: Yeah, Consumer Reports or some moral equivalent will say our job is to read EULAs, and by the way, you shouldn't download this app. But you're exactly correct that relying on users to read pages of tiny print is basically-- they're not going to do it. They're going to hit Next and then keep on going. OK, so any questions up to this point?

I think that the rules for how information flows through the system are fairly straightforward. So as we were discussing, it's basically taint from the right hand side goes to the left side. Sometimes, though, these information flow rules can have somewhat counterintuitive results. So imagine that an application is going to implement its own linked list class. So it's going to define some simple class up here called ListNode and it's going to have an object field for data. And then it will have a ListNode object which represents the next thing in the linked list.

Suppose if the application assigned some tainted data to this field here. Some sensitive data derived from a GPS or whatever. So one question you might have is what happens when we calculate the length for this list. Should the length of the list be tainted? It may strike you as a bit counterintuitive that the answer is probability no, at least in the way that TaintDroid and a lot of these systems define information flow. So what does it mean to add a node to the linked list.

It basically means three things. So the first thing you do is you allocate a new list node to contain this new data that you want to add. Then the second thing you do is you assign to the data field of this new node. And then the third thing that you do is you do some type of patch up of the next pointers to actually splice the node into the list.

What's interesting is that this step here doesn't actually involve the data field at all. Just looking at these next values. Right, so what's interesting is that since only these data objects are tainted, how we calculate the length of a list. We basically start from some head node and we traverse these next pointers, and we count how many we traverse. So that algorithm is not going to touch the tainted data at all.

So interestingly, even if you have a linked list that's filled with tainted data, then just calculating the length of that list won't actually result in the generation of value that is tainted at all. So does that makes sense? That may seem a little bit counterintuitive, and this is one of the reasons why, for example, like when we were talking about the array, for example. They say array.length, I'm not going to generate any taint for that. It's because of reasons like this.

If you wanted a stronger assurance about-- not stronger assurance. But if you actually want to calculate the length of the list to generate a kind of value, we could imagine that your implementation, it's a bit goofy, but you can just decide to touch data for no real semantic reason other than to generate taint in the resulting length. Or, as I'll discuss towards the end of the lecture, you could actually use a language which allows you the programmer to define your own types of taint. And then you can actually define your own policies for things like this.

One nice thing about TaintDroid is that you as a developer, you don't have to label anything. TaintDroid basically does that for you. It says here's all the sensitive stuff that can be a source, here's all the sensitive stuff that can be a sink. You as a developer, you're ready to go. But if you want that pointer to be controlled, you might have to build some of the policies yourself.

All right, so in terms of performance overhead of TaintDroid, what does that look like? The overheads actually seem to be pretty reasonable. So there's going to be memory overhead, and that's the memory overhead, essentially, of storing all of these taint tags. And so there's going to be CPU overhead, and this is basically to assign, propagate, and check those taint calculations. And that's because of overhead like here. So any interpreting for the Dalvik VM, we're actually doing additional work. So looking at the source, looking at this 32 bit taint information, we're doing the or operations that we discussed before, and so on and so forth. So that's computational overhead.

These overheads actually seem to be pretty moderate. So for memory, the authors report about 3% to 5% in terms of the extra RAM space you need to store those taint tags. So that's not too bad. The CPU overhead is higher, which I think makes sense. They're both somewhere between, let's say, 3% and about 29% CPU overhead. And the reason why I think it's reasonable to see why that's higher is because you can imagine that every time you step into the interpreter loop, you're having to look at these tags and do some operations. So even though it is all these bitwise operations, you have to do that all the time. So that seems like it's going to get painful, whereas basically, the overhead for this, OK, so you put a couple extra integers in memory somewhere. That doesn't seem, maybe, too bad.

Even on it's high end, 29%, in of itself maybe that's OK, because Silicon Valley keeps telling us that we need phones that have like quad cores and whatnot, so probably have a lot of spare cycles sitting around. So maybe that's not all that crushing. Although there might be a problem with battery life. So even if you have these extra cores, you might not want your phone getting hot in your pocket as you're just sitting there, just sort of churning and calculating some of this stuff.

I think for here, the main issue here would be if this is bad for your battery. If it's not bad for your battery, then probably even at that high end, that may not be that bad. So that is essentially an overview of how TaintDroid works. Any more questions before we--

AUDIENCE: Do you tag something that also has been there all the time? Do you tag every variable, or only tag the ones that have this?

PROFESSOR: Yes, so you basically tag everything. So in theory, there's nothing that prevents you from not allocating any taint information for stuff that has no taint at all. I think the problem, then, with it-- then once something gains even one bit of taint, then you have to do dynamic sort of layout changes. So what if on the stack, this local here, then it had a taint, so now you're allocating with this, and it does get taint. Or you have that extra taint flag live on the heap, and you're going to see how it rewrites the stack, and then someone made your code-- so we're going to see how that works. So in practice, typical use is like shadow memory somehow, so every byte in the application is backed up by some byte of extra information somewhere. And in the case of TaintDroid, that shadowing actually lives alongside of the actual variable itself. Anyone has another question? OK. Cool.

This system essentially tracks information at the level of these high level Dalvik VM instructions. So one thing you might think to yourself is, could we track taint at the level of x86 instructions or the ARM instructions. One reason why that might be useful is because then we could actually understand how information flows through arbitrary applications, not just ones that are running inside this tricked out VM that requires you to run Java and so on and so forth. So why not track taint at that level.

It turns out that you can, in fact, do that. So there are projects that we looked at at tracking taint at this low level. What's nice is that you maybe get that increased coverage. You don't throw a line into [INAUDIBLE] for how, for example, Java code interacts with native code methods. It's all eventually going to result down to x86 instructions executed, so that removed a lot of the manual effort that you as a developer have to do to sort of understand it's the taint semantics if you use native methods. But the problem with that, if we track at this low level, it can be very expensive to do this. You can also get a lot of false positives. So if they're spec'd to the expense, there's also this issue of correctness.

As you may know, x86 is an adversarially complex instruction set. There's all kinds of crazy things that it can do. I don't know if you've ever seen an x86 instruction manual, they're huge. So they'll have one huge manual that's this thick, and then it'll say this is instructions whose letters start with M through P, and there'll be this full on series about that. So it's actually pretty tricky to think about what it means to actually track taint at the level of x86 instruction. Because even seemingly simple instructions, like sometimes at, they're setting all types of internal processor registers and flags and things like that. So it's very difficult to describe in the first place.

If you could do that, it's also oftentimes very expensive. You're sort of looking at things at a very, very low level. So the amount of state you have to track might get very large very quickly. It might be a very sensitive computational clause. Then there's this issue of false positives. This is actually pretty devastating. You can get into bad problems if you ever have kernel data that improperly gets tainted. And if this happens, maybe because your infrastructure's trying to be ultraconservative, it doesn't want to miss anything, so it says well, I'm going to err on the side of security. And I'm going to taint some of this kernel data structure, then what you get here is this exciting term they call taint explosion.

What this basically means is that at a certain point, there are certain things that if they end up getting tainted, they're involved in so many calculations that essentially everything in your program gets polluted. It's like one of these things in Dungeons and Dragons where you touch this evil thing and eventually death spreads throughout your body.

This is very bad, because if you can't tightly constrain the way that taint flows through the system, then eventually what's going to end up happening is that you let this run for a while, the system's going to say you can't do anything. You can't send anything over the network, you can't display anything on the screen, because everything in your system seems like it's been tainted by some sensitive error, even if that's not the case.

One way that this can happen is if somehow the stack pointer or the break pointer get tainted. If this happens, you're probably in a world of hurt. You can imagine that all of the instructions in x86, for example, that access the stack, they all go through ESB. So the stack register gets corrupted somehow, that's bad. If the break point register gets bad, a lot of times when you want your equivalents to access local variables, it has to go the EBP indirectly. So if anybody ever touches those in terms of taint, it's basically game over. So there's a link in the lecture that's about a paper that acknowledges some of this stuff and basically says that we have to be very careful when we do taint tracking at this low level because very quickly, if you're looking at how this works in the Linux kernel, there are certain optimizations the Linux kernel would do to make its code fast, but will result, unintentionally, in the break pointer or the stack pointer getting tainted. And once that happens, you can't really do anything useful with the taint tracking system.

AUDIENCE: So how do you do this [INAUDIBLE] programs? It seems like you have all these register files in the CPU.

PROFESSOR: Yeah, so great. So all those register files, it hangs back to the correctness case. So unless you are very, very good at understanding x86 architecture, there are going to be things that you miss. It terms of computation level, how do you actually do this thing. There's this-- I think the most popular way, and I could be wrong about this. So when I say it's popular, the way I know about, because I'm a knowledge [INAUDIBLE], right. There's this system submitter called Bochs, I think it's spelled like this. They actually have something called TaintBochs, which actually does x86 level innuation of flow.

And it's actually an interpreter, you can think of it as. So it's going to take your entire OS and all your applications, and it's going to look at each x86 instruction and try to simulate what the hardware would do. So you can imagine this is very, very slow. What's nice about that is you don't require any hardware support, and then it's relatively straightforward to tweak your software model of how things work, if you discovered that you weren't tracking some registered files or something like that.

AUDIENCE: So the ideal solution would be architectural support.

PROFESSOR: Yeah, so there have been techniques to do that, too. That gets a little bit subtle because, for example, if you look here you've looked at how we've allocated the taint state next to the variables themselves. So if you bake in that support in the hardware, it can be very difficult to, for example, change the way you want the layout to work. Because then it's like baked into the silicon. You could imagine doing some of this because at a high level-- where do we have it. So the Dalvik VM and TaintDroid is executing these high level instructions and it's assigning taint at this level. You can imagine doing that at the hardware level, too. So actually, if this is the silicon, you can probably make that work. So that's definitely possible. You had a question?

AUDIENCE: What does TaintDroid do with information built from branching and permission tests.

PROFESSOR: Oh, we're going to get to that in a second. So just hold that thought, we're going to get to that.

AUDIENCE: I'm curious, how long was it to things like buffer overflow because all the things are so nested together [INAUDIBLE]?

PROFESSOR: That's a good question. So presumably, one would hope that in a language like Java there are no buffer overflow, right. But you can imagine in a language like C, for example, where you didn't have this protection, maybe there's something catastrophic that could happen or somehow, if you did a buffer overflow and then you were able to overwrite taint tags and you could set this to zeros, then you could just let your data exfiltrate.

AUDIENCE: I think if it's super predictable, like one every other one for the next q variables, there's no stacking--

PROFESSOR: I was going to say, that's exactly right. So you run into somewhat similar issues like what we can discuss with the stack canaries, because basically we have this data on the stack, like in this particular layout, that you don't neither want to make it impossible to overwrite, or if it is overwritten, one that's hacked in some way. So you're exactly right about that.

So you can in fact do taint tracking at this low level although it may be expensive and a little bit difficult to get right. So you might say well, why don't we just punt on this whole issue of taint tracking in the first place and instead we're just going to look at the things that the program tries to output over the network, let's say, and just do a scan for data that seems sensitive. That seems to be much more lightweight, you don't have to do this dynamic instrumentation of all the things the program's doing.

The problem with that, though, is that that will only work as a heuristic. In fact, if the attacker knows that this is what you're doing, then it's pretty easy to subvert that. So if you're just sitting there and you're trying to do a grep for numbers, Social Security numbers, then the attacker can just use base 64 encoding, or do some other wacky thing, compress it. It's actually trivial to get past that type of filter. So in practice, that's completely insufficient from the security perspective.

Now let's get back to the question that you brought up, which was basically how can we track flows through things like branches, for example. So this is basically going to lead us to a topic that's called implicit flows. And so an implicit flow occurs typically when you have a tainted value that's going to affect the way that another variable is assigned, even though that implicit flow variable doesn't directly assign variables. This will make more sense with a concrete example.

Let's say that you have an if statement that does something like, it's going to look at your INEI and it's going to say if it's greater than 42, maybe I'm going to assign 0 to x. Otherwise I'm going to assign 1. So what's interesting here is that we're looking at this sensitive data here and we're doing some comparison of it up here, but when we're assigning to x down here, we're not actually assigning something that is directly derived from the sensitive data here. This is an example of one of these implicit flows. Because the value of x is actually dependent on this thing here, but the adversary, if they're clever, can sort of structure their code in a way that there's no direct assignment.

Now note that even here, instead of just assigning to x, you can just say let's try to send something over the network. You might say over the network x is 0, or x is 1, or something like that. So that's an example of one of these implicit flows that a system like TaintDroid cannot actually handle. So do people sort of see the problem here at a high level? Yes. This is called an explicit flow as contrast to those direct flows like from the assignment operator.

AUDIENCE: What if [INAUDIBLE] a native power function that did exactly [INAUDIBLE]? Because the output in that case would be, right?

PROFESSOR: Well, let's see. So it depends. So if I understand your question correctly, you're saying there could be some native function that does something that's sort of equivalent to this, and so for example, TaintDroid wouldn't know necessarily, because it can't look inside this native code to see this type of thing. The way that the authors claim that they would handle that is that they would say for native methods that are defined by the VM itself, they would look at the contract that method exposes and they might say things like I take these two integers and then return the average. So then the TaintDroid system would say we trust that the native function does that, so we need to figure out what the appropriate tainting policy should be.

However, you are correct that if something like this was sort of hidden inside and for whatever reason wasn't exposed to the public-facing contract, then the manual policy that the TaintDroid authors came up with might not catch this implicit flow. It might actually allow information to leak out somehow. But I mean for that matter, there might even be a direct flow in there that the TaintDroid authors couldn't see and you might still have an even more direct leak.

AUDIENCE: So in practice, this seems very dangerous, right? Because you can literally send the whole [INAUDIBLE] value by just looking at this last three--

PROFESSOR: That's right. We had class a few times where you'd sit in a while loop and you'd try to construct these implicit flows to do these types of things. There's actually some ways that you can think about trying to fix some of this stuff. At a high level, one approach you can do to try to prevent this stuff is you can actually assign a taint tag to the PC. Then essentially you taint it with the branch test. So the idea here is that we as humans can look at this code here and we can tell that there's this implicit flow here, because we know that somehow to get here, we had to look at the sensitive data.

So what does that mean at the implementation level? That means that to get here, there's something about the PC that has been tainted by sensitive data. To say that we have gotten here is to say the PC has been set to here or to here.

At a high level we could imagine that the system would do some analysis and it would say that at this point in the code, the PC has no taint at all. At this point, it gets tainted somehow by the INEI, and at this point here, it's going to have that taint.

So what will end up happening is that if x is a variable that initially shows up with no taint maybe we'll say OK, at this point, it's actually going to give the taint of the PC which is actually going to taint it there. So there's some sublety here that I'm glossing over, but at a high level that's how you can capture some of these flows here by actually looking and seeing how the PC is getting set, and then trying to propagate that to the targets of these if statements.

Does that all makes sense? OK. And if you're interested in learning more about this, come talk to me, there's been a lot of research into this kind of stuff.

However, you can imagine that the system I just described may be too conservative once again. So imagine that instead of having this code here, this was also 0. So in this dump case, there's absolutely no reason to taint x with anything related to the INEI, because you didn't actually leak any information in either of these branches. But if you use it with a naive PC tainting scheme, then you might over-estimate how much x has been tainted by. So I should say there's some subtlety you can do to try to get around some of these issues, but it's a little bit tricky. Does this all make sense? All right.

AUDIENCE: Just a question.

PROFESSOR: Oh, sorry.

AUDIENCE: When you get out of the if statement, so you're out of the branch, do you [INAUDIBLE] taint out?

PROFESSOR: Yeah, so typically, yes. So like down here the PC taint would be cleared. So it would only be set inside these branch things here. And the reason for that is because essentially, by the time you get down here, you get down here regardless of what the INEI was. So yeah, you clear that. It's a good question. Let's see.

You talked about how you might be able to taint at this very low level, even though that might be expensive, one reason why it might be useful is because it will actually allow you to do things like see what your data lifetimes look like. So a couple lectures ago, we talked about the fact that a lot of times key data, for example, will live in memory longer than you think that it should.

So you can imagine that even if some of the x86 or ARM level taint tracking is expensive, you can imagine using it to form an audit of your system and actually tainting, let's say, some secret key that the user entered, and just seeing where that goes throughout your system. It's an offline analysis, it's not facing customers, so it's OK for it to be slow. That might actually really help you to figure out oh, this data's getting into the keyboard buffer, it's getting into the x server, it's getting to wherever. So even if it's slow, that can still be very, very useful. So I just wanted to mention that briefly.

One interesting thing you might think about is the fact that as I mentioned, TaintDroid is nice because it constrains the universe of taint sources and taint sinks. But as the developer, maybe you want to actually explicitly assert some more fine grain control over the labels that your program interacts with. So now as a programmer, you want to be able to say something like this. So you query some int, and let's say we call it x, then you associate some label with it. Maybe the name of this label is that Alice is the owner of this data, but Alice permits Bob, or something labeled with Bob, to be able to see that.

TaintDroid doesn't let you do this, because it essentially controls that universe of labels. But maybe as a programmer you want to be able to do a thing like this. You can imagine that your program has various input channels and output channels, and all of these input and output channels, they all have labels, too. And these are labels that you as a programmer get to actually pick, as opposed to the system itself trying to say here's this group of fine set of things. So maybe say for input channels, you know the read values, maybe they get the label of the channel.

That's very similar to how TaintDroid works right now. So if you read something from the GPS, that read value is the taint of the GPS channel, but now you as a programmer can choose what those labels are. And then you could imagine that for output channels that label will channel has to match some label value we've written.

You can imagine other policies here as well. But the basic idea is that there are actually program managers that allow you the developer to pick what the labels are and what the semantics for those labels can be. So what's nice about some of these is they do require the programmer to do a little bit more work, but the outcome of that work is that static checking-- and by static checking I mean checking that's done at compile time-- can catch many types of information flow bugs.

So if you're diligent about labeling all of your network channels and screen channels with the appropriate permissions, and you're very diligent about leaving your data like this, what can happen is that at compile time, when you compile your program and your compiler can tell you things like hey, if you were to run this program, then you actually have an information leak that this particular piece of data will pass an equal channel, which is untrusted.

And at a high level, the reason why static checking can catch a lot of these bugs is because usually speaking, when you think of some of these annotations, they're somewhat similar to types. So the same way that compilers can catch errors involving types and installing type language, you can imagine that the compiler in a language like this can codes some calculus over this label, and in many cases, determine hey, if you would actually run this program, this would be a problem. So you really need to fix the way that the labels work, maybe you need to explicitly declassify something, so on and so forth.

AUDIENCE: You can't just [INAUDIBLE]?

PROFESSOR: Yeah, yeah, that's right. So depending on the language, these labels can associate people with IO ports, all that kind of stuff. That's exactly right. So this is just interesting to know about, because TaintDroid has a very nice general introduction to this information flows stuff, but there's actually some really hardcore systems out there than can express much richer semantics in the control of a program with respect to information flow.

And you know, too, that when we talk about static checking and being able to catch many bugs, it's actually preferable to catch as many bugs using static checking and static failures as opposed to dynamic checking and dynamic failures. There's a very subtle but powerful reason for why that is. The reason is that, let's say that we defer all of the static checks to the runtime, which you could certainly do. There's no reason you couldn't take all the static checks and give you a name for it. The problem is that the failure or success of these checks is actually a covert channel, perhaps.

So the attacker could actually feed your program some information and then see whether it crashed or not. And if it crashed, it might say, aha, you've passed some dynamic check of information flow, that must mean something was secret about this value I sort of cajoled you into computing. So you want to try to make these checks as static as possible to the greatest possible extent.

If you want more information on this kind of stuff, maybe a good place to start, a word to search is Jif. It's a very influential system that built some of these issues of label computation. So you can start there and sort of roll forward. My co-professor actually has done a lot of good work on this, so you could ask him about that if you want to talk more label stuff. That's sort of interesting to know that TaintDroid is actually fairly restrictive in the expressiveness of the labels it allows you to look at. There are systems out there that allow you to do more powerful stuff.

Finally, what I'd like to talk about is what we can do if we want to track information flows in some of these legacy programs, or through programs that are written in C or C++ that don't have all the fancy runtime support. So there's a very cute system, some of the same authors on this paper that looks at this issue of how can we track informational leaks in a system which we don't want to modify the application at all. This is the TightLip system.

So the basic idea is that they introduce this notion of what they call doppelganger processes. TightLip uses doppelganger processes evolved. So the first thing it does is it periodically scans a user's file system and it looks for sensitive file types. This might be things like your mail file, your word processing documents, so on and so forth. So what it's going to do for each one of these files is it's going to produce a scrubbed version. So for example, if it finds an email file, it's going to replace the to or the from information with, let's say, a string of the same length but just dummy data. Maybe all spaces or something like that. It does this as a background task.

Then the second thing it's going to do, at some point a process is going to start executing, and so then TightLip is going to detect when and if the process tries to access a sensitive file. And if such an access does take place, TightLip is going to spawn one of these doppelganger processes. And so what the doppelganger process looks like is very similar to the original process that tried to touch that sensitive data, but the key difference is that the doppelganger, which I'll abbreviate DG, reads the scrubbed data.

So imagine that-- so the process is executing, it tries to access your email file. The system spawns this new process, the doppelganger, that doppelganger is exactly the same as that original one, but it is now reading from the scrub data instead of the real sensitive data.

What happens then. Essentially, TightLip, we're going to run those two processes in parallel. It needs to just watch them and see what they do. And so in particular, we're going to see, do the processes issue the same system calls with the same arguments. And if that's the case, then presumably those system calls do not depend on the sensitive data. So in other words, if I start a process that tries to open some sensitive file, I feed it basically junk data, I let it execute. If that doppelganger process still does the same things that the regular process would have done, then presumably it wasn't influenced by that sensitive data at all.

So essentially doppelganger will let these processes run, TightLip will let these processes run, and then check the system calls here. And then it might happen that in some case the sys calls divert. So in particular, what if the doppelganger starts doing things that the regular version of the process would not have done, and then the doppelganger tries to make a network call. So just like in TaintDroid, when that doppelganger tries to make a network call, that's when we say aha, we should probably stop what's happening right now and then do something.

So if the system calls diverge, then the doppelganger makes a network call, then we're going to do something. So we're going to either raise an alert to the user or whatever. Kind of like in TaintDroid, but at this point there's a specific policy you can add in some particular system you're going to use. But this is sort of an interesting point at which you can say well, somehow that doppelganger process was affected by that sensitive data that was returned. That means that maybe if the user did not think that a particular process was going to get exfiltrated data, now the user can actually do an audit of that program to figure out why that program returned send that data over the network. So does anyone-- go ahead.

AUDIENCE: So if you're hitting something like a word file or whatever, you kind of have to know what you're zeroing out and what you're [INAUDIBLE].

PROFESSOR: Good question, that's right. So I was going to discuss some limitations, and one of the limitations is precisely that. You need to have per file type scrubbers. So you can't just take your email scrubber and use it for Word. And in fact, if those scrubbers miss something, so if they don't redact everything, then this system may not catch all the possible sensitive data leaks. So you're exactly right about that. But I think-- go ahead.

AUDIENCE: So if I understand, why should the process look at the data before saying go ahead? Why wouldn't you just send the stuff in?

PROFESSOR: Why would the process--

AUDIENCE: If the process plans to input data, [INAUDIBLE]?

PROFESSOR: Oh, no, no. From the perspective of the doppelganger, I mean, it may try to, in fact, look and see things like does this email address make sense, for example, before it tries to send it out. But the doppelganger process, it shouldn't know that it's gotten this weird scrubbed data. So this gets back a little bit to the question we were just talking about. If your scrubber doesn't scrub things in a semantically reasonable way, the doppelganger may, in fact, crash, for example. It expects things in this sort of format, but it's not.

But at a high level, the idea is that we're trying to trick the doppelganger into doing what it would do normally, but on data that's different in the original version and see if there will be that divergence. So one drawback is that, like we're discussing, this basically puts the scrubbers in TCB and if they don't work properly, doppelgangers might crash, you might not be able to catch some violations, things like that.

But the nice thing about this is that it works with legacy systems. So we don't have to change anything about the application itself runs. We just have to make some fairly minor changes to the OS kernel to be able to track the system call stuff, and then things sort of work. It's very, very nice. And the overhead of the system is essentially the overhead of running an additional process, which is fairly low in a modern operating system.

This is just sort of a neat way to think about how to do some type of limited taint tracking without doing heavyweight changes to the runtime without requiring changes from the OS-- or sorry, from the application.

AUDIENCE: Are we only doing parallel or waiting for each one? Are we running both processes and then after that we can just check that the system calls are the same? Like when do we check--

PROFESSOR: Yeah, two questions. So as long as the doppelganger process does things that the OS can control and keep on the local machine, you can imagine running the doppelganger process and the regular one forward. But as soon as the doppelganger tries to affect external state, so maybe the network is doing this and that. Maybe you can think of some other linked sources like that. Maybe there's something like pipes, for example, that the kernel doesn't know how to create doppelganger state for. At that point you have to stop it and then declare success or victory, basically. Any other questions? All right, well, that's the end of the lecture. Have a good Thanksgiving. See you next week.