Noname manuscript No.
(will be inserted by the editor)

Lifting Inter-App Data-Flow Analysis to Large App Sets

Florian Sattler · Alexander von Rhein · Thorsten Berger · Niklas Schalck Johansson · Mikael Mark Hardø · Sven Apel

the date of receipt and acceptance should be inserted later

Abstract Mobile apps process increasing amounts of private data, giving rise to privacy concerns. Such concerns do not arise only from single apps, which might—accidentally or intentionally—leak private information to untrusted parties, but also from multiple apps communicating with each other. Certain combinations of apps can create critical data flows not detectable by analyzing single apps individually. While sophisticated tools exist to analyze data flows inside and across apps, none of these scale to large numbers of apps, given the combinatorial explosion of possible (inter-app) data flows. We present a scalable approach to analyze data flows across Android apps. At the heart of our approach is a graph-based data structure that represents inter-app flows efficiently. Following ideas from product-line analysis, the data structure exploits redundancies among flows and thereby tames the combinatorial explosion. Instead of focusing on specific installations of app sets on mobile devices, we lift traditional data-flow analysis approaches to analyze and represent data flows of all possible combinations of apps. We developed the tool Sifta and applied it to several existing app benchmarks and real-world app sets, demonstrating its scalability and accuracy.

Florian Sattler
University of Passau, Germany

Alexander von Rhein
CQSE GmbH, Germany

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Niklas Schalck Johansson
IT University of Copenhagen, Denmark

Mikael Mark Hardø
IT University of Copenhagen, Denmark

Sven Apel
University of Passau, Germany

1 Introduction

The growing popularity and adoption of mobile devices—such as smartphones and tablets—has led to a tremendous rise of mobile apps. By January 2014, Apple's app store offered more than one million apps [2] and had a yearly revenue of $10 billion. Other app-store providers, including Google and Microsoft, experienced a similar growth. The large number of apps available and the increasing diversity of mobile devices lead to very different sets of apps installed on mobile devices today.

Privacy of data is an increasing concern. While apps often process private data, such as passwords, device identifiers, or position data, they also commonly possess unrestricted access to communication channels, which may not be trustworthy. To prevent security escalation, mobile operating systems employ a range of methods, such as sandboxing of apps, dedicated communication mechanisms (e.g., ANDROID intents), and a permission system for accessing sensitive data. Additionally, various analysis techniques have been developed to detect so-called *tainted data flows*—flows of data from private sources to untrusted public sinks [3, 18].

Yet, as apps are allowed to communicate with each other, a *combination* of apps can create a privacy leak even if individual apps are considered safe [8, 31, 33]. For instance, an app could obtain the current location and send it—accidentally or maliciously—to a second app, which then forwards it via the Internet to an untrusted party. Such scenarios are hard to detect as they could, in principle, involve a chain of many apps [17]. Malicious apps can even intercept or eavesdrop on unsecured communication between apps.

The presence of critical inter-app data flows depends on the set of apps installed on a device. Consider an accidental privacy leak, where an app sends private information to apps that can display it. If multiple target apps are installed, most systems display a choice dialog, possibly creating awareness of a potential privacy leak. When only one alternative, potentially malicious, app is present, communication occurs without user interaction. Consequently, all possible combinations of apps of a given set would need to be analyzed to detect inter-app leaks, whether accidental or malicious.

Unfortunately, inter-app data-flow analysis is expensive and difficult to scale to larger app sets or even to a whole app store. First, the communication between apps is often redundant, since many apps send similar messages, leading to substantial numbers of flows (many apps are also cloned or use common code [35, 23]). Second, the representation of flows is prone to a combinatorial explosion in the number of apps, when flows arise from apps that may communicate. So, installing a new app may double the number of possible inter-app flows. Recent taint-analysis tools for ANDROID are reasonably precise in detecting critical data flows, tackling all the peculiarities of ANDROID apps (e.g., permissions, ANDROID API, intents), but they do not scale well to large sets of apps.

A major problem of existing approaches lies in the representation of inter-app data flows, which does not exploit *redundancies* between and inside apps. More importantly, they do not explicitly consider *variability* [4]—an app can

be installed or not, thereby contributing to the global data flows that may exist. Instead of duplicating detected flows, variability inside flows should be represented explicitly. Recognizing synergies, we adopt concepts known from product-line analysis [34,36,20,16,25], which incorporates variability to avoid redundancies and to tame the combinatorial explosion.

Our overarching goal is to explore how inter-app data flows can be efficiently represented. To this end, we present a scalable graph-based data structure representing flows annotated with *presence conditions* [11]—Boolean expressions over the presence and absence of apps. We compare this *variational* data structure to the structure used by the state-of-the-art tool DidFail [17]. Furthermore, we lift an analysis approach that analyzes data flows inside individual apps to whole app sets by extending and combining existing tools—that is, we make the approach *variability-aware*; we use and aggregate the analysis results in a graph that is *variational* [36] to efficiently represent inter-app communication.

Overall, we strive to answer two research questions:

RQ1: Can a variability-aware data-flow analysis based on a variational representation of inter-app data flows maintain accuracy?

RQ2: Does a variability-aware data-flow analysis based on a variational representation of inter-app data flows scale to very large sets of real-world apps?

To address these questions, we evaluate the our variability-aware approach by means of two third-party community benchmarks and a set of 51 935 analyzed real-world apps that we mined from the Google Play app store. It is important to note that, while aiming at scalability, our tool maintains an accuracy that is similar to existing tools focusing on intra-app analysis, which we also evaluate with two third-party benchmarks and with our own benchmark. As a further feature, our approach supports the *incremental* generation of the inter-app data-flow graph: When new apps are added or changed (e.g., a new app is uploaded to the app store or developers publish a new version of an existing app), the inter-app data-flow graph can be updated with information for such apps, instead of generating a new graph.

Finally, to illustrate possible insights one can gain when analyzing large-scale inter-app data-flow graphs using our approach, we implemented a standard analysis that traverses the lifted graph and reports tainted inter-app data flows. For example, we use the graph to reason about global communication patterns, identifying central apps that are responsible for many global flows.

Overall, we make the following contributions:

- An efficient variational, graph-based representation of inter-app data flows, which captures Android-specific information (sources and sinks of potentially private data and inter-app communication metadata). The graph benefits from redundancies between data flows and from the optionality of apps (variability).
- An algorithm implemented in our tool Sifta to efficiently construct variational inter-app data-flow graphs.

- A variability-aware taint-propagation analysis based on the graph (reporting malicious flows). We evaluated its accuracy in one experiment with three benchmarks, and its scalability in three experiments using three other, larger-scale benchmarks, comparing SIFTA to other state-of-the-art tools.
- Two new benchmarks for the community: IACBENCH with 9 apps (accuracy) and a large-scale benchmark with 51 935 real-world apps (scalability).

SIFTA, links to all other tools in our evaluation, and information on how to replicate our results are available on a supplementary Web site: http://www.fosd.net/siftaeval/.

2 Background and Motivation

In this section, we describe the app communication mechanisms of ANDROID and discuss existing analysis strategies and their limitations. We distinguish between *intra-app* communication (when components inside one app communicate) and *inter-app* communication (when components of different apps communicate).

2.1 Android Apps and the Intent Mechanism

ANDROID apps are delivered in ANDROID *application packages* (APKs) and consist of multiple components that communicate with each other. Components can be GUI elements (*activities*) shown to the user or non-visible elements that process or store information (*services*, *broadcast receivers*, and *content providers*). Components have a dedicated lifecycle and are encapsulated. They communicate via dedicated messages, called *intents*,[1] both for intra-app and inter-app communication. Intents contain various pieces of data, such as routing and payload information.

Intents can be *explicit* or *implicit*. The former identify the target component directly using its fully qualified name. The latter describe the minimal capabilities a target component needs to fulfill, which are then matched against the maximal capabilities of components defined in *intent filters*. Such capabilities could be the ability to show a URL or to display an image of a certain type. If multiple components of installed apps qualify, ANDROID displays a choice dialog and lets the user select. Usually, intents pass information to other components. However, they can also query information (e.g., user information from a data-storage component), initiating an information flow back to the sender. With these so-called *intent results*, the target app sends a second intent back to the source app.

[1] Other means of communication (e.g., shared files, native code) exist, but are outside of the scope of this paper.

2.2 Intra-App Communication

Intent-based communication is the primary mechanism for data exchange between components inside an app. For example, an activity could send data entered by the user to a service that processes the data. Here, an *explicit* intent is typically used to unambiguously identify the receiver.

Analysis of communication inside an app is important to detect data flows that leak private data by accident. For example, a developer of a popular Android app might want to analyze her own app to confirm that private user data are not passed to third-party components used in the app. In this intra-app scenario, the set of components is known.

Several analysis tools focus on an intra-app scenario. One comparatively precise tool is IccTA [18], which relies on FlowDroid [3] and Epicc [29]. IccTA composes all components of an app into one "super" component subsuming all the flows. A challenge is to connect components—that is, mapping intent calls of one component to incoming intents of another component. The parameters of an intent object, which is dynamically instantiated at run time, need to be known and matched to intent filters. For this purpose, Epicc (or alternative tools, such as IC3 [28]) performs a static analysis to retrieve the intent parameters. Once the "super" component is created, it is analyzed with FlowDroid, a precise inter-procedural data-flow–analysis tool. The intra-component data flows reported by FlowDroid connect sources and sinks, which are Android API methods, intent calls, or incoming intents.

2.3 Inter-App Communication

Communication between apps (i.e., their inner components) is realized using the same intent-based mechanism as for intra-app communication. The main difference is that the set of installed apps is not pre-determined. An implicit intent can be processed by different apps (e.g., different e-mail clients) in different mobile-device configurations, with different implications for data privacy.

Fig. 1 shows a simple inter-app communication. The app LocationReader-App is one of many alternative apps that read the current location from the GPS device, enhance it with context information (e.g., near landmarks), and send it via an intent (Loc). If installed, each of the two apps FitnessApp and MaliciousApp can receive the intent (determined by their intent filters, shown as little rectangles). If the latter obtains the data, they are forwarded over the Internet to an untrusted third party.

Similar scenarios have been reported in the literature [5,33,8,31,17]. Ideally, Android's permission system should prevent apps from accessing private data without user consent. However, Android permissions are not sufficient for this, as commonly stated in the literature (see Sec. 8). In the scenario of Fig. 1, MaliciousApp might lack permission to read GPS data from the Android API, but can still get it by interacting with LocationReaderApp. It does not matter

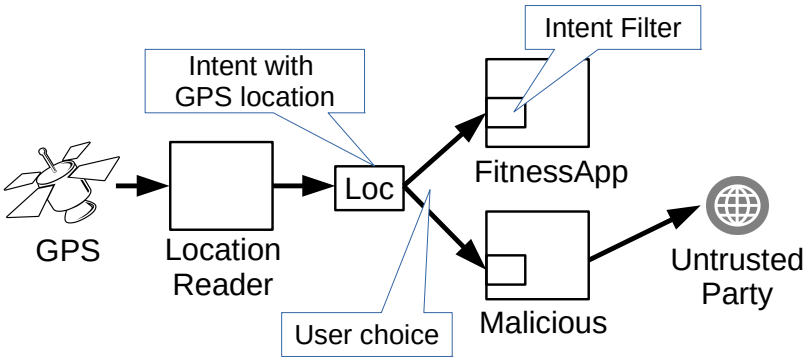


Fig. 1: Inter-app communication example, where GPS location information is forwarded to untrusted receivers

whether LocationReaderApp sends the data out via an intent or has a component that is accessible via an intent, and whether both happens accidentally or whether the app was maliciously developed to enable this scenario. This problem is generally known as *permission re-delegation* [31,8] (a.k.a. *confused-deputy problem* [15]).

Analyzing inter-app communication is important for maintainers and users of app stores or pools. It is desirable to ensure that each possible combination of apps respects privacy of user data and that no inter-app data-flow leak exists. Even without actual leaks, it is desirable to identify apps that heavily forward data, which in combination could be exploited for privacy leaks in the future. This scenario is more complex than intra-app communication, since apps can be present or absent (resulting in different global flows—this is why we pursue a variability-aware approach). Furthermore, apps are regularly added, removed, and updated. Thus, analysis results of inter-app communication should be kept updated after each change in the pool, without the need of re-analyzing the entire pool (this is why we pursue an incremental approach).

To analyze inter-app communication, one could, in principle, use tools that have been developed for intra-app analysis, since both kinds of communication rely on intents. Such an approach is taken by DidFail [17], which, like IccTA, relies on FlowDroid and Epicc. DidFail runs FlowDroid to obtain data flows within each component on each app. Based on the intent parameters obtained with Epicc, DidFail connects the possible outgoing intents to intent receivers and builds a global data-flow graph involving all apps.

2.4 Limitations of Existing Tools

Both IccTA and DidFail rely on the assumption that the set of components is known, invariable, and rather small. IccTA's approach would cause scalability problems when inter-app communication is analyzed, as the generated "super"

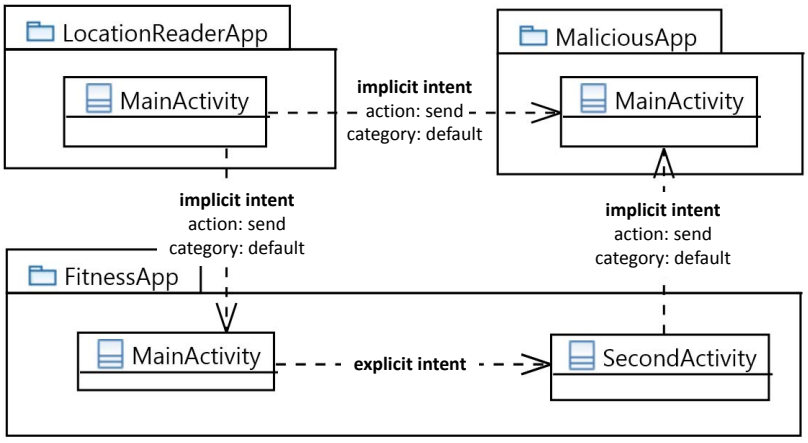


Fig. 2: Example of inter-app communication[2]

component easily becomes large, with many—likely redundant—flows. But the IccTA developers focus on intra-app communication in their experiments anyway [18]. In contrast to IccTA, DidFail addresses inter-app communication explicitly. Yet, it stores detected flows in a simple graph data structure without exploiting redundancies between flows, harming scalability (as we will demonstrate).

To understand how a larger app set can influence the number of flows, consider the following thought experiment, illustrated in Fig. 2. We build a scenario based on the three apps of Fig. 1: LocationReaderApp obtains private data, and sends it with a valid intent to FitnessApp. However, MaliciousApp can also receive the intent—its presence establishes a data flow to an untrusted network receiver outside the phone. MaliciousApp has the permission to access the Internet, but not to obtain GPS data from the Android API. Furthermore, we extend the scenario slightly, adding (1) a typical internal data flow between two components inside FitnessApp and (2) another accidental leak from FitnessApp to MaliciousApp, so that we have two privacy leaks.

Now, consider a larger scenario, where we have an additional alternative app for each of the three apps. The alternative apps have roughly the same functionality (the same data sources and sinks and the same intents), but they are implemented by different developers. In this scenario, the number of flows grows: For example, both variants of LocationReaderApp can send information to both variants of MaliciousApp. Now, there are 12 leaks in total. In general, the number of flows is cubic in the number of apps of each kind, which shows that an efficient inter-app analysis has to exploit redundancies between flows. Even though this thought experiment shows an extreme example, our experiments (Sec. 5.2) confirm this problem for DidFail. A key insight is that the problem

[2] For readability, we do not show additional, doubled or tripled, edges and nodes that get added during the thought experiment.

lies in the representation of the underlying graph and of the flows. DidFail does not address sharing between apps or redundancies between flows.

The limitations of existing tools motivated us to develop our own tool, called Sifta, which addresses these challenges when analyzing inter-app communication. We reused parts of DidFail's code, but completely re-implemented the graph construction and the identification and analysis of tainted flows.

2.5 Variability-Aware Analysis and Variational Data Structures

We assume that significant redundancies in the communication paths between apps exist. There are many reasons that back this assumption, such as the existence of commonly used intents (e.g., ACTION_VIEW, which requests that data is displayed to the user) or duplicated code [35,23]. Consequently, exploiting redundancy using a dedicated data structure should have considerable influence on the size of inter-app data-flow graphs and their analysis time.

Technically, we draw on ideas from variability-aware product-line analysis [34]. In particular, we use the concept of *presence conditions* [11] Boolean expressions denoting which apps need to be present to enable a given data flow. Using presence conditions, we can compress the data-flow graph and make it variational such that its generation and analysis scales much better than for a non-variational repesentation (e.g., as used in DidFail).

We borrow the presence-condition idea and its efficient encoding and analysis from recent advancements in product-line analysis, such as variability-aware static analysis [20,6], type checking [16,25], and model checking [9,1] as well as variational data structures [36,10].

3 Representing Inter-App Flows

The communication between apps is typically analyzed in two steps. First, information about individual apps (e.g., using static code analysis) is collected and stored in a suitable data structure. Second, the stored data are analyzed for interesting facts. This two-phase process avoids the need to deal with app internals (e.g., source code) in the second step. The key factor is how to abstract from and store app internals. In Sec. 3.1, we describe a basic representation of inter-app data flows in the form of a graph; in Sec. 3.2, we introduce our efficient, variational representation; and, in Sec. 3.3, we describe the construction of variational inter-app data-flow graphs in detail.

3.1 Representation without Sharing

We first present a graph data structure that is *not* variational. The graph is defined as a set V_{DF} of nodes, with $V_{DF} \subseteq \mathit{Int} \cup \mathit{PrivSrc} \cup \mathit{PubSnk}$, and a set E_{DF} of directed edges, with $E_{DF} \subseteq V_{DF} \times V_{DF} \times \mathit{Comp}$. V_{DF} contains all intent objects (*Int*), private-source methods (*PrivSrc*), and public-sink

methods (*PubSnk*). For each component $comp \in Comp$ that receives data from a private-source method or an intent object *src* and that delivers data to an intent object or a public-sink method *snk*, the set of edges contains the triple $(src, snk, comp) \in E_{DF}$. A path through the graph represents a potential data flow from a private source through a number of components (possibly across multiple apps) to a public sink.[3] As a component is automatically present when the app it belongs to is installed, we only store app names on edges (instead of component names).

Recall our thought experiment from Sec. 2.4: With an increasing number of apps, the graph quickly becomes very large and its generation and analysis expensive. The reason is that often different apps have (partly) similar functionality. For example, they receive data from the same sources (*Int* or *PrivSrc*) and send data to the same sinks (*Int* or *PubSnk*). Thus, the graph has many edges that differ only in the app component, such as (a, b, c_1) and (a, b, c_2). Fig. 3a shows an example with four flows of private data (GPS location and private key) to a public sink (Internet). The middle edges of the flows have the same source (Int_1) and the same sink (Int_2), and only differ by the app enabling this edge. The key point is that similarities among data flows are not shared. Such a representation is used in DidFail [17].[4]

3.2 Variational Representation

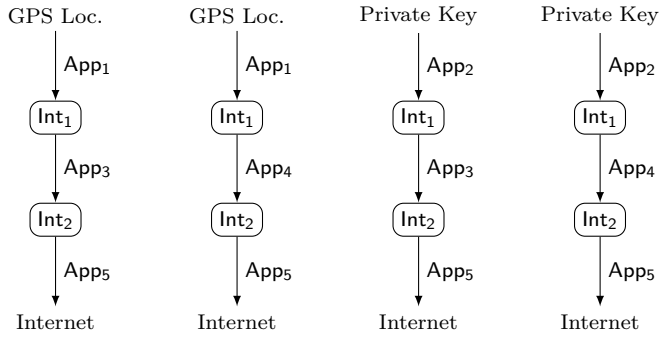
Our innovation is to represent flows within and across apps in a variational fashion. The difference is that each edge in the graph is annotated with the condition when it is present in the system. This presence condition (cf. Sec. 2.5) is a predicate over the (optional) apps in the pool. Each path in the graph represents a *variational flow* corresponding to multiple *concrete flows* (e.g., flows in the DidFail representation).

We define the set of edges such that each holds a set *comps* of components (or, equivalently, a predicate over *Comp*): $E_{VA} \subseteq V_{DF} \times V_{DF} \times \mathcal{P}(Comp)$. Instead of mapping each edge to a component, we now map each edge to a *set* of components (or, equivalently, a predicate over component identifiers). The semantics is that an edge $(a, b, comps)$ is present in the graph (or on the mobile device) iff one of the components in *comps* is installed.

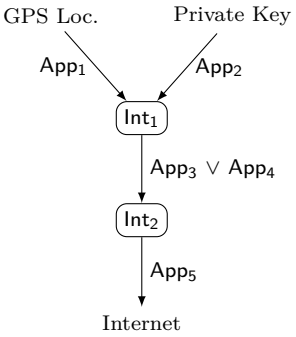
Fig. 3b shows the same scenario as Fig. 3a, but using a variational representation. The four edges from Int_1 to Int_2 are replaced by one, with a presence condition denoting the components that give rise to the flow. This lifted representation is efficient when app sets contain many inter-app flows that share common parts (intents or partial flows). Such sharing can be caused by common

[3] The flow is only a potential flow, as our analysis is static and can produce false positives (as taint analysis, in general)

[4] Our experiments are based on the DidFail variant published at the SOAP workshop [17]. More recently, the authors describe an improvement of DidFail [7], however the focus of the improvement is DidFail's accuracy, not its scalability. Therefore, our analysis of DidFail's scalability does still hold even though the accuracy of the new DidFail version is better than suggested in our experiments (Section 5.1).



(a) non-variational (DIDFAIL)



(b) variational (SIFTA)

Fig. 3: Example of an inter-app data flow, non-variational and variational. Edges are annotated with app names instead of component names.

intents used by many apps (e.g., ACTION_VIEW) or by code in differently named components that process information in the same way (e.g., through code duplication [35,23]).

It is important to note that this variability encoding does not introduce any false positives or false negatives, as is always ensured in variational data structures [36] and variability-aware analysis [34].

3.3 Graph Construction

We now describe step by step the construction of a variational inter-app data-flow graph based on results that we obtain from FLOWDROID and EPICC. The goal is to create a directed graph where a path represents a potential data flow from a private source through a number of nodes (possibly across multiple apps) to a public sink. There are various possibilities of flows among the constituents of the graph. Table 1 shows all possible intra-app flows together with examples of methods from *PrivSrc* and *PubSnk* representing start and end points.

Table 1: Potential intra-app data flows

Kind of Flow	**Example**
Source → Sink	Src: TelephonyManager.getDeviceId Snk: Log.v
Source → Intent	Src: Location.getLatitude Snk: startActivity[1]
Intent Filter → Sink	Src: getIntent[1] Snk: FileOutputStream.write
Intent Filter → Intent	Src: getIntent[1] Snk: startActivity[1]
Source → Intent Result	Src: Location.getLongitude Snk: startActivityForResult[1]
Intent → Intent Result	Src: getIntent[1] Snk: startActivityForResult[1]
Intent Result → Sink	Src: onActivityResult[1] Snk: HttpClient.execute
Intent Result → Intent	Src: onActivityResult[1] Snk: startActivity[1]

[1] non-qualified methods belong to ANDROID's Activity class

For illustration, we introduce a runnning example in Fig. 4. It shows the data flows within an app App_1, which consists of two components Comp_1 and Comp_2. In our representation, components may contain sources (Src) (API method calls representing user input, for example) and sinks (Snk) as well as intent calls (Int) and intent receivers (IntRecv), each of which are sources and sinks on their own.

3.3.1 Source and Sinks

For each app, we identify the private sources and public sinks that participate in intra-component flows using an intra-app data-flow analysis, such as provided by FLOWDROID. All identified sources and sinks become nodes in the graph, connected with edges representing these flows. Each flow uniquely belongs to a component $comp \in Comp$, which is annotated to each edge of the graph. The result is an intermediate graph like the one in Fig. 4 with $V_{DF} \subseteq PrivSrc \cup PubSnk$ and $E_{VA} \subseteq V_{DF} \times V_{DF} \times Comp$.

3.3.2 Intents, Intent Results, and Intent Filters

Next, we need to handle flows across components (inside apps). Specifically, we need to connect flows ending in an intent (e.g., using $\mathsf{startActivity} \in PubSnk$) with flows starting with an intent (e.g., using $\mathsf{getIntent} \in PrivSrc$). We obtain

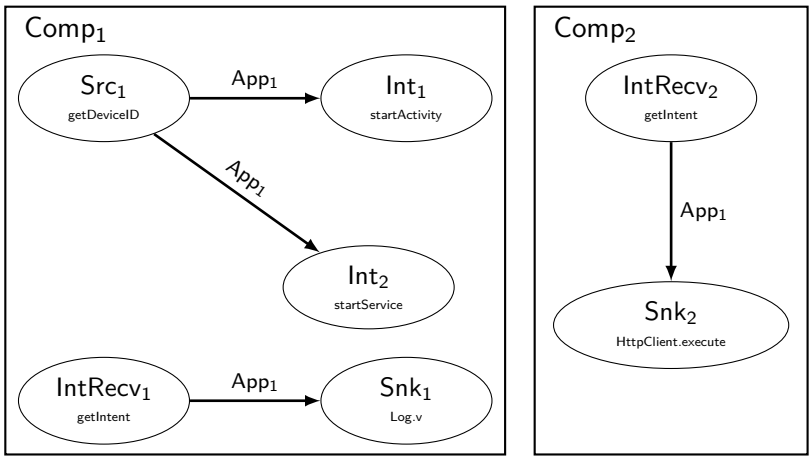


Fig. 4: An app with two components (Comp_1, $\mathsf{Comp}_2 \in \mathit{Comp}$) with intra-component flows, connecting a source ($\mathsf{Src}_1 \in \mathit{PrivSrc}$) with two intent calls (Int_1, $\mathsf{Int}_2 \in \mathit{PubSnk}$) as well as two intent receivers ($\mathsf{IntRecv}_1$, $\mathsf{IntRecv}_2 \in \mathit{PrivSrc}$) with two sinks each ($\mathsf{Snk}_1$, $\mathsf{Snk}_2 \in \mathit{PubSnk}$). The apps involved are annotated to the edges.

details about the intent by one of the various static analysis tools, such as Epicc [29] and IC3 [28], that extract parameters of the instantiated intent object from source or byte code. Likewise, we obtain intent-filter information from the app's manifest file, relating it to the intent-filter method that starts the flow (e.g., $\mathsf{getIntent} \in \mathit{PrivSrc}$) by package and component name.

Explicit Intents: If a method that starts a flow has no corresponding intent filter, it corresponds to an explicit intent. We completely remove the respective intent and intent-receiver nodes and directly connect the flows. This eliminates any information that the resulting inter-component flow was based on an explicit intent, which is not needed anymore and keeps the graph concise. The rationale is that explicit intents are almost exclusively used for intra-app and not inter-app communication. New apps that are added later cannot connect to already processed explicit intents or the receiving method. The elimination of explicit intents already has considerable influence on performance, as it reduces the flows within one app dramatically, and allows us to focus on sources, sinks, and implicit intents. Fig. 5 shows the example of Fig. 4 with the explicit intent $\mathsf{Int}_1 \in \mathit{PubSnk}$ and its matching receiver $\mathsf{IntRecv}_2 \in \mathit{PrivSrc}$ resolved.

Implicit Intents: Next, we match flows ending in an implicit intent method with flows starting with an intent-receiver method that has a corresponding intent filter. Since such spots can later be extended (see Sec. 4.2), we keep the intent method and the intent-receiver method as nodes in the graph. We directly connect the intent method with the sink of the matching flow. More precisely, as we will explain shortly, we actually replace the intent-method node with an intent-object node, which abstractly represents an intent object

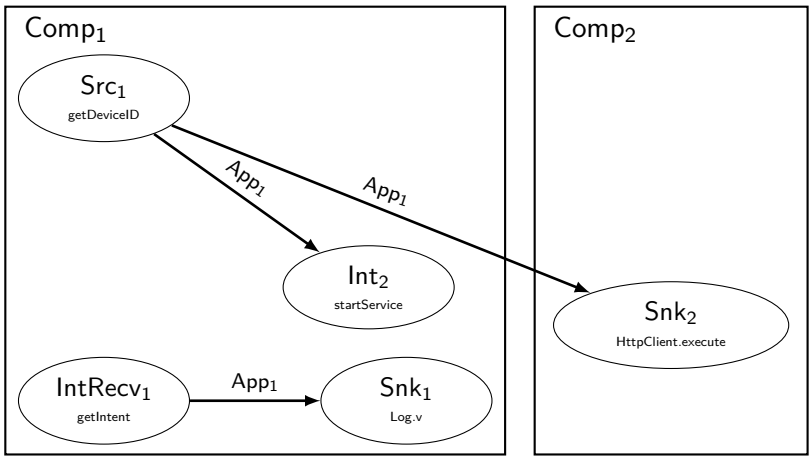


Fig. 5: Example of Fig. 4 with flows via explicit intents resolved

by its metadata. Furthermore, note that we also take services and broadcast receivers into account, which also communicate via intents and only differ in the method used to start the intent. Fig. 6 shows an example with explicit and implicit intents resolved.

Whether an intent can be accepted by an intent filter is decided by ANDROID based on the intent object's metadata, the component's intent filter, the type of the component, and the method used to call the intent. An intent object is defined by an action key, a list of categories, and a MIME type.[5] An intent filter is defined by a list of action keys, a list of categories, and a list of MIME types. Based on this information, ANDROID matches intent objects with intent filters to deliver the intent object to a component [12].

To collapse the same intent-object nodes across apps, we describe them purely based on their properties, and only add the components (respectively the app names, as discussed in Sec. 3.2) of the flows they belong to (from source to intent, and from intent receiver to sink) to the respective edges in our variational graph representation. Specifically, we define intent-object nodes for implicit intents as $Int_{implicit} = Keys \times \mathcal{P}(Categories) \times \lfloor URIs \rfloor \times \{service, activity, broadcast\ receiver\}$, where *Keys* is the set of all possible action keys, *Categories* the set of all possible *Categories*, and *URIs* the set of all possible MIME types that can be used as for data specification in ANDROID intents. Note that the latter is optional.[6]

This representation allows collapsing the graph, since all nodes are app-independent and only described based on their properties. If a flow between the source getDeviceId $\in$ *PrivSrc* and the sink Log.i $\in$ *PubSnk* is found in two

[5] The MIME standards define content types (e.g., JPEG, GIF, or AVI) of data attached to communication messages. They are also used in e-mail and HTTP protocols. There, clients use MIME types to determine how attached data should be opened.

[6] $\lfloor URIs \rfloor = URIs \cup \bot$, where $\bot$ represents an absent MIME type.

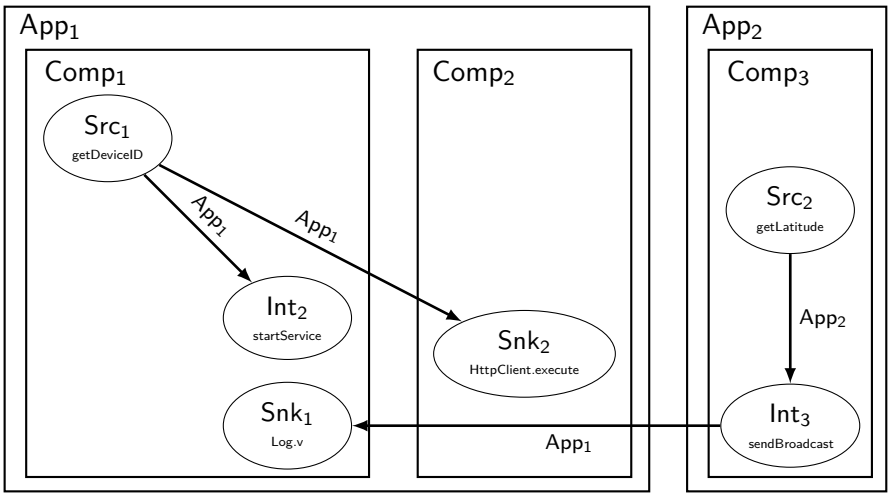


Fig. 6: Extended example of Fig. 5 with flows via explicit and implicit intents resolved

applications, the graph will still only have two nodes, but with two app names on the edge between them. So we exploit the sharing of elements in different apps, where components such as ad libraries and common patterns used in multiple apps will not increase the number of nodes or edges in the graph, only the information associated with each edge.

Intent Results: Finally, we need to handle intent results—another form of inter-app communication. They are initiated by intent methods (e.g., startActivityForResult) that also return a result (another intent object) to the sender app by calling the target activity's setResult method. An intent result includes several possible flows, whereas the first two are the same as for ordinary intents: (i) in the source app, from a private source to method startActivityForResult; (ii) in the target app, from getIntent to a sink; (iii) in the target app, from getIntent or a source to setResult; and (iv) in the source app, from onActivityResult to a sink. We add the respective flows as edges to the variational graph. Since we need to keep the information whether an intent node represents an intent result, we finally define $Int_{implicit} = Keys \times \mathcal{P}(Categories) \times \lfloor URIs \rfloor \times \{service, activity, broadcast\ receiver\} \times \{intent, intent\ result\}$.

The Final Graph: Using this construction and our definitions of $Int_{implicit}$, $PrivSrc$, and $PubSnk$, we obtain our final, variational graph with $V_{DF} \subseteq Int_{implicit} \cup PrivSrc \cup PubSnk$ and $E_{VA} \subseteq V_{DF} \times V_{DF} \times \mathcal{P}(Comp)$. Recall that in this directed graph, a node represents either a start or end point of a potentially critical data flow, that is, from a private source to a public sink, or an intent that forwards information. Furthermore, explicit intents have been resolved already at this point (see above), and edges represent apps that receive

and process intents, receive data from private sources, or forward data to public sinks.

Note that such a variational representation is concise and still amenable to data-flow analysis. It might appear surprising, since apps are essentially represented as edges. But, intents are the central means of inter-app communication in Android. It would have been possible (and might seem more intuitive) to represent apps as nodes, but this representation would either lead to dangling edges or some temporary additional nodes that might need to be replaced once more apps are added to the graph.

4 Implementation

We implemented our variability-aware approach in the tool Sifta. It reuses some code from DidFail, such as the parser for Epicc output files. Sifta implements all concepts discussed in Sec. 3 and, in addition, supports services and broadcast receivers, which are types of Android components that are not covered by DidFail.

4.1 A Two-Phase Approach

Like DidFail, Sifta uses a two-phase approach. In the first phase, it uses FlowDroid and Epicc[7] to analyze one app at a time. FlowDroid generates information on (i) which intents contain private information and (ii) which information from intents is sent to public sinks of an app. Epicc provides detailed information on the metadata of the intents, which is necessary to match them to intent filters of other apps (cf. Sec. 3.1). In the second phase, Sifta performs intent-matching procedures as described in the Android API and generates the inter-app data-flow graph (cf. Sec. 3.2 and Sec. 3.3). It uses the FlowDroid and Epicc output from the first phase and the manifest files (containing details of intent filters) of the apps. Based on this information, Sifta (and DidFail) determines which intent is matched by which component's intent filter. In addition to DidFail's matching criteria, Sifta implements matching of MIME types, as specified in the Android API. Finally, note that the first phase may fail (cf. Sec. 5.2) on real-world apps. In such cases, FlowDroid or Epicc usually hit timeouts. Improving these third-party tools is well beyond the scope of this paper, and some failures are to be expected, as we use static-analysis tools on real-world apps that might actively prevent analysis by code obfuscation. Still, our two-phase design allows to easily use results from other tools in the first phase.

Recall that paths in the graph correspond potentially to one or more private-data leaks. When edges in a path have alternative apps (i.e., a presence condition consists of more than one app), the path corresponds to multiple

[7] There are alternatives to Epicc, such as IC3 [28], but our considerations and results are not affected by this choice, as we discuss in Section 6.2.

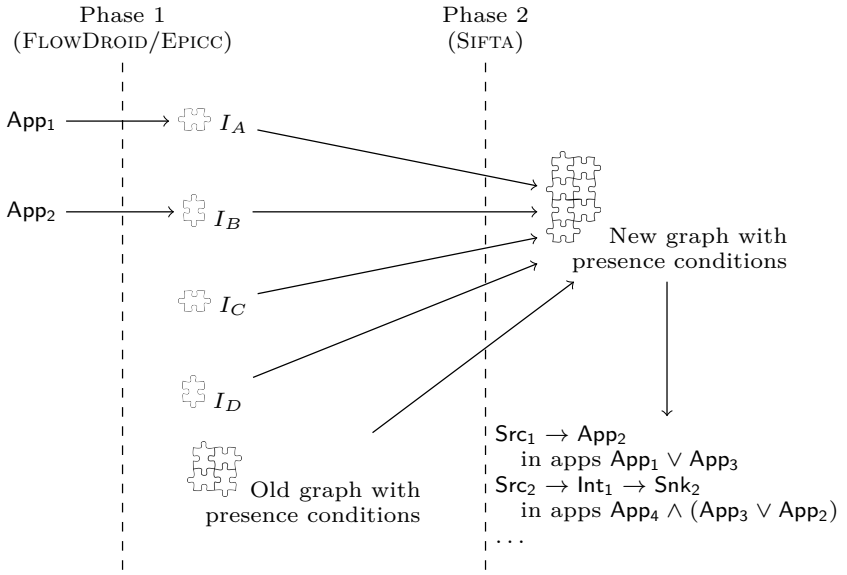


Fig. 7: Sifta's inter-app analysis. I_A, I_B, I_C, and I_D represent intermediary results generated by the first phase. I_C and I_D are reused from a previous run of the analysis. The intermediary results are added to the old graph, which is also reused from a previous analysis run.

concrete leaks. In contrast to product-line analysis [34], our presence conditions are simple (disjunctions), though, such that we do not need SAT queries to generate the graph or to derive feasible flows from it.

4.2 Incremental Graph Construction

Furthermore, we implemented an incremental graph-construction procedure—a feature that we needed for our largest experiment (see Sec. 5.2). As a considerable computation effort lies in the first phase, we support reuse of already computed partial results. This includes two types of intermediate results, as illustrated in Fig. 7. The apps $\mathsf{App_1}$ and $\mathsf{App_2}$ are analyzed for the first time. Two other apps have been analyzed before (results from phase 1 are reused), and an old graph with information about more apps exists already. In the first phase, $\mathsf{App_1}$ and $\mathsf{App_2}$ are analyzed by FlowDroid/Epicc. In the second phase, Sifta uses the newly generated results and the reused results and integrates them into the existing graph. To this end, Sifta keeps data structures (dictionaries) that save the information about intent filters, which were not added to the graph (cf. Sec. 3.3), and about implicit intents that did not match any filter so far. These need to be re-considered when new apps are added, since new flows and connections could arise in the graph. Finally, Sifta produces

an updated graph containing all the edges of the old graph and the new edges introduced by the new apps.

This persistence and reuse of results enables SIFTA to analyze large-scale, evolving sets of apps in short time. If only few apps change, SIFTA does not need to analyze the entire app set from scratch, but can reuse old results if they are still valid (when the apps have not changed). To update a graph, we remove all updated apps from the graph (delete the app from all presence conditions) and integrate the FLOWDROID/EPICC results for the updated apps.

4.3 Taint Propagation

Once the data-flow graph has been generated, it can be used in various ways. An example, which we implemented in SIFTA, is a standard taint analysis implemented with a depth-first graph traversal algorithm: We simply report all paths from sources to sinks in the graph. These paths correspond to potentially malicious data flows. This is, in fact, a taint-propagation analysis as the (*tainted*) private data is forwarded along the path until it reaches a sink. The apps on edges along the path constitute the presence condition of the data flow.

5 Evaluation

In a series of experiments, we evaluated the accuracy and scalability of our approach, comparing it to the other state-of-the-art tools DIDFAIL and ICCTA. In Sec. 5.1, we discuss our evaluation of the accuracy of SIFTA on benchmark sets comprising a total of 44 test cases with inter-component and inter-app leaks (**RQ1**). Yet, accuracy is only a necessary condition and highly relies on the underlying data-flow–analysis tools we use. Since our main contribution is a scalable approach for inter-app communication scenarios, we present our analysis of large sets of real-world apps in Sec. 5.2, where we compared the variational representation of SIFTA to the non-variational representation of DIDFAIL, measuring to what extent our approach is able to exploit redundancies in app communication (**RQ2**).

5.1 Experiment 1: Accuracy

To answer **RQ1**, in the first experiment, *E1*, we measured the accuracy of SIFTA by calculating precision and recall of detected privacy leaks using a ground truth of established, third-party community benchmarks and our own hand-crafted benchmark. To understand the accuracy that is achievable with state-of-the-art tools, we compare our results to those obtained by ICCTA (only intra-app) and DIDFAIL (also inter-app). Overall, we analyzed three different sets of apps:

– IACBench contains 9 app sets (two apps per set) created by us to cover basic (intents with and without results, comprising activities, services, and broadcast receivers) and advanced inter-app flows (e.g., loops).
– ICC-Bench[8] contains 9 apps with intra-app flows developed by the authors of Amandroid [37].
– DroidBench[9] comprises 23 apps testing inter-component communication (provided by the IccTA authors [18]) and 3 sets of apps testing inter-app communication (provided by the DidFail authors [17]), among many more apps not relevant for our approach.

Our own benchmark IACBench contains test cases with critical data flows via implicit intents across apps from the source TelephonyManager.getDeviceId to the sink Log.i. Details on IACBench are provided in Table 2. For the third-party benchmarks ICC-Bench and the parts of DroidBench that we use in our evaluation, we refer to the literature: Amandroid [37], DidFail [17], and IccTA [18]. All benchmarks comprise apps developed to test whether analysis tools capture specific means of communication. The apps are much smaller and cleaner than real apps and are thus ideal to compare the tools' accuracy.

Table 2: IACBench test cases

	Test Case	Description
Basic	startActivity	intent from Activity to Activity via startActivity
	startService	intent from Activity to Service via startService
	bindService	intent from Activity to Service via bindService
	sendBroadcast	intent from Activity to BroadcastReceiver via sendBroadcast
	sendOrderedBroadcast	intent from Activity to BroadcastReceiver via sendOrderedBroadcast
Advanced	multipleIntents	two identical intents from the same source to the same sink
	loop	intent from Activity to Activity, but the first Activity can also receive its own intent, creating a loop
	intentChain	intent from Activity1 to Service, to Activity2, to Activity3, back to Activity2 (result), to BroadcastReceiver
	identicalIntentFilter	intent sent to three different components (Activity, Service, BroadcastReceiver), each of which has the same intent filter

5.1.1 Methodology and Setup

We ran Sifta and DidFail on all benchmarks and measured precision and recall. While Sifta focuses on inter-app communication, it can still analyze intra-app flows. Thus, we do not only compare against DidFail, but also against IccTA, which is specialized on inter-component, intra-app communication. Consequently, we can run IccTA only on the ICC-Bench and DroidBench-ICC benchmarks, not on IACBench. We ran experiment *E1* on a Ubuntu

[8] Obtained from the authors of Amandroid.
[9] http://github.com/secure-software-engineering/DroidBench/

14.04 workstation with four cores (Intel Xeon Processor X3470 @ 2.93GHz). Timeouts and memory consumption were not an issue for these rather small test cases.

5.1.2 Results

Table 3 shows all precision and recall values of *E1*. Note that, to better compare accuracy values, we calculated two versions, one incorporating failed test cases (e.g., cases were developers did not implement functionality to handle a test case, such as specific kinds of intents or broadcast receivers) in the overall set and one not. If we compare the overall results from Table 3 we see that SIFTA outperforms DIDFAIL in precision and recall, even without including negative n/i values. Furthermore, SIFTAS precision is even near our ground truth ICCTA. Next, we discuss the individual benchmarks, emphasizing test cases where SIFTA produced worse results than DIDFAIL or ICCTA. Table 4 provides detailed information on all test cases.

Table 3: Precision and recall values of experiment *E1*. Values in parentheses include not implemented test cases in the calculation ('n/i' in Table 4.

Benchmark		DIDFAIL	SIFTA	ICCTA
IACBENCH	Precision	100% (50%)	100% (100%)	n/a
	Recall	80% (80%)	100% (100%)	n/a
ICC-BENCH	Precision	100% (100%)	100% (100%)	100% (100%)
	Recall	67% (67%)	56% (56%)	89% (89%)
DROIDBENCH IAC	Precision	100% (33%)	100% (100%)	n/a
	Recall	100% (33%)	82% (82%)	n/a
DROIDBENCH ICC	Precision	0% (0%)	83% (83%)	95% (95%)
	Recall	0% (0%)	79% (79%)	100% (100%)
TOTAL	Precision	85% (41%)	91% (91%)	96% (96%)
	Recall	42% (28%)	80% (80%)	96% (96%)

IACBENCH focuses on inter-app communication. Thus, we could not evaluate ICCTA on this benchmark. SIFTA solved all tests correctly. DIDFAIL could not solve four test cases, because it lacks support for services and broadcast receivers.

Table 4: Results of experiment *E1*: accuracy evaluation (IccTA results according to Li et al. [18])

Benchmark	**Test Case**	DidFail	Sifta	IccTA
IACBench	startActivity	+	+	n/a
(basic)	startService	n/i	+	n/a
	bindService	n/i	+	n/a
	sendBroadcast	n/i	+	n/a
	sendOrderedBroadcast	n/i	+	n/a
(advanced)	multipleIntents	+	+	n/a
	loop	+	+	n/a
	intentChain	⊖	+	n/a
	identicalIntentFilter	+	+	n/a
ICC-Bench	Explicit1	⊖	+	+
	Implicit1	+	+	+
	Implicit2	+	+	+
	Implicit3	+	+	+
	Implicit4	+	+	+
	Implicit5	+	⊖	+
	Implicit6	+	⊖	+
	DynRegister1	⊖	⊖	+
	DynRegister2	⊖	⊖	⊖
DroidBench	sendBroadcast1	n/i	+	n/a
(IAC)	startActivity1	+	+	n/a
	startService1	n/i	+	n/a
DroidBench	startActivity1	⊖	+	+
(ICC)	startActivity2	⊖	+	+
	startActivity3	⊖	+	+
	startActivity4	⊕	⊕	−
	startActivity5	⊕	−	−
	startActivity6	−	⊕	−
	startActivity7	−	⊕	⊕
	startActivityForResult1	⊖	+	+
	startActivityForResult2	⊖	⊖	+
	startActivityForResult3	⊖	⊖	+
	startActivityForResult4	⊖	+	+
	startService1	n/i	+	+
	startService2	n/i	+	+
	bindService1	n/i	+	+
	bindService2	n/i	⊖	+
	bindService3	n/i	⊖	+
	bindService4	n/i	+	+
	sendBroadcast1	n/i	+	+
	stickyBroadcast1	n/i	+	+
	insert1	⊖	+	+
	delete1	⊖	+	+
	update1	⊖	+	+
	query1	⊖	+	+

true positive:	+	(analysis reported an existing leak)
true negative:	−	(no leak and no leak reported)
false positive:	⊕	(a reported leak does not exist)
false negative:	⊖	(analysis misses an existing leak)
not implemented:	n/i	(DidFail on services or broadcasts)
not applicable:	n/a	(intra-app tool on inter-app scenario)

On some apps in the ICC-Bench benchmark set, Sifta failed to report privacy leaks. In particular, in the test cases Implicit5 and Implicit6, Sifta reported no flows as opposed to DidFail. The reason is a limitation of the underlying tool Epicc and of DidFail, which ignores the faulty Epicc output. In both cases, the flows are enabled by MIME types set on the intent objects in the code (Intent.setDataAndType). Apparently, Epicc does not handle this function, as it does not include the MIME type in its output. Based on this output, Sifta assumes that no MIME type is given and the intent does not match the intent filter in the test case. DidFail does not test for MIME types and therefore, by accident, correctly reports a flow. Furthermore, in the test cases DynRegister1 and DynRegister2, intent filters are registered dynamically and not declared in the manifest file. Epicc does not find such intent filters, which are therefore not visible to Sifta or DidFail. IccTA fails to detect the leak in DynRegister2, because the app uses string operations, which cannot be parsed by IccTA [18].

DroidBench is a much larger benchmark that tests many possible communication paths. Table 4 shows that Sifta reports correct results much more often than DidFail, but not as often as IccTA. The test startActivity4 has an intent that uses an URI scheme (http:) that is not listed in the test's intent filter. Therefore, the intent does not match the filter. Sifta does not test for URI schemes, because this information is not provided by Epicc and FlowDroid. The tests startActivity6 and startActivity7 check whether information retrieval from an intent is handled correctly. In these cases, an intent with private information is accepted by an intent filter, but, instead of the private information, other information is retrieved from the intent. The information available to Sifta contains no details on *which* information is retrieved from an intent. As long as the intent with private information is accepted and information from that intent is sent to a public sink, Sifta reports a flow. The bindService tests transfer private data via an intent to a service that logs the data. In bindService2 and bindService3, FlowDroid did not report that an intent is sent, therefore the flow is invisible to Sifta. The tests startActivityForResult2 and startActivityForResult3 failed because Sifta cannot handle some aspects of the return communication in startActivityForResult intents. We intentionally omitted these aspects for scalability reasons (see Sec. 5.2). Finally, the inter-*app* communication tests (IAC) of DroidBench were all solved correctly by Sifta.

E1 demonstrates that our variability-aware tool Sifta produces more accurate results than DidFail. Yet, it is less accurate than IccTA, which was to be expected as IccTA, for each test, combines all components and analyzes them in one run. Sifta relies on necessarily filtered information gained in separate per-component analyses, but this is exactly the lever that enables large-scale inter-app analysis.

Surprisingly, Sifta has some wrong results where DidFail's results are correct. This is not caused by Sifta's graph reduction (which does not influence the set of reported flows, as discussed in Sec. 3.2), but by additional matching criteria (for the MIME types, cf. Sec. 4) that we implemented.

Regarding **RQ1**, we can conclude that SIFTA's variability-aware approach maintains a reasonable degree of accuracy.

5.2 Experiments 2–4: Scalability

To answer **RQ2**, evaluating the scalability of SIFTA and DIDFAIL, we used three sets of apps:

- Experiment *E2*: ICCRE is a set of 523 real apps deployed with ICCTA. These apps leak private user data through inter-component communication [18].
- Experiment *E3*: MALGENOME is a set of 1260 real apps published by the ANDROID Malware Genome Project [38]. They are known to be malicious, 51.1 % harvest user data, but not necessarily using inter-app communication.
- Experiment *E4*: GOOGLEPLAYSET is a set of 172 779 apps that we randomly downloaded from Google Play, covering various categories and developers. We started off by obtaining popular apps (see Sec. 6.1, for details about the selection process).

5.2.1 Methodology and Setup

In *E2*, we compared SIFTA against DIDFAIL, however, given DIDFAIL's scalability limitations, we were not able to use it in *E3* and *E4*.

We ran *E2* on a Ubuntu machine with 32 Cores (AMD Opteron 6386 SE @ 2,8 GHz) and 100 GB reserved RAM. Because DIDFAIL and SIFTA are both based on the data-flow information generated by FLOWDROID and EPICC, we ran this pre-analysis separately. First, FLOWDROID and EPICC analyzed all ICCRE apps with a timeout of 10 minutes. This pre-analysis generated results for 324 of the 523 apps. We excluded uninteresting flows that have no influence on other apps. Then, we let DIDFAIL and SIFTA build their data-flow graphs based on this output. For both tools, we measured the time needed both to generate the graphs and to report detected critical flows. To evaluate the scalability of DIDFAIL and SIFTA, we generated subsets of increasing size from the ICCRE app set. We ran DIDFAIL and SIFTA on each subset (without reusing results from smaller subsets).

For *E3* and *E4* (MALGENOME and GOOGLEPLAYSET), given their size, we switched to a cluster of 17 nodes, each with an Intel Xeon E-5 2690v2 CPU @ 3,0 GHz, 10 cores and 2 hyperthreads per core. We allowed 6 GB RAM and 20 minutes each for FLOWDROID and EPICC.

5.2.2 Experiment E2 (ICCRE)

Fig. 8 shows the result of the scalability experiment on ICCRE. Even for only five apps, SIFTA generates the graph faster than DIDFAIL. For larger app sets, the difference between the tools gets larger (speedup of up to 7620). We stopped the experiment for DIDFAIL after analyzing the app set with size 100, as the effect was clear.

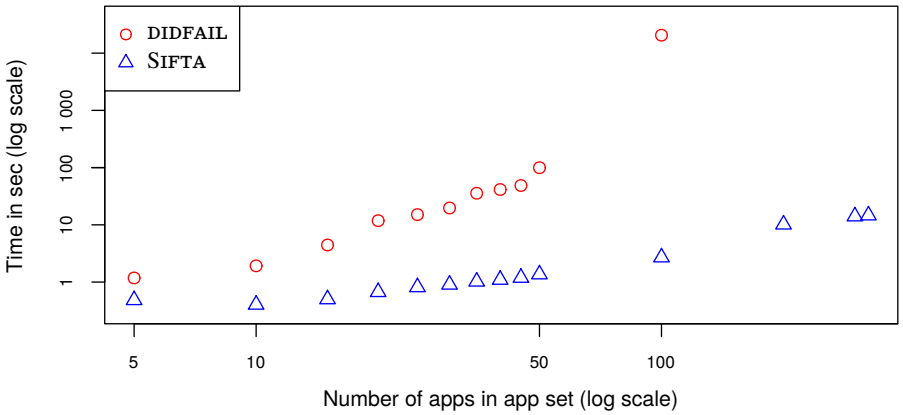


Fig. 8: Results for SIFTA and DIDFAIL on ICCRE. Both axes have a logarithmic scale.

A closer look at the output of DIDFAIL and SIFTA reveals the reason for this difference in scalability. For the app set with size 100, DIDFAIL generates a data-flow graph with 1610 nodes and 51 709 edges. SIFTA's graph has only 51 nodes and 96 edges—illustrating the effectiveness of our compressed variational representation. Even for the largest app set with 324 apps, SIFTA's graph has only 66 nodes. This result shows that there is large potential for storing inter-app data-flow graphs more compactly without losing information. It is the variability-aware approach that achieves this compression and that enables efficient analysis of inter-app communication on large app sets.

5.2.3 Experiment E3 (MALGENOME)

We analyzed the MALGENOME benchmark set only with SIFTA (DIDFAIL does not scale to this size). The analysis ran in two phases: In the first phase, FLOWDROID and EPICC ran on each of the 1260 apps. This phase is computationally very expensive. It took about 50 hours (sum across all cluster cores). This phase failed on 421 of the 1260 apps due to 20-minute timeouts or other errors outside SIFTA. In the second phase, we applied SIFTA to generate a global graph of inter-app and intra-app communication. This generation took only 26 seconds. We had to drop 9 further apps due to parsing errors on the FLOWDROID or EPICC output.

The resulting graph contained 283 flows representing 839 apps. 248 flows are intra-app flows that go directly from a private source to a public sink. These would also be found by other tools that focus on intra-app communication. However, we also found 35 flows that involve two or more apps and therefore cannot be found with intra-app analysis. The maximum number of apps annotated on an edge is 220 (average is 10), which means that we have a high degree of sharing in the graph. If we would use a tool like DIDFAIL, which is not variability aware, it would produce 220 clones of this edge instead of a

single edge. This shows the benefit of our representation even if there are no inter-app data leaks.

5.2.4 Experiment E4 (GooglePlaySet)

To evaluate the scalability of Sifta on even larger app sets, we downloaded 172 779 apps from the Google Play store. Our strategy to select the apps was to start with one of the most popular apps, facebook, to scan its Web site, following links to apps listed under "similar" and "more from developer". This process was continued, allowing us to download apps across various app categories. The strategy targets apps that are likely to communicate, leading to a dataset suitable to evaluate Sifta's scalability. The mining script and the list of apps are available on our supplementary Web site. Then, we used Sifta to analyze inter-app communication and to build the data-flow graph. Next, we report on the time needed to execute Sifta and on characteristics of the generated graph.

The first phase of Sifta (running FlowDroid and Epicc) was executed on the AMD Opteron Cluster that we also used for *E2*. This phase generated results for 51 935 of the initial 172 779 apps. The others mainly failed due to FlowDroid timeouts. This phase of the analysis took 1705 days (4.6 years) in total (sum of wall times consumed by cluster nodes). We set a timeout of 20 minutes each for FlowDroid and for Epicc. The rather low yield of this phase can be explained by the fact that we rely on research tools on a very diverse set of real apps. Although FlowDroid is one of the most precise tools for data-flow analysis of Android apps [3], improving it to an industrial strength is an effort that was not taken yet.

Next, we allocated the results generated by the first phase and ran the second phase of Sifta, to generate the global variational data-flow graph. We executed this phase on the previously described Intel Xeon workstation, because it has not been parallelized so far. We first tried running the graph generation for all apps at once, however the machine's main memory was not sufficient. After loading less than half of the apps, the process already used more than 5.7 GB. Instead, we used the incremental graph-generation feature of Sifta (cf. Sec. 4): We partitioned the results of the first phase into four sets and generated the graph in four steps, as shown in Fig. 9. The graph generation took 13 minutes, and each step used less than 3.5 GB RAM.

Overall, the graph contains 126 205 variational flows from a private source to a public sink. The graph has 1387 nodes and 5848 edges. The maximum flow length is 8: 5154 of the flows pass through 8 apps before leaking private information.[10] The edges' presence conditions contain an average of 32 (median 3) apps. The maximum of 14 164 apps has an edge that represents a set of intra-app data flows from Bundle.getBoolean to Bundle.putBoolean. This makes sense as these very common API methods can be used to read/write data from/to Bundle objects, which are the payload of intents. Fig. 10 shows the

[10] We provide histograms for the path lengths of E3 and E4 on our Web site.

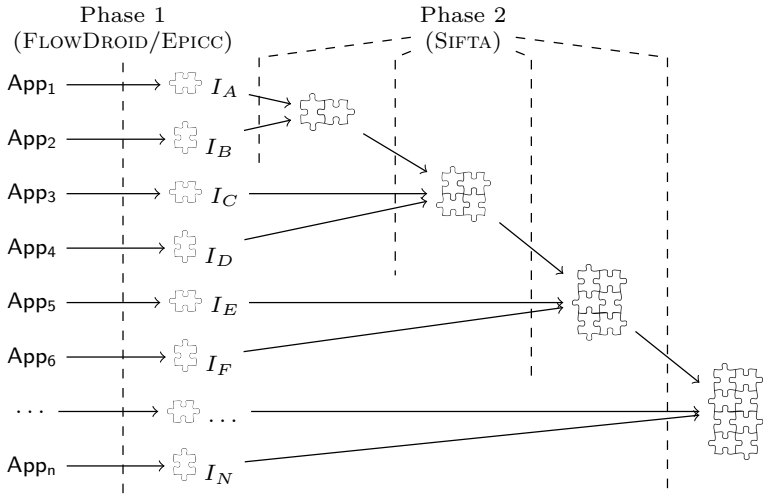


Fig. 9: Incremental setup for *E4*. We chose this setup due to memory limitations when building the graph from all apps at once. In our experiment, each partition comprised a quarter of the 51 935 apps.

frequency of presence condition sizes (number of unique apps) in the graph. Sizes lie between 10 and 200 apps for many edges. That is, each of these edges would be repeated 10 to 200 times in a graph without sharing. This demonstrates that, for large app sets, it would be infeasible to generate and store the graph in a non-variational manner.

Regarding **RQ2**, we can conclude that the construction of inter-app data-flow graphs is much more efficient using a variational representation (in terms of size) than using a non-variational representation without sharing. This way, we are able to scale the analysis of app sets to a substantial size, which is not possible otherwise.

6 Threats to Validity

6.1 External Validity

The external validity of our evaluation depends on the choice of (i) the app benchmark sets and on (ii) the tools we compare Sifta with.

ICC-Bench and DroidBench are established third-party benchmarks used also in other studies. To evaluate accuracy, we also created IACBench to include cases not covered by ICC-Bench and DroidBench, especially, advanced communication scenarios, such as loops, intent chains, and recognition of multiple identical intents. IACBench is publicly available at our supplementary Web site. To evaluate scalability, we go beyond existing bench-

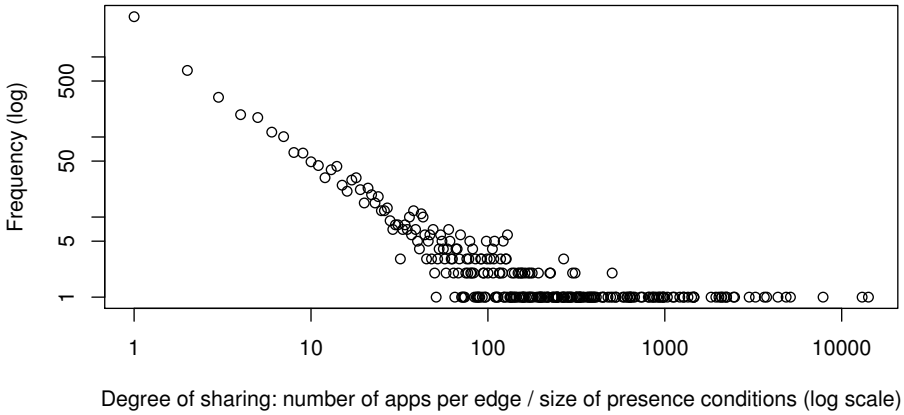


Fig. 10: Frequencies of presence-condition sizes illustrating the reason for sharing in inter-app flows. The graph shows how often different numbers of apps on edges in the graph occur. Both axes have a logarithmic scale.

marks (IccRE with 523 and MalGenome with 1260 apps) by analyzing 51 935 real-world apps from Google Play.

Empirical studies on real-world apps are commonly prone to the app-sampling problem [22]. In fact, obtaining a truly random sample of apps of an app store is almost impossible, due to the store's size. However, this problem does not affect our evaluation, since we do not aim at such a representative sample. Instead, we sought to obtain apps that are likely to communicate as described in Sec. 5.2.

In our accuracy and scalability experiments, we compared Sifta to two state-of-the-art tools for intra-app (IccTA) and inter-app communication analysis (DidFail). Further tools exist for intra-app analysis (e.g., PermissionFlow [33], CHEX [21]), but we chose the most recent and mature tool IccTA, focusing specifically on analyzing data flows. Although IccTA is more precise than Sifta, this limitation is acceptable, given that our focus is scalability. Achieving more precision is possible, but requires significant effort for creating an industry-strength tool. For scalability, our comparison is limited to DidFail, as it is the only other tool that supports inter-app communication. We tried another tool, Covert [32], that is also developed for inter-app communication, however, it failed with a Java reflection exception, which we were not able to solve, even with support from the authors. For comparison, in their publication, Covert is evaluated with "hundreds of apps", which is an order of magnitude smaller than our evaluation.

6.2 Internal Validity

In Sifta, we implemented the matching procedure of intents to receiver components, which is essentially a re-implementation of the Android systems'

intent-matching algorithm. For this purpose, we relied on Android's documentation. Yet, a threat to validity is that we may have missed (possibly undocumented) corner cases of intent matching or that we mis-interpreted the documentation. However, our results show that Sifta agrees with IccTA on most ICC-Bench test cases, which indicates proper matching.

In our experiments, we found that FlowDroid reports many false-positive flows on real apps (experiments *E3* and *E4*). Usually, these arise from private data being stored in class fields and intents being instantiated in the same class. Reproducing the detected data flows "in the wild" is often impossible, since many apps require extensive setup (e.g., registration of user accounts). Instead, we looked at several of these flows manually. We found that, in many cases, private data are visible to the code that generates the intent, but they are not attached to the intent. FlowDroid reports a flow in these situations. After consulting with a FlowDroid developer, we implemented a filter that removes such flows from FlowDroid's output. For similar reasons, we filter intent results and intents with empty actions. However, this may remove too many flows. Still, we argue that missing a few true positives is better than reporting thousands of false-positive privacy leaks, which would render the analysis useless. We only used this filtering in experiments *E3* and *E4*, which aimed at scalability.

Furthermore, in cases where Epicc cannot extract the correct intent parameters, it returns a wild card '*' (i.e., meaning we have to assume every intent parameter is possible), which can lead to more false positives. To deal with false positives in inter-app communication, Octeau et al. [27] developed a static analysis that determines, for each inter-component link, its correctness probability. Based on these probabilities, they can eliminate 95 % of the links and focus on a smaller number of flows. This approach could be combined with our scalable data-flow representation, but could also mean that we exclude existing leaks due to a low correctness probability. Additionally, we can also replace Epicc with a more accurate tool, such as IC3 [28]. In fact, we built a parser to integrate IC3 and reevaluated the accuracy test cases from *E1* with Sifta using IC3. Essentially, we found no influence on the precision of Sifta and, more importantly, there was little to no influence on the graph size, which is given as input. That is, the choice of using Epicc or IC3 does not practically affect the runtime of our Sifta analysis. We could further confirm this by reanalyzing 500 aps from our scalability analysis. We stress that our analysis does not add new imprecisions and, therefore, can always be combined with newer and more precise tools.

7 Outlook: Analyzing Large-Scale Inter-App Data-Flow Graphs

To illustrate the potential of analyzing large-scale variational inter-app data-flow graphs, we share further insights obtained from analyzing the GooglePlaySet. As a promising use case, the graphs we generate allow us to reason about the

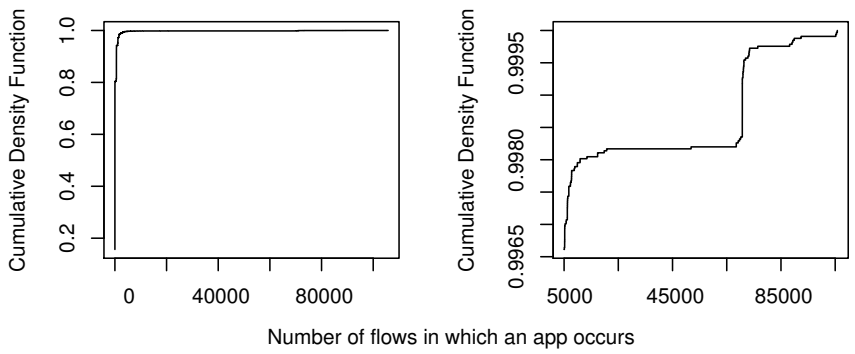


Fig. 11: Cumulative Density Function (CDF) of the number of flows in which apps participate. The left plot shows all apps, the right plot only apps that are part of >5000 flows.

positions of apps in the global communication structure, which is novel for large app sets.

As an example, let us look at how often apps occur in the flows' presence conditions. The rationale is that apps that enable many flows have a larger potential for being exploited maliciously—it may be desirable to further analyze them. Fig. 11 shows cumulative density plots illustrating how often apps occur in flows. The plots show which fraction of the apps (y axis) participates in, at most, x flows. The left diagram shows the data from all apps, the right focuses on apps participating in more than 5000 flows. Both show that a very large fraction of apps participates only in few flows and that few apps participate in very many flows (right-hand side of the right plot)—a classic Pareto distribution. Looking closer at these apps may be beneficial, as securing their inter-app communication would have significant impact. Table 5 shows the top five of these apps (our supplementary Web site lists all).

Table 5: Top five apps occurring in flow presence conditions

Rank	**App**	**# Flows**
1	com.doeiqts.SpeakHereNow	105 776
2	pl.imoney.praca	105 485
3	com.t3hh4xx0r.pwiccer	105 202
4	mbk.yap.pl4a	92 419
5	com.vnandroid.news4mobile	89 892

As a further example, let us take a look at the app that appears on the most edges (Table 5). It is called 'Speak Here Now' and belongs to the category 'Social Networks' in Google Play. It was quite often downloaded (10 000–50 000 downloads), asks for many permissions (e.g., access to all networks), and can receive a wide range of intents through broad intent filters. It has been updated last in May 2014. The app enables voice actions on the Android device. For example, a user can dictate and send tweets or specify music to be played. Moreover, there is a library that allows other apps to accept custom voice commands (recorded by Speak Here Now). Therefore, it is plausible that this app has access to private information (e.g., microphone) and can be exploited.

Let us now specifically look at apps that forward data between apps (i.e., occur in the middle of a flow). Our analysis cannot determine whether they are definitely malicious or not. However, forwarding apps are particularly sensible apps that can be exploited for constructing data leaks, with or without knowledge of the app author. Forwarding apps are typically unsuspicious because they do not necessarily use permissions, such as Internet access. Any app (forwarder or other) that participates in a flow can be removed to prevent the corresponding leaks. In our data set, we identified 88 forwarder apps (including all apps from Table 5), which receive data from an intent and deliver it to another intent. We provide the full list of forwarder apps on our supplementary Web site.

To further illustrate the potential of digging deeper into the inter-app data-flow graphs, we analyzed one forwarder app manually by decompiling and inspecting it. This app, called 'Tentacle' (category 'Business' in Google Play) provides telemarketing support (e.g., sales, customer service) for small companies. It integrates deeply with Android's call management. We found that its component BrowserCallActivity has a fairly broad intent filter, which can receive many intents (e.g., any intent with an Android VIEW or CALL action key). Once received, the app obtains a phone number from the intent, which is then internally passed to another intent (trying to make a phone call) sent by the component CallActivity. The app could, in principle, be used to forward private data (a number) by appending it to an attacker's phone number and causing 'Tentacle' to call this number. Also, the call intent sent by 'Tentacle' could be intercepted by another app, using 'Tentacle' as a forwarder, which would allow an arbitrary string to be leaked (with 'Tentacle' in the middle of a flow).

In summary, while well beyond the scope of this article, we believe there lies great potential in analyzing large-scale app stores based on inter-app data-flow graphs. Without a variational representation, this would be infeasible.

8 Related Work

8.1 Privacy Leaks in Apps

Our variational data-flow representation and variabiliy-aware analysis has various applications in software engineering (build secure apps, prevent accidental flows) and security analysis (detect privacy leaks or high-risk apps).

Privacy leaks inside and across mobile apps have been studied extensively [5]. Several researchers argue that the permission system used in Android is insufficient to prevent tainted data flows. For instance, permissions are too coarse-grained [26,30] and surprisingly rarely used in practice [29].

Enck et al. [14] studied 1100 popular Android apps, analyzing their use of libraries and misuse of private information. They found that apps often access personal information, combined with account information. Many apps also heavily use ad libraries [24], forcing acquisition of many permissions.

8.2 Data-Flow Analysis of Android Apps

To the best of our knowledge, our approach is the first to effectively scale inter-app data-flow analysis to large app sets. According to a recent literature review [19], most tools focus on intra-app analysis. A main challenge of inter-app analyses is the support of scalability for market-scale analyses. Inter-app analysis is addressed by some tools (e.g., DidFail), but they do not optimize for analysis of very large app sets such as Sifta.

Apart from DidFail, many tools focusing on intra-app communication exist. Yet, most stop at component boundaries, such as PermissionFlow [33], which does not incorporate intents and their flows, or FlowDroid [3], which we use for our component analysis. Some tools can track data flows across components. The most notable intra-app, but inter-component analysis tools are Amandroid [37] and IccTA [18]. We explained the difference between Sifta and IccTA already in Sec. 2. Amandroid is similar in accuracy to IccTA [18], and also resolves flows across components (using its own points-to analysis, where we use Epicc). We considered Amandroid for our accuracy experiments, but were not able to execute it. Anyway, Amandroid targets intra-app analysis only (as confirmed by the developers).

All these tools differ in their accuracy and how they handle the peculiarities of Android, such as the main Android library and native calls. Our approach is able to use these and other underlying tools and leverage them to create highly compressed data-flow graphs effective in identifying tainted data flows.

Dynamic analysis tools, such as TaintDroid [13], track data flows across applications at run time. While these conceptually provide the highest accuracy, they are limited by the dynamic analysis, not being able to confirm the absence of tainted flows. Most importantly, they can only analyze fixed sets of apps.

Octeau et al. address the problem of large numbers of false positives when analyzing flows between apps, which results in an explosion of potential inter-

app flows [27]. Their approach, which is based on probabilistic modeling, is orthogonal to our approach. While we aim at compressing the inter-app communication graph by representing flows more efficiently, they aim at removing flows that cannot occur during execution (false positives).

8.3 Analysis of Software Product Lines

Our variational graph representation is inspired by product-line analysis, where all possible products or systems (exponentially many, in the worst case) need to be described in a compact representation (e.g., code and models). From this representation, individual (variant-specific) representations can be derived, or statements about all possible variants can be made (e.g., all variants are consistent or safe) [34]. A mobile-device setup (a specific combination of apps) can be seen as a specific variant of a product line, where combinations of selectable features—apps in our case—constitute a system variant [4].

The analysis of product lines relies on variability awareness [34]. For instance, in product-line model checking, presence conditions are used to compactly store program states incorporating variability [9, 1]. A survey of variability-aware analyses gives an overview of related techniques [34]; Walkingshaw et al. [36] provide a broader perspective on variational data structures, discussing applications in product-line analysis and beyond.

9 Conclusion

We presented a variability-aware approach to inter-app data-flow analysis of mobile apps. It effectively tames the combinatorial explosion that previous analysis techniques faced. At its heart is a variational inter-app data-flow graph that explicitly takes variability—the diversity of apps that can be installed on a mobile device—into account. Its scalability is superior, proven on a large benchmark set of 51 935 real-world apps from Google Play, which is well beyond related work (few hundreds [17]). At the same time, our approach's accuracy can compete with state-of-the art tools, which primarily focus on intra-app flows. Our tool Sifta and a replication package are freely available on our supplementary Web site.

Perspectively, our approach enables a whole class of analyses that needs to reason about all possible combinations of apps. For demonstration, we implemented a taint analysis on top of it, which illustrates the potential to identify malicious data flows across multiple apps and to gain insight by analyzing global communication patterns.

While our focus was on scalability, further improving the precision of the static data-flow analysis is an important direction for future work. Specifically, adding more information to the graph, such as type information, could enhance the precision, by ruling out flows that cannot exist due to type incompatibilities. Since this type information would also be conditional, research in this direction

would require identifying effective ways to combine type information with presence conditions, inferring (conditional) types in data flows across apps, and incorporating this type information into intra-app data-flow analysis.

Acknowledgments

We thank Eric Bodden, Steven Arzt, Li Li, Fengguo Wei, and Yajin Zhou for helpful discussions on our implementation, on their tools (IccTA and Amandroid), and for making their benchmark sets available. The work has been supported by the German Research Foundation (AP 206/4 and AP 206/6).

References

1. S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. ICSE*, pages 482–491. IEEE, 2013.
2. Apple. App Store Sales Top $10 Billion in 2013. http://www.apple.com/pr/library/2014/01/07App-Store-Sales-Top-10-Billion-in-2013.html, 2014.
3. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*, pages 259–269. ACM, 2014.
4. T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She. Variability Mechanisms in Software Ecosystems. *Information and Software Technology*, 56(11):1520–1535, 2014.
5. N. Al Bidani and M. Vigant Raffay. A Systematic Literature Review of Mobile Inter-Application Security. Master's thesis, IT University of Copenhagen, 2014.
6. E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL$^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes instead of Years. In *Proc. PLDI*, pages 355–364. ACM, 2013.
7. J. Burket, L. Flynn, W. Klieber, J. Lim, W. Shen, and W. Snavely. Making DidFail Succeed: Enhancing the CERT Static Taint Analyzer for Android App Sets. Technical Report CMU/SEI-2015-TR-001, Software Engineering Institute, 2015.
8. E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. In *Proc. MobiSys*, pages 239–252. ACM, 2011.
9. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. ICSE*, pages 335–344. ACM, 2010.
10. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. GPCE*, pages 422–437. Springer, 2005.
11. K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *Proc. GPCE*, pages 211–220. ACM, 2006.
12. S. Dienst and T. Berger. Static Analysis of App Dependencies in Android Bytecode, 2012. Tech. note, available at http://informatik.uni-leipzig.de/~berger/tr/2012-dienst.pdf.
13. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM TOCS*, 32(2):5:1–5:29, 2014.
14. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. USENIX*, pages 21–21. USENIX Association, 2011.
15. N. Hardy. The Confused Deputy (or Why Capabilities Might Have Been Invented). *ACM SIGOPS*, 22(4):36–38, 1988.
16. C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. OOPSLA*, pages 805–824. ACM, 2011.

17. W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *Proc. SOAP*, pages 1–6. ACM, 2014.
18. L. Li, A. Bartel, T. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proc. ICSE*, pages 280–292. IEEE, 2015.
19. L. Li, T. Bissyande, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. Technical report, University of Luxembourg, Fraunhofer SIT/TU Darmstadt, University of Wisconsin and Pennsylvania State University, 2016.
20. J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. ESEC/FSE*, pages 81–91. ACM, 2013.
21. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. CCS*, pages 229–240. ACM, 2012.
22. W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. The App Sampling Problem for App Store Mining. In *Proc. MSR*, pages 123–133. ACM, 2015.
23. I. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. Hassan. A Large Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software*, 31(2):78–86, 2014.
24. I. J. Mojica, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan. On Ad Library Updates in Android Apps. *IEEE Software*, 2015. Online first.
25. S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proc. ICSE*, pages 140–151. ACM, 2014.
26. Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proc. ASIACCS*, pages 328–332. ACM, 2010.
27. D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, , and Y. Le Traon. Combining Static Analysis with Probabilistic Models to Enable Market-Scale Android Inter-Component Analysis. In *Proc. Int. Symp. Principles of Programming Languages (POPL)*, pages 469–484. ACM, 2016.
28. D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. ICSE*, pages 77–88. IEEE, 2015.
29. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An essential Step Towards Holistic Security Analysis. In *Proc. USENIX*, pages 543–558. USENIX Association, 2013.
30. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-centric Security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.
31. A. Porter Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-delegation: Attacks and Defenses. In *Proc. USENIX*, pages 22–22. USENIX Association, 2011.
32. A. Sadeghi, H. Bagheri, and S. Malek. Analysis of Android Inter-App Security Vulnerabilities Using COVERT. In *Proc. ICSE*, pages 725–728. IEEE, 2015.
33. D. Sbîrlea, M.G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic Detection of Inter-Application Permission Leaks in Android Applications. *IBM Journal of Research and Development*, 57(6):10:1–10:12, 2013.
34. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.
35. N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proc. SIGMETRICS*, pages 221–233. ACM, 2014.
36. E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *Proc. Onward!*, pages 213–226. ACM, 2014.
37. F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. CCS*, pages 1329–1341. ACM, 2014.
38. Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proc. SSP*, pages 95–109. IEEE, 2012.