PROFESSOR: All right. Let's get started. So today we're going to talk about Android security, which is-- you can sort of think of it as an interesting case study of a system that was designed with quite a bit of attention paid to security in the first place. So this is perhaps in contrast to many of the systems we've looked at so far, like Unix, or the web, browsers, where security was in many ways bolted on after the fact and wasn't really designed in quite the same way as you see in this paper, where these guys who designed the Android were very worried about particular classes of attacks, and constructive mechanisms. Then figured out a better way of structuring applications that is going to allow us to enforce security policies in a better way in Android.

And the cool thing about it is that it's actually a pretty widely used system. So unlike some research papers that might propose a new architecture, this actually gets used in practice. And there's lots and lots of Android devices out there. And we can talk about how well some things have worked out, how well some things didn't pan out. But we will, I guess, look at what parts of the design they thought were important, what did they miss, what, in practice, turns out to matter or not.

But it's kind of interesting. In some ways, it also uses existing systems that we've talked about. So Android is built on top of Unix. It's just a Linux kernel running underneath the entire phone. So in many ways, they use some of the familiar mechanisms you guys have seen in Lab 2 already, where you use Unix user IDs and groups and all these things to separate applications from one another. But in Android's case, they have a very different way of setting up user IDs and file permissions, et cetera, than in a typical Linux system.

So I guess let's start out by talking about what is the threat level? What are these

guys worried about on a phone? What's going on? What are they trying to protect against? What's the threat model? Yeah.

AUDIENCE: Applications that want to do malicious things?

PROFESSOR: Yeah. So they worry about-- I guess there was these applications that are going to run on the phone. And they might be malicious. And I guess there's-- well, that there's outright malicious applications that are just out to get you, maybe steal your private data. So things you might worry about-- there's data, there's things that might cost money, like sending an SMS message maybe, or making a phone call. There's maybe using the internet, et cetera.

So these are the, presumably, things you want to guard against or protect on your phone. And then there's things that go wrong. So presumably, there's malicious applications, because these guys want to allow users to install apps written by developers that Google has never heard of themselves. Or it might be that apps just have bugs themselves, that you have a well-meaning developer, but they forget to do something. And it would be nice to help these guys as well to build applications that remain secure despite the fact that the app developer isn't an expert in exactly every kind of vulnerability that might be exploited in their application.

So one thing we could do is, actually, we can-- because Android is what we use, we can look at various vulnerability reports. So there's this database called CVE that catalogs lots of common vulnerabilities in software systems. And it's actually kind of interesting. There is a number of reports, of course, of bugs in Android. And many of them are of the flavor you guys have already seen in the class. So there's still buffer overflows in some parts of Android. There's bad default choices for crypto systems. People forget to initialize the random number generator sometimes and generate predictable keys. So all these things do still happen. It's software. It's not immune from any of the other problems we've seen so far.

But one cool thing is that there doesn't seem to be a huge number of these issues. So they crop up from time to time. But by large, can fix these issues. And the system remains reasonably secure after you fix these bugs. So in many ways, this, I

think, design is working reasonably well. So we'll look at it, I guess, in more details later on as to which parts of the design are working to various degrees. But it seems to be a reasonably well thought out design. Or at least much more so than desktop Unix applications that you've seen so far. All right.

So maybe one way to approach this is to figure out how we're going to protect data and various services that might cost you money, et cetera, from malicious applications is first to understand what does an application look like in an Android system. And then we'll talk about how various permissions or privileges are configured in that application and enforced. So Android applications are quite different from what you've seen so far in terms of desktop apps or web applications.

So instead of being a monolithic piece of code with a main function that you start running, and you just keep going forever, they're actually much more modular. And the application, in the case of Android, is-- actually, you can think of it as a collection of components. And the paper talks about four kinds of components that the Android framework provides to you or gets the developer to think in terms of. And the components are roughly-- there's something called an activity. You might have an activity component. And this is just a thing that has a user interface. So these are things that actually display things to the user, or take user input, touches, key presses, et cetera.

In terms of security, the Activity thing has an interesting security property that you probably want to make sure your user input is going to one activity at a time. So the framework-- I believe [? an ?] Android actually ensures that there is only one activity that's getting your user input at a time. So if you are running your bank application, you can be reasonably confident that there's not other applications in the background grabbing the screen presses corresponding to your PIN number in your bank app. So having the framework be aware of these different activities helps enforce some security properties with respect to user input. All right. So these guys are the user interface components of an application.

And then there's three other types of components that mostly help an application

structure its own sort of logic and interaction with other components. So there is something called a service component. And this guy just runs in the background. So you might have a service component that monitors your location, like in the application these guys describe in the paper.

Or you might have services that pull things from the network in the background, et cetera. These guys have a content provider component. And you can think of these guys as just SQL database you can define. Or you can define a couple of tables with a schema, et cetera. And you can run SQL queries all over the data stored in that application. And having it be a component is going to allow the framework to control access to this database to say who's allowed to run queries against it.

And then there's something kind of unusual that hasn't shown up in other systems--something called a broadcast receiver. And this guy is going to be used for receiving messages from other parts of the system. So we'll talk about how applications interact with one other in terms of messages. All right. So this is some very high-level logical view of how you can think of an Android application. But in reality, all these are just Java classes or Java code that the developer writes.

And there's just some standard interface for an activity, for a service, for a broadcast receiver, for a content provider that you implement. But clearly, this is all just Java code. And this application box is really just a Java runtime that runs on top of your phone. And it's just a single process on the Linux kernel running on your phone. And all these components are just different classes or pieces of code running inside of this Java runtime process. That make sense? That's how it sort of translates to traditional processes that you might understand otherwise.

And the other thing that sort of shifts with an application is what's called a manifest. So this is code that the application developer writes or compiles. But there's also this manifest that sits on the side, which is a text or an XML file, really, that describes all these components and how other parts of the system should interact with this application.

So in particular, this manifest is going to talk about things called labels that we'll talk

about in a second that define both the privileges of this application in terms of what it should be allowed to do as well as the restrictions on who else should be able to interact with the different components of this application. That make sense? Questions about how that works?

AUDIENCE: Is the label something like, this app cannot do the phone call, or this app can send--

PROFESSOR: Yeah. So these labels are going to be things like, well, this application can dial a phone, or can send an SMS message, or can use the internet. So there's really two kinds of labels. So we can draw them out here. So each application has a list of labels that describe the privileges that the application has. So these are something like maybe DIAL PERMISSION for dialing a phone, maybe INTERNET, et cetera. So we'll talk about how they're described in a little bit.

So these are privileges that the application has. But then you can also stick labels on top of individual components. And there, they have a different meaning. So these are privileges that the application has. If you have a label [? on a ?] component, it's a requirement on anyone that talks with the component to have the corresponding label.

So in their example, maybe you have some sort of a FRIEND VIEW privilege. So you are able to view the locations of your friends. So that's like a privilege you might have in an application. So you're allowed to do this. But then in order to enforce this privilege, you might actually put this FRIEND VIEW label onto a particular component. So you might say, well, my content provider, the database storing the location of all my friends, might have the FRIEND VIEW label attached to it.

And what this means is that anyone that wants to access this database better have this label in their privilege set. So that's how you specify permissions. You can think these are as like generalized user IDs or group IDs from Unix, except they're arbitrary strings, which make them slightly more flexible. You don't run out of these guys. You don't worry so much about who gets the number 25 or silly things like that.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So turns out these guys aren't-- at least in the design, weren't super careful in scoping these guys out. So you could totally have two applications that decide to introduce the same label. So these labels are application-defined, partly. And you could have two applications, like Facebook and Google+. They say, oh, we want to both create a new permission string that's View Your Friends In A Social Network. All right. Well, they're the same string.

So by convention, what happens is that these strings are actually longer than what I am drawing out. And they have the domain of the Java-style domain of the application defining it in the string. So DIAL PERM, I think, is actually something like com.google.android.dialperm. I might have slightly screwed this up. But roughly, these are the kinds of strings that show up in permissions. So if you have well-meaning applications, then they won't collide in terms of these permission strings.

But it turns out that nothing actually enforces this, unfortunately, in Android. So you do have some potential problem that arise as a result of this, that-- I don't know why they weren't fixed. It's a little bit tricky to fix them. Maybe these guys thought, well, let's do them now, [? for ?] maybe they weren't thinking of these issues. Anyway. So we'll see what arises if you have two applications that conflict on the label names that we get. All right. Make sense? All right.

So this is what a single application looks like. It's a bunch of Java code, a manifest describing the permissions for the application, and the required restrictions on all the components. And then in order to communicate between applications, for the most part, it's done through what's called an intent, which is an Android thing that the developers of this framework introduced.

And an intent-- you can think of it as a structured message that-- we'll see how these components of an intent are going to be used in a second. But roughly, the intent has three important things. There's other fields, of course. But the main thing is the name of a component to which you want to send a message. There's the action that you want the component to take and the data, along with a MIME type,

that you want to send to this other component.

So just as an example-- this is a little abstract-- but what you can imagine is that this component is maybe-- you could imagine com.android.dialer/Dial or something. So this is how you name a component in Android. You specify the name of the application, which is kind of like this Java inverted domain name. Like, com.android.dialer is the name of an overall application to which you want to send an intent. And then you write something like /Dial. And Dial is the name of a component. Would [? lend ?] that target application to which you are sending this message. OK. So that's how you name the particular component where you want to send the guy.

The action, there's a predefined set of actions. You could stick in your own things as well. But you might have something like, I think, android.intent.DIAL. So this is a pre-defined string or, by convention, a string that applications put on the Action field if they want the phone dialer to dial a phone number for them. So this is how you stick something in here.

There's other actions. Like, if you want to view a document, you will stick something.view in the Action field. This'll tell the receiving component that you just want to view this object instead of dialing the phone number that's in the object, perhaps. And finally, the data is basically an arbitrary URI or URL for the data that you want to send along with this message. So it could be something like a telephone, colon, some digits to dial the phone number. It could be an actual HTTP URL that you want to view or open. It might be any other applications you find URI as well. So this is how you send these messages.

And the way these messages are actually routed through the system is with the help of the Android runtime itself that sits underneath all these applications. So you can think of the Android runtime as being somewhere between the applications and the kernel. It's not quite correct, but maybe let's try to draw some picture to clarify what the architecture of this thing looks like.

So you have one application that's running on Android. Perhaps you have another

application. These are all boxes that are basically these guys-- a separate application with a bunch of components internally. Of course, these guys are all processes running on top of the Linux kernel. So that's providing some degree of isolation between the applications.

And then there's what the paper calls the reference monitor. And this guy is going to mediate all the intent-level interactions between the different apps here. So if App 1 wants to send a message to App 2, they actually are going to send a message to the reference monitor first. So this is how you send all intents in Android, is that you create one of these intent messages. And you basically send it over some pipe to this reference monitor.

So Android basically has its own implementation of pipes for sending these kinds of intents, called binder. And every Android application, by convention, is going to open a binder connection over to the reference monitor, so the reference monitor can get intents from this application as well as send messages over to this application.

So in our case, if App 1 writes an intent for App 2 to the reference monitor, the reference monitor is going to figure out where this intent should go and relay it over here to App 2. So Application 2 can maybe start an activity, or receive a message, or do a SQL query, et cetera, for App 1. Does that make sense, roughly, in terms of what's going on? Yeah, question.

AUDIENCE: Does the label checking happen once it gets to that, or does the reference monitor?

PROFESSOR: Ah, yeah, yeah, yeah. So the reference monitor, hugely, importantly, is in charge of checking all the permissions that are represented by these labels. So you could imagine different things going on in terms of checking the permissions in the apps themselves. So why do these guys actually do the checking in the reference monitor instead of in the applications? Would it make sense to do the checking in the app? Suppose we stuck the label checks into App 1. Would that be reasonable? Yeah?

AUDIENCE: Well, that seems like a bad idea, because if someone compromises the

[INAUDIBLE] and is able to its behavior and get past the checking.

PROFESSOR: Yeah. Right. So you probably don't want to stick it in the sender, because you don't really want to trust the sender. So if you install a particular malicious application, if you want to be able to handle, that application isn't going to be guaranteed to do the correct checks for us. So that seems a little unfortunate. So we're not allowed to do the checks on the sender site.

What about doing the checks on the receive side in App 2? What about that? Yeah?

AUDIENCE: You could, but it would need crypto, and you would need a PKI. So it would be much more [INAUDIBLE].

PROFESSOR: OK. So you're thinking this would have to be crypto and have to have a PKI involved. So I'm not sure you have to have crypto, because the kernel can tell you exactly where things are coming from. So you could still have the reference monitor telling you, oh, this is coming from App 1. So you don't really need the PKI in that sense. It doesn't have to be crypto-related. I think you need crypto generally when you're talking over the network, when there's nothing common that you can trust. Here, I think it's not so much about crypto. Any other reasons why you would actually maybe want it in that reference monitor? Yeah?

AUDIENCE: You might want to shift the burden away from the developer, who ends up making a lot of mistakes [INAUDIBLE].

PROFESSOR: Right. I think a huge part of it is app bugs. If you don't really want silly bugs that the application developer makes to compromise the security. So to the extent possible, I think you want to factor out common functionality into code that the developer doesn't have to worry about so much, or doesn't even have a chance of screwing up. So this seems like partially a good reason for sticking to the reference monitor. Yeah?

AUDIENCE: It could also be because you want to minimize the trusted surface of the entire system. So you want to make the reference monitor so small that it can actually be [? orbited ?] on its own, or have some component that deals with the actual label

checking that can be checked on its own.

PROFESSOR: Right. Yeah. So that seems like a financially reasonable plan as well, because the security of the role system depends on the reference monitor being correct. Well, you could see this going either way. Actually putting the logic into the reference monitor makes the reference monitor bigger. So you could make the RM smaller by delegating some work to the apps. Although, then there is some library you have to audit. So it's not exactly clear.

I think the one other example I came up with for why to do this-- well, I guess there was two things. One is just simplicity. I think it's easier to do the checks all in one place in many ways. So you can sort of-- as you were saying, that you could really look at this and say, oh, yup, the checks are being done. They're always being done on every message. So that's convincing or good from a software engineering perspective, perhaps.

Another thing is that these intents have two addressing modes. In particular, in the paper, they describe what are called implicit and explicit intents. So explicit intents are ones where you specify some component. And you actually say it has to go to this particular component. So for these explicit intents, it's actually totally fine to do the checking on the receiver side, because well, you know where you're going to send it. You can send it there. And if it doesn't want to allow you to send the message, it'll drop it on the floor or reject it somehow.

But then there's also implicit intents in Android, where you don't know, as a sender, exactly which application you want to receive your message. So this might happen if you just want to view a picture or you want to dial a phone number, but you don't actually know which phone dialer the user has installed. Maybe it has a Google Voice, Voice-over IP dialer, or Skype, or who knows what. So in those case, these implicit intents actually skip the component name and just say, I want this action to be handled with this data by some application out there.

And in this case, it's the job of the reference monitor to find an application that's suitable for handling that kind of message-- dialing a phone number, viewing a PDF,

or a JPEG image, or what have you. And in that case, the reference monitor can actually take permissions into account when choosing a suitable application. So it might be that there are some very sensitive PDF viewer application you have installed, and it's capable of viewing PDFs, but you don't want it accessible to most apps. So maybe the permissions on it don't allow that application to receive PDF View messages from the rest of the system.

So in this case, the reference monitor will look at this and say, well, you're not allowed to send your request there, but maybe another application is willing to handle your request. So this sort of simplifies the user interface or user interaction here by matching an available system, including and considering the permissions that are available to the sender. That make sense? Does that make sense? Any questions? Yeah.

AUDIENCE: Would the reference monitor ever be a bottleneck?

PROFESSOR: It could be, yeah. So a lot of these messages are sent through the reference monitor. And I don't know whether it's actually currently multi-thread or not. You probably could make it multi-thread. I think the logic it's implementing doesn't involve maintaining a lot of shared state. So you could probably process many of intent messages in parallel if need be. I suspect that you could probably avoid it being a bottleneck.

For bulk things, Android has an RPC mechanism that the paper talks about, where if you want to send a lot of operations to another application, you actually send what's called a bind intent to the reference monitor, saying, I want a direct connection to this application. And if you send a bind intent to the reference monitor and it [? forwards the app ?] to this app, then you're going to establish this sort of bound channel between these two apps and then send lots of messages directly [INAUDIBLE]. So if some application is worried about an interface which is performance-critical, they'll probably do this. Yeah, question?

AUDIENCE: Why does the [INAUDIBLE]? Because every single label needs to match [INAUDIBLE].

PROFESSOR: Ah. So here, it's not the case that you get direct access inside of App 2. It's not that you can directly manipulate all the stuff in the address space or the objects of App 2. You just get a channel that the other application is willing to look at messages of and do something sensible with these messages. So it's up to the Application 2 here to look at these messages and do something sensible with them, not to allow arbitrary code execution or arbitrary access.

But in this application, I think they have two operations where you can add a new friend or enable or disable tracking through this interface. So there's well-defined messages that you define. And you're going to implement probably one of these [? surface ?] components that's responsible for taking a message, and checking that it's sensible, and executing that operation. Question?

AUDIENCE: Well, so I guess intents are usually human-initiated, right?

PROFESSOR: Oftentimes yeah.

AUDIENCE: Yeah. And humans are pretty slow. So it's unlikely that the reference monitor is going to be any kind of bottleneck.

PROFESSOR: Yes, that's probably true. Yeah. It depends on exactly, yeah, I guess how you're using intents. It's a little bit of a bummer. In the paper, they say that the permissions add [? buying ?] time are checked by the reference monitor. But the permissions on individual RPC calls between these applications are not checked by the reference monitor, because you have this direct channel between two applications.

So presumably, actually, I don't know exactly why they chose to do it this way. Perhaps it's to get away from having a reference monitor be any kind of bottleneck for [? handy ?] communication. But it means that the permissions for individual RPC operations between the applications have to be done in software inside of the application logic, which is a little bit unfortunate if we want to avoid the application developer making these kinds of mistakes and maybe forgetting to check the permissions on some RPC calls.

So in some ways, if you are purely worried about security, it might have been nicer to forward all the RPCs through the reference monitor as well, because then the reference monitor is going to make sure that it checks permissions on every RPC call instead of just at the time you establish a channel for future RPC calls between two apps. That make sense? All right. OK.

So let's try to figure out-- one interesting thing [? to do ?] to try to contrast is, why did these guys-- before we dive into a little bit more details, why did these guys design a whole new app model for Android applications as opposed to-- we've seen already. There's desktop applications, there's web applications. Why did these guys invent a whole new way of writing software?

Because in some ways, this is confusing for the developer, because I am used to writing my nice little C program with a main function. I look at this and say, well, what the hell? I mean, what am I going to do with-- I have to define four kinds of components, and I have to send intents instead of just having a C struct and writing [? straight line ?] code.

So what are the pros or cons of existing app models? So we have, I guess, desktop apps and web apps. Why do we need a third column, so to say? Because what are the nice things about these guys? Yeah?

AUDIENCE: Well, the model has completely changed now, right? Because I think on desktop apps, you don't put as much trust in the developers as you put in mobile apps. And you have a bunch of more users that are less [? inexperienced ?] than the desktop users who end up having a bunch of apps that really want to [? isolate ?] from each other.

PROFESSOR: It could be. So you think in the desktop case, we don't have to trust the developers as much?

AUDIENCE: Of course you do. But it seems that there's always like your son or your cousin that you take care of their desktop if it goes bad.

PROFESSOR: [LAUGHS]

AUDIENCE: But with your phone, there's a different problem altogether.

PROFESSOR: I guess it's cool that the phones don't need a cousin to take care of them. So that's great, right? But in terms of security, one thing on the desktop apps is that you can't install-- or it's really hard to install new apps or maybe-- well, probably not quite exactly true, because you can always click an executable and install an app on a desktop case.

But I guess people don't install apps regularly sort of tend to maybe-- because it depends on the usage model of a desktop app. But typically, you have a fixed set of software you are running as opposed to on a web app, one cool thing is that it's very easy to run new apps. You just visit a website, and there's nothing really you have to do other than click on a link. And off you are on some new site running some new Application. So that's a pretty nice property of web apps.

One bummer about desktop apps is that there's actually no isolation at all between applications. That's perhaps somewhat related to the fact that it's hard to install an app, because you really are trusting it fully with all the data on your machine when you're installing it. There is no isolation between one app that you install in your laptop and probably any other thing running there, or any of the data that you are storing on that computer.

Whereas in the web app case, there's some reasonable isolation. As long as you believe the same origin policy is correctly [INAUDIBLE] by the browser, then you're in pretty good shape. It's reasonably safe to probably go to some arbitrary website and start using their application. It's not going to tamper with other sites that you have open in your browser, assuming they don't exploit some browser bugs.

So far, it looks like the web apps are the winning plan. They're easy to use, they have isolation. Why don't these guys use web apps for Android? Yeah?

AUDIENCE: So web apps are starting to become like an operating system in themselves, right? So you have Firefox OX, which is basically just a web mobile OS.

PROFESSOR: Right. OK. So you're arguing that actually these guys were mistaken. They shouldn't have built a new Android stack. They should have just done a giant web browser as your phone.

AUDIENCE: Well, at least Mozilla has shown that it's possible.

PROFESSOR: Right. OK, fair enough. So it's at least more reasonable to go with the web apps route rather than the desktop route, at least for phone. Yeah?

AUDIENCE: [INAUDIBLE] phone call from a web apps, you need [? an entire new ?] API for the web app interface with the phone.

PROFESSOR: Right. So the one limitation that might be fixable, of course, but is still there is maybe there's no APIs for some of the devices. This is increasingly becoming less so. Like, for a camera or for a GPS, these are slowly being added to the web case. But there is probably not quite an API for accessing your phone yet, or sending an SMS message, and things like that.

Another bummer in web apps is there is actually limited sharing that you can do. So we were just talking about implicit intents in Android, where you could just say, well, I want to view this JPEG picture, but who knows what application's going to open it? Or I want to view this PDF file. Or I want to share this picture I just took with my camera with a friend through email, but I don't know what email application you're using.

Let's just ask the reference monitor to find me some email program that's going to send this picture out. So it's something you can actually do in Android. But it's kind of hard to do in a web app case, because every interaction, you have to refer to a particular URL. So if you don't know what PDF viewer someone is using, you might not know what URL to open to ask it to view the PDF [? potentially. ?] Question?

AUDIENCE: [INAUDIBLE]. But JavaScript is very hard to read [INAUDIBLE].

PROFESSOR: Yeah. So one bummer is that yeah, this is all JavaScript. [LAUGHS] So it's potentially unfortunate. But it's maybe not as good in terms of performance. Maybe

it's hard to understand what it's doing. It might be hard to compile efficiently, et cetera. Yeah.

Because one nice thing about the desktop [INAUDIBLE] is that sharing is easy. So one side effect of having all your files be accessible in every app is that, well, you just share. It's very easy to access any data you have. And in some ways, if you really want a lot of sharing, this is great. Like, I probably could-- it might be a little bit tricky to implement some things on Android that are easy to do in the desktop case. So in the desktop case, if I want to compile a piece of software, I'm going to run [? Make. ?] That's going to run GCC and maybe other programs.

And they're all collaborating on the same C source code in the single directory. They're all building it. It might be a little bit trickier to do in the Android case, where data is kind of associated with a primary application, but storing it in a [? quantum ?] provider. So it might be a little bit tricky to have an Android world where I have my source code stored somewhere, and then I install a C compiler, and [? a Make ?] program, and an assembler, and other things. And they all sort of work together. It's a little bit harder to do. You probably could do it somehow.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So you could probably work around it in some ways. So I think Android is certainly general purpose enough to let you somehow do it. But it's probably not quite as natural to do this in Android as it would be in a desktop case. Not that I'm arguing the desktop case. But it's not quite as secure. But yeah. Any other comments? Yeah?

AUDIENCE: Another thing might be the case that we're optimizing for different web apps [INAUDIBLE] constrained. I'm not sure what [INAUDIBLE] tend to be constrained by. But [? web apps ?] tend to be constrained by both RAM and processing. And much more so than either desktop or web apps, certainly.

PROFESSOR: Yeah. So it might be that solid engineering decisions around how to optimize these things are going to be different. I guess one unfortunate thing about web apps, at

least at the time these guys were designing Android, is that it was difficult to run a web app offline. If your phone doesn't have strong enough cellphone reception, then it might be hard for you to run the application if some parts of it fell out of the cache. I think slowly, as you were pointing out, the web app world is catching up to Android.

But many of these limitations are getting fixed or sort of improved. So it might be that these days, web apps are a reasonable model for starting a new phone platform. But five years ago, this Android world was necessary, because the web apps weren't quite there. Probably not quite there still yet, but nowadays, it might be easier to push the web apps all the way to where Android is rather than start from scratch. All right.

So I guess we can still talk about what Android did even though maybe today you wouldn't have done it the same way. But I guess in terms of isolation, we can start talking maybe about security a little bit more. Android relies on the Linux kernel, as I mentioned, to isolate these apps from one another. So what happens is that the Android framework actually sets user IDs so that this application has perhaps UID 1,001. This application runs as UID 1,002. And the reference monitor is basically [INAUDIBLE]. So I think it might run as UID 0, although I forget the detail. But I think it does run as UID 0 on Android.

So the Linux kernel is largely responsible for keeping the apps separate from one another. And mostly, interactions between user IDs happens through these intents. And then there's a little bit of details in terms of things that the Linux kernel actually enforces in terms of which UID is allowed to do which operation as well. So we'll talk about that in a second.

One interesting question is why did these guys choose Java? Like, what's the role of Java in Android? Why is Java there at all? Is it enforcing anything? Yeah.

AUDIENCE: I think it enforces [? text messaging ?] and [? downstreaming. ?]

PROFESSOR: Right. OK. So what do we get out of it? Is it like for security, for probability? One

other way to think of it is suppose we took away Java and made all the apps written in C, or, like, Assembly, or don't require anything at all for that matter. Would anything break? Yeah.

AUDIENCE: You have vulnerabilities [INAUDIBLE] override these important values.

PROFESSOR: Uh-huh. Yeah, it could be. So like an app could have some buffer overflow in it. So how bad would that be?

AUDIENCE: It could override with permissions.

PROFESSOR: Which permissions?

AUDIENCE: Like the latency.

PROFESSOR: Yeah. Well, actually, let's see. So as we were talking about, the reference monitor is the thing that does all the label checking for us. And the reference monitor actually, in Android, stores a list of all the installed applications along with the labels that correspond to all those applications. So it's probably true that you don't want any kinds of bugs in the reference monitor regardless of what language it's written in.

So having the reference monitor being written in a type-safe language is probably a good thing in general. And I like Java. I can sense that it's a type-safe language and has all these nice properties. But if an application were to be written in C and had a buffer overflow, it wouldn't be able to corrupt directly the labels stored in the reference monitor here. So that wouldn't be as big of a deal. Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: It could be. So why would that be helpful?

AUDIENCE: There's some system [INAUDIBLE] that it can actually override C?

PROFESSOR: Yeah. So in principle, it might be nice to avoid the applications talking directly to the Linux kernel. [INAUDIBLE] in Android, this is not the case. So actually, Android applications can make arbitrary system calls if they want to. And in fact, applications

I should say can shift with arbitrary components written in C or Assembly for performance reasons. So some games [? shift ?] with the computationally intensive parts written in C. And they sort of talk to it from Java as well.

AUDIENCE: So I guess part of it is being able to use all of the stuff that's been written for Java. They wanted to make it easy to build applications for developers. And one of the easy ways of doing that is being able to take advantage of the massive Java libraries out there.

PROFESSOR: Mm-hm. Yeah. So I think one big reason for using Java is the usability. They wanted to-- I think Java has little to do with security and mostly with programmability, ease of development.

One other thing that I think is going on is that-- well, to contrast with iPhones, for example. iPhones also really want ease of development. But they use Objective C, which has buffer overflows if you try hard enough. And it is specific to a particular hardware. It doesn't maybe have all the same libraries. I think the main reason why the Android guys went with Java is probably the fact that they weren't manufacturing devices at the same time. So the iPhones knew exactly it was going to be this ARM processor running their phone. So they could compile to ARM and be done with it. And it's more efficient, because battery matters a lot on a phone.

And the Android guys use Java, which probably is slightly less power-efficient or CPU-efficient, because it involves this JRE, et cetera. But the cool thing about it, it's actually portable between architectures. So if you have a phone that has a MIPS processor, or an ARM processor, or an x86 processor, the Java application can be run on all three of these kinds of devices. And the Android guys wanted their platform to be usable on any kind of hardware or phone. So that's probably one big reason for why they use Java in all these cases, and probably less so in terms of [? staying ?] security consideration for Java.

So in fact, it turns out that the Java runtime doesn't really provide any security purpose for the application and is just sort of a nice convenience thing, as well as providing all the abstractions that the developer should think in terms of. But in

terms of isolation, it's mostly up to the kernel and the reference monitor to keep these guys in line. That make sense? Any questions?

AUDIENCE: Doesn't the ease of development also kind of translate into some security [INAUDIBLE]? Because if you write that reference monitor in C, I can see much more ways to make a mistake.

PROFESSOR: Yeah. So you're absolutely right. Actually, I shouldn't have said that ease of development has nothing to do with security. This is completely silly, because you want to make it as easy as possible to write correct code. And it's all about covering mistakes. So in some ways, having a system where it's easy to write correct code is the most important security consideration to have. So in some sense, you're right, that it avoids the bugs.

But you don't want your application written in C. Or I don't know why Apple has Objective C. It's actually a little bit of a problem in this regard, because you could easily have buffer overflows in your application. And if that application matters a lot, then it's vulnerable potentially. Not with respect to compromising other applications, but you're [? all like, ?] bank app, I don't want that bank app written in C.

AUDIENCE: Right. Yeah.

AUDIENCE: Is the reference monitor written in Java or C?

PROFESSOR: So in Android, the reference monitor is largely written in Java, yes. There are some native hooks that it needs in order to be able to talk to these intent interface-- pipes, basically, to talk to the binder, they need some native code. They need some native code to spawn these applications in the first place, et cetera. But by and large, most of the logic is written in Java. So it's actually a reasonably safe plan, I think. Any other questions about this? All right.

So I guess let's try to figure out what are these application UIDs used for other than to keep applications separate from another in terms of their processes? So I guess the main thing that applications need to use the UID for or that we need to somehow support is the ability to share access to shared resources and shared

data in the system.

And we already saw one mechanism for doing it, which is to send intents to the reference monitor. But there's a bunch of things that, in Android, are not done through intents to the reference monitor. And they have to do probably with performance, or why-- basically, why isn't everything sent through intents is probably because there are some performance considerations. You don't want to invoke the reference monitor in every single thing you do in the system.

And there's a couple of things that an Android are like this. The simplest one is probably network access. So if you want to talk to the internet, you just open a socket, very much like you would on a standard Linux application today. The application can just ask the kernel, I want a socket, I want to connect to this machine. Go for it. So network access happens to work this way.

Access to removable storage. So if you have an SD card in your phone, that also directly goes through the kernel. Or more generally, any kind of file system access or direct access to the file system, at least, goes directly through the kernel, because there is already a file system there. And you want to avoid probably [INAUDIBLE] performance overheads on that.

And also, for most devices that are hardware, Android allows the application to directly talk to the device instead of mediating the access through the reference monitor. So this is things like probably your camera, your GPS device, compass, et cetera. And these guys just show up in Android in Linux as something like /dev/camera. And this is just a Linux device you could open, and get the camera data out, and control the camera in whatever ways you want, et cetera.

And the cool thing is that if you want to do some specialized things to this device, you're not restricted to what Java allows you to do. You can always write C code or even Assembly that directly talks to the kernel and performs the necessary operations on this Linux device, making arbitrary system calls. And you could wrap this up in a Java native interface to expose it to the rest of your Java-based application here. Yeah.

AUDIENCE: But you still have the checks in these calls, right? So when you open a socket, someone has to check if they're allowed to open a socket [INAUDIBLE].

PROFESSOR: Yeah. So this is an interesting thing. Now these things are outside our intent-based model. So how are we going to protect these guys? So this sort of boils down to doing something very similar to what you guys did in Lab 2, which is you [INAUDIBLE] you want to enforce it using UIDs. Basically, the Android framework is responsible for carefully orchestrating the UIDs and GIDs of the applications and of these things to enforce whatever policy was specified in terms of labels.

So the way this works out is that for every one of these resources, there is a pre-defined label string that defines the privilege to access this resource. So for this network access thing, for example, I think there's a string, something like android.permissions.INTERNET. So this is a label that an application can ask for.

And if an application has this label in its set of privileges, then it should be able to access the network. And the way this is enforced-- so the label is the policy side of this. How do you specify what should happen? And the enforcement mechanism is a small change to the Linux kernel in Android, where in order to make any network-related system calls, you have to be a member of some magic group. This is not at all how things work in Linux traditionally.

But in Android, there is some magic group [INAUDIBLE] I think it's GID 3003. And the kernel has this number hard-coded in it. And if the process has this group ID in its group list, then it's allowed to use a network-related system call. And if it doesn't have this group ID in its group list, then it's not allowed to make any network-related system calls at all. So this is how Android is able to translate these label-- sort of maintain a single, coherent policy system that's in terms of these label screens, but enforce it in different ways. So sometimes it gets enforced by the reference monitor. And other times, it gets enforced by setting GIDs or UIDs appropriately.

The same thing actually happens with SD cards. There's another GID that corresponds to having access with the SD card. And there's a string that gets

translated into this GID, effectively. And same for the file system. I guess in the file system, things are a little bit trickier. I guess the SD card is in the file system and has a specific GID for accessing the entire SD card.

Another sort of other files in the phone's file system-- there, the policy isn't so much controlled by labels, but rather by Android's design, which is that each application has a private directory that it can use to store whatever files it wants. And in particular, the content provider, the SQL database that you're going to use as an application, is stored in your private directory. And the policy is only the application's UID can access that private directory, and no one else can access it directly.

And then for devices, there's also a plan very similar to network access, where there's a permission string for accessing GPS, the camera, et cetera. And for each one of these, there's a GID that is used in the permissions on that device. So for example, dev/camera is owned by some magic GID. And any application that should have access to that camera has that GID in its [INAUDIBLE]. All make sense? It was hopefully fairly similar to what you guys did in Lab 2. Well, not the label part, but the using UIDs and GIDs to get somewhere [INAUDIBLE]. Make sense? All right.

So one interesting question is why do these guys have such a course-grained plan for handling the SD card? Why don't they have different apps act [? with ?] different parts of the SD card? I probably have lots of stuff in my SD card that I don't want applications to-- you know, to have access to all of it. Make sense? Yeah.

AUDIENCE: Maybe it has something to do with [INAUDIBLE] that a user has to approve those permissions. And the less of them you have, the more likely it is to actually [INAUDIBLE].

AUDIENCE: Could be. Well, that's definitely a problem in general in Android, and we'll get to it in a minute. But I think for the SD card, it's actually a slightly different concern. It has to do with the fact that you want these SD cards to inter-operate with the rest of the world. So this is like one of the places where Android isn't free to make arbitrary decisions, because you want these SD cards to have a standard file system, namely FAT, because that's the de facto standard for storing data on an SD card these

days. And as a result, you can't have a sophisticated file system there that stores permissions for each file. And as a result, it's going to be difficult for you to somehow segregate these files and give different apps different access. Yeah?

AUDIENCE: Do you still think of [INAUDIBLE]?

PROFESSOR: Yeah. So it might be that you could give each application a different subdirectory on the SD card to have access to. But then it would mostly be a way to give each application additional storage and not the ability for an application to read existing content from your SD card. Because the existing content might be in some other directory. So you're right. It might be that you could have more fine-grained things, like well, you want access to the entire SD card addressed to a per app directory there. And that probably has to do with not overwhelming the user with too many choices in terms of these permissions. But yeah, you're right, you could probably do some combination of these two. All right.

So I guess one interesting [? thing to ?] talk about is how do we decide whether an application should have a particular set of labels that it should have access to in this label set? So where do these guys actually come from? Like, who decides that this application should have DIAL PERM and INTERNET and FRIEND VIEW permissions in Android? Yeah.

AUDIENCE: The developer [INAUDIBLE].

PROFESSOR: Yeah, it's a little bit of a, you know, well, I guess an upfront system, where the developer has to, first of all, enumerate all the things it'll ever need in the future or as their applications going to run. And then the user is responsible for looking over this list and deciding whether it's OK, whether they should allow these application to be installed or not.

And this way, the user is still an important part of the system in terms of security, because the user could almost always approve any set of permissions you want or that the developer wants. So in many ways, it's actually quite flexible, because unlike something like iOS-- where it's actually difficult to share between apps, or it's

hard, for example, for a third-party iPhone application, I think, to access arbitrary other components of the phone, and dial a phone, or send an SMS message, or find a JPEG viewer, et cetera-- here, it's quite flexible.

But sort of the cost to it is that you have to get the developer or the user to check that the developer is asking for a sensible set of permissions or the user trusts this developer with these privileges. So it's a little bit unfortunate. It's probably one of the biggest, actually, security problems in practice with Android, which is that users are quite willing to give away these permissions if they really want an application. I search for some application, and I click Install. And if it doesn't look like a particularly long list and there's nothing that pops out right away, I'll probably click OK. And I might not spend the time to really understand whether these permissions are necessary.

The other slight bummer is that [INAUDIBLE] the time this paper was written, Android decided that the user had only two choices-- either install the app or not install the app, which is a little bit of a binary choice. And the user is presumably going to just say, well, yeah, I want the app. What else am I going to do? I need to do this thing.

And I think more recently, I think some version of Android, I think 4.3, introduced a more fine-grain scheme, where the user is actually allowed to pick and choose from these permissions where-- well, you are still presented with a list of permissions that the developer wants for their application to have. But the user can now, in a more fine-grained manner, remove some of these permissions from the application. And it's unclear how an average user is supposed to use this, because it's probably quite hard to go through this list and make decisions about this. But at least the API is starting to show up there. I haven't seen any significant uses of it so far. But it might be nice. All right. [INAUDIBLE]. Yeah?

AUDIENCE: [INAUDIBLE]. it just lets you take away.

PROFESSOR: Sorry?

AUDIENCE: The [INAUDIBLE] just lets you take away [INAUDIBLE].

PROFESSOR: Yeah. So I think that's basically what this new version of Android lets you do, which is you can-- instead of taking away-- well, it's not labelled strings. It's descriptions of these permissions. But you can actually now-- something, I think, called Android Permission Manager lets you, for every app, list all the labeled strings that the app has permission for. And you could, I think, remove these things on an individual basis if you feel strongly about them. I don't know how many users [INAUDIBLE]. Yeah, question?

AUDIENCE: Whenever the labels [? don't match up, ?] does it hard-fail, or is it just that doesn't work [INAUDIBLE]?

PROFESSOR: Well, I think it depends on exactly what the application is trying to do that is going to require that label. So if the application is going to send an intent and sending that intent requires a particular label like DIAL PERM, well, it might be that you're going to send the intent to the reference monitor. And the reference monitor is going to say, well, there's no application that is willing to accept your message. So maybe then it's up to the application to do something sensible in response.

Another possibility is that maybe it's a network access, and you don't have access to that, and you're going to make a socket system call, or you're going to say, connect to this IP address. And the kernel says, E PERM, you can't do that. And who knows what the application is going to do in that case? Maybe it'll throw a null pointer exception somehow.

So one argument against doing this is that Android applications, at least originally, weren't written to expect some of their accesses to fail, because they were told, the manifest is all or nothing. Either the user approves your app, or they don't. So application developers, perhaps rightly so, wrote code that perhaps crashes or does something not unexpected if some access provision fails. So it might be that by taking away permissions, you're going to cause the app to crash if it needs that access.

So it's not like, well, you know, you have this nice app, and it needs access to the camera. But if you turn it off, it'll just put on some dummy picture instead. Maybe it'll just crash instead. So it's not great. You might imagine much more sophisticated systems which, if you take away some of the access to a camera, are going to provide a fake camera that just has a black screen all the time. So this is not what Android does. But you could imagine [? alternate ?] situations where this might happen. All right. Any other questions here? All right.

So we've looked a little bit at sort of where these strings come from in the label of an Android app. But who defines these strings? Like, where does the meaning of a string come from? You can list all kinds of strings in your manifest file. But how do you decide which strings matter? Where does this string INTERNET or FRIEND VIEW come from in the first place? Who gives it meaning in the system? Any ideas?

Well, I think it's mostly-- the way to think of it is that almost none of these strings should be magic or pre-defined ahead of time. Almost all these strings are basically contracts between two applications so one application is willing to export something under the protection of some label string. And another application wants to ask for that permission to talk to the application that defines this component.

So these labels are typically-- you can think of them as being defined by the application that provides some protected service. So if you have this DIAL PERM permission, it's presumably something that has to be defined by an application that defines what it means to dial a phone number. So probably the dialer application in your phone, that's the thing that defines the string and says that, yeah, this thing, DIAL PERM, exists, and my components are going to be protected by it.

And then other applications that want to interact with this guy, with the dialer, can now request this DIAL PERM permission for themselves. And of course, there are some built-in things that we looked at here, like the internet permission, the camera permission, et cetera. But you can sort of think of them as the Android framework being the initial application that is in charge of providing access to this resource and defining the string that's going to protect the resource as well.

What does it mean? What's sort of associated with a label string in Android other than the fact that the string goes into the application's label here when they want to ask for this permission and in the component label as well? So there's a couple of things that is associated with the label in Android. So a label, in addition to a string, also has a couple of interesting properties.

So there's a type of a label. And there's, at least in sort of a modern Android, there's three types you might care about. There's a normal type of a label, or permission label. There's dangerous and signature. And the application that defines this permission in the first place gets to choose the type or all these other fields for a label that we're going to talk about in a second.

So what's the point of a type of a label? Why do labels have types in Android? Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, I guess so. Yeah. So why not make everything dangerous, I guess, at that level? Well, I guess, yeah. So maybe the interesting thing is like, what are the semantics of these types? So if something is dangerous, then you are right. It actually warns the users when you're installing an application, and the application asks for access to a particular label that's of type Dangerous. Then the user has to look at this message and say, yes, I'm willing to give this dangerous permission to this new application.

For normal-type labels, applications can ask for them, and the user doesn't get prompted if the application requests this normal type of permission. Is this silly? What's the point of a permission if everyone just gets it? Is there a reason why we should have this? Like, one example of a normal permission in Android is setting your wallpaper. So if you have an application that's going to set your wallpaper, I can, as an application developer, say in my manifest that I want to set your wallpaper. And if you click Install, it's going to say, well, there's nothing interesting going on here. You don't need to give it any permissions. Yeah.

AUDIENCE: Well, these permissions usually require you to [INAUDIBLE], right? So if an

application wants to change your desktop wallpaper, it will ask you. The system will ask you, do you want to change your wallpaper?

PROFESSOR: Nope.

AUDIENCE: No?

PROFESSOR: No, it'll just change the wallpaper. It's access to this API call to change the wallpaper. If I have this permission, I can make this API call. Yeah?

AUDIENCE: Maybe the application developer wants to make sure they don't do this accidentally?

PROFESSOR: Yes, I think that's one reason why you might want to have these permissions, is to help the application developer do the right thing. So if you worry that your application might accidentally do something or it might have bugs in it that others will exploit, knowing that there's some set of permissions that you do or don't have prevents your application from being abused in these ways. So if you have a benign application that never needs to set the wallpaper, you probably don't want to ask for this permission, because if a VLAN gets compromised, then it will be better for the user, on whose phone your application's installed. It's sort of a least privilege property.

I guess another thing is that it maybe allows some sort of auditing, both from a developer standpoint, where they can look and see, well, what are the things I should be concerned about here? And as well as from a user's perspective. If your phone is flickering with a wallpaper change every second, you can go and see who has this permission. Even though I didn't have to approve it, I can at least go and check who is potentially doing this right now.

So these normal permissions are kind of like a good security measure-- or probably more of a good auditing measure. And [? aren't ?] generally used for really interesting things like [INAUDIBLE] data or accessing things like cameras or things that cost money.

So there's also this third thing, this signature permission. So one interesting thing in

Android is that you can define a permission that is only accessible to applications that are signed with the same developer key as the application that defined the permission in the first place. So if I have, I guess in the FRIEND VIEW example in the paper, if the friend tracker defined some permission with this signature type, then only other applications signed by the same developer key are going to be able to get this signature permission. What's the point of this thing? Why not just make them dangerous or, I don't know, something else? Why do we need a third type? Any reason? Yeah?

AUDIENCE: Operations [INAUDIBLE] same developer?

PROFESSOR: Yeah. So it might be that this developer has some internal APIs that they don't actually mean to expose to the outside world. But they just want to couple their own applications to each other. So maybe Facebook, hypothetically, could write multiple applications. They might have one application that's pre-fetching content from Facebook servers, another application that fuses, another application that tracks your location. And they want all these components to interact with one another. And they can define the signature permission to do this.

And presumably, one reason why you might not want to do this-- might not want to tag this permission as a dangerous permission is for mostly the same reasons as the [INAUDIBLE] was talking about, which is that if you really know who should be allowed to get this permission, you don't want to allow the user to screw this up. So the user could always be tricked by someone-- or could be tricked by someone into accepting a malicious application that asks for some permission that's really dangerous. But [? it doesn't ?] need to be dangerous, you could just define this signature. And the user doesn't even have a choice of giving away some application's internal privileges. That's one nice thing about this permission type. Make sense?

So there's other stuff associated with the label that mostly has to do with describing the permission to the user. So there's some sort of a description here that is the sort of English-level explanation of what this permission entails. And it's this description

that pops up when you're asked to install a new application. So the Android framework will look through all the label strings in the manifest of the application you're about to install, and display to the user the descriptions for all those labeled strings, saying, OK, you're about to give away the privileges to dial your phone, or you're about to give this application the permission to send SMS messages on your behalf, et cetera. That make sense? All right.

So one interesting question is, what happens if a malicious application defines a label for some other app? These labels are just free-form strings. So what happens if you're a malicious application and you say, oh, I have this new, great permission. It's called DIAL PERM. And [INAUDIBLE] dangerous. And the description does nothing. Is it good or [INAUDIBLE]? Yeah?

AUDIENCE: So [INAUDIBLE] domains [INAUDIBLE].

PROFESSOR: Yeah. So you hope so. Unfortunately, it's not actually enforced. So by convention, all these permission strings should have Java-style reversed domain names. But there's no strict association between the labels that an application defines and the application's own Java-style name. And for that matter, there's nothing that enforces that an application's Java-style name be tied to anything, because we have no way of knowing whether the public [INAUDIBLE] developer signing a particular application corresponds to com.google.something or edu.mit.something.

So in fact, one slight weakness in Android that was there at least when I checked a while ago-- it probably is still there-- is that the label definitions are sort of first come, first served. So when you first install an application, it defines a particular label, you get to decide what type that label string is and what is the description of this label string. So for system permissions, this is probably not a big problem, because the system permissions, or the ones for built-in applications like [? compiler, ?] get defined first. But applications that come later are unable to redefine them. So at least the framework enforces that.

But certainly, one bummer is that if you install a malicious application first and then some important application later, the malicious application can potentially subvert

the labels used by the later well-meaning app. So in the paper's FRIEND VIEWER example, you could actually-- if you're a malicious developer, you could first trick the user into installing these applications defines this FRIEND [? VIEW ?] permission to be a normal permission with a description stream saying, oh, this does nothing interesting at all. And then the FRIEND VIEWER applet gets installed later. It can't redefine this label. It's already been defined. And consequently might be unable to prevent the user from giving away this FRIEND VIEW permission to other apps.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Well, in principle, the framework could do this. When I tried this, you don't get warned at all. So if you install an application that defines a label that's already been defined, it does nothing. It simply ignores the VIEW label definition and uses the old one. Perhaps that's the problem where it really goes wrong. You should at least say, well, I refuse to install this application, because it's defining a label that already exists.

AUDIENCE: Owned by this other application.

PROFESSOR: Yeah. And it's owned by a different key even. Yeah. So at least there's a chance to fix this potentially. I haven't seen this fixed yet, but maybe I've not been keeping track of it. Anyway. So that's kind of an interesting problem that you really have to keep track of these names and figure out who owns a name and getting that right is actually quite important.

So one interesting problem that shows up in Android has to do with these broadcast receivers or sort of sending messages between applications. So one interesting problem is when you're sending-- well, I guess I should first describe how these messages work with broadcast receivers. So broadcast receivers are used for one application being able to announce some event to every other application in the system. So as we saw before, intents typically go to a particular component, like view a JPEG image. But for some events, like, the system boots up. Or my friends are nearby. You might want to announce this to every application that cares. And this is what these broadcast receivers are for.

But you actually start worrying when you have these messages being sent between two applications, both in the broadcast receiver case and in other cases, you probably care about two things. You might want to authenticate where the message is coming from. So you want to know who sent this message. Can I trust them? And also, you want to potentially control where this message goes to-- who is able to receive this message.

And initially, it seems like an Android device didn't quite get these things quite correctly in several ways. In particular, the broadcast receivers-- well, if you are sending a broadcast message to all the other components in your system, I think in the initial version of Android, you just sent this message. And other applications could either subscribe or not subscribe to these messages.

So if you have a FRIEND VIEWER application that it'll subscribe to these messages by setting the right action or date or data time or MIME type in their Intent filter, but most applications could always subscribe to all broadcast events in the system. And you are able to watch everything that's going on on the phone, or everything that's being broadcast.

So the Android framework added an extra sort of argument for applications to be able to specify who should be able to see a broadcast message. So when you're sending a broadcast message, there's the obvious argument, which is the message you want to send, which is basically an intent. But then you can also specify an optional label that describes who should be able to receive this message. So instead of broadcasting to everyone in the system, you can say, well, only other applications that have a certain permission should be able to receive this broadcast message from me.

So this way, you could send out sensitive information, like the locations of your friends, in the paper's example, and make sure that only those applications that are allowed to see the list of your friends will actually get your broadcast message. So this is how, on Android, you can actually control who receives the message that you're sending out, at least in the broadcast case. Question? No. [? Sorry. ?]

How do you authenticate where a message is actually coming from? So suppose that in Android, you register for-- in the paper's example, you have your FRIEND VIEWER and you receive a message saying, yep, this friend is nearby. How do you know this actually came from the right component? Can you actually convince yourself of this? Yeah?

AUDIENCE: Using the kernel codes? Wouldn't you trust the kernel [INAUDIBLE]?

PROFESSOR: Potentially, yeah. Well, this binder thing is going to tell the reference monitor, here's where the intent came from. And then the reference monitor is going to forward this intent to your receiver application. And somewhere in there is the name of the applet that sent this guy. How should you check whether this is a reasonable app that should be sending these intents? Is there a way to do this in Android? I guess you're right. Strictly speaking, yeah, the source is always authenticated. You know exactly which app sent the message. But what do you do with a source name? How do you check whether it should be sending these Friend is Near messages? [INAUDIBLE]. Yeah?

AUDIENCE: [INAUDIBLE] label [INAUDIBLE].

PROFESSOR: Yeah. So one way to do it is to actually stick a label on the broadcast receiver. So one thing you could is say, well, the only people that are allowed to send a message to the broadcast receiver are people with the Friend Tracker maybe label. So if you stick such a label on your broadcast receiver, then you know that only messages sent from applications with this label are going to get to you through your broadcast receiver. That's one way to filter who's able to send messages to you is by restricting them by label.

So this works in many cases. Android also provides a more specific function that you can use. It's called Check Privilege. And you could sort of say-- or Check Sender Privilege, I think. And you could ask the framework whether the sender of the intent you're looking at has a certain label in its provisions. So this way, you can also reason about what privileges does the sender of a particular message have in

situations where maybe the framework doesn't provide quite the right mechanisms in the manifest to do this.

One place where this Check Sender Privilege function turns out to be particularly useful is in the case of this RPC interaction between two applications, where the reference monitor isn't actually involved in mediating the RPC-looking applications. But you still want to ask, is the application of [INAUDIBLE] privileged for this kind of operation? So this way, you can manually invoke-- or check what's in the manifest of an application. Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Well, so I guess-- [INAUDIBLE].

AUDIENCE: You can use the signature to enable it.

PROFESSOR: Yeah, so one thing that stops is presumably the [INAUDIBLE]. If the label is a signature type label, then that's going to stop it if it's not [INAUDIBLE].

AUDIENCE: That requires the same signature as--

PROFESSOR: That's right. Yeah, yeah. So that requires it be the same, exact developer. Another thing that stops it is that maybe the permission is labelled as Dangerous. So then the user is going to see this on their screen. When you install an app, it says, this app requires the permission of viewing your friends, or your friends' locations.

AUDIENCE: But that's taken from the description.

PROFESSOR: That's right.

AUDIENCE: So what's [INAUDIBLE] from just giving a different description?

PROFESSOR: So I should say, this description comes from the application that initially defines this label. So the label is initially-- so there's two things going on in the manifest. You can ask for access to existing labels. So I can say, well, this is an application. I want access to the FRIEND VIEW permission. And a separate thing you can do in the

manifest is say, I am defining a new kind of label. So then the first Friend Viewer application-- or the Friend Tracker application, presumably, is going to say, well, I have an application, it has these permission. But also, I'm defining a new kind of label. It's called this. It's type Dangerous, and it has this description.

So that's a place where you have to really get the-- this is that first come, first served problem. But as long as the application that should define the label is installed first, then you're in good shape. Any subsequent application is going to ask for the label just by its string name. And then the Android framework will fish out the description and the type from the application that first defined it. All right. Any other questions here? All right.

So I guess this gives you some sense of how Android works. So one cool thing about it is that you can actually get this manifest that, to a larger extent, describes the security properties of an application. So this is one thing that the developers of the Android framework were going for-- something called mandatory access control, where you can actually specify the security policy of an application separate from the application itself, and in fact, have that security policy be enforced by this reference monitor regardless of what that application itself is doing.

So it seems like kind of a nice property to be able to audit an application by looking at this manifest file. You can mostly think of it as a development nicety rather than a strict enforcement mechanism, because if an application really wants to get around its own manifest, it can probably do so. But it seems like a nice way of being able to understand what's going on with an application in terms of security without having to dive down into the Java code. Of course, it matters, but you can still get a high-level sense of what's going on in an application from the manifest.

One bummer, I guess, as we were talking about here, there are some situations where the Android framework turns out to be not quite expressive enough in the manifest. And you have to still write code that talks about security checks. It would be, in some ways, nice if there were no security checks at all in the code and all the security checks went to the manifest. But that would mean exposing things like all

the RPCs to the framework and so on. That's potentially a bit of a trade-off that these guys are making.

And I guess one other thing is it's actually kind of hard to change the manifest file after you design the system. So one perhaps surprising thing is that the Android framework hasn't changed in terms of security very much since it was released or since this paper was written five years ago, or six years ago now. Because once the application starts using this framework, it's hard to say some existing application's going to break. So you basically have to maintain backwards compatibility to a large extent. So you don't get a chance to sort of do it over again.

I guess one interesting thing that did happen to Android since this paper came out is that the Android guys borrowed an idea from Apple and are doing now much more server-side analysis of applications. So Apple, on the iPhone side, is pretty aggressive in terms of having the Apple server checking all the applications from developers for various guidelines, including security properties.

And the Android servers, or this Android market or whatever, now also does quite a bit of analysis of applications submitted by developers to make sure they are not malicious in some loose sense. So that's kind of a cool thing. And it is largely [? orthogonal ?] to this security architecture. So this security architecture works on your phone. But then for any other security problems, the server can evolve defenses over time as need be. And those tend to be more on the phishing side, where the human is being tricked into doing something rather than on exploiting some specific vulnerability in the kernel, perhaps, and so on. Make sense? Any other questions? All right.

So we'll see you guys on Wednesday, hopefully. We'll talk about an extension of Android for data privacy.