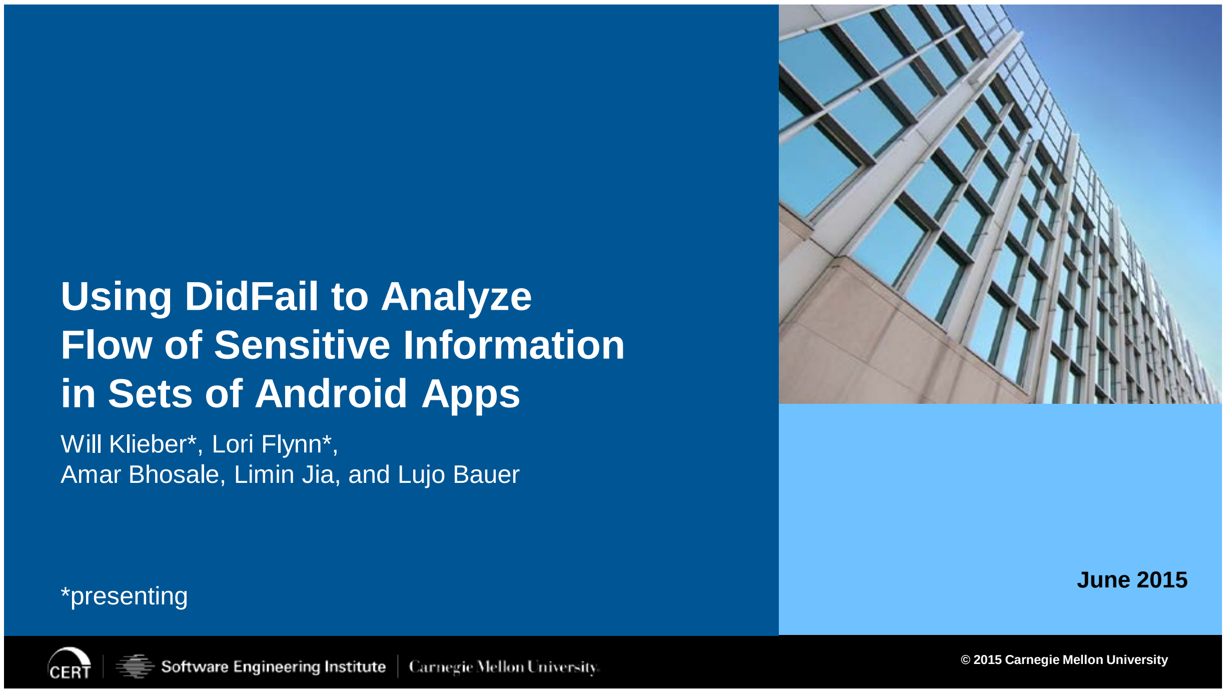
Using DidFail to Analyze Flow of Sensitive Information in Sets of Android Apps - Lori Flynn & Will Klieber

Table of Contents



Using DidFail to Analyze Flow of Sensitive Information in Sets of Android Apps

Will Klieber*, Lori Flynn*,
Amar Bhosale, Limin Jia, and Lujo Bauer

*presenting

June 2015

CERT | Software Engineering Institute | Carnegie Mellon University

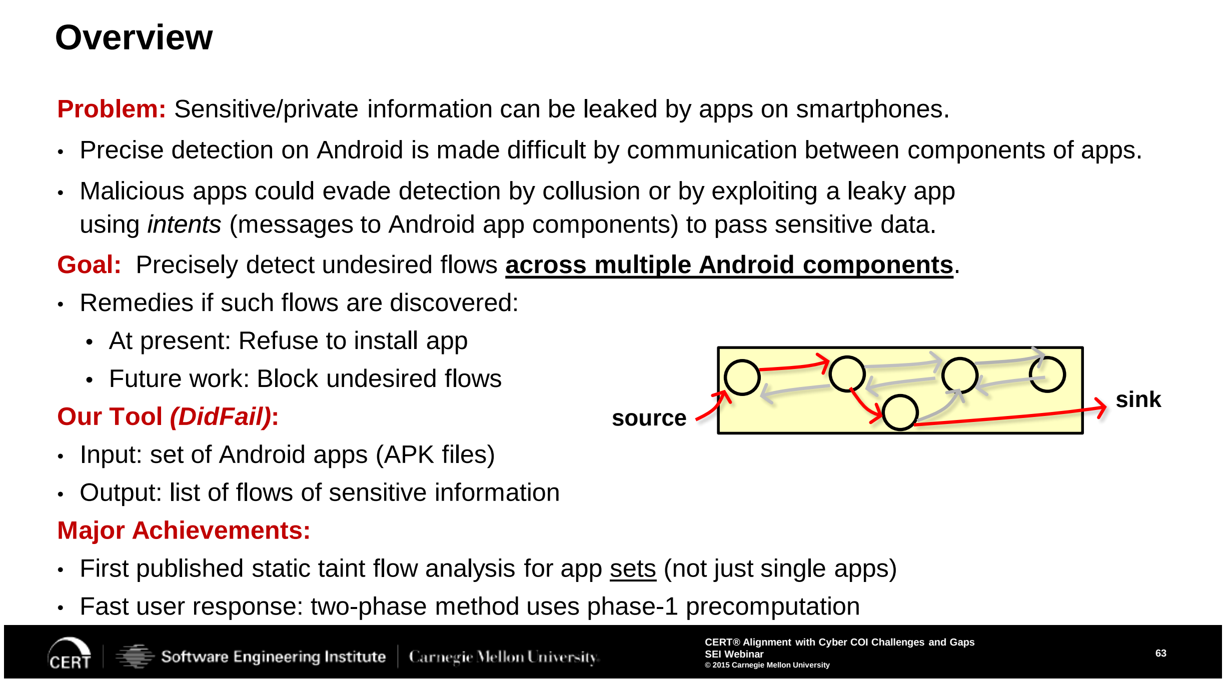
© 2015 Carnegie Mellon University

**062 Shane: And welcome back to the SEI virtual event, CERT Alignment with Cyber COI Challenges and Gaps. Just a reminder, anybody that's on Twitter, be sure to follow @SEInews and use the hashtag CERTCyber to follow along in the conversation.

Our next topic is using DidFail to analyze flow of sensitive information in sets of Android apps by Dr. Lori Flynn and Dr. Will Klieber. Dr. Lori Flynn is a researcher in the secure coding initiative within CERT. Her work includes research and development of new static analyses and secure coding standards. Dr. Will Klieber is a researcher within CERT and author of DidFail. His work is focused on static analysis of Android apps and detection of potentially malicious Java source code. Prior to joining the CERT division, Klieber was a doctoral student in CMU's computer science department. And now, I would like to turn it over to Dr. Will Klieber. Will, all yours.

Will Klieber: Thanks. I'm Will Klieber. Today, Lori Flynn and I will be talking about using DidFail to analyze the flow of sensitive information in sets of Android apps.

Overview



Overview

Problem: Sensitive/private information can be leaked by apps on smartphones.

- Precise detection on Android is made difficult by communication between components of apps.
- Malicious apps could evade detection by collusion or by exploiting a leaky app using *intents* (messages to Android app components) to pass sensitive data.

Goal: Precisely detect undesired flows **across multiple Android components**.

- Remedies if such flows are discovered:
  - At present: Refuse to install app
  - Future work: Block undesired flows

Our Tool *(DidFail)*:

- Input: set of Android apps (APK files)
- Output: list of flows of sensitive information

Major Achievements:

- First published static taint flow analysis for app sets (not just single apps)
- Fast user response: two-phase method uses phase-1 precomputation

CERT | Software Engineering Institute | Carnegie Mellon University
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University
63

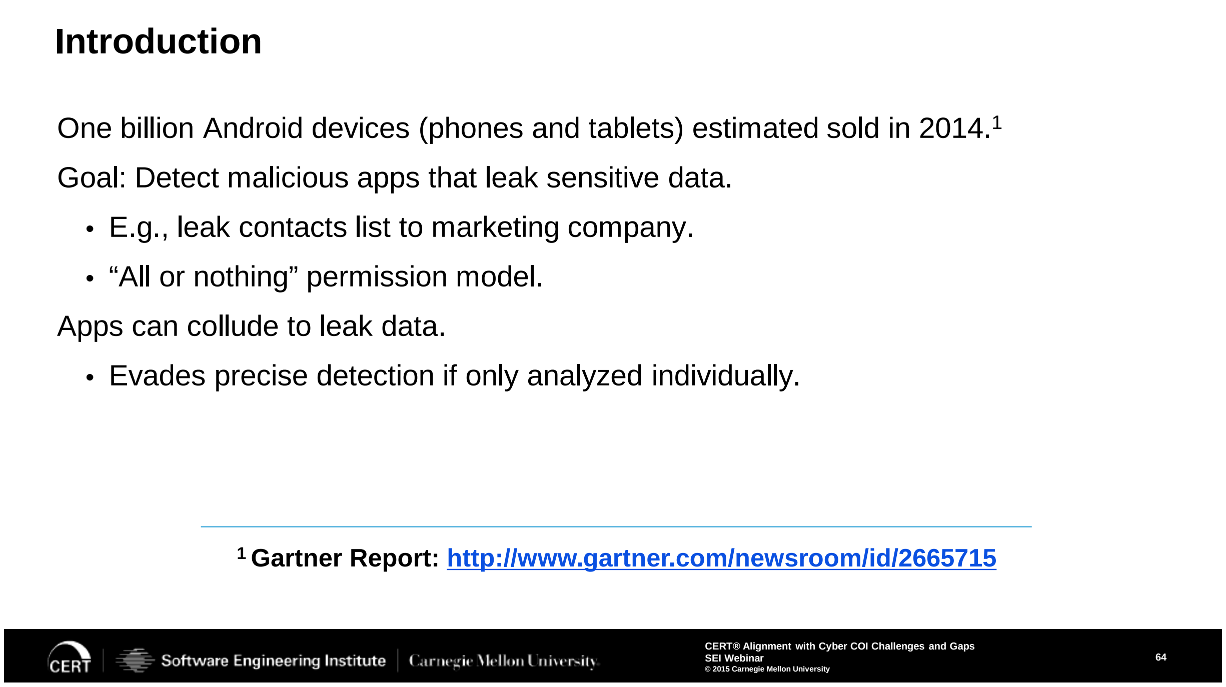
**063 The problem that we address is the leakage of sensitive information on apps on smartphones, especially on the Android platform. Android has a complex communication system, which can make it difficult to detect flows of information between apps. Malicious apps can take advantage of this.

For example, suppose that you have one app that has permission to read from a source of sensitive information and another app that has Internet permissions. In that case, if the two apps can communicate with each other, you can have a flow from your sensitive information to the Internet.

So, previously, static analysis tools were not able to precisely detect these types of flows. They would either have many false positives or many false negatives. The goal of DidFail, of our project, is to precisely detect these flows that happen across multiple Android components. Currently, if such a flow is discovered, the only remedy would be to refuse to install the app. But as future work, we would like the ability- we are investigating the ability to block these undesired flows while still allowing the user to install the app and use the remaining functionality.

Our tool, DidFail, takes as input a set of Android apps' APK files, and as output, produces a list of flows of sensitive information. DidFail was the first published static taint flow analysis for sets of apps, not just single apps. DidFail uses a two-phase computation method, which in many common use scenarios allows for a fast user response.

Introduction



Introduction

One billion Android devices (phones and tablets) estimated sold in 2014.[1]

Goal: Detect malicious apps that leak sensitive data.

- E.g., leak contacts list to marketing company.
- "All or nothing" permission model.

Apps can collude to leak data.

- Evades precise detection if only analyzed individually.

[1] **Gartner Report: http://www.gartner.com/newsroom/id/2665715**

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

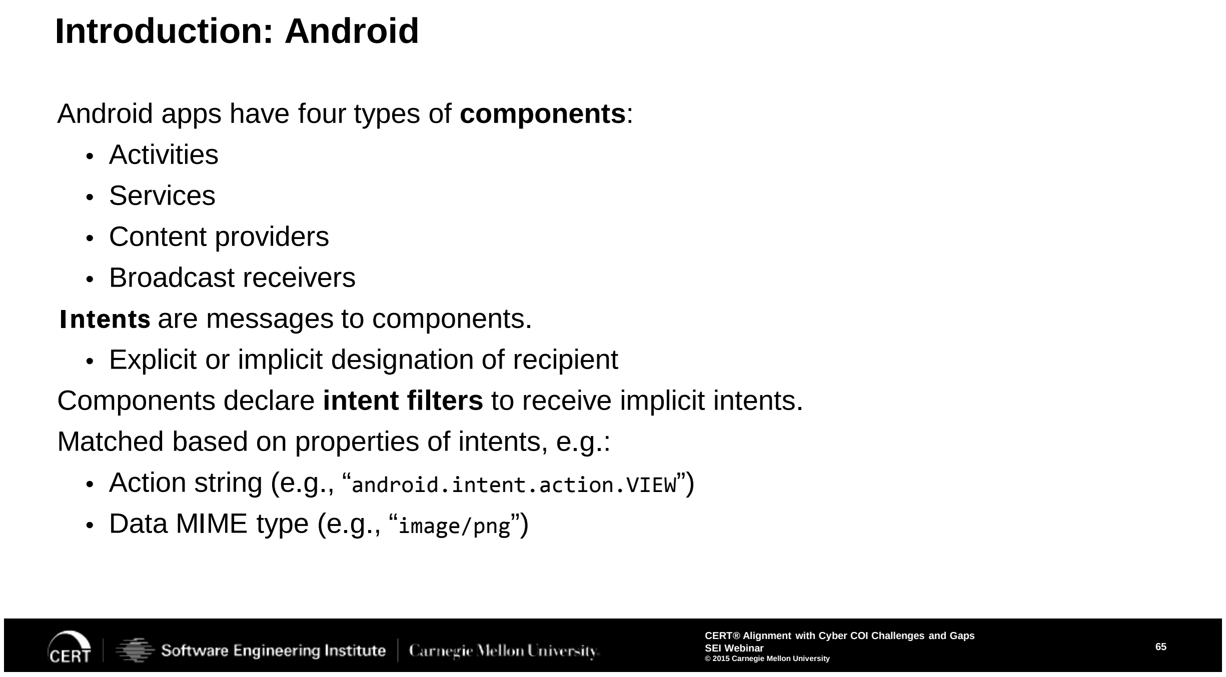
64

**064 Android is the most popular mobile operating system in the world, with an estimated one billion devices sold last year. The goal of our project, DidFail, is to detect malicious apps that can leak a user's sensitive data. So, for example, a malicious app might read your list of contacts and leak that list to a marketing company.

This problem is made worse by Android's all-or-nothing permission model. So, in Android, when you go to install an app, you're presented with a list of permissions that the app needs. And you get to choose to either install the app and grant it all permissions or refrain entirely from installing the app.

Now, with new versions of Android that are under development, this situation is being improved where the user can selectively choose to grant certain permissions to apps. But currently, only this all-or-nothing permission model exists.

Introduction: Android



Introduction: Android

Android apps have four types of **components**:

- Activities
- Services
- Content providers
- Broadcast receivers

Intents are messages to components.

- Explicit or implicit designation of recipient

Components declare **intent filters** to receive implicit intents.

Matched based on properties of intents, e.g.:

- Action string (e.g., "`android.intent.action.VIEW`")
- Data MIME type (e.g., "`image/png`")

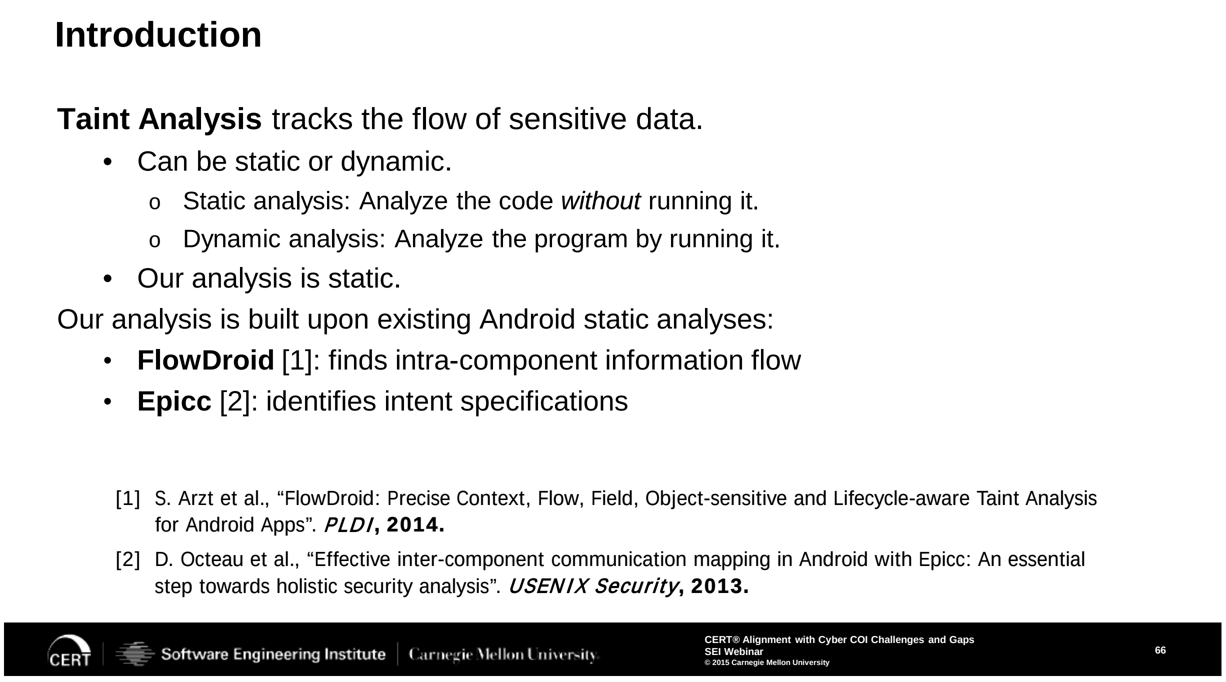
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

65

**065 Android apps have four types of components: activities, services, content providers, and broadcast receivers. An intent is a message sent to a component Android app. An intent may either explicitly designate its recipient by the name of the component that's supposed to receive the intent, or it can implicitly designate the recipient by listing properties and letting the Android OS choose which would be a suitable receiver for the intent.

Components can declare intent filters if they wish to receive implicit intents. And then the Android OS matches a sent-implicit-intent to a receiver based on the properties of the sent intent such as the action string and the data MIME type.

Introduction



Introduction

Taint Analysis tracks the flow of sensitive data.

- Can be static or dynamic.
  - Static analysis: Analyze the code *without* running it.
  - Dynamic analysis: Analyze the program by running it.
- Our analysis is static.

Our analysis is built upon existing Android static analyses:

- **FlowDroid** [1]: finds intra-component information flow
- **Epicc** [2]: identifies intent specifications

[1] S. Arzt et al., "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". ***PLDI*, 2014.**

[2] D. Octeau et al., "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis". ***USENIX Security*, 2013.**

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

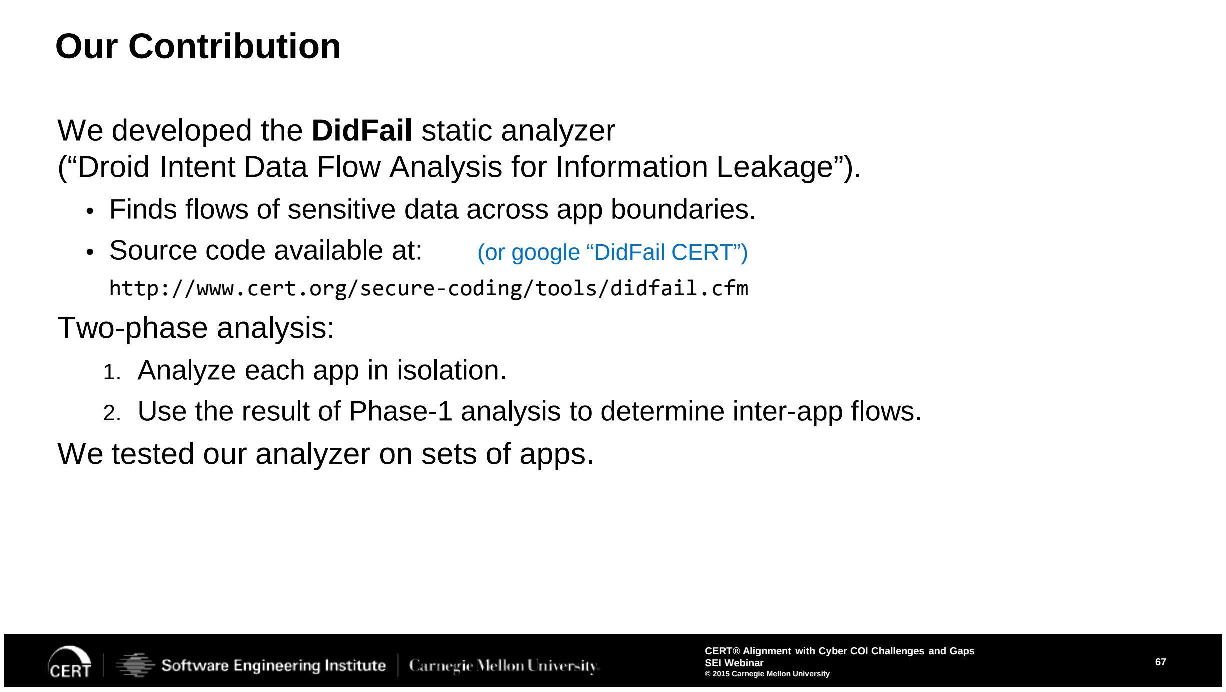
66

**066 Taint analysis tracks the flow of sensitive data. Taint analysis can be either static or dynamic. In a static analysis, the tool analyzes the source code of the program without running it. A benefit of static analysis is that if the static analysis-- if it faithfully models the program and the program's environment, then it can detect all possible behaviors of the program. A downside of static analysis is that it is difficult to completely faithfully model the environment in which the program is running, which can lead to both false positives and false negatives.

In contrast, with dynamic analysis, we actually run the program and observe its behavior as its running. A downside of dynamic analysis is that you can only detect programs that are-- you can only detect properties of the program that are actually exercised in the particular traces that you execute. An advantage is that you always faithfully represent the environment because you are running in the actual environment.

So, our analysis, DidFail, is a static analysis. And it builds upon two existing standard Android static Analyses: FlowDroid, which finds information flow within a single component, and Epicc, which identifies properties of intents such as the action string and the data MIME type, which are used to-- when such an intent is sent, Android uses-- the OS uses those properties to determine which component should receive the intent.

Our Contribution



Our Contribution

We developed the **DidFail** static analyzer
(“Droid Intent Data Flow Analysis for Information Leakage”).

- Finds flows of sensitive data across app boundaries.
- Source code available at: (or google “DidFail CERT”)
  `http://www.cert.org/secure-coding/tools/didfail.cfm`

Two-phase analysis:

1. Analyze each app in isolation.
2. Use the result of Phase-1 analysis to determine inter-app flows.

We tested our analyzer on sets of apps.

CERT | Software Engineering Institute | Carnegie Mellon University

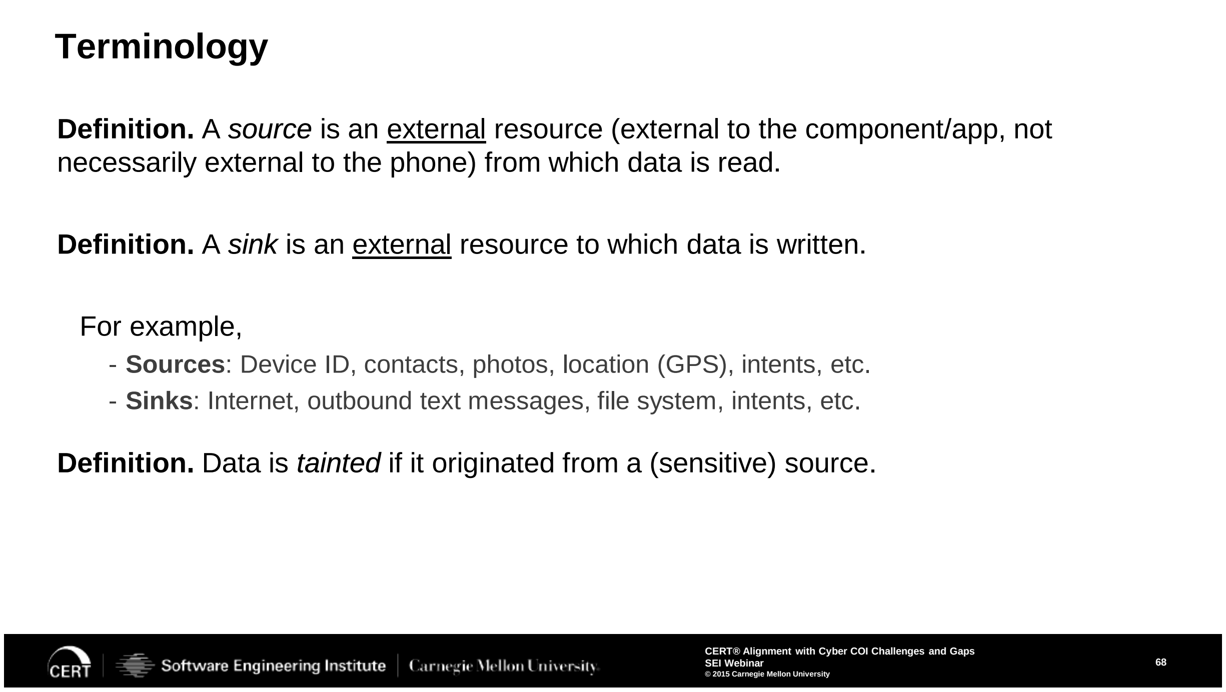
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

67

**067 Our main contribution is the development of the DidFail analysis and tool. "DidFail" is an acronym for Droid Intent Data Flow Analysis for Information Leakage. DidFail finds flows of sensitive data across app boundaries. DidFail is open source. And the source code is available at the URLs seen in this slide. Or you can just Google "DidFail CERT", and it should be the first result.

DidFail uses a two-phase analysis. In the first phase, DidFail analyzes each app in isolation. And then in the second phase, we use the results of the first phase to determine what flows are possible across app boundaries. We've also tested our analyzer on several sets of apps.

Terminology



Terminology

Definition. A *source* is an external resource (external to the component/app, not necessarily external to the phone) from which data is read.

Definition. A *sink* is an external resource to which data is written.

For example,

- **Sources**: Device ID, contacts, photos, location (GPS), intents, etc.
- **Sinks**: Internet, outbound text messages, file system, intents, etc.

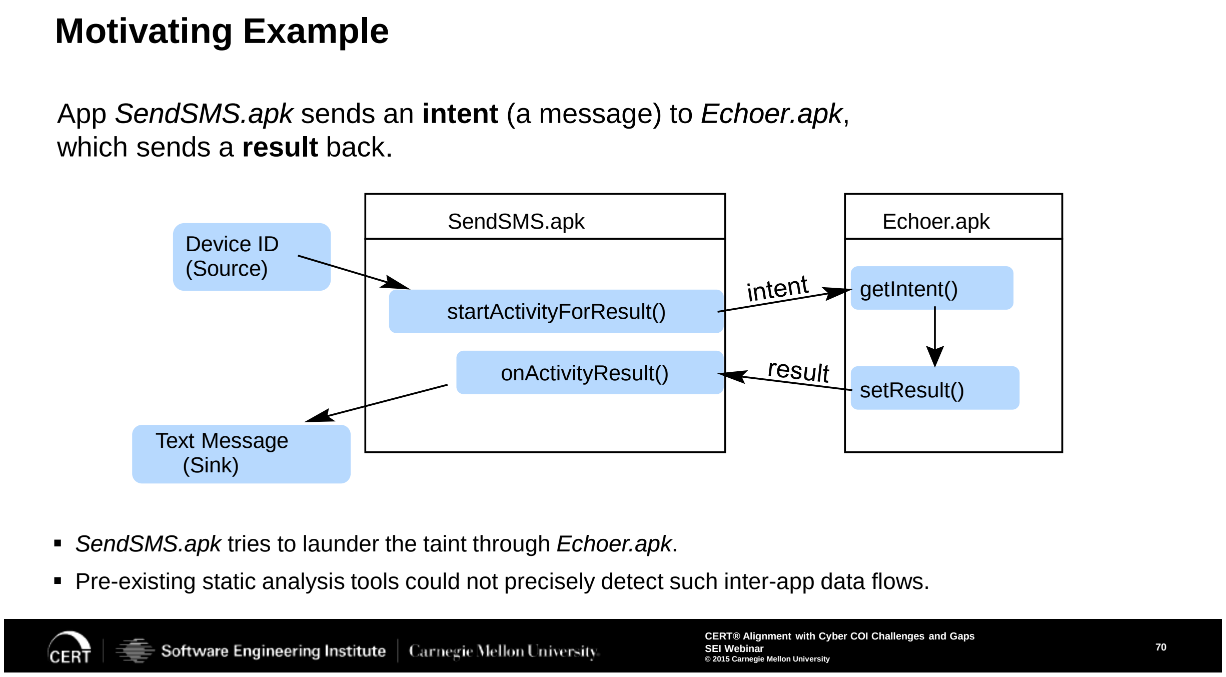
Definition. Data is *tainted* if it originated from a (sensitive) source.

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University
68

**068 We say that a source is an external resource (external to the app or component, but not necessarily external to the phone) from which data is read. And a sink is an external resource to which data is written. For example, sources include the device ID of the phone, the user's list of contacts that's stored in the phone, photos that are stored in the phone, the location, and the physical location of the device as determined by GPS. And we also consider intents that are received to be sources.

Sinks include sending information to the Internet, sending a text message, writing to the file system, or sending an intent. We say the data is tainted if it originated from a sensitive source. And with DidFail, all sources are either Android API functions that can be called and return a value, or parameters of callback methods. And DidFail comes with a list of sources that we consider sensitive. And the user can add to or remove from that list.

Motivating Example



Motivating Example

App *SendSMS.apk* sends an **intent** (a message) to *Echoer.apk*, which sends a **result** back.

- *SendSMS.apk* tries to launder the taint through *Echoer.apk*.
- Pre-existing static analysis tools could not precisely detect such inter-app data flows.

CERT | Software Engineering Institute | Carnegie Mellon University

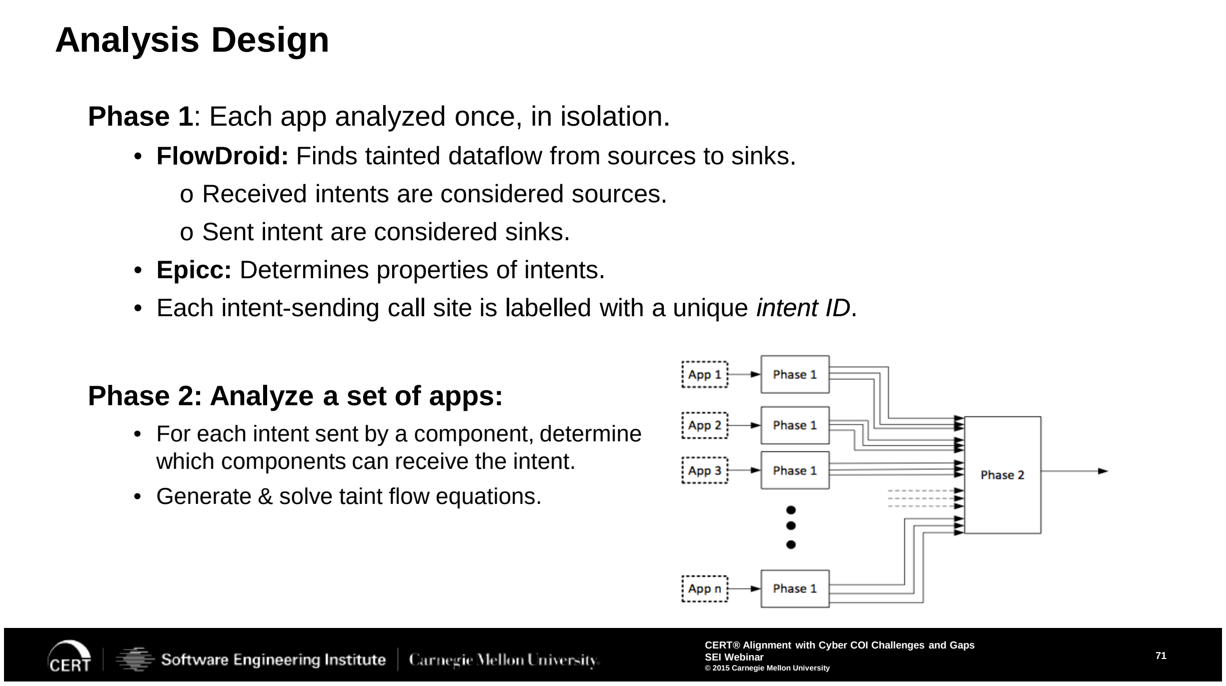
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

70

**070 Let's consider a motivating example where we have a set of two apps, SendSMS and Echoer. What happens here is that SendSMS reads the device ID. And then it sends that information to Echoer. Echoer takes the device ID and then just echoes it back to SendSMS. And then SendSMS takes that data and writes it as a text message. So, in effect, what's happening is that SendSMS is trying to launder the taint through the Echoer.

Before DidFail, previously existing static taint analyses could not precisely detect these inter-app flows. They would either have many false negatives or many false positives, depending on how they were set up.

Analysis Design



Analysis Design

Phase 1: Each app analyzed once, in isolation.

- **FlowDroid:** Finds tainted dataflow from sources to sinks.
  - o Received intents are considered sources.
  - o Sent intent are considered sinks.
- **Epicc:** Determines properties of intents.
- Each intent-sending call site is labelled with a unique *intent ID*.

Phase 2: Analyze a set of apps:

- For each intent sent by a component, determine which components can receive the intent.
- Generate & solve taint flow equations.

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

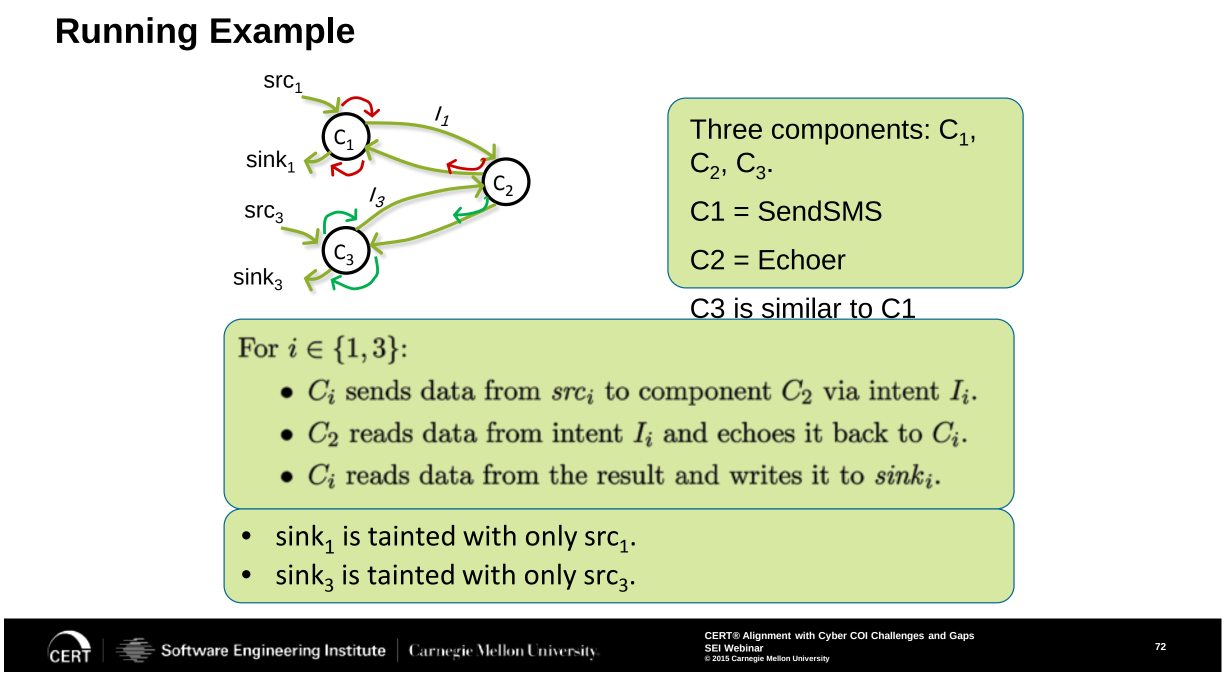
71

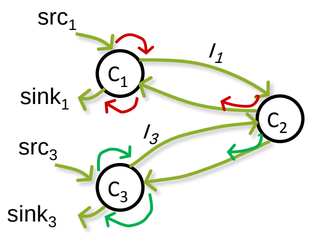
**071 So, DidFail operates in two phases. In the first phase, we analyze each app in isolation once. We use FlowDroid to find tainted data flow from sources to sinks inside components, where received intents are considered sources and sent intents are considered sinks.

We also use Epicc to determine properties of intents that are used in matching sent intents to receivers. Each intent-sending call site is labeled with a unique intent ID.

In phase two, we analyze a set of apps. For each intent that is sent by a component, we determine which components can possibly receive the intent. We generate and solve taint flow equations which we use to calculate the final taints of each possible sink.

Running Example


Running Example



Three components: C_1, C_2, C_3.

C1 = SendSMS

C2 = Echoer

C3 is similar to C1

For $i \in \{1,3\}$:

- C_i sends data from src_i to component C_2 via intent I_i.
- C_2 reads data from intent I_i and echoes it back to C_i.
- C_i reads data from the result and writes it to $sink_i$.

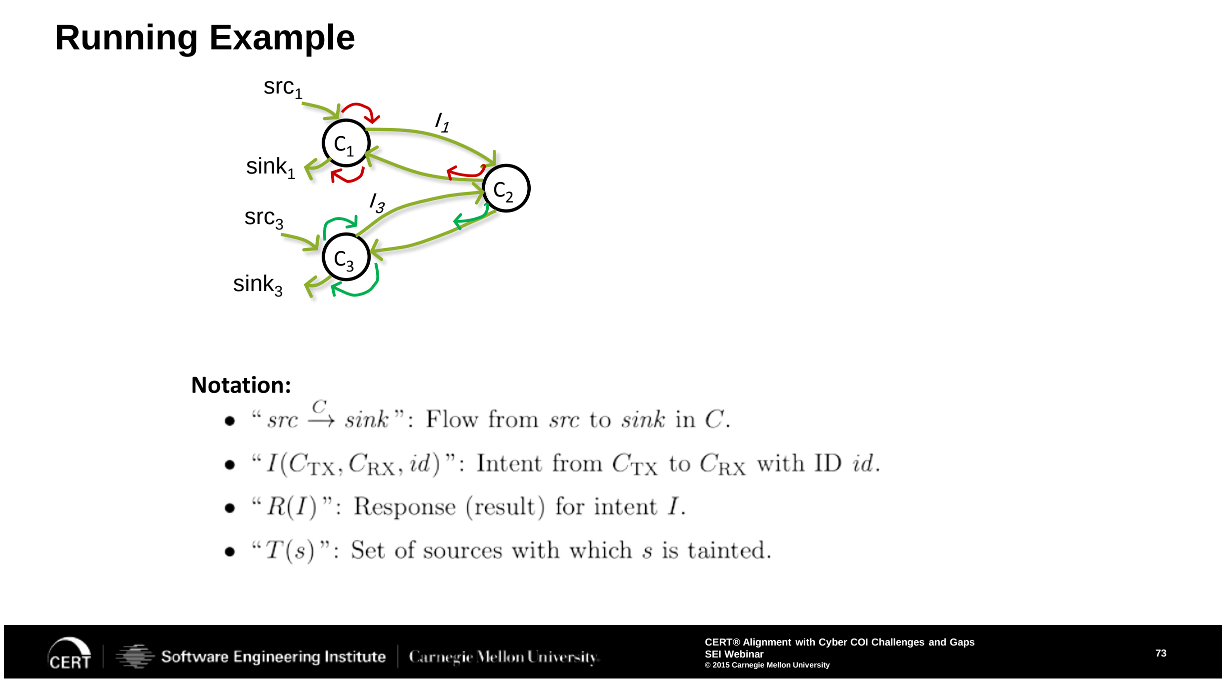
- $sink_1$ is tainted with only src_1.
- $sink_3$ is tainted with only src_3.

**072 So, let's consider a running example where we have three components, C1, C2, and C3. C1 is similar to the SendSMS component of the previous example. And C2 is the Echoer. And C3 is similar to C1.

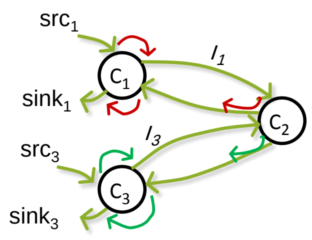
So, let's see what happens in this example. So, C1 reads data from Source 1 and sends it to component C2 via intent I1. Then C2 reads data from intent I1 and sends the data back to C1. Finally, C1 reads the data from the results and writes it to the sink. And of course, C3 operates similarly to C1. So, our final result is that Sink 1 is tainted only

with Source 1. And C3 is tainted
only with Source 3.

Running Example



Running Example



Notation:

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\mathrm{TX}}, C_{\mathrm{RX}}, id)$": Intent from C_{TX} to C_{RX} with ID id.
- "$R(I)$": Response (result) for intent I.
- "$T(s)$": Set of sources with which s is tainted.

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

73

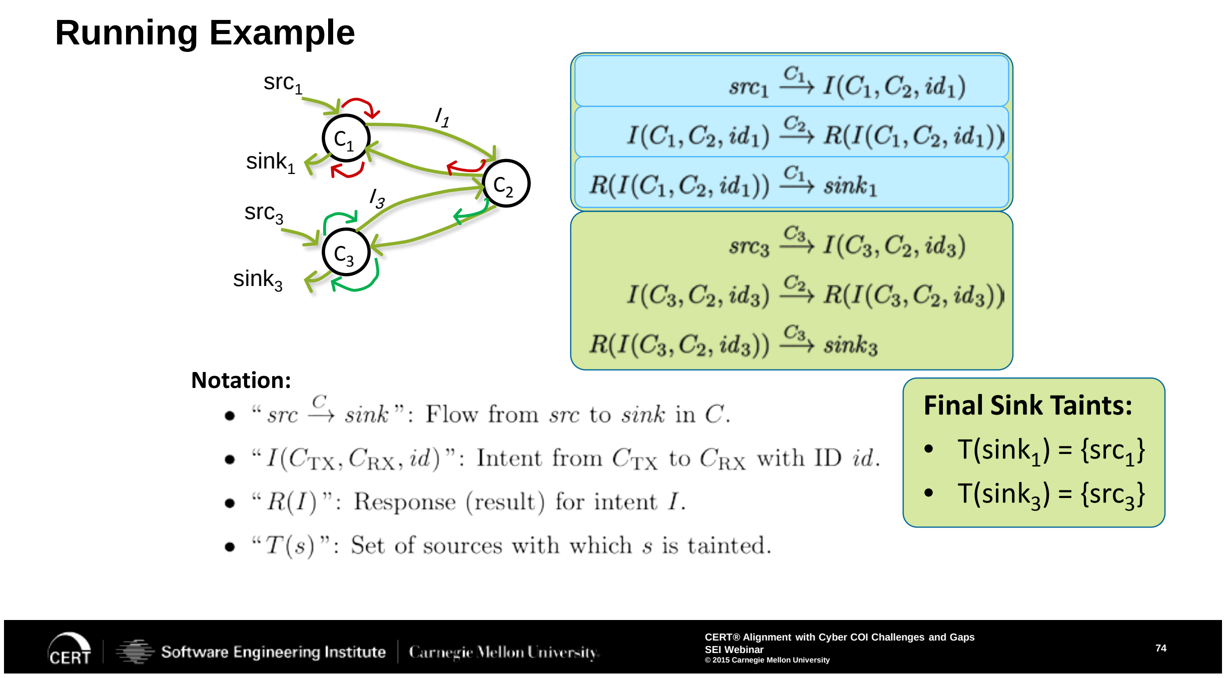
**073 Now, let's describe some notation that we're using. We write sink arrow-- I'm sorry, we write "source arrow sink" to denote that there is a flow from source to sink. And we write the component in which the flow happens above the arrow.

As shown on this slide, an intent is identified by a tuple of three items, C_TX, which is the component that sends the intent, C_RX, which is the component that receives the intent, and a unique intent ID. So, each line of the source code that can possibly send an intent is labeled with a unique ID.

The next bullet point on the slide: If you have an intent R, then you write

R of I to denote the result, the response that is sent for that intent. And given a source or a sink S, we write T of S to denote the set of sources from which S has tainted data.

Running Example



Running Example

$$src_1 \xrightarrow{C_1} I(C_1, C_2, id_1)$$
$$I(C_1, C_2, id_1) \xrightarrow{C_2} R(I(C_1, C_2, id_1))$$
$$R(I(C_1, C_2, id_1)) \xrightarrow{C_1} sink_1$$

$$src_3 \xrightarrow{C_3} I(C_3, C_2, id_3)$$
$$I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$$
$$R(I(C_3, C_2, id_3)) \xrightarrow{C_3} sink_3$$

Notation:

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\mathrm{TX}}, C_{\mathrm{RX}}, id)$": Intent from C_{TX} to C_{RX} with ID id.
- "$R(I)$": Response (result) for intent I.
- "$T(s)$": Set of sources with which s is tainted.

Final Sink Taints:

- T(sink$_1$) = {src$_1$}
- T(sink$_3$) = {src$_3$}

CERT | Software Engineering Institute | Carnegie Mellon University

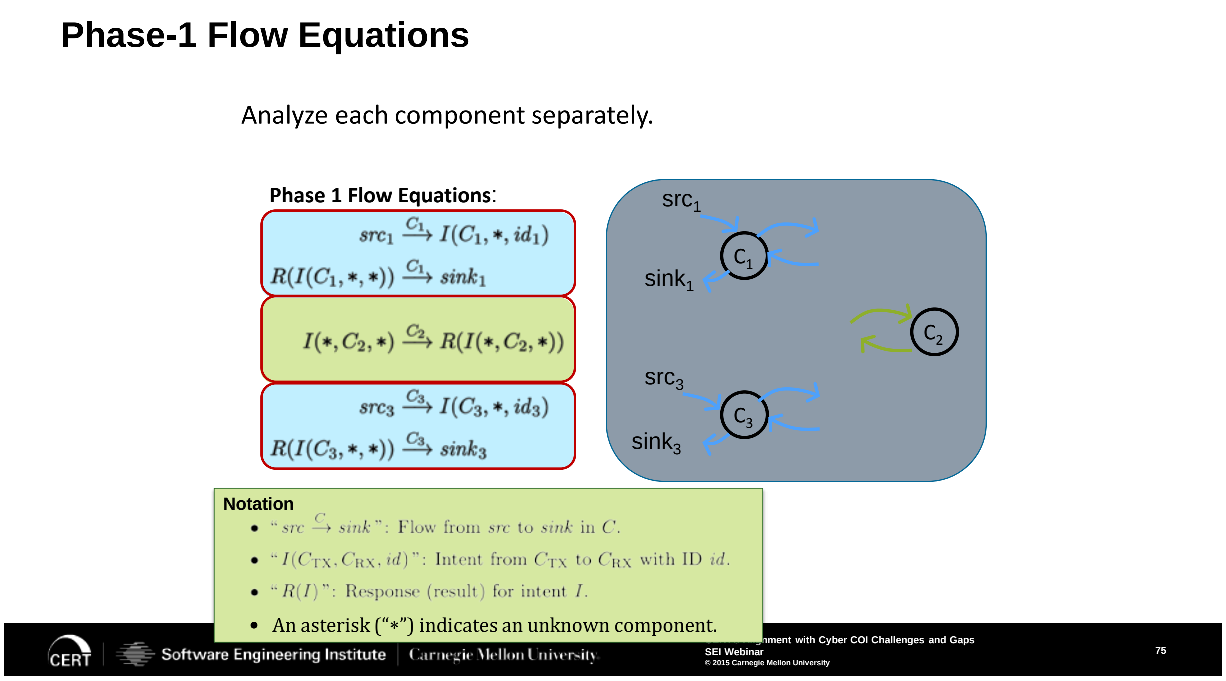
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

74

**074 So, let's write the flow equations for this example. In here we have a flow from Source 1 to an intent. And the intent is sent from C1 to C2. And it has an ID: ID1.

Then we have a flow from that intent to component C2. So, it's the same intent as in the previous equation. And C2 takes that information and just echoes it back as a response to the intent. And then there's a flow from that response that C2 sends back. C1 takes that response and then just writes that data to the sink. And C3 behaves similarly to C1. So, our final result is that the taint of Sink 1 consists only of Source 1 and the taint of Sink 3 consists only of Source 3.

Phase-1 Flow Equations



Phase-1 Flow Equations

Analyze each component separately.

Phase 1 Flow Equations:

$$src_1 \xrightarrow{C_1} I(C_1, *, id_1)$$
$$R(I(C_1, *, *)) \xrightarrow{C_1} sink_1$$
$$I(*, C_2, *) \xrightarrow{C_2} R(I(*, C_2, *))$$
$$src_3 \xrightarrow{C_3} I(C_3, *, id_3)$$
$$R(I(C_3, *, *)) \xrightarrow{C_3} sink_3$$

Notation

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\mathrm{TX}}, C_{\mathrm{RX}}, id)$": Intent from C_{TX} to C_{RX} with ID id.
- "$R(I)$": Response (result) for intent I.
- An asterisk ("*") indicates an unknown component.

CERT | Software Engineering Institute | Carnegie Mellon University

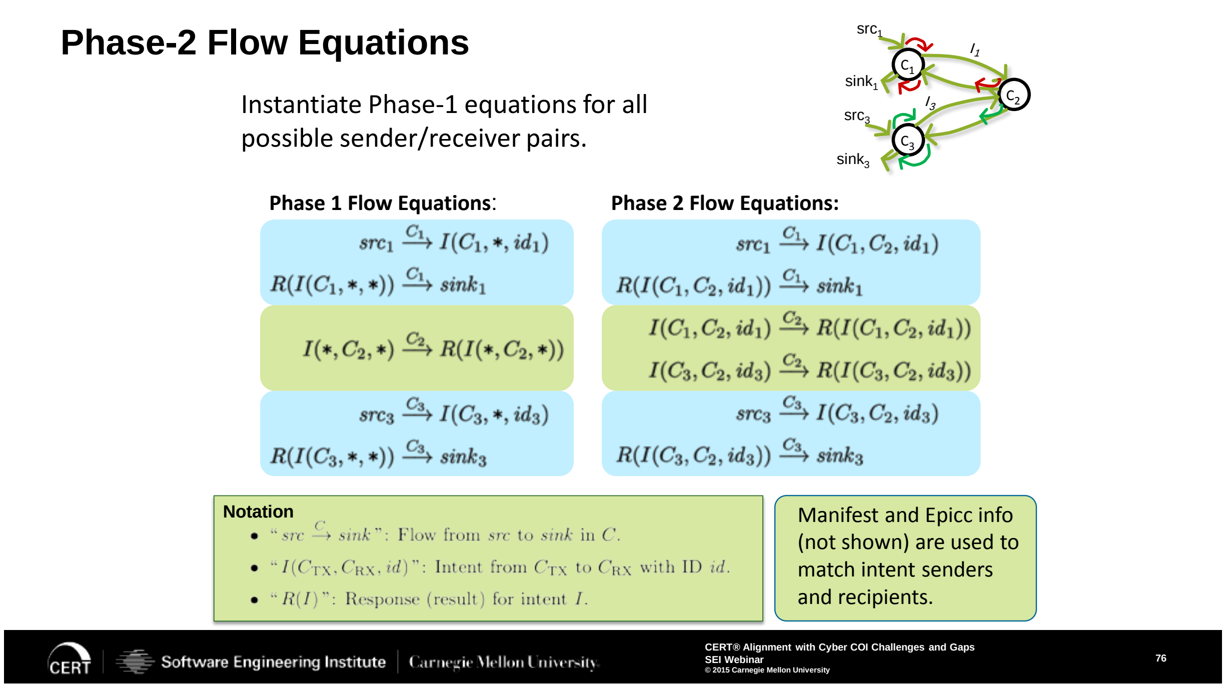
nment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

75

**075 So, in phase one, we analyze each component separately. So, for C1, we don't know-- we know that C1 sends an intent. But we don't know what the-- who the recipient is. So, as shown on this slide, we use an asterisk in place of the recipient. So, we said that there's a flow from the source to an intent, but we don't know who receives the intent. So, we use an asterisk for the recipient.

And then we also know from analyzing the program that there is a flow from-- that component C1 accepts a response from that intent. And then it takes that response and writes it to Sink 1. Then for C2, we know that C2 takes an intent from an unknown sender, and then just echoes that information back as a response. And C3 behaves similarly to C1.

Phase-2 Flow Equations



Phase-2 Flow Equations

Instantiate Phase-1 equations for all possible sender/receiver pairs.

src_1
I_1
C_1
$sink_1$
C_2
src_3
I_3
C_3
$sink_3$

Phase 1 Flow Equations:

$$src_1 \xrightarrow{C_1} I(C_1, *, id_1)$$
$$R(I(C_1, *, *)) \xrightarrow{C_1} sink_1$$
$$I(*, C_2, *) \xrightarrow{C_2} R(I(*, C_2, *))$$
$$src_3 \xrightarrow{C_3} I(C_3, *, id_3)$$
$$R(I(C_3, *, *)) \xrightarrow{C_3} sink_3$$

Phase 2 Flow Equations:

$$src_1 \xrightarrow{C_1} I(C_1, C_2, id_1)$$
$$R(I(C_1, C_2, id_1)) \xrightarrow{C_1} sink_1$$
$$I(C_1, C_2, id_1) \xrightarrow{C_2} R(I(C_1, C_2, id_1))$$
$$I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$$
$$src_3 \xrightarrow{C_3} I(C_3, C_2, id_3)$$
$$R(I(C_3, C_2, id_3)) \xrightarrow{C_3} sink_3$$

Notation

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\mathrm{TX}}, C_{\mathrm{RX}}, id)$": Intent from C_{TX} to C_{RX} with ID id.
- "$R(I)$": Response (result) for intent I.

Manifest and Epicc info (not shown) are used to match intent senders and recipients.

CERT | Software Engineering Institute | Carnegie Mellon University

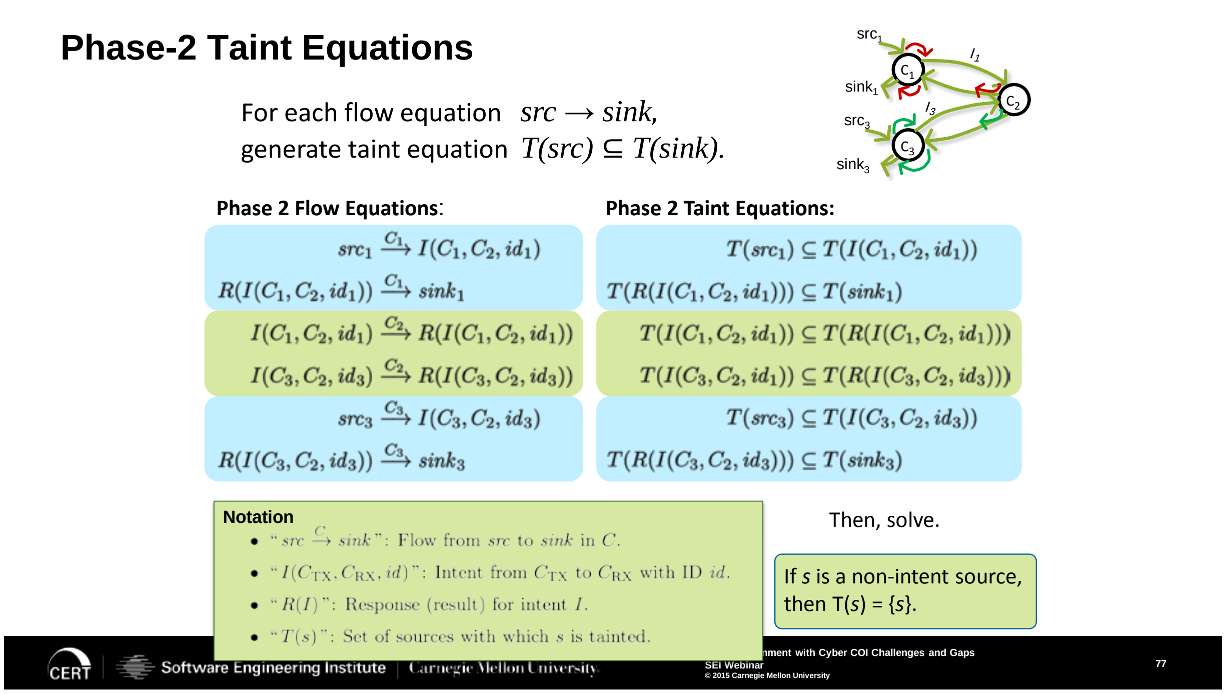
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

76

**076 In phase two, we take the information from phase one, we take those phase-one flow equations and instantiate them for all possible sender/receiver pairs. So, for C1, there's only one possible instantiation. The only possible recipient, the asterisk in this flow equation is C2. And likewise, the only-- when C1 receives a response, it's going to be from C2. So, we substitute in C2 for the asterisk. And the only possible intent ID is ID1 in this case.

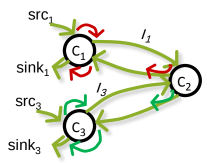
For C2, there are two possible instantiations. The first possible instantiation would be C1 for that asterisk. And the other possible instantiation would be for C3. So, this single equation in phase one gets instantiated as two equations in phase two. And then for C3, it behaves similarly to C1.

Phase-2 Taint Equations



Phase-2 Taint Equations

For each flow equation $src \longrightarrow sink$,
generate taint equation $T(src) \subseteq T(sink)$.



Phase 2 Flow Equations:	Phase 2 Taint Equations:
$src_1 \xrightarrow{C_1} I(C_1, C_2, id_1)$	$T(src_1) \subseteq T(I(C_1, C_2, id_1))$
$R(I(C_1, C_2, id_1)) \xrightarrow{C_1} sink_1$	$T(R(I(C_1, C_2, id_1))) \subseteq T(sink_1)$
$I(C_1, C_2, id_1) \xrightarrow{C_2} R(I(C_1, C_2, id_1))$	$T(I(C_1, C_2, id_1)) \subseteq T(R(I(C_1, C_2, id_1)))$
$I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$	$T(I(C_3, C_2, id_1)) \subseteq T(R(I(C_3, C_2, id_3)))$
$src_3 \xrightarrow{C_3} I(C_3, C_2, id_3)$	$T(src_3) \subseteq T(I(C_3, C_2, id_3))$
$R(I(C_3, C_2, id_3)) \xrightarrow{C_3} sink_3$	$T(R(I(C_3, C_2, id_3))) \subseteq T(sink_3)$

Notation

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{TX}, C_{RX}, id)$": Intent from C_{TX} to C_{RX} with ID id.
- "$R(I)$": Response (result) for intent I.
- "$T(s)$": Set of sources with which s is tainted.

Then, solve.

If *s* is a non-intent source, then T(*s*) = {*s*}.

CERT | Software Engineering Institute | Carnegie Mellon University

…nment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

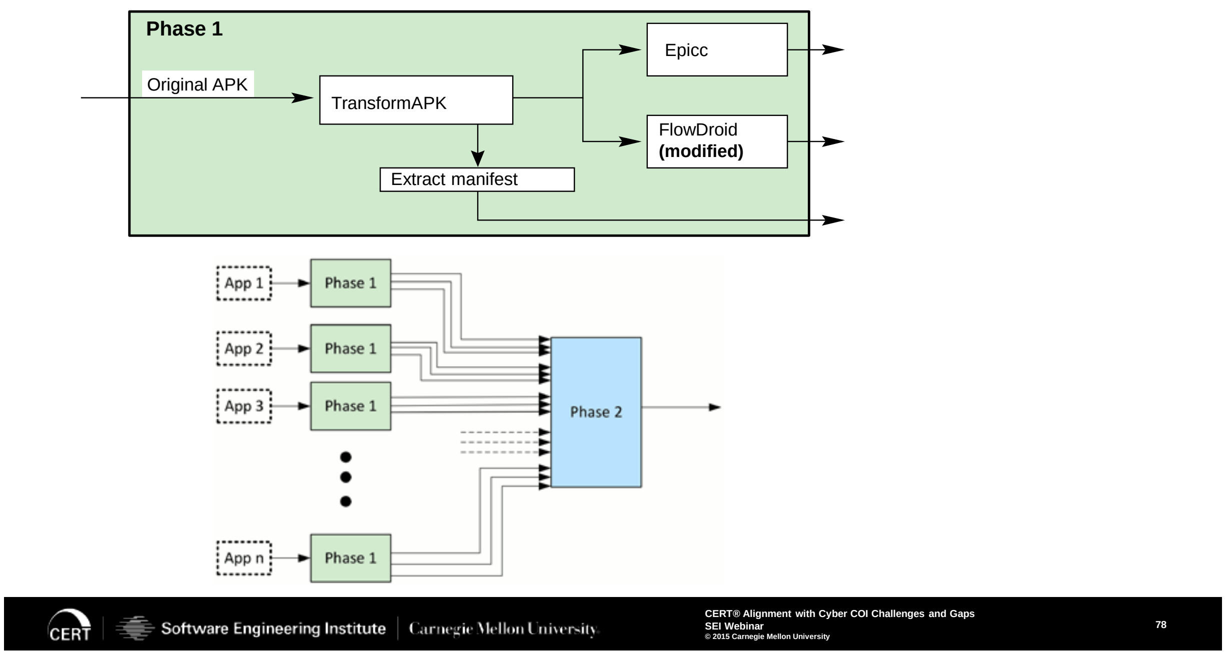
77

**077 Now, this slide shows the same two equations that were on the previous slide. For each one of these equations that indicate that there's a flow from a given source to a sink, we generate a taint equation that says that the taintedness of this sink must be at least that of the source. So, the taint of the source must be a subset or equal to the taint of the sink.

So, we just take all of the equations from all of the phase-two flow equations and write equivalent taint equations. And we also, for a non-intent source, we also have the taint of that source is just a set consisting exactly of itself.

And then we take all of these equations and solve them.

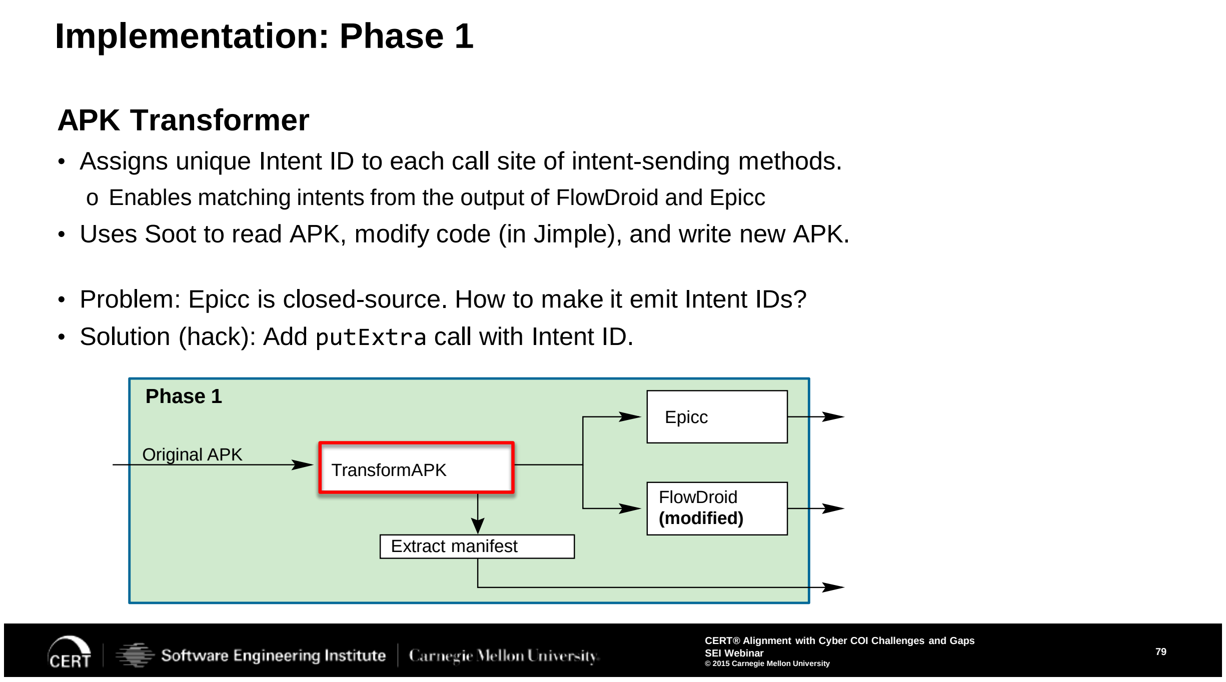
Phase 1


Phase 1
Original APK
TransformAPK
Extract manifest
Epicc
FlowDroid (modified)
App 1
App 2
App 3
App n
Phase 1
Phase 2
Software Engineering Institute
Carnegie Mellon University
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University
78

**078 Lori Flynn: So, in this slide, we show the processes that happen in phase one and phase two. At the top, we see the phase one. It has, as input, the original APK, or original app, which gets transformed. And we extract the manifest from the transformed APK. We also run the Epicc tool on the transformed app. And we run our modified FlowDroid on the transformed app.

Then phase two is shown on the bottom of the slide. And its inputs for each app in the -- for the app set that we're analyzing, it has three inputs per app in that set. And they're just the outputs of phase one. And in phase two, we analyze for possible taint flows.

Implementation: Phase 1



Implementation: Phase 1

APK Transformer

- Assigns unique Intent ID to each call site of intent-sending methods.
    - Enables matching intents from the output of FlowDroid and Epicc
- Uses Soot to read APK, modify code (in Jimple), and write new APK.

- Problem: Epicc is closed-source. How to make it emit Intent IDs?
- Solution (hack): Add `putExtra` call with Intent ID.

Phase 1
Original APK
TransformAPK
Epicc
FlowDroid **(modified)**
Extract manifest

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

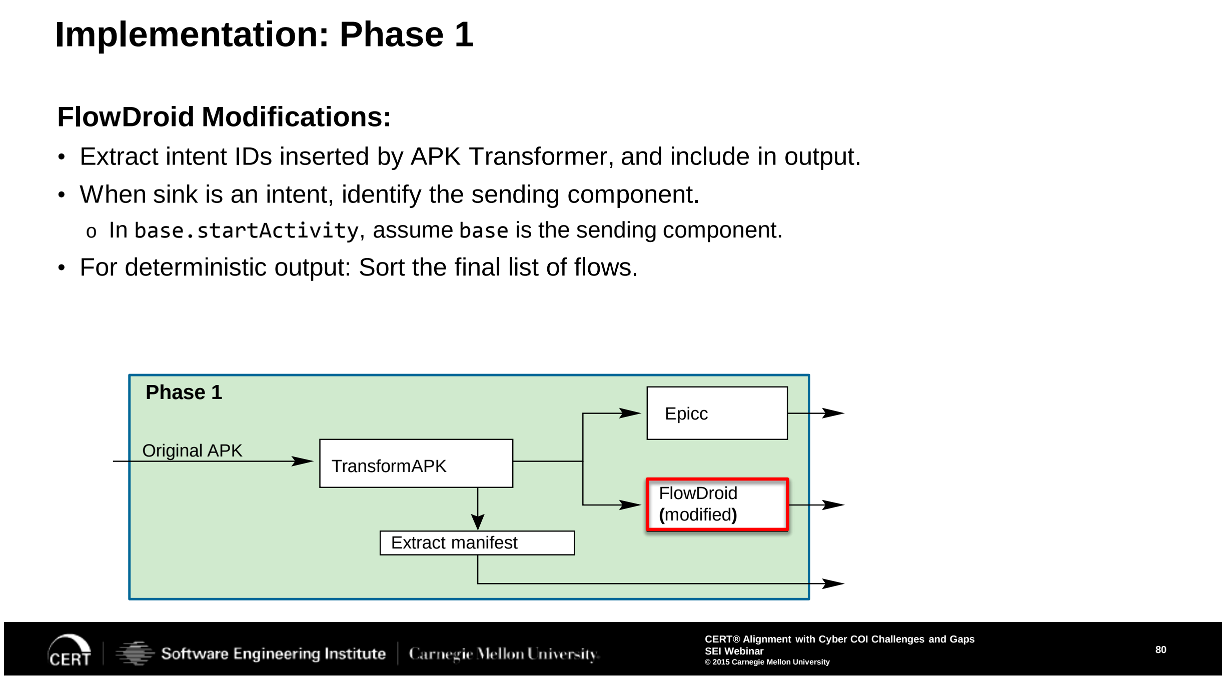
79

**079 In phase one, we have an APK transformer that assigns a unique intent ID to each call site of intent sending methods. This enables matching intents from the output of FlowDroid and Epicc.

The APK transformer uses the Soot framework to read the APK to modify the code in Jimple, which is an intermediate representation. And then it writes the new APK back out.

The reason that we do this transformation is because we have this problem: We needed Epicc to emit intent IDs, but Epicc is closed source. So, we couldn't modify the code. So, instead, we modified the APK. Epicc outputs information about putExtras that are in intents, which are sent. So, our solution was to add an extra-- or add a putExtra call with this unique intent ID to the app code directly before each intent gets sent.

Implementation: Phase 1



Implementation: Phase 1

FlowDroid Modifications:

- Extract intent IDs inserted by APK Transformer, and include in output.
- When sink is an intent, identify the sending component.
  - In `base.startActivity`, assume `base` is the sending component.
- For deterministic output: Sort the final list of flows.

Phase 1
Original APK
TransformAPK
Extract manifest
Epicc
FlowDroid (modified)

CERT | Software Engineering Institute | Carnegie Mellon University
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University
80

**080 In phase one, we modified FlowDroid-- or we use a modified FlowDroid in phase one. We modified FlowDroid to extract the intent IDs, which we had inserted into that APK using the transformer. And we include that in the output.

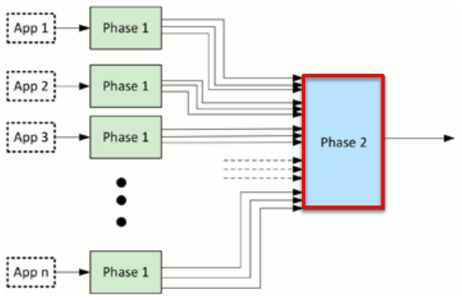
When the sink is an intent, we identify the sending component. And in order to get deterministic output, we sort the final list of flows that FlowDroid outputs.

Implementation: Phase 2

Implementation: Phase 2

Phase 2

- Input: Phase 1 output.
- Generate and solve the data-flow equations.
- Output:
  1. Directed graph indicating information flow between sources, intents, intent results, and sinks.
  2. Taintedness of each sink.



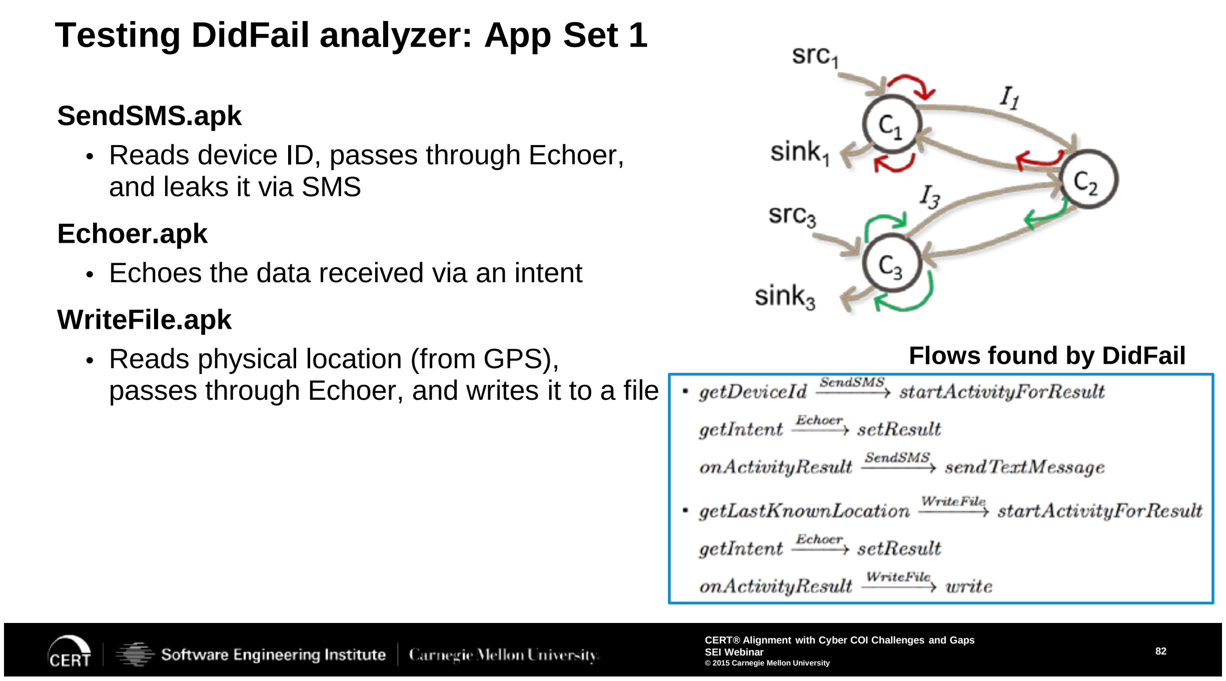
CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

81

**081 The input to phase two is the phase one output for each of the apps that we're analyzing. Then in phase two, we generate and solve the data flow equations. And our output consists of a directed graph that indicates information flow between sources, intents, intent results, and sinks, and the taintedness of each sink.

Testing DidFail analyzer: App Set 1



Testing DidFail analyzer: App Set 1

SendSMS.apk

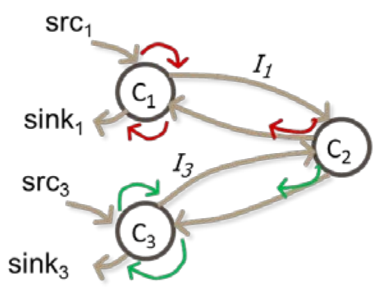
- Reads device ID, passes through Echoer, and leaks it via SMS

Echoer.apk

- Echoes the data received via an intent

WriteFile.apk

- Reads physical location (from GPS), passes through Echoer, and writes it to a file



Flows found by DidFail

- $getDeviceId \xrightarrow{SendSMS} startActivityForResult$
  $getIntent \xrightarrow{Echoer} setResult$
  $onActivityResult \xrightarrow{SendSMS} sendTextMessage$
- $getLastKnownLocation \xrightarrow{WriteFile} startActivityForResult$
  $getIntent \xrightarrow{Echoer} setResult$
  $onActivityResult \xrightarrow{WriteFile} write$

CERT | Software Engineering Institute | Carnegie Mellon University

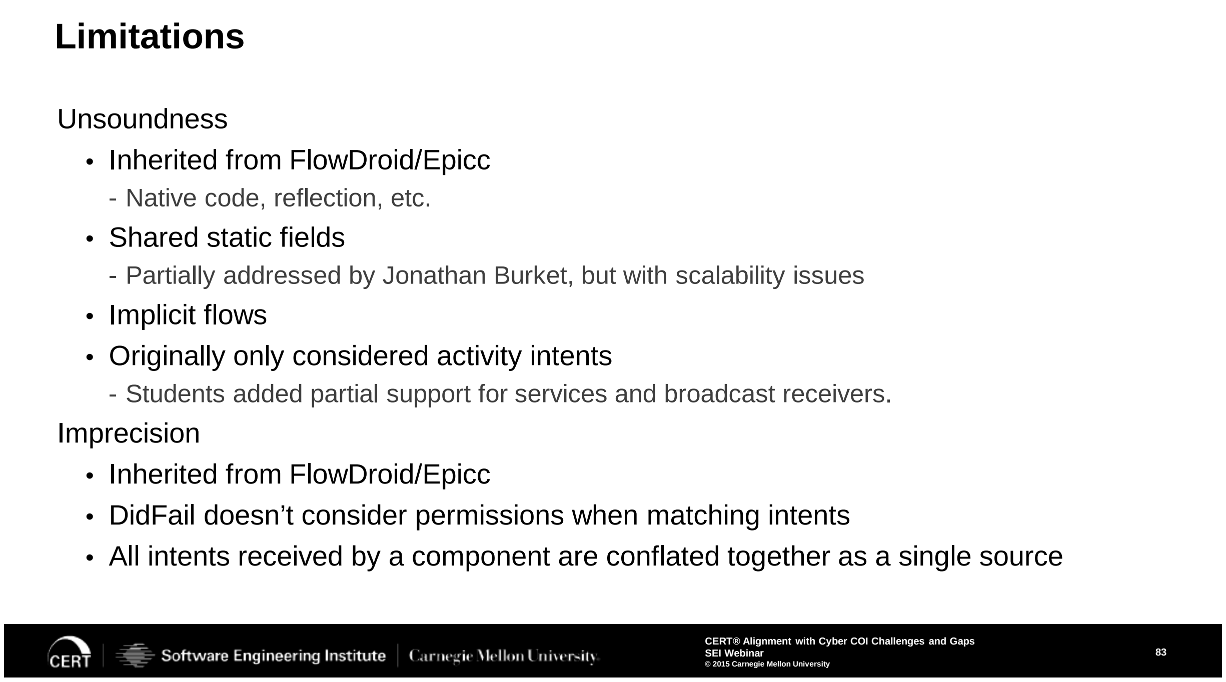
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

82

**082 We tested the DidFail analyzer on a toy set of apps, which instantiate the example that Will was talking about in the previous slides, that running example. So, we have the SendSMS app, which reads a device ID, passes it through the Echoer, and leaks it via SMS. So, that's like the C1 in this figure.

We have the Echoer app, which echoes the data received via an intent. That's our C2. And our WriteFile app reads a physical location from GPS, passes it through the Echoer, and writes it to a file. So, that's our C3. And DidFail found those flows.

Limitations



Limitations

Unsoundness

- Inherited from FlowDroid/Epicc
  - Native code, reflection, etc.
- Shared static fields
  - Partially addressed by Jonathan Burket, but with scalability issues
- Implicit flows
- Originally only considered activity intents
  - Students added partial support for services and broadcast receivers.

Imprecision

- Inherited from FlowDroid/Epicc
- DidFail doesn't consider permissions when matching intents
- All intents received by a component are conflated together as a single source

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

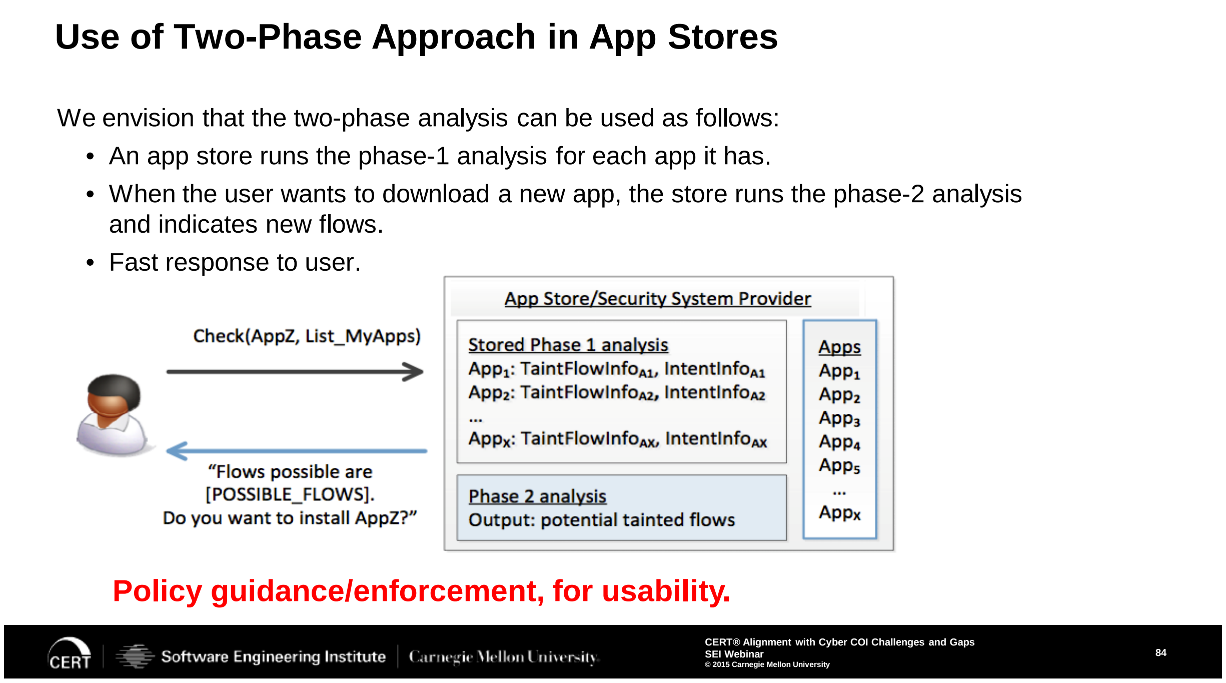
83

**083 DidFail has some limitations, including both sources of unsoundness and of imprecision. Some sources of unsoundness are inherited from FlowDroid and Epicc. For instance, they don't analyze native code or reflection. We also have soundness limitations due to analysis of taint flow through shared static fields. Some of those limitations have been addressed by work by Jonathan Burket in one of our branches of DidFail late last year. However, that work currently has some scalability issues. It uses a lot of memory and CPU power. So, we're working on scalability issues for that analysis.

DidFail does not analyze implicit flows. For instance, if receiving or sending an intent by itself conveys information aside from data within the intent, we don't analyze that. Our original release of DidFail only considered activity intents. However, we've worked with some grad students late last year who added partial support for services and broadcast receiver components.

We have some sources of imprecision which were inherited from FlowDroid and Epicc. Additionally, currently, DidFail doesn't consider permissions when matching intents. However, we intend to add that analysis in the future. Also, currently, all intents received by a component are conflated together as a single source. And we intend to enhance DidFail's precision with regard to that in the future as well.

Use of Two-Phase Approach in App Stores



Use of Two-Phase Approach in App Stores

We envision that the two-phase analysis can be used as follows:

- An app store runs the phase-1 analysis for each app it has.
- When the user wants to download a new app, the store runs the phase-2 analysis and indicates new flows.
- Fast response to user.

App Store/Security System Provider

Check(AppZ, List_MyApps)

Stored Phase 1 analysis
App$_1$: TaintFlowInfo$_{A1}$, IntentInfo$_{A1}$
App$_2$: TaintFlowInfo$_{A2}$, IntentInfo$_{A2}$
...
App$_X$: TaintFlowInfo$_{AX}$, IntentInfo$_{AX}$

Apps
App$_1$
App$_2$
App$_3$
App$_4$
App$_5$
...
App$_X$

"Flows possible are [POSSIBLE_FLOWS]. Do you want to install AppZ?"

Phase 2 analysis
Output: potential tainted flows

Policy guidance/enforcement, for usability.

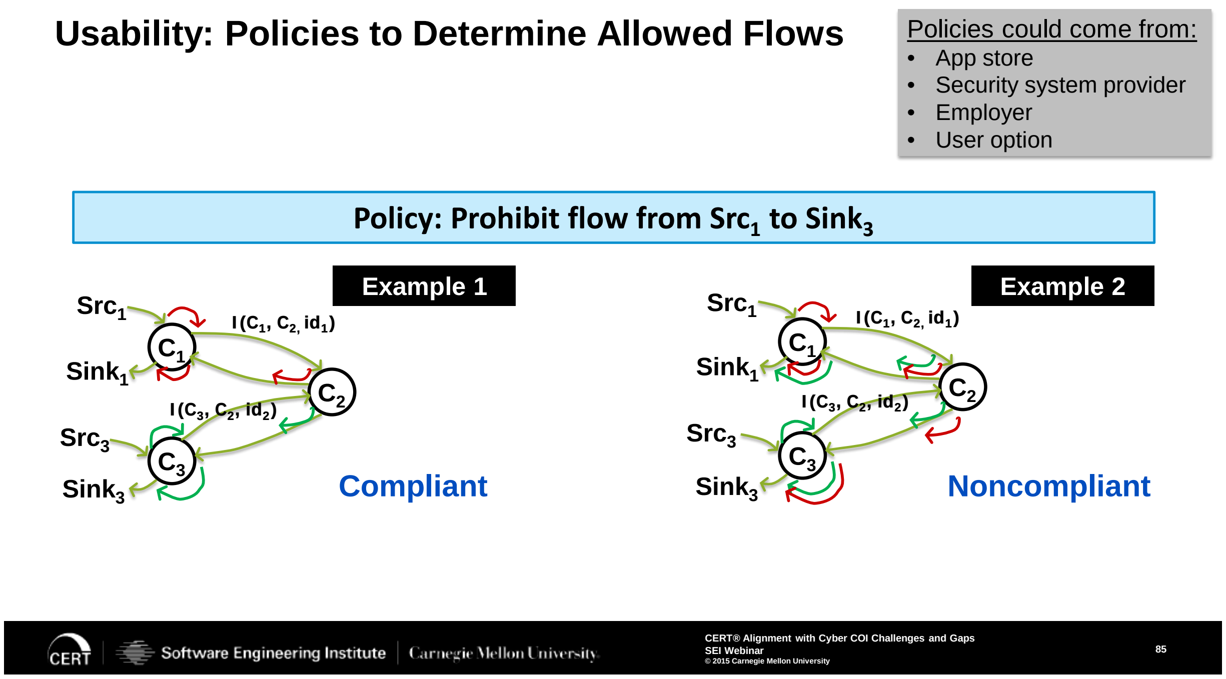
CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

84

**084 We envision that the two-phase analysis can be used as follows. An app store runs the phase-one analysis for each app that it has ahead of time. Then when a user wants to download a new app, the store would run the phase-two analysis and indicate new flows. There would be a fast response to the user since our phase-two is quite fast. We would-- for usability reasons, we think that policy guidance and enforcement would be helpful.

Policies could come from: App store Security system provider Employer User option



Usability: Policies to Determine Allowed Flows

Policies could come from:

- App store
- Security system provider
- Employer
- User option

Policy: Prohibit flow from Src_1 to $Sink_3$

CERT | Software Engineering Institute | Carnegie Mellon University

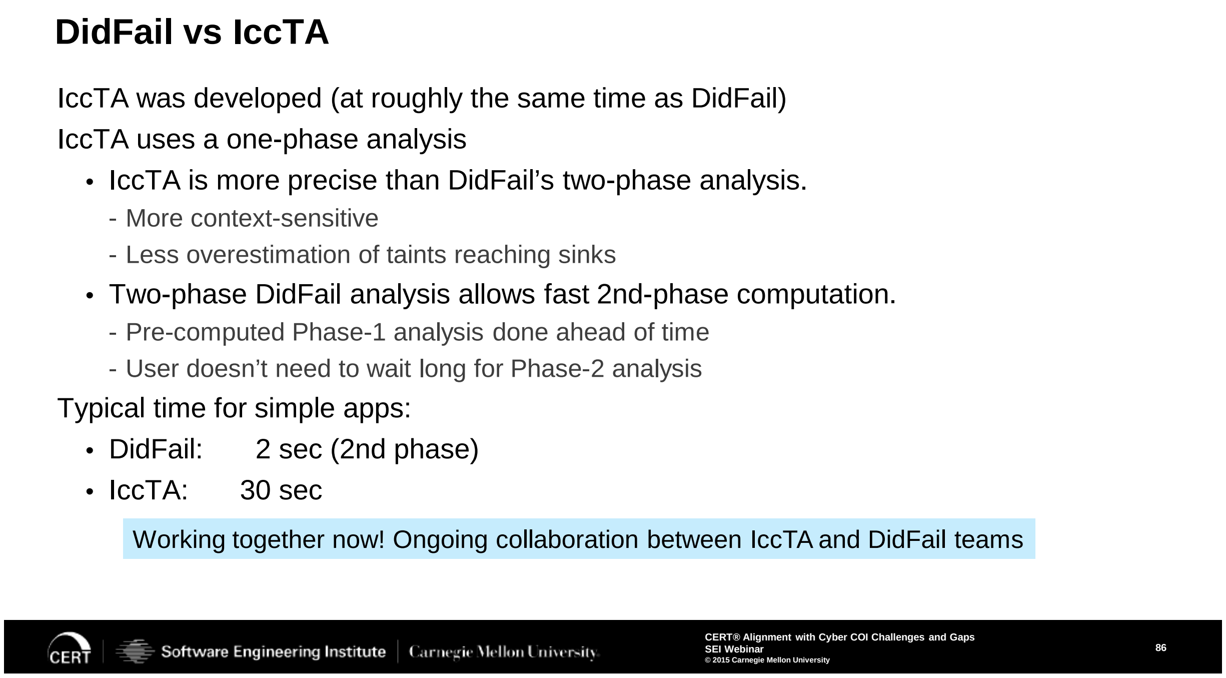
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

85

**085 We-- so for instance, policies could come from an app store, from a security system provider, from an employer. Or a user could set an option aided by an easy-to-use interface on their phone. An example policy is shown on this slide. For instance, there could be a policy prohibiting a flow from Source One to Sink Three. In the Example One set of apps on the left, you can see that the data flows are compliant. The Source One's taint is shown as a red arrow. And you can see that it doesn't reach the Sink Three.

However, the set of apps on the right in the Example Two set of apps is non-compliant. So, you can see that that Source One's red taint does indeed reach the Sink Three. And DidFail would be able to do analysis to determine if the set of apps was compliant or non-compliant with that type of policy.

DidFail vs IccTA



DidFail vs IccTA

IccTA was developed (at roughly the same time as DidFail)

IccTA uses a one-phase analysis

- IccTA is more precise than DidFail's two-phase analysis.
  - More context-sensitive
  - Less overestimation of taints reaching sinks
- Two-phase DidFail analysis allows fast 2nd-phase computation.
  - Pre-computed Phase-1 analysis done ahead of time
  - User doesn't need to wait long for Phase-2 analysis

Typical time for simple apps:

- DidFail: 2 sec (2nd phase)
- IccTA: 30 sec

Working together now! Ongoing collaboration between IccTA and DidFail teams

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

86

**086 An analyzer named "IccTA" was developed at roughly the same time as DidFail. IccTA uses a one-phase analysis for Android app sets. And it looks for taint flows. IccTA's analysis is more precise than DidFail's analysis. It's more context-sensitive, and it has less overestimation of taints reaching sinks.

However, our two-phase DidFail analysis allows very fast second-phase computation. The precomputed phase-one analysis gets done ahead of time. And then the user, when they want to install an app, they don't have to wait long for the phase-two analysis. A typical time for very simple apps is that DidFail takes two seconds for the second phase. And IccTA takes thirty seconds.

We are currently working with the IccTA team on a collaborative project. And we hope to develop an analyzer that has the best of both worlds where we still use the two-phase analysis with a very fast user response. But we hope to add a lot more precision to the analysis.

Installing DidFail

Installing DidFail

Main DidFail website

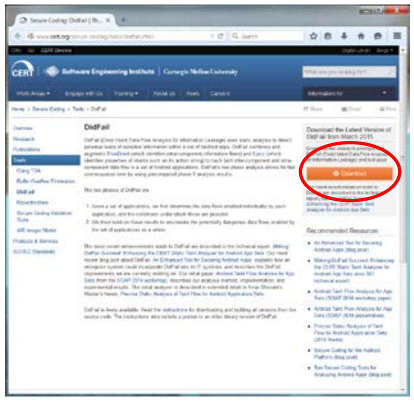
- http://www.cert.org/secure-coding/tools/didfail.cfm

Detailed install instructions are on the download website

- https://www.cs.cmu.edu/~wklieber/didfail/install-latest.html

There are 3 branches

- Static fields (Dec. 2014)
- Services and broadcast receivers (Dec. 2014)
- Improved DEX conversion (Nov. 2014)



Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar

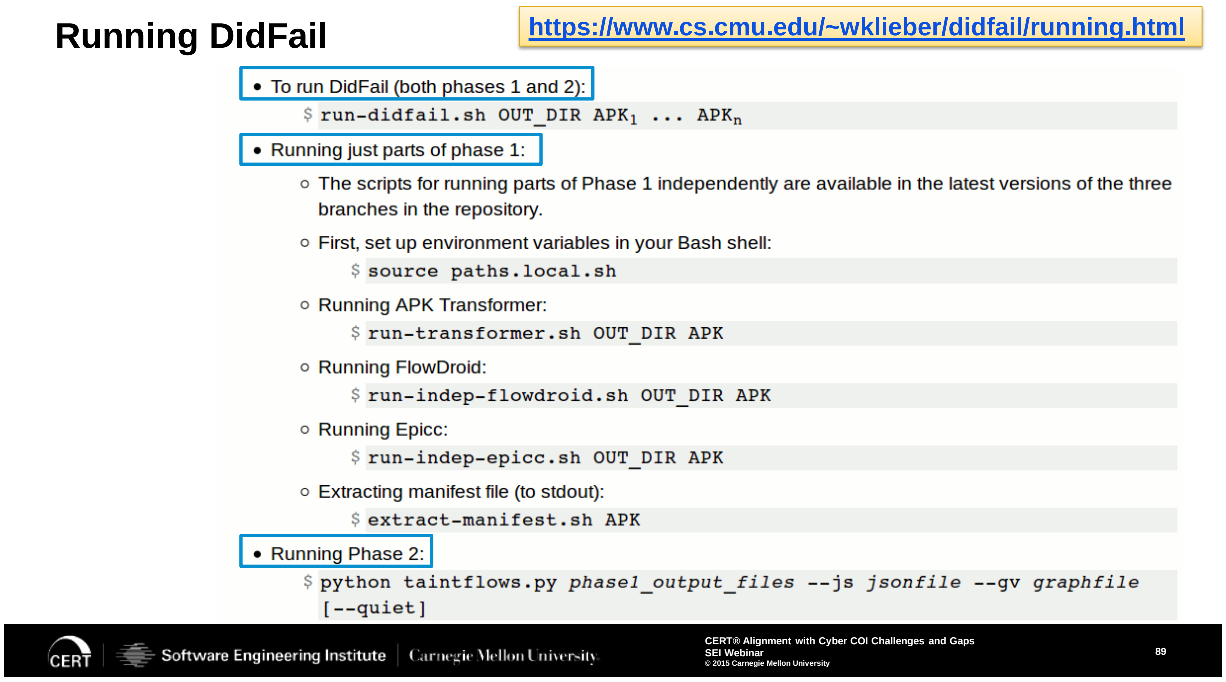
88

**088 In order to install DidFail, you can go to the main DidFail website which is on the CERT secure coding webpage at the URL listed on the slide. You can see a screenshot of that webpage on the slide. And if you click on that big orange button, you get to the install and download website.

There are three branches of DidFail. Two of them were mostly completed last December. They're the static fields analysis branch I mentioned earlier and the services and broadcast receivers branch which added the analysis of those two components to DidFail. In the future, we plan to integrate those two most recent branches and simply have an argument for running DidFail to determine whether you run the-- you use the static field analysis or not, since the static field analysis is a heavy user of memory and computation.

And then we have an earlier version which improved DEX conversion compared to our original DidFail. And that improved DEX conversion is incorporated in both of the most recent branches.

https://www.cs.cmu.edu/~ wklieber/didfail/running.html



Running DidFail

https://www.cs.cmu.edu/~wklieber/didfail/running.html

- To run DidFail (both phases 1 and 2):

```
$ run-didfail.sh OUT_DIR APK1 ... APKn
```

- Running just parts of phase 1:
  - The scripts for running parts of Phase 1 independently are available in the latest versions of the three branches in the repository.
  - First, set up environment variables in your Bash shell:
    ```
    $ source paths.local.sh
    ```
  - Running APK Transformer:
    ```
    $ run-transformer.sh OUT_DIR APK
    ```
  - Running FlowDroid:
    ```
    $ run-indep-flowdroid.sh OUT_DIR APK
    ```
  - Running Epicc:
    ```
    $ run-indep-epicc.sh OUT_DIR APK
    ```
  - Extracting manifest file (to stdout):
    ```
    $ extract-manifest.sh APK
    ```
- Running Phase 2:

```
$ python taintflows.py phase1_output_files --js jsonfile --gv graphfile
  [--quiet]
```

CERT | Software Engineering Institute | Carnegie Mellon University

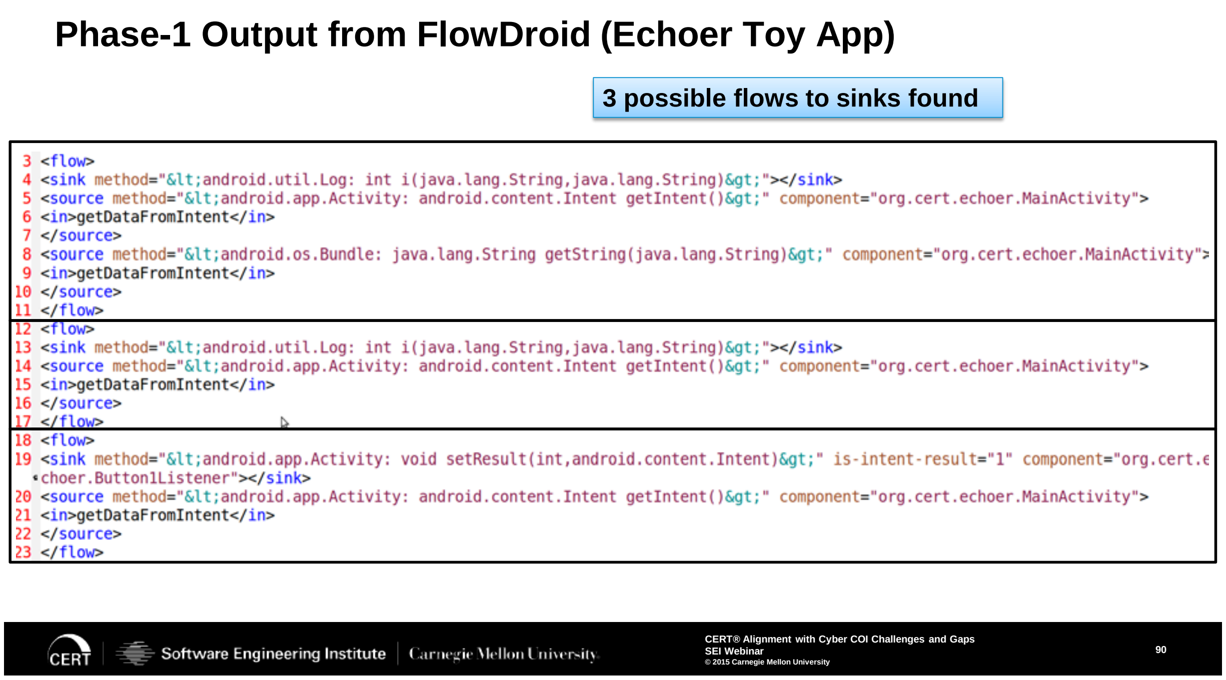
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

89

**089 There are three categories of running DidFail. And we have-- and you can see commands for running DidFail on this slide. The three types of commands you can use are the first one is to run both phases one and two of DidFail all together, or all with one command. You can also run just parts of phase one separately. For instance, you can run just the transformer. You can run just the modified FlowDroid tool. You can just do Epicc or simply extract the manifest file.

And you can also run phase two separately. After you've done the phase-one analysis on each app, you can look at possible taint flows for different sets of apps in combination.

Phase-1 Output from FlowDroid (Echoer T oy App)



Phase-1 Output from FlowDroid (Echoer Toy App)

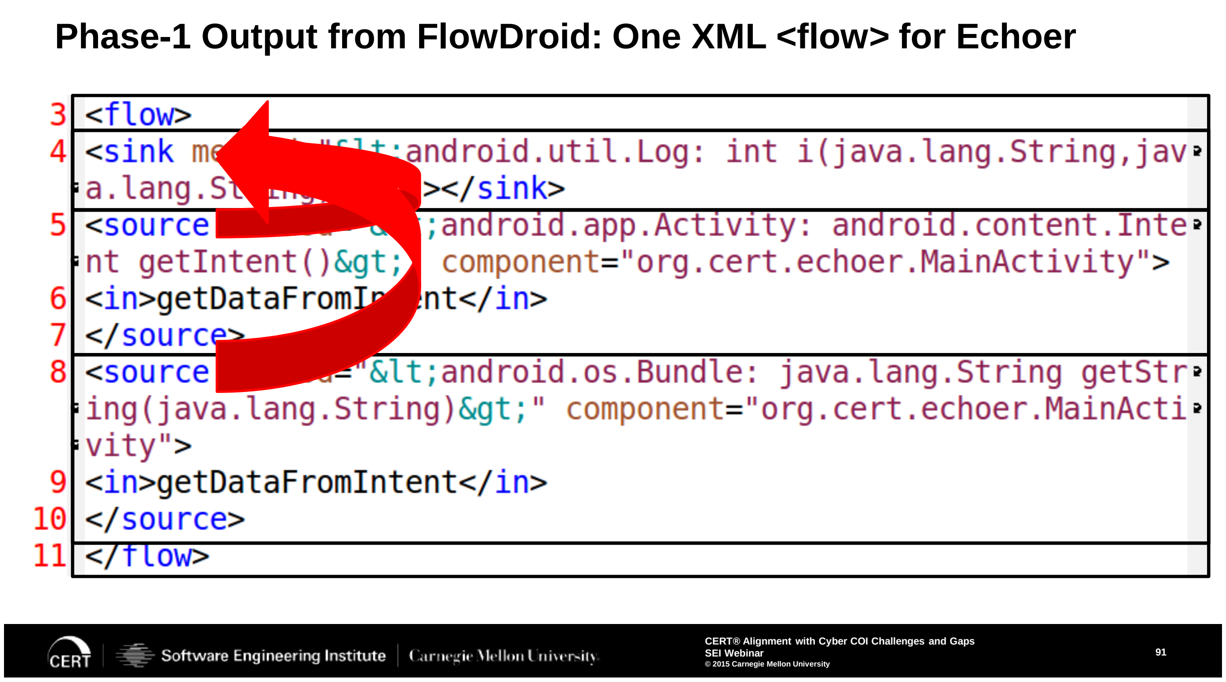
3 possible flows to sinks found

```
<flow>
<sink method="&lt;android.util.Log: int i(java.lang.String,java.lang.String)&gt;"></sink>
<source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
<in>getDataFromIntent</in>
</source>
<source method="&lt;android.os.Bundle: java.lang.String getString(java.lang.String)&gt;" component="org.cert.echoer.MainActivity">
<in>getDataFromIntent</in>
</source>
</flow>
<flow>
<sink method="&lt;android.util.Log: int i(java.lang.String,java.lang.String)&gt;"></sink>
<source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
<in>getDataFromIntent</in>
</source>
</flow>
<flow>
<sink method="&lt;android.app.Activity: void setResult(int,android.content.Intent)&gt;" is-intent-result="1" component="org.cert.e
choer.Button1Listener"></sink>
<source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
<in>getDataFromIntent</in>
</source>
</flow>
```

CERT
Software Engineering Institute | Carnegie Mellon University
CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University
90

**090 This slide shows output from FlowDroid for the Echoer toy app. Three possible flows to sinks were found where each flow indicates-- where each flow has one sink. We're going to zoom in on one of these flows. And you can see that the first line in the XML flow indicates the sink. In this case, it's a write to the log. That's followed by one or more sources. In this case, the source is a getIntent. So, the source was a received intent. And there's a second source for this flow as well.

Phase-1 Output from FlowDroid: One XML <flow> for Echoer



Phase-1 Output from FlowDroid: One XML <flow> for Echoer

```
<flow>
<sink me[illegible]"&lt;android.util.Log: int i(java.lang.String,java.lang.St[illegible]></sink>
<source [illegible];android.app.Activity: android.content.Intent getIntent()&gt;  component="org.cert.echoer.MainActivity">
<in>getDataFromI[illegible]nt</in>
</source>
<source [illegible]="&lt;android.os.Bundle: java.lang.String getString(java.lang.String)&gt;" component="org.cert.echoer.MainActivity">
<in>getDataFromIntent</in>
</source>
</flow>
```

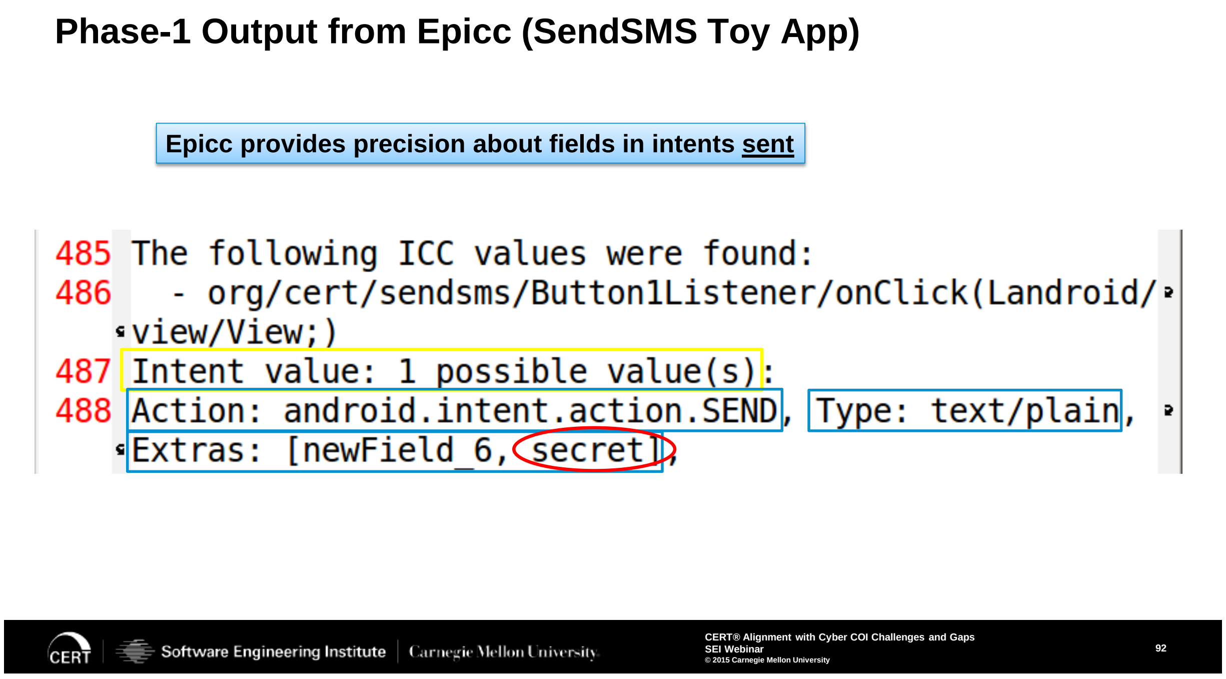
CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

91

**091 So, there's one tainted data flow that was found to be possible to the sink from the intent. And another flow was found to be possible from the other source.

Phase-1 Output from Epicc (SendSMS Toy App)



Phase-1 Output from Epicc (SendSMS Toy App)

Epicc provides precision about fields in intents sent

```
485 The following ICC values were found:
486   - org/cert/sendsms/Button1Listener/onClick(Landroid/
    view/View;)
487 Intent value: 1 possible value(s):
488 Action: android.intent.action.SEND, Type: text/plain,
    Extras: [newField_6, secret],
```

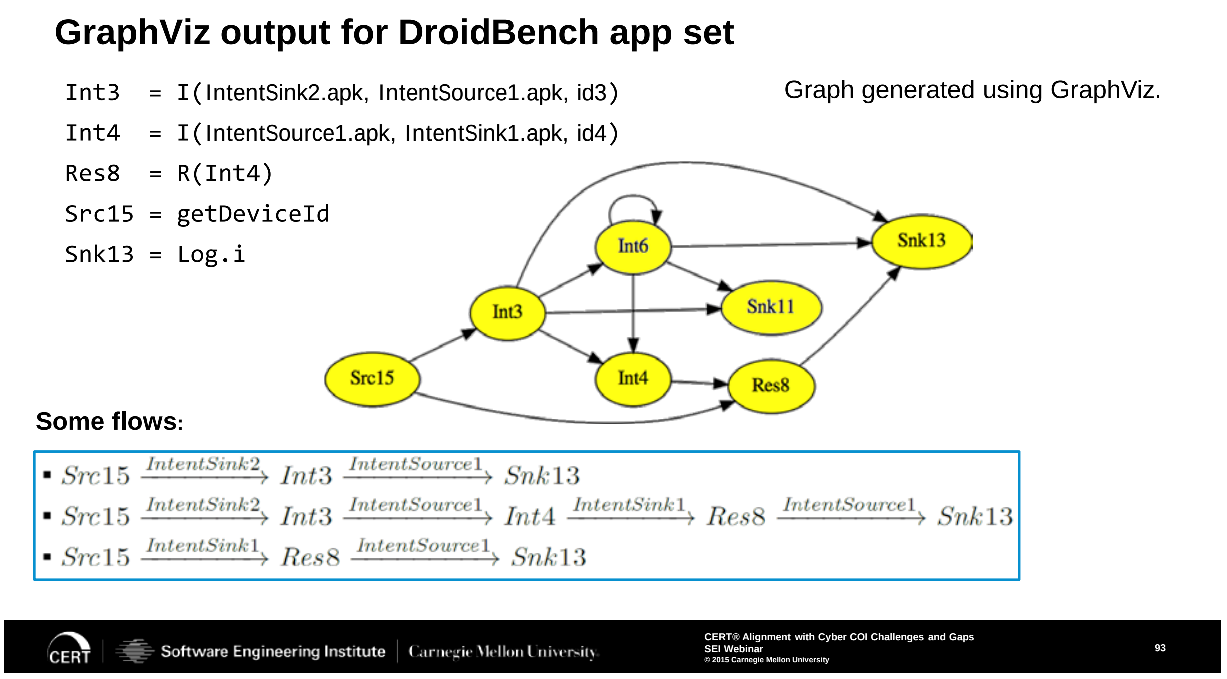
CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

92

**092 Here we have some of the output from Epicc. You can see the output lines start at 485. So, this is output for the SendSMS toy app from Epicc. So, Epicc provides precision about fields in intents which are sent. And what's found here is that there's only one set of possible values for the sent intent from this app. And those values are that the action string must be action.intent.action.SEND. The data type is text/plain. And there are two extras, the new_field and the secret, which for our example set of apps, that's what we used to send the tainted data.

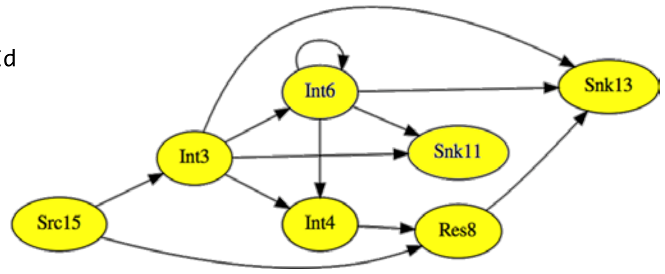
GraphViz output for DroidBench app set



GraphViz output for DroidBench app set

```
Int3  = I(IntentSink2.apk, IntentSource1.apk, id3)
Int4  = I(IntentSource1.apk, IntentSink1.apk, id4)
Res8  = R(Int4)
Src15 = getDeviceId
Snk13 = Log.i
```

Graph generated using GraphViz.



Some flows:

- $Src15 \xrightarrow{IntentSink2} Int3 \xrightarrow{IntentSource1} Snk13$
- $Src15 \xrightarrow{IntentSink2} Int3 \xrightarrow{IntentSource1} Int4 \xrightarrow{IntentSink1} Res8 \xrightarrow{IntentSource1} Snk13$
- $Src15 \xrightarrow{IntentSink1} Res8 \xrightarrow{IntentSource1} Snk13$

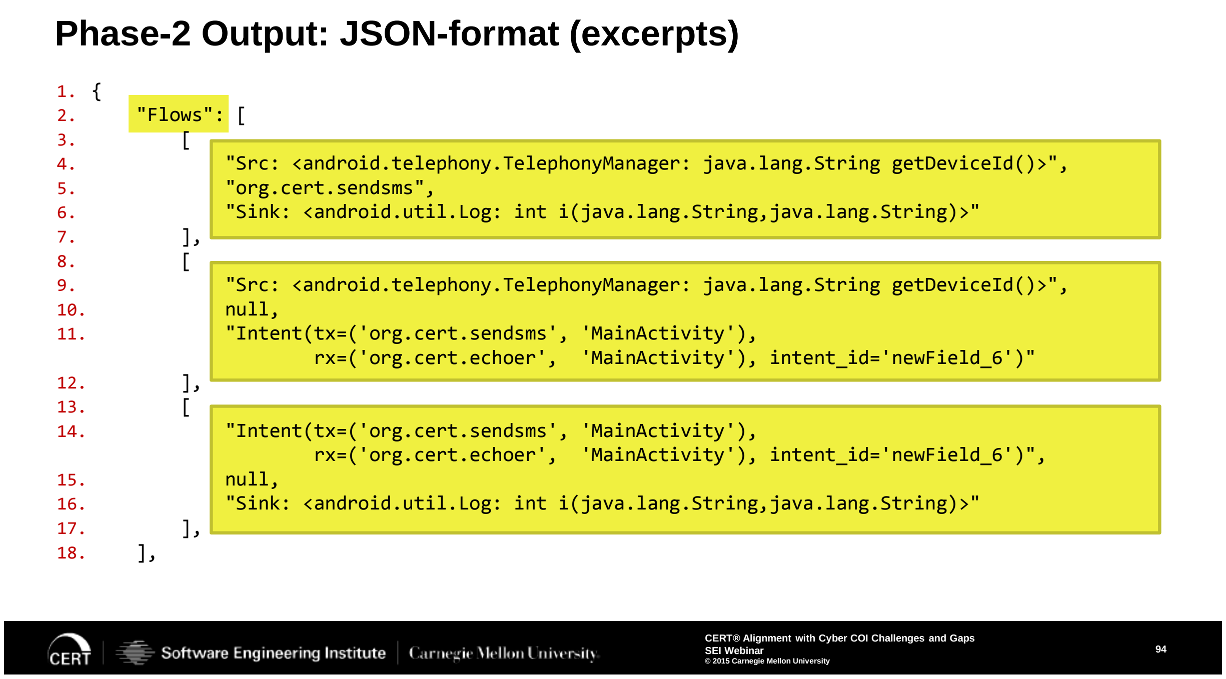
**093 Will Klieber: One of the
outputs from DidFail is a GraphViz file
which we can then feed to GraphViz
to generate a visualization of this
graph. So, in this case, all the paths
on this graph from a source to a sink
indicate possible flows of information.
So, for example, here we have Source 15,
which, as shown in the
legend, corresponds to this
getDeviceID function that reads your
phone's device ID. And then there's a
flow from that source to this intent,
Intent 3, which is an intent sent
from one app to another app. And
then from Intent 3, there's
another flow to Sink 13.

And Sink 13 is this log function.
And the log function is considered a
sensitive source because at least in
some older versions of Android, if an
app writes to a log, then other apps

can also read that log function. So, in a sense, logs are not private to an app. They're basically public information on the phone that any app can read. So, they're considered-- so that function is considered, at least in older versions of Android, to be a sensitive sink.

And then we can show these flows here. Like here's the flow from Source 15 to intent three to Sink 13.

Phase-2 Output: JSON-format (excerpts)



Phase-2 Output: JSON-format (excerpts)

```
{
    "Flows": [
        [
          "Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>",
          "org.cert.sendsms",
          "Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>"
        ],
        [
          "Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>",
          null,
          "Intent(tx=('org.cert.sendsms', 'MainActivity'),
                  rx=('org.cert.echoer',  'MainActivity'), intent_id='newField_6')"
        ],
        [
          "Intent(tx=('org.cert.sendsms', 'MainActivity'),
                  rx=('org.cert.echoer',  'MainActivity'), intent_id='newField_6')",
          null,
          "Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>"
        ],
    ],
```

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

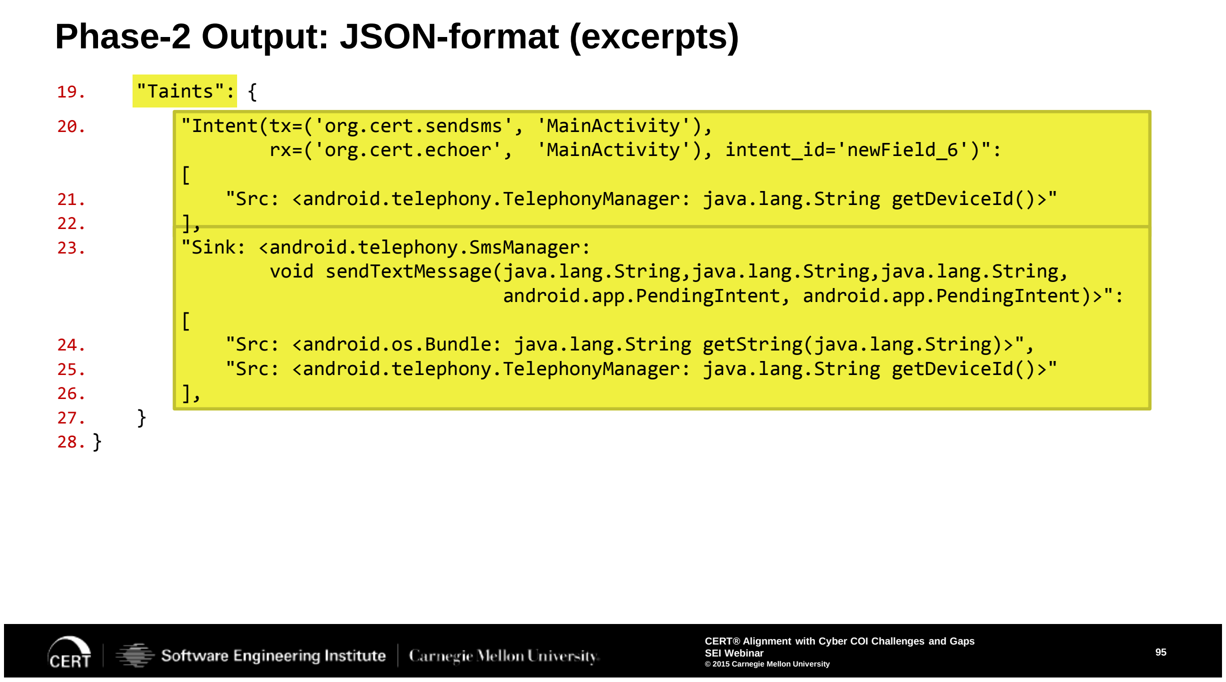
94

**094 In addition, we generate a list of flows in JSON format, JavaScript Object Notation. So, here you can see there's a flow within a component. So, we identify each flow by a tuple of three items, the source, the component in which the flow happens, and the sink. So, in this case, we have a sink-- the source is this getDeviceID function. And data flows from the source to this log function. And it happens within this one component.

Here, shown on this slide, we have a source-- a flow that involves an intent. So, our source is going to be the getDeviceID method of this TelephonyManager object. Now, when a source or a sink is an intent, we don't include a component in the second field because, in a sense, the flow doesn't really happen within a single component. So, it doesn't make sense to have one here.

And here the intent, we show that it's being transmitted by this SendSMS application. It's being received by the Echoer. And it has intent ID 6. Here, the final flow on this slide, the source is an intent. In fact, it's the same intent that was a sink on the previous flow. It's being sent from SMS to Echoer. And then Echoer is reading that information and writing it to this sink, this log function.

Phase-2 Output: JSON-format (excerpts)



Phase-2 Output: JSON-format (excerpts)

```
    "Taints": {
        "Intent(tx=('org.cert.sendsms', 'MainActivity'),
               rx=('org.cert.echoer',  'MainActivity'), intent_id='newField_6')":
        [
            "Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>"
        ],
        "Sink: <android.telephony.SmsManager:
               void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
                                    android.app.PendingIntent, android.app.PendingIntent)>":
        [
            "Src: <android.os.Bundle: java.lang.String getString(java.lang.String)>",
            "Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>"
        ],
    }
}
```

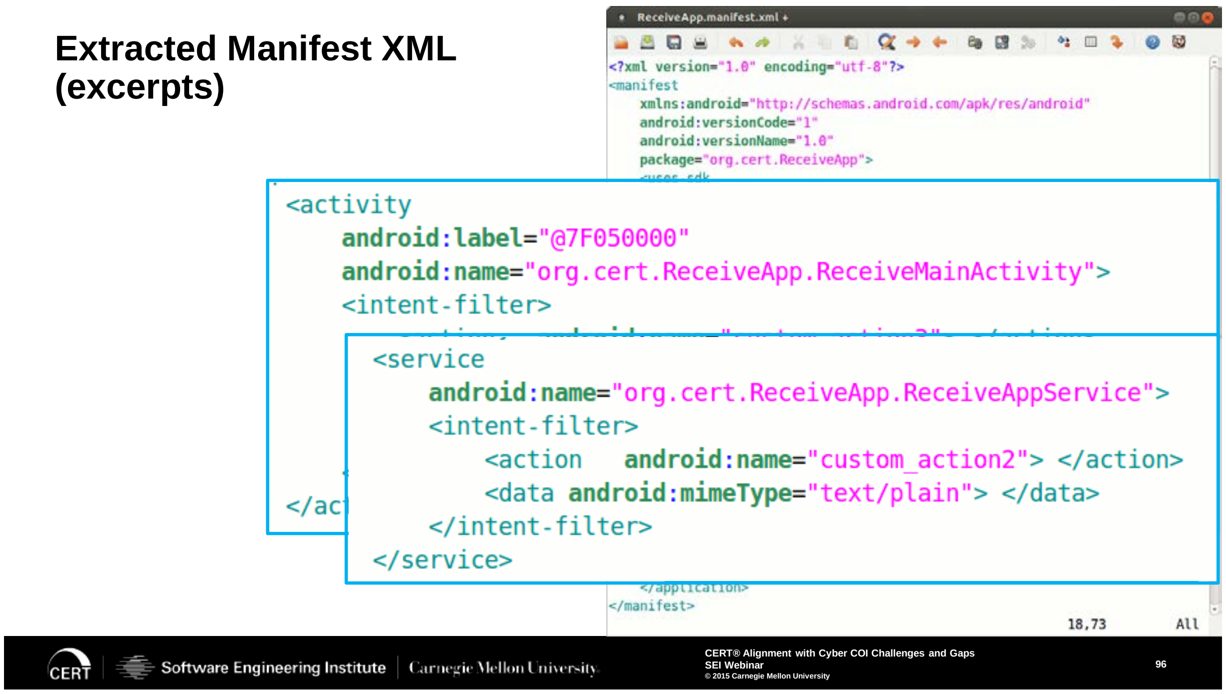
CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

95

**095 We also include the final taints of each sink. So, the first sink up here is this intent, the same intent that was shown on the previous slide. And it is tainted only with this getDeviceID function. Here we consider another sink, the sendTextMessage function, which is considered a sink here. And it is tainted with two sources, this os.Bundle.getString, which we don't really care too much about. But what we do care about is the second source, which is the getDeviceID method of this TelephonyManager.

Extracted Manifest XML (excerpts)


Extracted Manifest XML (excerpts)

ReceiveApp.manifest.xml +

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0"
    package="org.cert.ReceiveApp">
```

```
<activity
    android:label="@7F050000"
    android:name="org.cert.ReceiveApp.ReceiveMainActivity">
    <intent-filter>
</ac
```

```
<service
    android:name="org.cert.ReceiveApp.ReceiveAppService">
    <intent-filter>
        <action   android:name="custom_action2"> </action>
        <data android:mimeType="text/plain"> </data>
    </intent-filter>
</service>
```

```
    </application>
</manifest>
                                          18,73         All
```

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

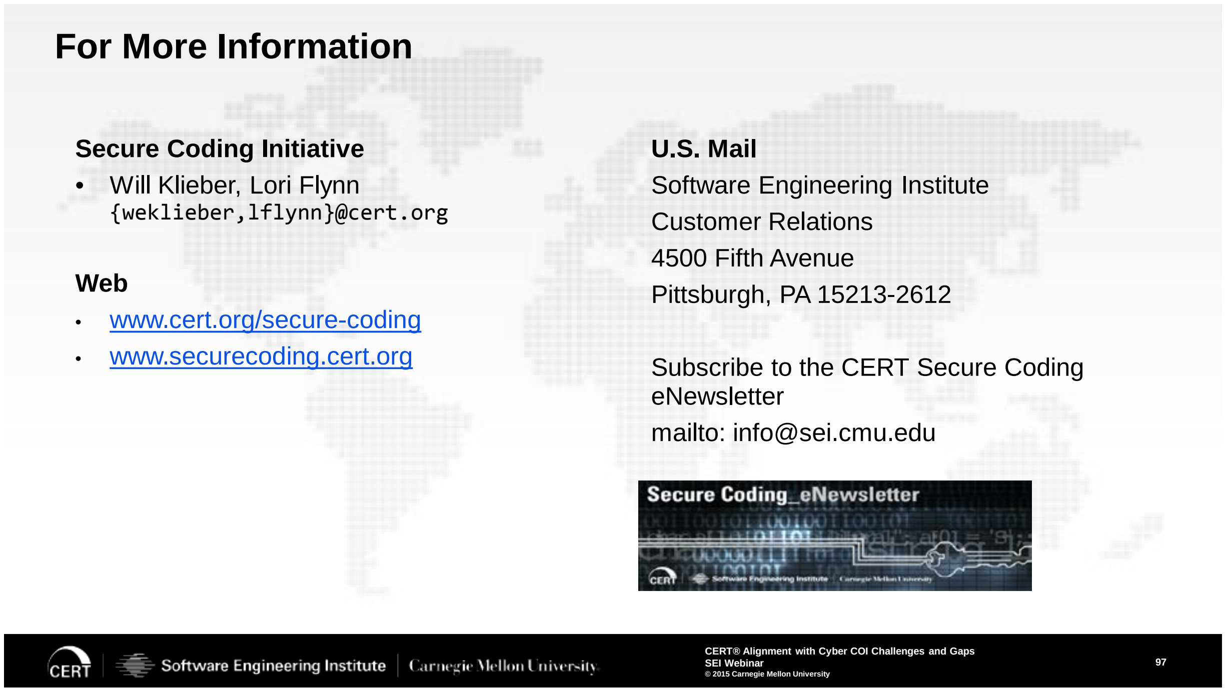
96

**096 We also produce excerpts from the-- we also extract the manifest XML file and use that-- we use information from the manifest in determining which possible components can receive implicit intents.

So, let's take a look here. I'm going to zoom in. So, we have this activity component. And we have the intent filter. So, this activity component can receive implicit intents that have this action string and this data type and this category.

And I'm going to zoom in on this part over here, services. So, this app also defines a service and gives an intent filter for it. It can receive implicit intents with the given action string and the given data MIME type.

For More Information



For More Information

Secure Coding Initiative

- Will Klieber, Lori Flynn
  `{weklieber,lflynn}@cert.org`

Web

- www.cert.org/secure-coding
- www.securecoding.cert.org

U.S. Mail

Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Subscribe to the CERT Secure Coding eNewsletter

mailto: info@sei.cmu.edu

Secure Coding_eNewsletter

CERT | Software Engineering Institute | Carnegie Mellon University

CERT® Alignment with Cyber COI Challenges and Gaps
SEI Webinar
© 2015 Carnegie Mellon University

97

**097 And this slide shows our contact information. Thank you for listening. Are there any questions?

Shane: All right, before we get to questions for Lori and Will, we've got a number of questions just rolling in through the day asking about whether the slides were available and if the archive is available. The event is being archived. An email will be sent out to all registrants within the next day or two with the location of the archive and how to access that. The slides are available now for anyone. If you just look at the webinar console, you'll see a files tab. And you can download all the presentation files today along with other work that CERT has done in cyber security.

So, one of the first questions for Lori
and Will asking from-- John wanted to
know: Will DidFail be available for
other operating systems?

Lori Flynn: DidFail is an Android-specific
analysis tool. It's created for
the Android lifecycle. It takes into
account the Android API calls,
Android sources and sinks, and
Android communication mechanisms
intents. So, at this point, I don't think
we would be able to modify DidFail
for other operating systems. But--

Shane: Would that make it a new
tool? Or is it just a matter of
stakeholders and funding to-- new
research? What would it take to draw
that out?

Lori Flynn: Definitely funding is
always good. And similar taint flow
analysis tools could be created for
other systems.

Shane: We had another question
come in. What about other platforms
like Linux? Could DidFail run on that?

Lori Flynn: No, DidFail is definitely
specific to the Android operating
system.

Shane: Okay, all right, let's get to
our next question here. Let me pull
them back up. And from Greg asking
about I believe a tool called
CyanogenMod has some privacy
enhancing capabilities. How would
you compare CyanogenMod to
DidFail? And is there any potential for
them to be used together? I guess

first of all, are you familiar with CyanogenMod?

Will Klieber: Mm-hmm.

Shane: Okay. Is there any potential for them to be used together?

Will Klieber: Yes. So, what CyanogenMod is is it enables existing Android devices to-- it enables the user to selectively enable or disable certain permissions, which can't be done on a stock Android phone. So, the way that you could use them together is you could see if you install certain apps, and there's a flow that you don't want to happen, what you can do is you can use CyanogenMod to disable permissions of these certain apps so as to disable that flow. So, for example, if you have one app that reads from a sensitive data source, and it communicates it to another app, then writes it to the Internet, then you can either disable the Internet functionality of the sender. Or you can disable the functionality of the app that reads the sensitive information so that it cannot read that sensitive information.

Shane: Okay. And just one last question we have in the queue is mentioning the SCALe method, which I know you guys are familiar with. How does that play into DidFail? Or is that a part of it? What is the relationship between the SCALe method, which I'm sure some of our viewers are not familiar with? How is that related to DidFail if at all?

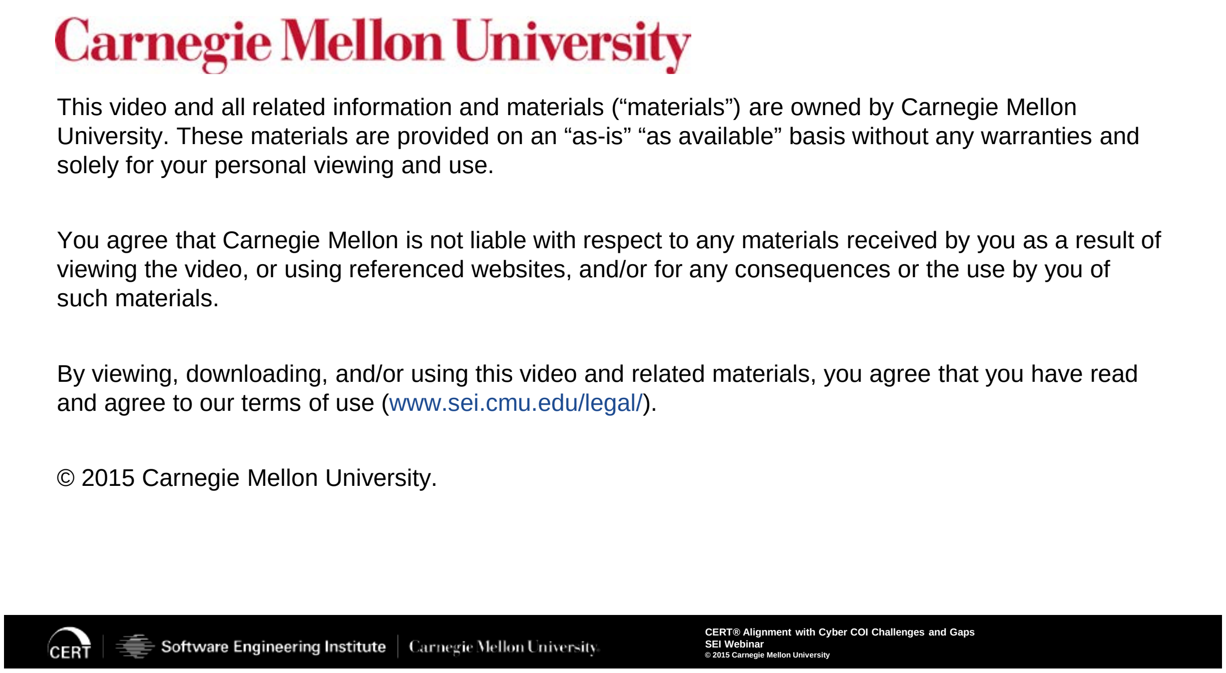
Lori Flynn: So, SCALe uses multiple analyzers on source code or binaries in order to diagnose coding flaws, coding flaws relevant to security that can be mapped to the CERT secure coding rules, to violations of the CERT secure coding rules. So, DidFail does indeed find a violation of a CERT secure coding rule which has to do with sensitive data not going to places it's not supposed to go to. However, currently DidFail is not fully integrated with the SCALe system, which has a web interface. And it's less-- the SCALe system is not so much geared toward tracing data flow as to analyzing particular diagnostics from multiple tools. Do you have anything to add?

Will Klieber: No, I think that covers it.

Lori Flynn: Okay.

Shane: Okay, that's all we have in the queue. So, Lori and Will, you guys are off the hot seat a little early. It will give us some time to set up for our final panel today, which will be our DevOps panel, which will be moderated by SEI CTO Dr. Kevin Fall. And we're going to have joining him will be Hasan Yasar and Joseph Yankel. So, again, thank you guys for your presentation today, very well done. And folks, we'll be back sharply-- or I guess, yeah. We'll be back at four oh five sharp to start the DevOps panel. So, look forward to starting it back up then.

Carnegie Mellon University



Copyright 2015 Carnegie Mellon University