Practical Precise Taint-flow Static Analysis for Android App Sets

IWSMA 2018

Will Klieber* (presenting),
Lori Flynn*,
Will Snavely*,
Michael Zheng*

*Software Engineering Institute (SEI)
Carnegie Mellon University
Pittsburgh, PA, USA

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-0979

Introduction

Goal: Detect malicious Android apps that leak sensitive data.

- E.g., leak contacts list to marketing company.
- Android allows users to grant/deny permissions, but flows of information may not be apparent to users.

Apps can collude to leak data.

- Evades detection if only analyzed individually.

Previous work that we build on:

- FlowDroid (TU Darmstadt, …): Analyzes source-to-sink flows in each component of an app individually.
- DidFail (SEI): Builds on FlowDroid to analyze flows that involve multiple apps.

Contribution of this paper:

- Make DidFail more precise (reduces number of false positives).

Outline of this talk

1. First half: Review of how DidFail works.
2. Second half: Contribution of this paper (improvement to DidFail algorithms).

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`")

Introduction

Taint Analysis tracks the flow of sensitive data.

- Can be static analysis or dynamic analysis.
- DidFail's analysis is static.

DidFail builds 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.**

DidFail

We developed a static analyzer called "**DidFail**" ("Droid Intent Data Flow Analysis for Information Leakage").

- Finds flows of sensitive data across app boundaries.
- Source code and binaries available at:
  `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.

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,

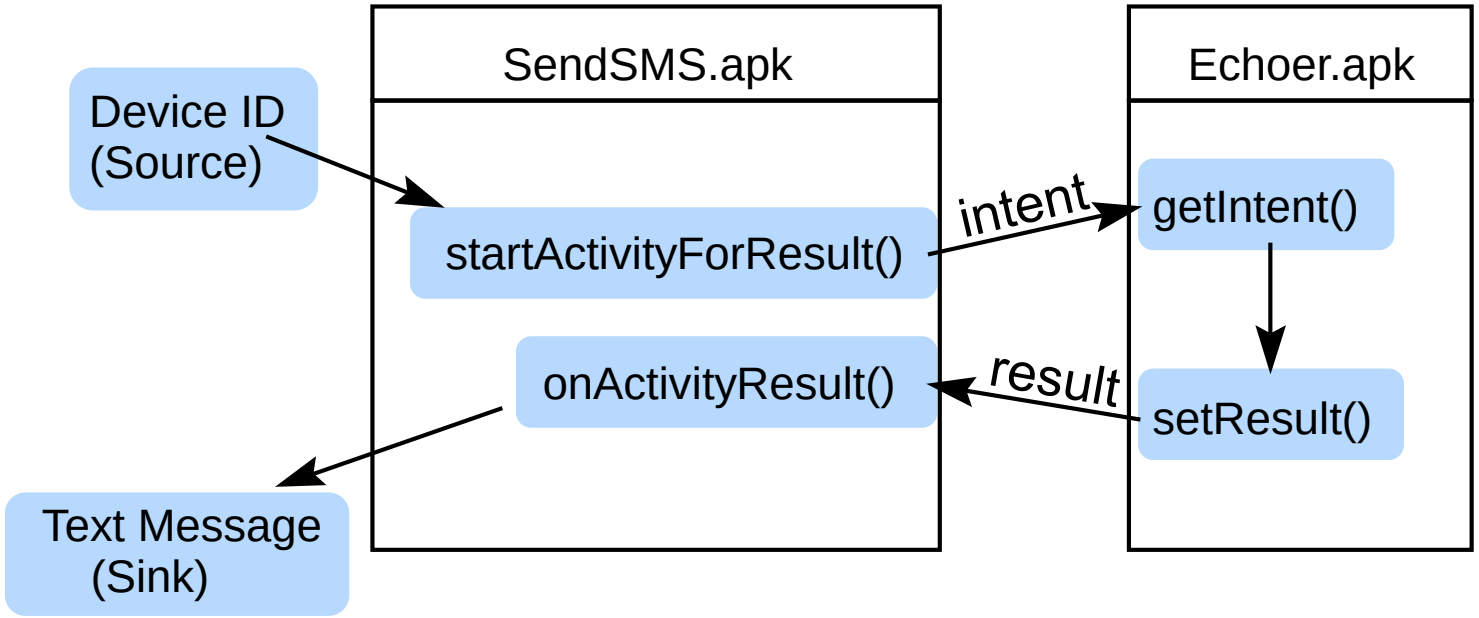
- **Sources**: Device ID, contacts, photos, location (GPS), intents, etc.
- **Sinks**: Internet, outbound text messages, file system, intents, etc.

Definition. A *taint flow* is data flow from a source to a sink.

Definition. A *full taint flow* is a taint flow where neither the source nor the sink are intents.

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*.

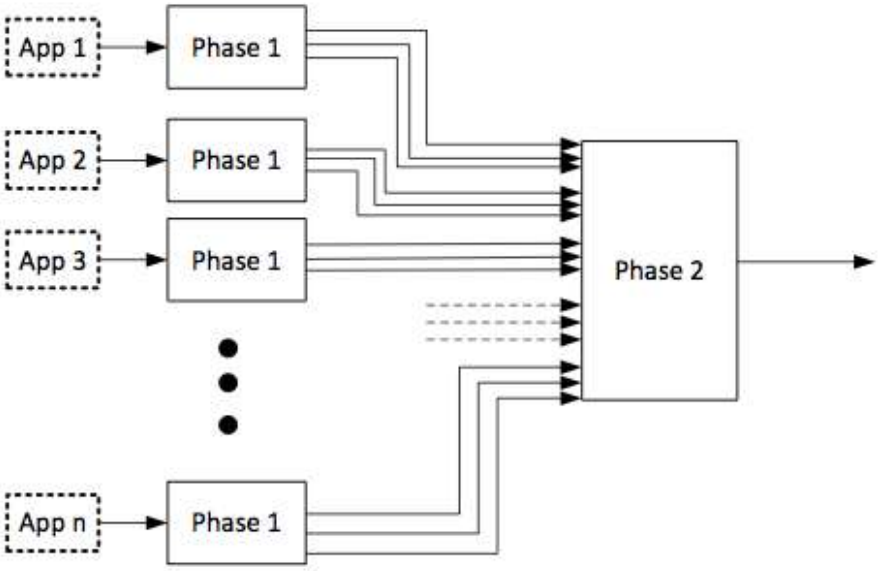
Analysis Design

Phase 1: Each app analyzed once, in isolation.

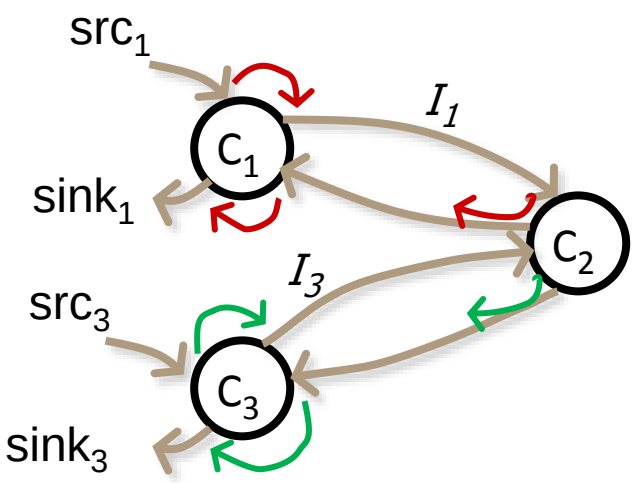
- **FlowDroid:** Finds tainted dataflow from sources to sinks.
  - Received intents are considered sources.
  - Sent intent are considered sinks.
- **Epicc:** Determines properties of intents.
- Each intent-sending call site is labelled with a unique *intent ID*.

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.



Running Example



Three components: C_1, C_2, C_3.

C1 = SendSMS

C2 = Echoer

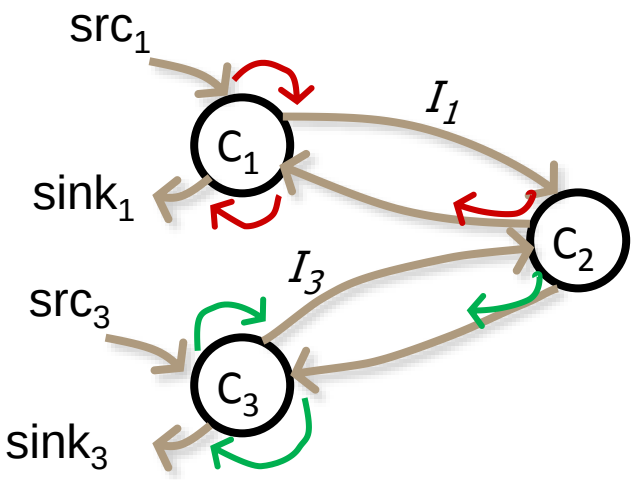
C3 is similar to C1

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.

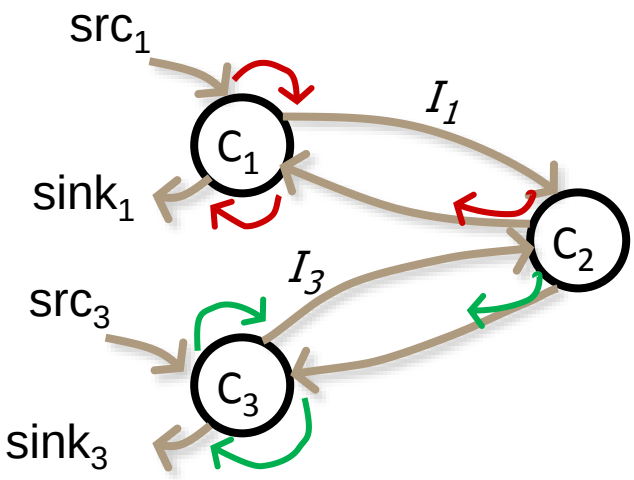
Running Example



Notation:

- "$src \xrightarrow{C} sink$": Flow from *src* to *sink* in C.
- "$I(C_{\text{TX}}, C_{\text{RX}}, id)$": 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.

Running Example



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

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

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

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

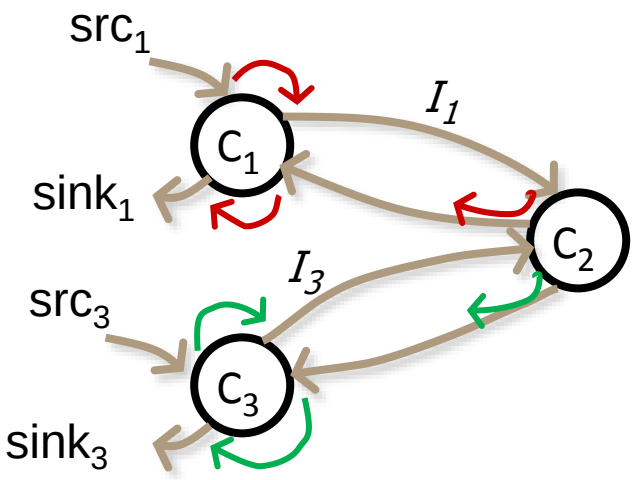
$$I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$$

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

Notation:

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\text{TX}}, C_{\text{RX}}, id)$": 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.

Running Example



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

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

Final Sink Taints:

- $T(sink_1) = \{src_1\}$
- $T(sink_3) = \{src_3\}$

Notation:

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\text{TX}}, C_{\text{RX}}, id)$": 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.

Phase-1 Flow Equations

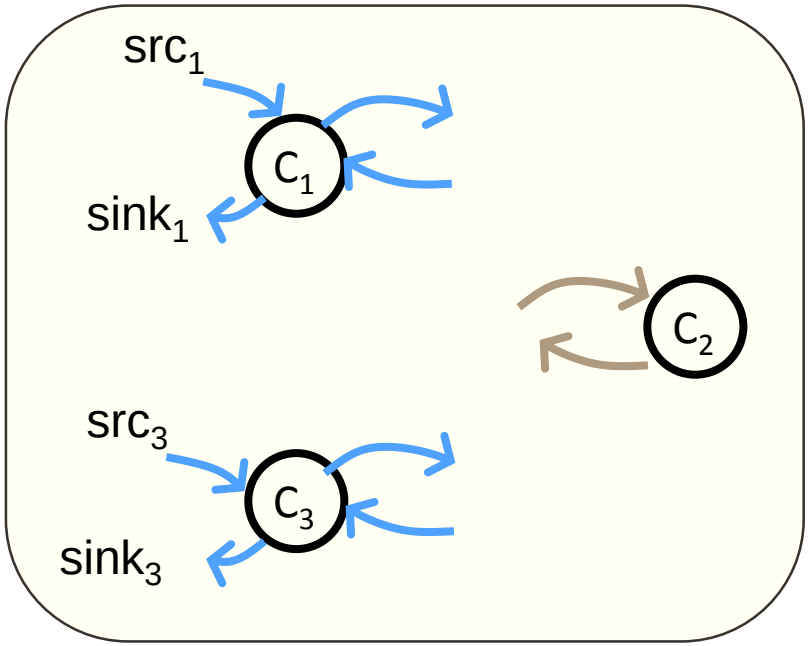
Analyze each component separately.

Phase 1 Flow Equations:

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

$$I(*, C_2, *) \xrightarrow{C_2} R(I(*, C_2, *))$$

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

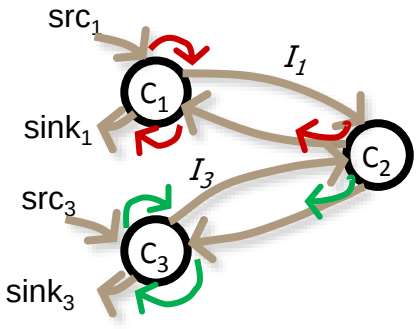


Notation

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

Phase-2 Flow Equations

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



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:

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

$$I(C_1, C_2, id_1) \xrightarrow{C_2} R(I(C_1, C_2, id_1))$$
$$I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$$

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

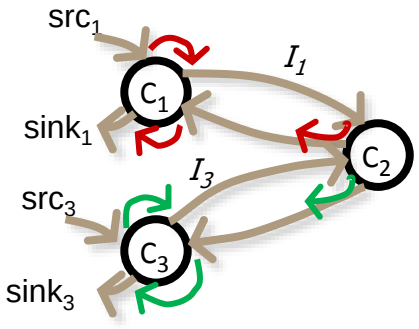
Notation

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\text{TX}}, C_{\text{RX}}, id)$": 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.

Phase-2 Taint Equations

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



Phase 2 Flow Equations:

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

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

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

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

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

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

Phase 2 Taint Equations:

$$T(src_1) \subseteq T(I(C_1, C_2, id_1))$$

$$T(R(I(C_1, C_2, id_1))) \subseteq T(sink_1)$$

$$T(I(C_1, C_2, id_1)) \subseteq T(R(I(C_1, C_2, id_1)))$$

$$T(I(C_3, C_2, id_1)) \subseteq T(R(I(C_3, C_2, id_3)))$$

$$T(src_3) \subseteq T(I(C_3, C_2, id_3))$$

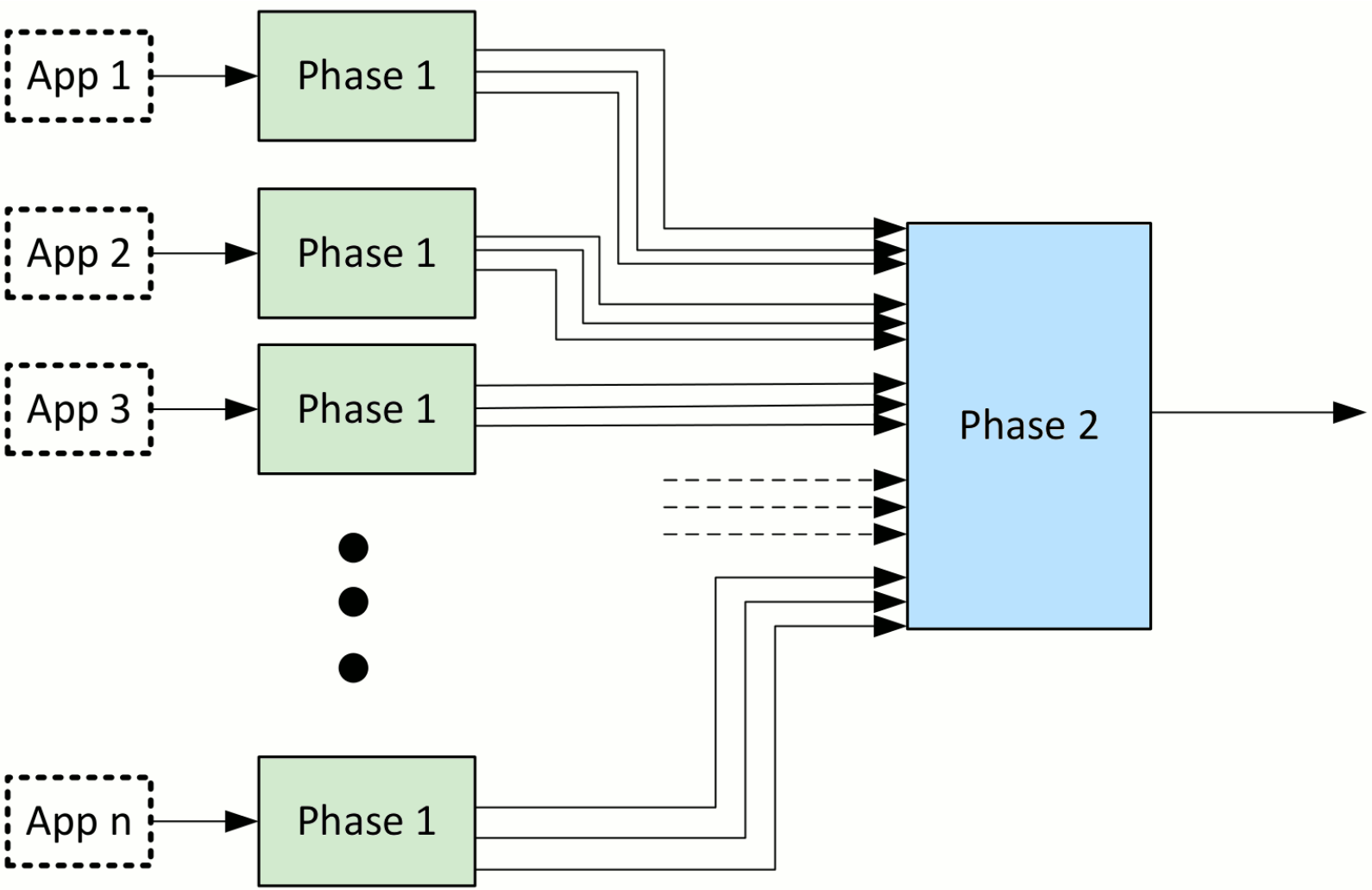
$$T(R(I(C_3, C_2, id_3))) \subseteq T(sink_3)$$

Notation

- "$src \xrightarrow{C} sink$": Flow from src to $sink$ in C.
- "$I(C_{\text{TX}}, C_{\text{RX}}, id)$": 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.

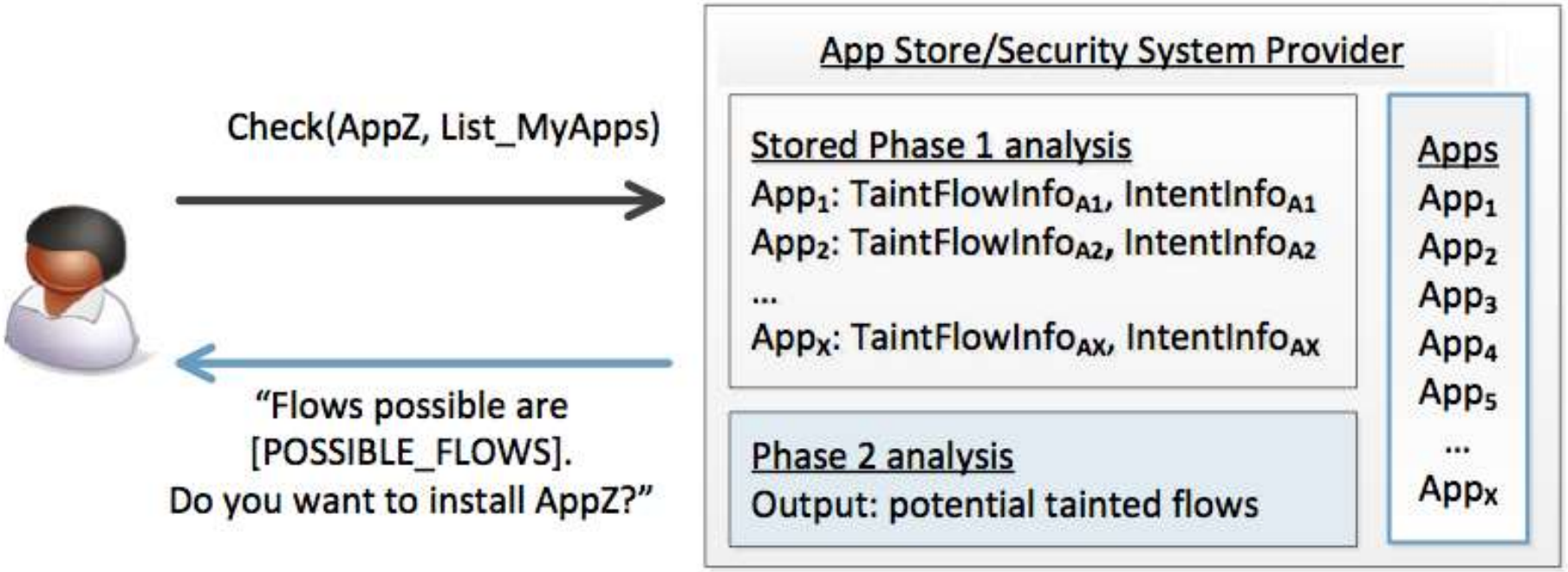
Then, solve.

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



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.



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

Motivating example for this paper's work

Let's consider an app that scans QR codes. Suppose it has two different behaviors, depending on action string of received intent:

- Action "`FollowURL`": Uses camera to read QR code, interprets it as a URL, and opens the URL in a web browser.
  - Flow: Camera → Internet
- Action "`ScanOnly`": Uses camera to read QR code, and returns it to the calling app.
  - Flow: Camera → IntentResponse

Original DidFail includes both of the above flows, even for a set of apps in which only ScanOnly is used. False positive!

For example, consider a Bitcoin wallet app that scans a QR code representing a private key.

Parameterized Component Summaries

Pseudocode for QR Scanner:

```
ir1 = getIntent();
if (ir1.action = "FollowURL"){
  visitURL(scanQR());
} else {
  sendIntentResult(scanQR());
}
```

Flow Equations

$$camera \rightarrow Internet,\ act = "FollowURL"$$

$$camera \rightarrow R(ir_1),\ act \neq "FollowURL"$$

Boolean formulas

- Common situation in apps:
  Branch on *action string* of received intent.
- Annotate each Phase-1 flow with a boolean formula.
  The formula indicates the **condition** under which the flow can happen.
- Atomic propositions: string equality between a property of the received intent and a string constant.

BF ::= AP
| BF ∧ BF
| BF ∨ BF
| ¬ BF
| true
| false

Modification to Phase 1

We use a simple path-sensitive intra-procedural flow analysis to compute the condition formula for each phase-1 flow.

Conservative approximation: Given a sink API call site *s*, we consider an associated flow from a source to *s* to be possible exactly when *s* is reachable.

Example 1:

```
x = readFile(...);
if (action == “foo”) {
    sendToInternet(x);
}
```

condition: *action* = “foo”

Example 2 (false positive):

```
x = null;
if (action == “foo”) {
    x = readFile(...);
}
sendToInternet(x);
```

condition: True

Modification to Phase 1

- Goal: For each sink API call site *s*, compute a condition formula *reach(s)* that represents the condition under which *s* is reachable.
- We use a simple dataflow analysis to identify which uses of local variables must refer to the received intent's action string (i.e., the result of `getIntent().getAction()` in an `Activity` context).
- We define *cond(p,s)* for all edges *(p,s)* in the control-flow graph as follows:
  - If *p* has the form "if (*e*) goto s_T else goto s_F" where *e* is an equality between a string constant *c* and the received intent's action, we define:
    - *cond(p, s_T)* = "`act` = *c*"
    - *cond(p, s_F)* = "`act` ≠ *c*"
  - If there is an unconditional edge from *p* to *s*, then *cond(p,s)* = true.
- Define *reach(s)* as the solution (least fixpoint) of:

$$reach(s) = \bigvee_{p \in preds(s)} reach(p) \wedge cond(p,s)$$

$$reach(entry) = \texttt{true}$$

where *preds(s)* is the set of predecessors of *s* in the control-flow graph.

Simplification to Canonical Form

We simplify the formulas using the following identities:

- if $c_1 \neq c_2$, $(act = c_1 \wedge act = c_2)$ is equivalent to `false`
- if $c_1 \neq c_2$, $(act \neq c_1 \vee act \neq c_2)$ is equivalent to `true`
- if $c_1 \neq c_2$, $(act = c_1 \wedge act \neq c_2)$ is equivalent to $act = c_1$
- if $c_1 \neq c_2$, $(act = c_1 \vee act \neq c_2)$ is equivalent to $act \neq c_2$

After simplification, each formula has one of the following forms:

- the constant true,
- the constant false,
- a disjunction of equalities, or
- a conjunction of disequalities.

Modification to Phase 2

Recall that in phase 2, all the phase-1 flows are instantiated for all possible sender/receiver pairs.

We modify phase 2 by deleting any instantiated flow whose condition formula is falsified by all possible action strings of the associated intent:

- For a flow of the form $\left(src \xrightarrow{C} sink, \varphi\right)$, where *src* is not an intent, we delete the flow if all possible intents receivable by *C* (including intents sent by the OS) have an action string that falsifies φ.
- For a flow of the form $\left(I(C_1, C_2, id) \xrightarrow{C_2} sink, \varphi\right)$, we delete the flow if φ is falsified by all possible action strings of intents that are sent at program location *id*.

Performance

Unfortunately we don't yet have a working implementation to get experimental results.

Based on theoretical arguments, we expect phase 2 to still be fast. There are two main possible slowdowns:

- The input to phase 2 may be larger (than in the original DidFail) because a single phase-1 flow in original DidFail may correspond to multiple flows (differing only in boolean formula).
  - Given a flow from original DidFail, the number of additional corresponding flows is bounded by the number of comparisons of the received intent's action string to distinct string constants.
- Detecting impossible phase-2 flows consumes time. The time to process a flow $\left(I(C_1, C_2, id) \xrightarrow{C_2} sink, \varphi\right)$ is $O(|\varphi| * |PossActs(id)|)$ where *PossActs(id)* denotes the possible action strings for *id*.

Conclusion

Problem: Colluding apps, or a combination of a malicious app and leaky app, can use intents (messages sent to Android app components) to extract sensitive or private information from an Android phone.

Goal: Precisely detect tainted flows across multiple Android components from sensitive sources to restricted sinks.

Two-phase DidFail analysis:

- Phase 1: Each app analyzed once, in isolation
  - Examine flow of tainted data from sources to sinks (including intents)
  - Examines intent properties to match senders and receivers
- Phase 2: For a particular set of apps
  - Instantiate and solve taint flow equations
  - **Fast** (due to pre-computation in phase 1)

New contribution of this paper: Increase precision of phase-1 summaries, to reduce the number of false positives, without compromising soundness.