Practical Precise Taint-flow Static Analysis for Android App Sets

William Klieber
Carnegie Mellon Univ, Software Engineering Institute
Pittsburgh, Pennsylvania
weklieber@cert.org

Lori Flynn
Carnegie Mellon Univ, Software Engineering Institute
Pittsburgh, Pennsylvania
lflynn@cert.org

Will Snavely
Carnegie Mellon Univ, Software Engineering Institute
Pittsburgh, Pennsylvania
lflynn@cert.org

Michael Zheng
Carnegie Mellon Univ, Software Engineering Institute
Pittsburgh, Pennsylvania
mzheng@cert.org

ABSTRACT

Colluding apps, or a combination of a malicious app and leaky app, can use intents (messages sent to Android app components) to exfiltrate sensitive or private information from an Android phone. This paper describes a novel static analysis method "Precise-DF" to detect taint flow in Android app sets (including flows involving multiple apps) that is precise, fast, and uses relatively little disk and memory space. Precise-DF re-uses the fast modular analysis of the DidFail static analysis tool, and adds context and therefore precision with parameterized summaries of potential data flows. We added Boolean formulas to DidFail's flow equations, to record conditions of control flow paths relevant to possible taint flows. The method that we have refined (a modular analysis with parameterized summaries of flow of sensitive information) is generally applicable to the class of problems involving taint flow analysis for software systems that communicate by message passing. This paper also describes how an enterprise architecture could use Precise-DF to analyze and enforce compliance with dataflow policies.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Automated static analysis**; **Software libraries and repositories**; **Formal software verification**; *Software notations and tools*; Software testing and debugging;

KEYWORDS

static analysis, Android, taint flow

ACM Reference Format:
William Klieber, Lori Flynn, Will Snavely, and Michael Zheng. 2018. Practical Precise Taint-flow Static Analysis for Android App Sets . In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany*. ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3230833.3232825

ACM ISBN 978-1-4503-6448-5/18/08…$15.00
https://doi.org/10.1145/3230833.3232825

1 INTRODUCTION

Sensitive or private information can be exfiltrated from an Android phone by colluding apps, or by a combination of a malicious app and a leaky app. [3, 13] The goal of our work is to precisely detect (i.e., with few false positives) malicious exfiltration of sensitive information from an Android phone, in a practical time and memory bound. We are focusing on false positives (precision) since they were a major obstacle faced in previous work with the DidFail tool [13]. To this end, we developed a new method for addressing the class of problems involving taint flow analysis for software systems that communicate by message passing. Our method is not expected to have any impact on false negative rates. We believe true positives detected by previous work will still be detected with the proposed modifications applied. Moreover, the proposed method does not detect any additional varieties of data flows, and therefore is not expected to detect issues that previous work missed.

Taint analysis aims to detect undesired flows from data sources to data sinks. We define sources as external resources (i.e., external to the app) from which data is read and sinks as external resources to which data is written. Tainted data is derived from a sensitive source. Static analysis of sensitive data flow between Android apps is challenging because there are many exit/entry points for control and data flow in the system, so trying to analyze potential flows across apps results in a combinatorial explosion.

Android Architecture. Android apps have four types of components: *activities*, which define a user interface; *services*, which perform background processing; *content providers*, which store and share data using a relational database; and *broadcast receivers*, which can receive broadcast messages from other applications. The primary method for inter-component communication, both within and between applications, is via *intents*. The app's manifest file specifies filters that are used by the system to determine if the app is eligible to receive a particular intent, using Android rules for matching filters to content of various intent fields. A component may also send an intent to itself, addressing itself either explicitly, or implicitly by setting its intent filter so that it can receive the intent it sent.

1.1 Related Work

Dynamic analysis tools execute code to analyze it. It is rarely possible to execute all possible execution paths with all possible combinations of inputs, although fuzzing and concolic analysis techniques are used to improve test coverage. An intent fuzzer [20] has been used to generate intents that crash apps. Availability of analysis time, disk space, and memory space are among the limiting factors for how much dynamic analysis can be done. Researchers found that the Google Bouncer (a tool used by the Google Play Store) does dynamic runtime analysis of an Android app in an emulated Android environment for 5 minutes (for each app) [15]. TaintDroid [5] is a realtime dynamic analysis tool that requires firmware modification. It functions by analyzing dataflow and detecting and preventing data with particular attributes (data marked “tainted” from particular sources) from reaching particular monitored sinks, in realtime.

Static analysis tools analyze code without executing it, and analyses often attempt to cover all possible execution paths and input. Defect findings may be wrong (false positives) and manual examination of defect warnings is time-consuming (expensive), so precision rates of the tools are important. Some taint-flow tools, like FlowDroid [11], only analyze single Android apps. IccTA [14] detects taint flows across multiple Android apps and is highly precise, but takes a large amount of time and/or memory (and sometimes will fail) because it analyzes app sets in a single phase. The StubDroid [1] method summarizes Java libraries, not Android intents, so the StubDroid method is one that could be used along with the method described in this paper.

COVERT [19] solves a problem that is similar to the one solved by Precise-DF but different in that COVERT looks specifically for inter-app communication that can be used to circumvent the Android permission system, whereas Precise-DF identifies all source-to-sink flows of sensitive information, regardless of the permissions of the involved apps. LetterBomb [12] automatically generates exploits for three types of inter-component communication vulnerabilities: interprocess denial of service, cross-application scripting, and fragment injection.

Our new algorithm builds on DidFail [13], so extensive detail about DidFail is provided in Section 2.

2 DIDFAIL ALGORITHM EXTENDED DETAIL

This section is a simplified excerpt from [13], intended to provide detailed understanding of parts of the algorithm used by (and in some ways modified by) Precise-DF.

DidFail [13] is a modular taint flow static analysis tool for Android app sets, that analyzes both inter-component and intra-component dataflow. It combines and augments the FlowDroid and Epicc [16] analyses to determine tainted flows both within and across applications. In DidFail’s first phase, static analysis is performed on Android apps. Determination is made of which dataflows are enabled individually by each app and the conditions under which these flows are possible. In the second phase, results of the first phase are used to enumerate the potential taint flows enabled by a particular set of applications. DidFail enables an organization (e.g., enterprise, app store, or security system provider) to pre-analyze apps, so that the analysis for potential dataflow problems (within the set of apps on the phone) is fast when a user requests to install a new app. Phase 1 of DidFail can be performed on one application at a time and, once completed, does not need to be run again. Phase 2 of DidFail (app set analysis) typically takes only seconds.

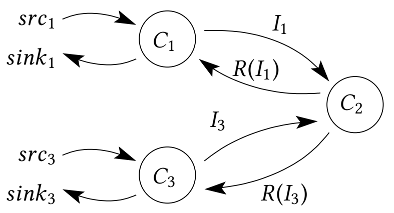


Figure 1: Running example described in Section 2.0.2. $R(I_i)$ **denotes the response to intent** I_i **(set using `setResult`).**

The FlowDroid static analysis is context-, flow-, object-, and field-sensitive and Android app lifecycle-aware [11]. FlowDroid performs a highly precise taint flow static analysis for Android, but its analysis is limited to single components. FlowDroid’s analysis uses a static list of Android API methods that correspond to sources and sinks. This list is produced by SuSi [18], a tool that uses machine learning to classify the methods exposed by the Android API.

The Epicc tool precisely and efficiently identifies properties (such as *action*, *category*, and *data MIME type*) of intents that can be sent and received by components [16]. For example, Epicc might identify that a particular app can only send intents with action `android.intent.action.VIEW` and MIME type `image/jpg`.

Soot [21] is a Java optimization and analysis framework. DidFail uses the Soot framework in several parts of the analyzer.

2.0.1 Analysis Method. DidFail’s taint flow analysis takes place in two phases. In Phase 1, each application is analyzed individually. Received intents are considered sources, and sent intents are considered sinks. The output of DidFail’s Phase-1 analysis, for each app, consists of (1) flows within each component, found by FlowDroid; (2) identification of the properties of sent intents, as found by Epicc; and (3) intent filters of each component, extracted from the manifest file.

An intent ID is assigned to every source code location that sends an intent (i.e., a source code location that consists of a call to a method in the `startActivity` family), as described in Section 2.1.2. Sent intents with distinct IDs are considered distinct sinks, while intents with the same ID are combined together.

Phase 2 of the analysis is carried out on a particular set of apps, using the output of Phase 1. The output of Phase 2 consists of all the source-to-sink flows found in the set of apps.

2.0.2 Example Scenario. Figure 1 shows an example of information flows between several components that cannot be precisely analyzed by existing tools.

Suppose that Component C_1 sends data to Component C_2 and receives data from it in return. Component C_3 interacts with C_2 in a similar fashion. We do not specify whether these three components belong to different apps or to a single app; the analysis is the same in either case. As depicted in Figures 1, for $i \in \{1, 3\}$:

(1) Component C_i calls `startActivityForResult` to send data from source src_i to component C_2 via intent I_i.
(2) Component C_2 reads data from intent I_i and sends that data back to component C_i by calling `setResult`.
(3) Component C_i, in method `onActivityResult`, reads data from the result and writes it to sink $sink_i$.

To be precise, the DidFail analysis should determine that (1) information flows from src_1 to $sink_1$ (but not $sink_3$), and (2) information flows from src_3 to $sink_3$ (but not $sink_1$). Note that FlowDroid by itself cannot produce this precise of a result, even in the case where the three components are part of a single app.

2.0.3 Phase 1. In this phase, DidFail analyzes each app individually. It identifies an intent by a tuple of (sending component, receiving component, intent ID). An intent sent from C_1 to C_2 with ID *id* is denoted by $I(C_1, C_2, id)$.

In Phase 1, when a component calls a method in the `startActivity` family, the recipient of the intent is unknown (because each app is analyzed in isolation in Phase 1, but the recipient can be a component in another app), so "$\star$" is used for the recipient field. Likewise, in the `onCreate` method, the sender of the intent is unknown, so "$\star$" is used for the sender field. If a component receives an intent I_1 and returns information via the `setResult` method, the returned information is denoted by $R(I_1)$.

The notation $source \xrightarrow{C} sink$ is used to denote that information flows from *source* to *sink* in component C. For this purpose, DidFail treats intents as both sources (in the component that creates and sends the intent) and sinks (in the component that receives the intent). Using this notation, the flows depicted in Figure 1 are represented as follows:

$$\begin{aligned}
src_1 &\xrightarrow{C_1} I(C_1, \star, id_1) \\
R(I(C_1, \star, \star)) &\xrightarrow{C_1} sink_1 \\
I(\star, C_2, \star) &\xrightarrow{C_2} R(I(\star, C_2, \star)) \\
src_3 &\xrightarrow{C_3} I(C_3, \star, id_3) \\
R(I(C_3, \star, \star)) &\xrightarrow{C_3} sink_3
\end{aligned}$$

The above flows constitute the desired output of the FlowDroid analysis. Although all the flows in the running example involve intents, in general the DidFail analysis will also find flows from non-intent sources to non-intent sinks.

The focus of this description is on intents sent and received by `Activity` components; however, DidFail similarly handles other types of components (services, content providers, broadcast receivers).

2.1 Phase 2

After all apps in the given set have been analyzed, DidFail enters Phase 2. Its goal is to find out how tainted information can flow between components. For each sent intent, DidFail finds all possible recipients, and it instantiates the Phase-1 flow equations (which have missing sender/receiver information) for all possible sender/receiver pairs, as described in detail in Section 2.1.1.

For the running example, the Phase-2 flow equations are as follows:

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

Let $T(s)$ denote the taint of s, that is, the set of sensitive sources from which s potentially has information. The goal of the analysis is to determine the taint of all sinks. Each phase-2 flow equation $s_1 \rightarrow s_2$ relates the taint of s_1 to the taint of s_2. If data flows from s_1 to s_2, then s_2 must be at least as tainted as s_1. Accordingly, DidFail generates a taint equation $T(s_1) \subseteq T(s_2)$. For the running examples, the taint equations generated are:

$$\begin{aligned}
T(src_1) &\subseteq T(I(C_1, C_2, id_1)) \\
T(R(I(C_1, C_2, id_1))) &\subseteq T(sink_1) \\
T(I(C_1, C_2, id_1)) &\subseteq T(R(I(C_1, C_2, id_1))) \\
T(I(C_3, C_2, id_1)) &\subseteq T(R(I(C_3, C_2, id_3))) \\
T(src_3) &\subseteq T(I(C_3, C_2, id_3)) \\
T(R(I(C_3, C_2, id_3))) &\subseteq T(sink_3)
\end{aligned}$$

Each non-intent source s is tainted with itself, i.e., $T(s) = \{s\}$. DidFail then finds the least fixed-point of the set of taint equations. The end result of Phase 2 is the set of possible source to sink flows.

2.1.1 Details of Generating Phase 2 Flow Equations. Above in Section 2.1, we said that DidFail instantiates the Phase-1 flow equations for all possible intent sender/receiver pairs. We now informally give the details of how it does this. For a more formal treatment, please see Section 3.3.1 of [13].

Let S be the set of sources and sinks (including intents and intent results) in the Phase 1 flow equations.

Let $is\_receivable(id, C_{RX})$ denote whether intents sent at the source-code location *id* can be received by component C_{RX}. An intent that explicitly designates its target component can be received by exactly that component. An implicit intent is considered receivable by C_{RX} if and only if and only if the action and category of the intent (as determined by Epicc) match the intent filter specified for C_{RX} in the application's manifest file.

For each Phase-1 flow, we instantiate the sources as follows:

- Instantiate $I(\star, C_{RX}, \star) \xrightarrow{C} sink$ by $I(C_{TX}, C_{RX}, id) \xrightarrow{C} sink$ for every pair (C_{TX}, id) such that $I(C_{TX}, \star, id) \in S$ and $is\_receivable(id, C_{RX})$ is true.
- Instantiate $R(I(C_{TX}, \star, \star)) \xrightarrow{C} sink$ by $R(I(C_{TX}, C_{RX}, id)) \xrightarrow{C} sink$ for every pair (C_{RX}, id) such that $I(C_{TX}, \star, id) \in S$, $R(I(\star, C_{RX}, \star)) \in S$, and $is\_receivable(id, C_{RX})$ is true.

Then we instantiate the sinks as follows:

- Instantiate $src \xrightarrow{C} I(C_{TX}, *, id)$ by $src \xrightarrow{C} I(C_{TX}, C_{RX}, id)$ for every component C_{RX} such that $is\_receivable(id, C_{RX})$ is true.
- Instantiate $src \xrightarrow{C} R(I(*, C_{RX}, *))$ by $src \xrightarrow{C} R(I(C_{TX}, C_{RX}, id))$ for every pair (C_{TX}, id) such that $I(C_{TX}, *, id) \in S$, $R(I(C_{TX}, *, *)) \in S$, and $is\_receivable(id, C_{RX})$ is true.

Finally we remove flows of the form $I_1 \xrightarrow{C} R(I_2)$ if $I_1 \neq I_2$.

2.1.2 APK Transformer. DidFail modifies the bytecode of the original APK to insert a unique ID at each intent-sending location in the the bytecode. When Epicc processes this modified APK, it prints the unique intent IDs.

2.1.3 FlowDroid (Modified). DidFail modified FlowDroid so that it prints an intent's unique ID if the FlowDroid sink is a sent intent.

2.1.4 Phase 2 Analysis. Each app in the app set undergoes its own separate Phase-1 analysis. The Phase-2 analysis output provides information about dataflows from a source to a sink, including intents if they are part of the data flow.

3 PRECISE-DF METHOD

We have developed an algorithm (Precise-DF) for a more practical (fast, low memory, precise) taint flow static analysis tool, based on the fast but imprecise tool, DidFail. To address the need for speed, we retained DidFail's modular analysis. We added context to the analysis with parameterized summaries of potential data flows, with the goal of reducing the number of false positives reported by our tool (thereby improving precision). Specifically, we added Boolean formulas to DidFail's flow equations to record a conservative overapproximation[1] of the condition under which the flow is possible. A Precise-DF Boolean formula (BF) is defined by the BNF grammar in Fig. 2. AP stands for "atomic proposition". The only type of atomic proposition we consider is string equality between the action string of the received intent (denoted by the variable *act*) and a string constant. (We may write "$act \neq foo$" as an abbreviation of "$\neg(act = foo)$".) In phase one of Precise-DF, we create a summary of each component within an app. (The second phase is a fast app-set analysis of possible taint flows.) This summary consists of the set of tainted flows found possible in the component, counting received intents as sources and counting sent intents as sinks. We have developed an initial implementation of Precise-DF (modifying DidFail code, including modifications to FlowDroid code incorporated in DidFail). However, initial experimental results show that more debugging is needed.

3.1 Phase 1

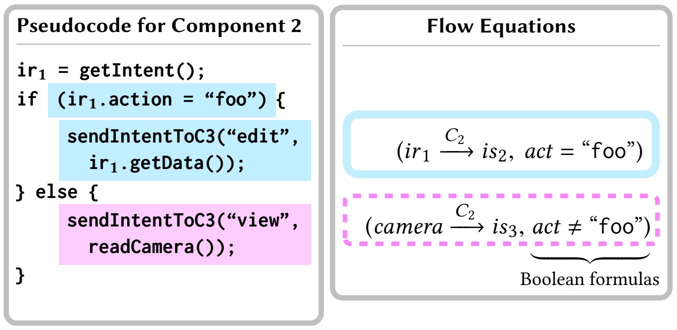
In Figure 3, the flow equation outlined in blue describes a possible taint flow in pseudocode on the left side highlighted in blue. Per-component code is analyzed and data for the resulting initial flow equations are stored by Precise-DF in Phase 1. Data for the part of the flow equation *before* the comma was also stored by the DidFail tool during its Phase 1. It indicates that there is a flow across component C_2 (the string above the arrow denotes the component ID), with the source to the left of the arrow (a received intent named ir_1) and the sink to the right of the arrow (a sent intent named is_2). The conditional expression in this pseudocode checks if the received intent's (ir_1) action string is equal to foo, and the possible taint flow in the blue-highlighted line below is only possible if that field indeed was foo. The possible taint flow highlighted in blue involves sending an intent with the possibly-tainted data field from the received intent (ir_1).

The pseudocode in the else block outlined in pink is only reached if this condition is false, meaning the action string was not equal to foo. In that case, tainted data could potentially flow from the camera to a sent intent (a sent intent is a sink). The flow equation in a dashed outline (pink) expresses that information, including the Boolean expression indicating that the action string must not equal foo. Similarly, flow equations for pseudocode in the example component 3 are shown in Fig. 4.

Compared to the number of initial Phase 1 flow equations from the DidFail tool, Phase 1 of Precise-DF will usually produce at least as many initial flow equations. In the depiction of phase 1 analysis in Fig. 5, Precise-DF shows more arrows for phase 1 flows because the figure differentiates flows detected (with different colors) involving

$$\boxed{\begin{array}{lcl} BF & ::= & AP \\ & | & BF \wedge BF \\ & | & BF \vee BF \\ & | & \neg BF \\ & | & \texttt{true} \mid \texttt{false} \\ AP & ::= & \texttt{act} = \textit{string-literal} \end{array}}$$

Figure 2: Grammar for Boolean Formulas

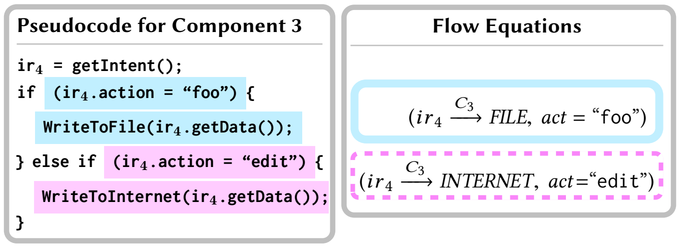


```
ir₁ = getIntent();
if (ir₁.action = "foo") {
    sendIntentToC3("edit",
      ir₁.getData());
} else {
    sendIntentToC3("view",
      readCamera());
}
```

$(ir_1 \xrightarrow{C_2} is_2,\ act = \text{"foo"})$

$(camera \xrightarrow{C_2} is_3,\ \underbrace{act \neq \text{"foo"}}_{\text{Boolean formulas}})$

Figure 3: Component 2 Pseudocode and Flow Equations



```
ir₄ = getIntent();
if (ir₄.action = "foo") {
  WriteToFile(ir₄.getData());
} else if (ir₄.action = "edit") {
  WriteToInternet(ir₄.getData());
}
```

$(ir_4 \xrightarrow{C_3} FILE,\ act = \text{"foo"})$

$(ir_4 \xrightarrow{C_3} INTERNET,\ act = \text{"edit"})$

Figure 4: Component 3 Pseudocode and Flow Equations

[1]It is an overapproximation in the sense that the flow is possible only if the recorded condition is satisifed.

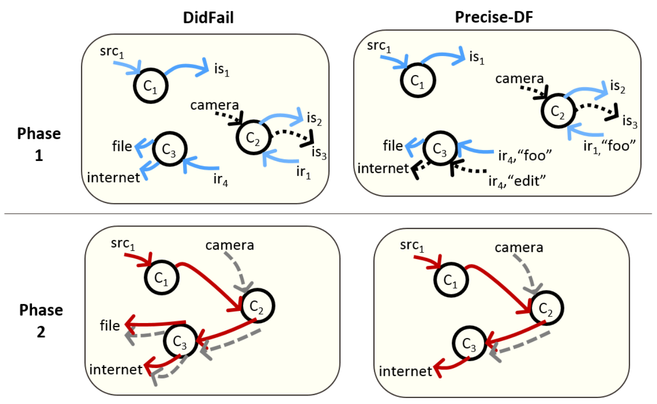


Figure 5: Comparison of Previous DidFail and Precise-DF

the same source and sink across the same component but different Boolean expressions.

As in [13], we identify an intent by a tuple of (sending component, receiving component, intent ID). The intent ID is the source-code location at which the intent is sent. An intent sent from C_1 to C_2 with ID *id* will be denoted by $I(C_1, C_2, id)$.

Precise-DF uses a simple path-sensitive intraprocedural flow analysis to compute the Boolean formula of each phase-1 flow equation. To do this, for each sink API callsite s, it computes a Boolean formula $reach(s)$ that indicates the condition under which the sink is reachable. For example, the sink `WriteToFile` in Figure 4 is reachable under the condition act = "`foo`". This Boolean formula is then associated with the flow equations for this sink.

The above-described analysis is done on the Jimple intermediate representation and proceeds as follows. First we identify which uses of local variables refer to a received intent's action string (i.e., the result of `getIntent().getAction()` in an `Activity` context). Next, we define $cond(p, s)$ for all edges (p, s) in the control-flow graph (CFG) as follows. If p has the form "`if (`e`) goto` ℓ" where e is an equality between a string constant c and a received intent's action string, we define:

$$cond(p, s_{\mathrm{T}}) = \text{“}act = c\text{”}$$
$$cond(p, s_{\mathrm{F}}) = \text{“}act \neq c\text{”}$$

where s_{T} is the statement that is jumped to if the condition e evaluates to true, and s_{F} is the statement that control falls through to if the condition e evaluates to false. For all other edges (p, s), we define $cond(p, s) = \texttt{true}$.

For a statement s, we write $reach(s)$ to denote the condition under which s is reachable. Note that, under a particular assignment to act, a given statement s is reachable only if s has a CFG predecessor p such that both $reach(p)$ and $cond(p, s)$ evaluate to `true` under the assignment. With this intuition in mind, we compute *reach* as the solution (least fixed point) to the follow equations:

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

```
int x = 0;
Intent i = getIntent();
if (i.getAction().equals("foo")) {
    x = i.getIntExtra("bar", 0);
}
Log.w("", x)
```

Figure 6: Code Illustrating Increased Precision Benefit

where $preds(s)$ is the set of predecessors of s in the control-flow graph, and *entry* is the first statement of the function. (Recall that this is an intraprocedural analysis.)

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.

Additional precision can be added to the taint analysis as follows: For each source, the taintedness of each variable (or, in general, each access path) is represented as a Boolean formula. The code in Fig. 6 illustrates a situation where this would be useful. The variable `x` is tainted if and only if the incoming intent's action is `foo`.

3.2 Phase 2

In phase 2 of Precise-DF, we find out how tainted information can flow between components. To do this, we identify which outgoing intents from one component can be incoming intents to other components. As in DidFail, the phase-1 flow equations get instantiated for all possible sender/receiver pairs. The DidFail phase-2 algorithm discussed in subsection 2.1 is modified as follows, to incorporate the Boolean formulas associated with the phase-1 flow equations:

- We delete any resulting phase-2 flow equation that is impossible in light of its associated Boolean formula. Specifically, we delete any flow equation of the form
$$\left(I(C_1, C_2, id) \xrightarrow{C_2} sink,\ \phi\right)$$
if $\phi \wedge (act \in \mathrm{EpiccAct}(id))$ is unsatisfiable, where $\mathrm{EpiccAct}(id)$ denotes the set of possible action strings (as determined by Epicc [16]) of intents whose intent ID is *id*. (Recall that the only free variable that can occur in ϕ is *act*.) This satisfiability check can be done in time $O(|\mathrm{EpiccAct}(id)| \cdot |\phi|)$ by plugging in each value in $\mathrm{EpiccAct}(id)$ for *act* in ϕ.

For example, considering the example in Figures 3–5, $(I(C_2, C_3, id_3) \xrightarrow{C_3} FILE,\ act$=“`foo`”$)$ would be deleted given that $\mathrm{EpiccAct}(id_3) = \{$“`view`”$\}$.

3.3 Performance

The two-phase analysis approach allows a mobile device management system (MDM) or app store to respond quickly (within 1 second or so) in typical cases because the first phase is the most time-consuming part of the analysis, and it can be precomputed for all apps in the app store. When a user requests to install a new app, only the 2nd phase needs to be run.

In original DidFail, phase 2 is typically fast, and we expect it to still be fast in Precise-DF. There are two main possible slowdowns in phase 2 for Precise-DF relative to original DidFail:

- The input to phase 2 in Precise-DF may be larger (than in original DidFail) because a single phase-1 flow in original DidFail may correspond to multiple flows (differing only in boolean formula) in Precise-DF. However, we expect (based on preliminary empirical investigation) that the number of phase-1 flows to be only a small factor larger. Given a flow from original DidFail, the number of additional corresponding flows in Precise-DF is bounded by the number of comparisons of the received intent's action string to a string constant.
- Deleting the impossible phase-2 flows (as detailed in Section 3.2) consumes time. As mentioned in subsection 3.2, the time to determine whether to delete a flow from an intent with ID id and boolean formula ϕ is $O(|\mathsf{EpiccAct}(id)| \cdot |\phi|)$. The boolean formula is either a constant or a disjunction of equalities or a conjunction of disequalities. The number of equalities/disequalities is bounded by the the number of string literals in the program text that are compared to the received intent's action string. The size of $\mathsf{EpiccAct}(id)$ is bounded by the number of string literals in the program text that are supplied as the sent intent's action string. Thus, both $|\mathsf{EpiccAct}(id)|$ and $|\phi|$ are expected to be small.

We say that *src directly flows* to *sink* if and only if there is a phase-2 flow equation $src \xrightarrow{C} sink$. We define the relation "transitively flows" as the transitive closure of the relation "directly flows". We say that a *full taint flow* is a pair $(src, sink)$ where *src* transitively flows to *sink* and neither *src* nor *sink* is an intent.

In Fig. 5, in phase 2 Precise-DF detects fewer *full taint flows* than DidFail for the same example app set.

3.4 Comparisons: DidFail and IccTA

Although the original version of DidFail successfully found potential inter-app taint flows (and was the first static analysis published for taint flow across multiple apps), its core analytical method (even with subsequent enhancements [2, 4, 9, 10]) is not precise enough for practical use. It produces too many false positives. The Precise-DF algorithm can only produce a smaller or equal set of identified full taint flows, compared to DidFail. Precise-DF produces a smaller set of full taint flows if Boolean formulas show any resulting phase-2 flow equation is impossible. Compared to IccTA, for analyses where IccTA doesn't time out, IccTA is an upper-bound for precision for Precise-DF. Similarly to DidFail, IccTA does inter-component and intra-component analysis and finds multi-component (including multi-app) flows. IccTA uses a single phase for all analysis, and retains a large amount of contextual information that allows IccTA to precisely determine if a flow is possible. Due to its modularity, Precise-DF is much faster than IccTA (considering Precise-DF's phase 2, which happens when the user tries to install a new app and/or when the security system compares app sets against policies) and also uses less memory. Both Precise-DF and IccTA use the FlowDroid algorithm for intra-component taint flow analysis, but IccTA must analyze (in one phase) possible flows across the entire app set, explaining its higher memory use and longer analysis time.

4 USE IN ENTERPRISE ARCHITECTURE

We call a two-phase static analysis for taint flow in Android app sets "Practical-DF" if it is sufficiently precise and fast while using low disk and memory space. (Our goal is to make Precise-DF an instance of a Practical-DF.) In an enterprise that uses Android devices, there will be multiple apps, numerous flows, and users unfamiliar with the concept of a dangerous flow. As recommended in [17], organizations need to create policies concerning permitted and forbidden dataflows. An enterprise could use a Practical-DF in conjunction with a security system to help set and enforce dataflow policies on Android devices, with compliance checks.

Enterprises must enforce policies governing smartphone use to avoid data exfiltration. Studies such as [8] have shown that end users, if presented with a request for permissions, generally approve the permission. The research results discussed in [7] indicate that often developers request unnecessary permissions and do not understand the true implications of permissions. Similar policies could be used for an enterprise's iPhone and Android devices at an abstract level, to prevent dataflows from a specified source to a specified sink (e.g., an organization prohibits data from corporate email to be sent out as a text message). Our work focuses on the mechanism rather than the policy, but a Practical-DF analytical mechanism would enable the enterprise to enforce policies it needs or wants.

4.1 Enterprise Use Details

Policies could be expressed as whitelists or blacklists:

- Whitelist example: A policy might specify that data from the microphone cannot flow to any sink except the public switched telephone network.
- Blacklist example: A policy might alert the user if data could flow from on-device storage to the internet.

When a user requests to install a new Android app, a security system such as the DoD's Mobile Device Management system (see p.6 of [6]) would use a Practical-DF analysis to determine if the app can be installed without violating policies. Similarly, if policies are updated, then on-device app set potential data flows get checked against blacklists or whitelists the policies describe. Users are provided with a user-friendly set of options if they must make a choice between apps to keep.

When flows are identified that don't comply with one or more policies, a variety of mitigations could be used:

- The system might refuse installation of the app. Alternatively, if the non-compliant flow relies on the existence of multiple installed apps, the system could give the option to remove previously-installed apps to prevent the non-compliant flow.
- The system could alert the user to the non-compliant flow and require escalation if the user still wants to install the app.
- The system could dynamically block non-compliant flows, which can be accomplished in various ways:
  - The app could be blocked from reading particular sources.

– The app could be blocked from writing to particular sinks.
– When an app sends an *implicit* intent, (i.e., the intent does not explicitly designate its recipient), the system could block that intent from being sent to certain other apps.

The dynamic changes would require modifications to the Android OS. Some systems that work within or replace the Android OS, such as LineageOS (formerly CyanogenMod) and TaintDroid[5], can perform those operations.

Beyond using a Practical-DF as a component in a system for setting, analyzing, and enforcing dataflow policies for Android devices, enterprises could use Practical-DF analyses for additional purposes. Security researchers could use a Practical-DF to check for dataflows that can be exploited in ways end users would not expect. Developers could also use a Practical-DF to check if an app being developed might accidentally enable dangerous dataflows.

5 CONCLUSIONS, LIMITS, FUTURE WORK

For a static analysis tool to be practical for finding taint flows in Android app sets, the tool must be fast, require only affordable and feasible memory and hard drive space, and it must be precise, because organizations cannot afford a lot of expensive engineer time trying to manually determine if each taint flow warning is a true or false positive. This paper describes a new static analysis algorithm for precise detection of taint flows in Android app sets, with fast user-time speed and relatively small disk space and memory requirements all due to a modular analysis. The paper also describes how an enterprise architecture could use such an analysis to enforce enterprise data-flow policies. The new analysis (like all other Android static analyses the authors are aware of) will not soundly analyze native code or uses of reflection in the apps. While our work applies the use of our method to Android security, the general method we refined (modular analysis with parameterized summaries of flow of sensitive information) is applicable to the class of problems involving taint flow analysis for software systems that communicate by message passing.

Future work planned includes testing Precise-DF on thousands of popular Android apps (including the top 100 apps in each of the Google Play Store categories), plus testing it on known malicious or leaky apps. Success metrics include analysis times, memory space, and disk space similar to DidFail but precision similar to IccTA, plus number of known flaws found (higher better).

6 ACKNOWLEDGEMENTS

Copyright 2018 ACM. All Rights Reserved. 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. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM18-0039

REFERENCES

[1] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the Android framework. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* IEEE, 725–735.

[2] Jonathan Burket, Lori Flynn, Will Klieber, Jonathan Lim, Wei Shen, and William Snavely. 2015. *Making DidFail Succeed: Enhancing the CERT Static Taint Analyzer for Android App Sets.* Technical Report CMU/SEI-2015-TR-001. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=434962

[3] Diwakar Dinkar, P Greve, K Landfield, F Raget, and E Peterson. 2016. *McAfee Labs Threats Report.* Technical Report. Intel Security, Tech. Rep.

[4] Karan Dwivedi, Hongli Yin, Pranav Bagree, Xiaoxiao Tang, Lori Flynn, William Klieber, and William Snavely. 2017. DidFail: Coverage and Precision Enhancement. (2017).

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI.*

[6] Greg Youst (DISA Chief Mobility Engineer). 2014. DoD's Strategic Mobility Vision: Needs and Challenges. http://csrc.nist.gov/groups/SMA/ispab/documents/minutes/2014-10/oct22_dod_mobility-needs-and-challenges_youst.pdf. (October 2014).

[7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. CCS.*

[8] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security.*

[9] Lori Flynn and William Klieber. 2015. An Enhanced Tool for Securing Android Apps. https://insights.sei.cmu.edu/sei_blog/2015/03/an-enhanced-tool-for-securing-android-apps.html. (2015).

[10] Lori Flynn and William Klieber. 2017. Automated Detection of Information Leaks in Mobile Devices. https://insights.sei.cmu.edu/sei_blog/2017/11/automated-detection-of-information-leaks-in-mobile-devices.html. (2017).

[11] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Programming Language Design and Implementation (PLDI).* http://www.bodden.de/pubs/far+14flowdroid.pdf

[12] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 661–671.

[13] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis (SOAP).* https://doi.org/10.1145/2614628.2614633

[14] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. *ICSE* (2015).

[15] Jon Oberheide and Charlie Miller. 2012. Dissecting the Android bouncer. *SummerCon2012, New York* (2012).

[16] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium.*

[17] Steve Quirolgico, Jeffrey Voas, Tom Karygiannis, Christoph Michael, and Karen Scarfone. 2015. *Vetting the security of mobile applications.* US Department of Commerce, National Institute of Standards and Technology.

[18] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proc. NDSS.* http://www.bodden.de/pubs/rab14classifying.pdf

[19] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of Android inter-app security vulnerabilities using COVERT. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 725–728.

[20] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA).* ACM, 1–5.

[21] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot — a Java Bytecode Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press.