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

Alexander von Rhein [1], Thorsten Berger [2], Niklas Schalck Johansson [3], Mikael Mark Hardø [3], and Sven Apel [1]

[1] University of Passau, Germany
[2] University of Waterloo, Canada
[3] IT University of Copenhagen, Denmark

Technical Report, Number MIP-1504
Department of Computer Science and Mathematics
University of Passau, Germany
September 2015

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

Alexander von Rhein
University of Passau

Thorsten Berger
University of Waterloo

Niklas Schalck Johansson
IT University of Copenhagen

Mikael Mark Hardø
IT University of Copenhagen

Sven Apel
University of Passau

ABSTRACT

Mobile apps process increasing amounts of private data, giving rise to privacy concerns. Such concerns do not only arise 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 Following ideas from product-line analysis, the structure exploits redundancies among flows and thereby prevents 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-word app sets, demonstrating its scalability while maintaining reasonable accuracy.

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 unlimited access to communication channels, which may not be trustworthy. To prevent privacy escalation, mobile operating systems employ a range of methods, such as encapsulation 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 inside an app [4, 17].

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, 28, 29]. 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 [16]. 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 (e.g., a picture) to apps that can display it. If multiple target apps are installed, most systems display a choice dialog, possibly creating awareness for 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 verified to detect inter-app leaks, whether accidental or malicious. Even without finding actual leaks, detecting apps or app combinations that forward data is important, as such high-risk apps could be exploited for realizing data leaks.

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 [31, 21]). 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 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.

We argue that the limitation mainly lies in the current representation of inter-app data flows, which does not exploit *redundancies* between and inside apps. More importantly, existing approaches do not explicitly consider *variability* [5]—an app can be installed or not, thereby contributing to the global data flows that exist. Instead of duplicating detected flows, variability inside flows should be modeled explicitly. Recognizing synergies, we adopt concepts known from product-line analysis [30, 32, 18, 15, 23], which incorporate variability, to reduce redundancies and avoid a combinatorial explosion.

We present a *variability-aware* approach to analyze inter-app data flows. It relies on a graph-based data structure representing flows annotated with *presence conditions*—Boolean expressions over the presence and absence of apps. Furthermore, we lifted an analysis approach that analyzes data flows

inside individual apps to whole app sets by extending and combining existing tools; we use and aggregate their results in a graph that efficiently represents inter-app communication.

We demonstrate the scalability of 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. At the same time, our tool maintains an accuracy that is similar to existing tools focusing on intra-app analysis, likewise evaluated with two third-party benchmarks and with our own benchmark. As a further feature, our approach supports an *incremental* generation of the inter-app communication graph: When new apps are added or changed, the graph can be updated with information for such apps, instead of generating a new graph. Finally, we implemented a standard analysis that traverses the lifted graph and reports tainted inter-app data flows. We use it to show that we can identify high-risk apps or combinations of them that enable inter-app data-flows and could be exploited for privacy leaks. We contribute:

- An efficient variability-aware, 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 optionality of apps (variability).
- An algorithm implemented in our tool SIFTA to efficiently build (i.e., merge results from lower-level tools) and condense the variability-aware graph.
- A standard 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 if possible.
- Two new benchmarks: IACBENCH with nine apps (to evaluate accuracy) and a large-scale benchmark with 51 935 real-world apps (to evaluate scalability).

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

In summary, our approach lifts a single-app data-flow analysis to a scalable analysis of large app sets by introducing a variability-aware data structure to efficiently represent inter-app flows. It can be used to analyze communication in large numbers of apps early, *before* concrete combinations of apps are requested by a customer. The long-term vision is to move analysis from the mobile device to the app store, checking all possible combinations of apps.

2. BACKGROUND AND MOTIVATION

We introduce app communication mechanisms in 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 (inter-component) 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.

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 inter-component 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 scenario, the set of components is known.

Several analysis tools address this scenario. One comparatively precise tool is IccTA [17], which relies on FLOWDROID [4] and EPICC [25]. IccTA composes all components of an app into one "super" component encoding all the flows. A challenge is to connect components—that is, mapping intent calls of one component to incoming intents of another component. Therefore, the parameters of an intent object, which is instantiated at run time, need to be known and matched to intent filters. For this purpose, EPICC 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 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 predetermined. 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 *LocationReaderApp* reads the current location from the GPS device and sends 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 [6, 29, 8, 28, 16]. In principle, Android's permission system

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

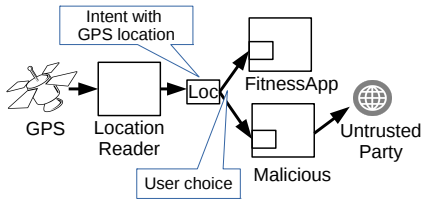


Figure 1: Inter-app communication example, where GPS location information is forwarded to untrusted receivers

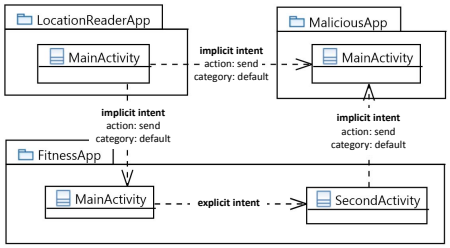


Figure 2: Example of inter-app communication

should prevent apps from accessing private data without user consent. However, Android permissions are not sufficient as commonly stated in the literature (see Sec. 7). In our scenario of Fig. 1, *MaliciousApp* might lack permission to read GPS data from the Android API, but can get it by interacting with *LocationReaderApp*. It does not matter 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* [28, 8] (a.k.a. *confused-deputy problem* [14]).

Analyzing inter-app communication is important for maintainers 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 high-risk apps that 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 [16], which, like IccTA, relies on FlowDroid and Epicc. DidFail runs FlowDroid to obtain data flows within each component in each app. Based on the intent parameters obtained with Epicc, DidFail connects the possible outgoing intents to intent receivers and builds a global communication 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"-component easily becomes large, with many (often redundant) flows. But the IccTA developers focus on intra-app communication in their experiments anyway [17]. Unlike IccTA, DidFail addresses inter-app communication explicitly. Yet, it stores detected flows in a list-like data structure without exploiting redundancies between the flows, harming scalability.

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, sending them 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. Furthermore, we extend the scenario slightly, adding a typical internal data flow inside *FitnessApp* and another accidental leak from *FitnessApp* to *MaliciousApp*, so that we have two leaks from a private source to a public sink.

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 increases: 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 has a cubic growth 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 example is extreme, our experiments (Sec. 5.2) showed similar results for DidFail's scalability. A key insight is that the problem 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, that addresses these challenges when analyzing inter-app communication. We reused parts of DidFail's code, but completely re-implemented the graph synthesis and the identification of tainted data flows.

2.5 Variability-Aware Data Structures

We assume that significant redundancies in the communication paths between apps exist. There are many reasons that back this assumption, such as commonly used intents (e.g., ACTION_VIEW, which requests that data is displayed to the user) or duplicated code [31, 21]. Consequently, exploiting redundancy using a dedicated data structure should have considerable influence on the size of inter-app flow graphs and their analysis time.

Technically, we use ideas from variability-aware product-line analysis [30]. In particular, we use the concept of *presence conditions*: 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 such that its generation and analysis scales much better than in the original DidFail implementation.

We borrow the presence-condition idea and its efficient encoding from other *variability-aware* approaches, such as variability-aware data structures [32, 10], static analysis [18, 7], parsing, type-checking [15, 23], and model checking [1].

3. REPRESENTING INTER-APP FLOWS

The communication between Android components 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 is 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 app internals and how to store the abstracted data. In Sec. 3.1, we discuss our core design considerations. In Sec. 3.2, we present a basic data structure that reflects the one used by DidFail [16]. In Sec. 3.3, we introduce our lifted, variability-aware representation.

3.1 Design Considerations

To represent app communication in the graph, we need to keep the information necessary to determine whether an intent can be accepted by a given component. Android makes this decision based on an intent's metadata and on a component's intent filter. An intent is defined by its sender component, an action key, a list of categories, and a mime type. An intent filter is defined by the component for which it controls incoming intents, by a list of action keys, a list of categories, and a list of mime types. Based on this information, Android matches intents with intent filters to deliver the intent to a component [11].

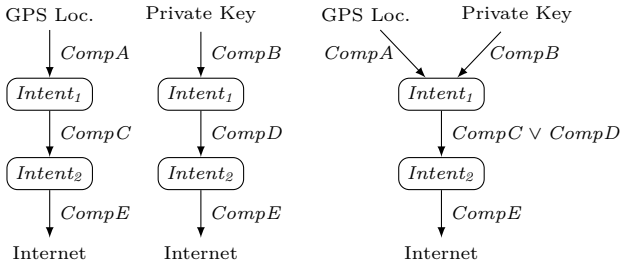
In Android, private data originate either from system API calls (e.g., location data) or from the user (e.g., a password entered in a text field). Likewise, to send data to untrusted receivers (private sinks), API functions are used (e.g., to send SMS, open network connections, or write into log files). To identify such private sources and public sinks, we rely on lists with API function signatures from previous work [3].

Based on these definitions, we can build an inter-app data-flow graph. We chose a directed graph as representation, in which 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. In this graph, we already abstract from many implementation details of the apps. For example, we do not consider sources of non-private data (e.g., the current time). Edges represent apps that receive and process intents, receive data from private sources or/and forward data to public sinks. We chose this node–edge mapping to avoid dangling edges and because intents are the central entities of inter-app communication.

We denote the set of all components with *Comp*, the set of all intents with *Int*, the set of all private source with *Priv*, and the set of all public sinks with *Pub*.

3.2 DidFail's Representation

Next, we present a graph data structure that is not variability-aware. It is similar to the data structure used internally in DidFail. The graph is defined as a set V_{DF} of nodes with $V_{DF} \subseteq Int \cup Priv \cup Pub$ and a set E_{DF} of directed edges with $E_{DF} \subseteq V_{DF} \times V_{DF} \times Comp$. V_{DF} contains all intents, private sources, and public sinks. For each component *comp* that receives data from a private source or intent *src* and that delivers data to an intent or public sink *sink*, the set of edges contains the triple $(src, sink, comp)$. 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.[2]



(a) basic (DidFail) (b) variability-aware (Sifta)

Figure 3: Example of a lifted, inter-app flow representation

Recall our thought experiment from Sec. 2.4: With an increasing number of apps, the graph quickly becomes very large and its generation expensive. The reason is that often different apps have (partly) similar functionality. For example, they receive data from the same sources (*Int* or *Priv*) and send data to the same sinks (*Int* or *Pub*). 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 two flows of private data (GPS location and private key) to a public sink (Internet). The second edges of the flows have the same source ($Intent_1$) and the same sink ($Intent_2$), and only differ by the inner component (middle edge).

3.3 Variability-Aware Representation

In Sifta, we represent flows in and across apps in a variability-aware 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. Finally, since a component is automatically present when its app is installed, we only store app names on edges (instead of component names).

Fig. 3b shows the same scenario as Fig. 3a, but using our lifted representation. The two edges from $Intent_1$ to $Intent_2$ are replaced by one, which has a presence condition denoting which components need to be installed to enable 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 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 [31, 21]).

4. IMPLEMENTATION

We implemented our approach in the tool Sifta. It is based on and reuses code from DidFail. Sifta implements

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

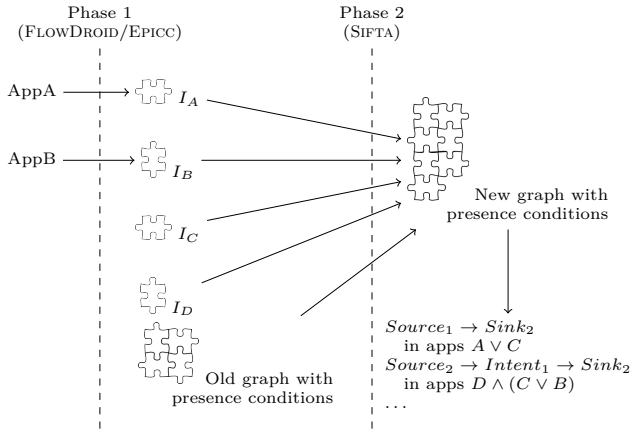


Figure 4: Sifta's inter-app analysis. I_A, I_B, I_C and I_D represent intermediary results generated by the first phase. I_C and I_D are reused from a previous run of the analysis. The intermediary results are added to the old graph, which is also reused from a previous analysis run.

all concepts discussed in Sec. 3.3 and additionally introduces handling of services and broadcast receivers, which are types of Android components that are not covered in DidFail.

Two-Phase Approach. Like DidFail, Sifta uses a two-phase approach. In the first phase, it uses FlowDroid and Epicc 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 data 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. 3.3). 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) determine 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 to potential private-data leaks, or to multiple concrete ones when edges have alternative apps (i.e., a presence conditions which more than one app). In contrast to product-line analysis [30], presence conditions in the graph are simple (disjunctions), such that we do not need SAT queries to generate the graph or to derive feasible flows from it.

Incremental Generation. Furthermore, we implemented an incremental graph-generation procedure—a feature that we needed for our largest experiment (Sec. 5.2). Since the main 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. 4. The apps *AppA* and *AppB* are analyzed for the first time. Two other apps were analyzed before (results from phase 1 are reused), and an old graph with information about more apps exists already. In the first phase, *AppA* and *AppB* 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. In the end, 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 apps have not changed). We would first remove all updated apps from the graph (delete the app from all presence conditions), and then integrate the FlowDroid/Epicc results for the updated apps.

Taint Propagation. Once the communication graph has been generated, it can be used in various ways. An example is a standard taint analysis: We simply report all paths from sources to sinks in the graph. These paths correspond to potentially malicious data flows. This is a taint-propagation analysis as the private (*tainted*) 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. 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 scenarios, we present our analysis of large sets of real-world apps in Sec. 5.2 where we measured to what extent our approach is able to exploit redundancies in app communication.

The potential leaks we detected in real-world apps can be used to inspect and fix apps. While this issue is orthogonal to our approach, we demonstrate in Sec. 5.3 that it can be used to identify high-risk apps that enable many flows.

5.1 Experiment 1: Accuracy

In the first experiment, *E1*, we measured the accuracy of our approach 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 currently achievable with state-of-the-art tools, we compare our results to those obtained by IccTA and DidFail. 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 (e.g., loops, intent chains) inter-app flows.
- ICC-Bench[3] contains 9 apps developed by the authors of Amandroid [33] with intra-app flows.

[3]Obtained from the authors of Amandroid.

- DroidBench[4] comprises 23 apps testing inter-component communication (provided by the IccTA authors [17]) and 3 sets of apps testing inter-app communication (provided by the DidFail authors [16]) among many more apps not relevant for our approach.

Our own benchmark IACBench contains test cases with undesired 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 used for our evaluation, we refer to the literature: Amandroid [33], DidFail [16], and IccTA [17].

All benchmarks comprise apps developed to test whether analysis tools capture the specific means of communication. The apps are much smaller and cleaner than real apps and are, thus, ideal to compare the accuracy of different tools.

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

Results. Table 1 shows precision and recall of *E1*. We discuss the different benchmarks, emphasizing test cases where Sifta produced worse results than DidFail or IccTA.

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.

On some apps from ICC-Bench, 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 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 [17].

DroidBench is a much larger benchmark that tests many possible communication paths. Table 1 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.

[4] http://github.com/secure-software-engineering/DroidBench

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 sendOrdered-Broadcast
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

The tests startActivity6 and startActivity7 check whether information retrieval from an intent is handled correctly. In these, 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 3 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.

These results demonstrate 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 combines all components of each test and analyzes them in one run. Sifta has to rely on the necessarily filtered information gained in separate per-component analyses, but this is exactly the lever that enables 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), but by additional matching criteria (for the mime types, cf. Sec. 4) we implemented.

5.2 Experiments 2–4: Scalability

To evaluate the scalability of Sifta and DidFail, we used three sets of apps:

- Experiment *E2*: IccRE is a set of 523 real apps coming with IccTA. These apps leak private user data through inter-component communication [17].
- Experiment *E3*: MalGenome is a set of 1260 real apps published by the Android Malware Genome Project [34]. They are known to be malicious, 51.1 % harvest user data, 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 sought to obtain popular apps that are likely to communicate (see Sec. 6, for details about the download process).

Methodology and Setup. In *E2*, we compared Sifta against DidFail, however, given DidFail's scalability limi-

Table 1: Results of Experiment *E1*: accuracy evaluation (IccTA results according to [17])

benchmark	test case	DidFail	Sifta	IccTA
IACBench (basic)	startActivity	+	+	n/a
	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 (IAC)	sendBroadcast1	n/i	+	n/a
	startActivity1	+	+	n/a
	startService1	n/i	+	n/a
DroidBench (ICC)	startActivity1	⊖	+	+
	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)
not applicable:	n/a	(intra-app tool on inter-app scenario)
false positive:	⊕	(a reported leak does not exist)
false negative:	⊖	(analysis misses an existing leak)
not implemented:	n/i	(DidFail on services or broadcasts)

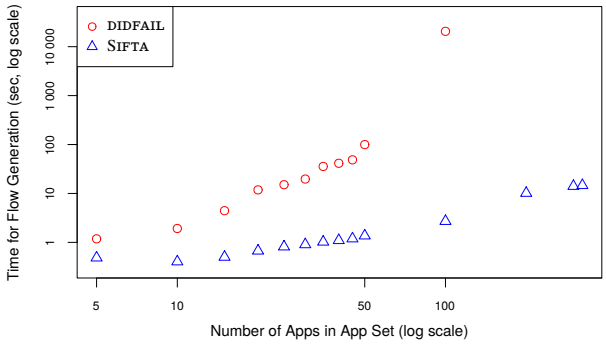


Figure 5: Results for Sifta and DidFail on IccRE. Both axes have a logarithmic scale.

tations, 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 the 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.

Experiment *E2* (IccRE). Fig. 5 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.

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 variability-aware 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 compact without losing information. Our variability-aware approach achieves this compression and enables efficient analysis of inter-app communication on large app sets.

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 the 20-minutes 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 31 seconds. We had to drop 9 further apps due to parsing errors on the FlowDroid or Epicc output.

The resulting graph contained 285 flows representing 839 apps. 250 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

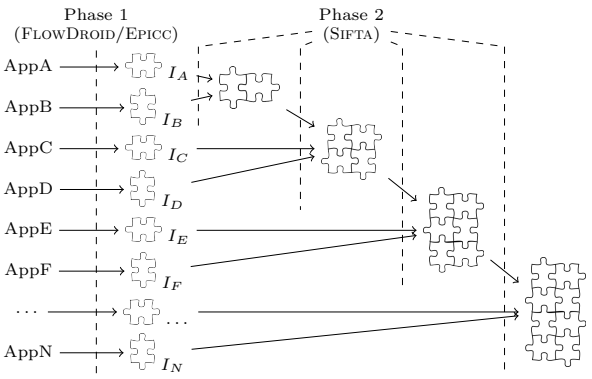


Figure 6: Incremental setup for *E4*. We chose this setup due to memory limitations when building the graph from all apps at once. In our experiment, each partition had a quarter of the 51 935 apps.

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.

Experiment *E4* (GooglePlaySet). To evaluate the scalability of Sifta on even larger app sets, we downloaded 172 779 apps from the Google Play store. We then 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 1704 days (4.6 years) in total (sum of 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 [4], 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 variability-aware 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. 6. The graph generation took 13 minutes, and each step used less than 3.5 GB RAM.

Overall, the graph contains 126 205 potential, 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. The edge's presence conditions contain an average of 31 (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

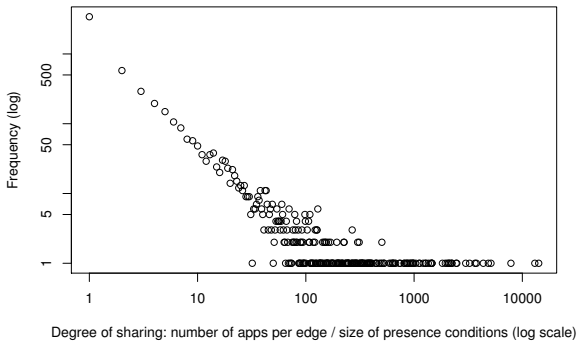


Figure 7: Frequencies of presence-condition sizes illustrating the reason for sharing in inter-app flows. The graph shows how often different numbers of apps on edges in the graph occur. Both axes have a logarithmic scale.

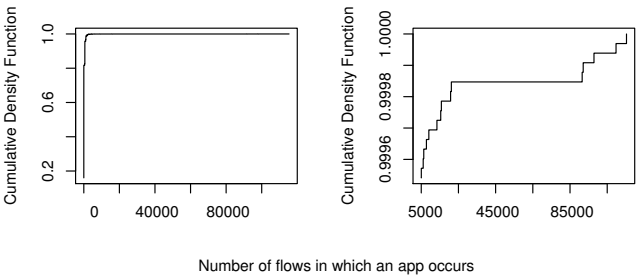


Figure 8: Cumulative Density Function (CDF) of the number of flows in which apps participate. The left plot shows all apps, the right plot only apps that are part of >5000 flows.

data from/to Bundle objects, which are the payload of intents. Fig. 7 shows the frequency of presence condition sizes (number of unique apps) in the graph. It shows that these 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 that does not tap into this sharing potential. For such large app sets, it would be infeasible to generate and store such a graph without variability awareness.

5.3 Identification of High-Risk Apps

Despite detecting actual leaks, our approach can be used to identify high-risk apps or app combinations. To demonstrate this potential, we provide insights obtained from analyzing the GooglePlaySet. Specifically, our approach allows to reason about the positions of apps in flows originating from app combinations, which is novel for large app sets.

Let us first look at how often apps occur in the flows' presence conditions. Apps that enable many flows have a larger potential for being exploited maliciously—it is certainly desirable to further analyze them. Fig. 8 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). These apps are of special interest, as securing their inter-app communication might have significant impact. Table 3 shows the top five of these apps (our supplementary website lists all).

Let us now take a look at the most frequent app, which

is called "Mild Tap Fashion-5" and belongs to the category "Communication" in Google Play. It was very rarely downloaded (10–50 downloads), asks for many permissions (e.g., phone-book access), and can receive a wide range of intents through a broad intent filter. It belongs to a series of apps from the same developer, all of which have similarly few downloads, have been almost never rated, and have been updated last in Sep. 2012. They also look very similar, essentially consisting of text fields for entering information which is then shared with contacts. We can safely conclude that these apps are relatively useless, abandoned apps in Google Play, but pose a risk for being exploited for privacy leaks.

Let us now specifically look at apps that forward data between apps (i.e., occur in the middle of a flow). These are particularly high-risk apps that can be exploited for constructing data leaks, with or without knowledge of the app author. We identified 23 "forwarder" apps (including all apps from Table 3), which receive data from an intent and deliver it to another intent. We provide the list of all "forwarder" apps on our supplementary website.

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, it is surprising that only 23 "forwarder" apps cause the large number of data flows in our graph. Yet, we can assume that in any realistic app pool, a certain number of such apps exist that highly complicate inter-app data-flow analysis. This empirical result again motivates our lifted, variability-aware approach to tame such complexity; it also shows that it might be fairly simple to reduce the risk for inter-app data leaks by securing or removing these apps.

6. THREATS TO VALIDITY

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 test 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 publicy available at our supplementary website. To evaluate scalability, we go beyond existing benchmarks in this area (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 [20]. In fact, obtaining a truly random sample of apps of an app store is almost impossible.

Table 3: Top five apps occurring in flow presence conditions

rank	app	flows
1	com.rekonsult.MTFashionAlert	115370
2	air.com.doitflash.ar.atelier	109743
3	com.krm.fbm	97888
4	com.merunetworks.IdentityWifi	92090
5	dk.southbound.instapaper	91538

However, this problem does not apply to our evaluation, since we do not aim at such a representative sample. Instead, we sought to obtain apps that are likely to communicate. Our strategy was to start with one of the most popular apps, facebook, to scan its website, following links to apps listed under "similar" and "more from developer". This process was continued, allowing us to download apps across various app categories. Yet, the strategy targeted 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 website.

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 [29], CHEX [19]), 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 on Sifta's 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 the only other tool supporting inter-app communication.

Internal Validity. In Sifta, we implemented the matching of intents to receiving 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 did not implement all (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 benchmark 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. We looked at several of these flows manually: The private data are visible to the code that generates the intent, but is 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 introduces a threat of removing 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 anyway.

7. RELATED WORK

Privacy Leaks in Apps. Our variability-aware data-flow representation and 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

extensively studied [6]. Several researchers argue that the permission system used in Android is insufficient to prevent tainted data flows. For instance, permissions are too coarse-grained [24, 26] and surprisingly rarely used in practice [25] (only 5 % of the publicly accessible components are protected). Furthermore, apps often ask for more permissions than are actually used [27], giving rise to accidental leaks.

Enck et al. [13] studied 1100 popular Android apps, analyzing their use of libraries and misuse of private information. They found that apps often access personal information, such as the IMEI number, often combined with account information. Many apps also heavily use of ad libraries [22], forcing acquisition of many permissions.

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.

Apart from DidFail, many tools focusing on intra-app communication exist. Yet, most stop at component boundaries, such as PermissionFlow [29], which does not incorporate intents and their flows, or FlowDroid [4], 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 [33] and IccTA [17]. We explained the difference between Sifta and IccTA already in Sec. 2. Amandroid is similar in accuracy to IccTA [17], 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. However, Amandroid targets only intra-app analysis (as confirmed by the developers).

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

Finally, dynamic analysis tools such as TaintDroid [12] 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.

Analysis of Software Product Lines. Our variability-aware flow representation is inspired by product-line analysis [9]. In product-line engineering, typically all possible products or systems (exponentially many, in the worst case) need to be described in a compact representation (e.g., code, models, requirements). 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, safe, structurally consistent) [30]. A mobile-device setup (a specific combination of apps) can be seen as a specific variant of a product line, where combinations of selectable options (a.k.a. *features*)—apps in this case—constitute a system variant [5].

In software product lines, the notion of presence condition is central. For model-based representations, Czarnecki and Antkiewicz [10] were the first to use presence conditions to annotate the optional parts of a model, from which concrete model variants can be derived by configuring the presence conditions. Walkingshaw et al. [32] provide a broader perspective on variability-aware data structures, discussing applications in product-line analysis and beyond.

The analysis of product lines relies on variability awareness, as analyzing all possible variants is usually intractable. For instance, in model checking of product lines, presence conditions are used to compactly store a graph of program states of an entire configurable system [1]. A survey of variability-aware analyses gives an overview of related techniques [30].

8. CONCLUSION

We presented a variability-aware approach to inter-app data-flow analysis of mobile apps. It effectively combats the combinatorial explosion that previous analysis techniques faced. At its heart is a lifted 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 [16]). 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 website.

The approach enables a class of analyses that needs to reason about all possible combinations of apps at once. We implemented a taint-propagation analysis on top of it, which identified potentially malicious data flows across a maximum of eight apps. We also demonstrated that we can now reason about such communication chains and the position of apps or app combinations in inter-app flows. We identified a small set of 23 "forwarder" apps that highly impact the potential flows in our set of 51 935 real apps. It is surprising that such a small set of apps is responsible for a large part of the reported flows. Running more heavyweight and more accurate analyses (e.g., model checking) on such identified apps, and potentially securing their communication could have a high positive impact on data flows and data security.

In future work, we strive for further insights into inter-app data-flow structures (e.g., conducting a more detailed network analysis on Sifta's data flow graph). Specifically, we want to determine whether clusters of apps or flows exist and what their characteristics are.

We also aim at improving Sifta. For instance, running a clone-detection on apps could significantly speed up Sifta's first phase by avoiding redundant executions of FlowDroid and Epicc, while maintaining the current accuracy of Sifta. Another promising track is to exploit dynamic variability within apps, which is commonly realized using configuration parameters or by checking Android API constants. Detecting and including this dynamic variability into our data-flow graph would allow reasoning about inter-app flows that depend on certain Android versions or app configurations.

Acknowledgment

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.

9. 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, and E. Bodden. Susi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. Technical Report TUD-CS-2013-0114, University of Darmstadt, 2013.

[4] 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.

[5] 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.

[6] N. Al Bidani and M. Vigant Raffay. A Systematic Literature Review of Mobile Inter-Application Security. Master's thesis, IT University of Copenhagen, 2014.

[7] 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.

[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] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[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] 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.

[12] 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.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. USENIX*, pages 21–21. USENIX Association, 2011.

[14] N. Hardy. The Confused Deputy (or Why Capabilities Might Have Been Invented). *ACM SIGOPS*, 22(4):36–38, 1988.

[15] 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.

[16] 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.

[17] 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, 2015.

[18] 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.

[19] 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.

[20] 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, 2015.

[21] 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.

[22] 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. preprint.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proc. CCS*, pages 627–638. ACM, 2011.

[28] 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.

[29] 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.

[30] 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.

[31] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proc. SIGMETRICS*, pages 221–233. ACM, 2014.

[32] 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.

[33] 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.

[34] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proc. SSP*, pages 95–109. IEEE, 2012.