Together Strong: Cooperative Android App Analysis

Felix Pauck
Paderborn University
Paderborn, Germany
fpauck@mail.uni-paderborn.de

Heike Wehrheim
Paderborn University
Paderborn, Germany
wehrheim@uni-paderborn.de

ABSTRACT

Recent years have seen the development of numerous tools for the analysis of *taint flows* in Android apps. Taint analyses aim at detecting data leaks, accidentally or by purpose programmed into apps. Often, such tools specialize in the treatment of specific features impeding precise taint analysis (like reflection or inter-app communication). This multitude of tools, their specific applicability and their various combination options complicate the selection of a tool (or multiple tools) when faced with an analysis instance, even for knowledgeable users, and hence hinders the successful adoption of taint analyses.

In this work, we thus present CoDiDroid, a framework for *cooperative* Android app analysis. CoDiDroid (1) allows users to ask questions about flows in apps in varying degrees of detail, (2) automatically generates subtasks for answering such questions, (3) distributes tasks onto analysis tools (currently DroidRA, FlowDroid, HornDroid, IC3 and two novel tools) and (4) at the end merges tool answers on subtasks into an overall answer. Thereby, users are freed from having to learn about the use and functionality of all these tools while still being able to leverage their capabilities. Moreover, we experimentally show that cooperation among tools pays off with respect to effectiveness, precision and scalability.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Domain specific languages.*

KEYWORDS

Android Taint Analysis, Tools, Cooperation, Precision.

ACM Reference Format:
Felix Pauck and Heike Wehrheim. 2019. Together Strong: Cooperative Android App Analysis. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338915

1 INTRODUCTION

In the field of software analysis, an ongoing race between attackers and defenders is taking place. The speed of this race is constantly increasing on both sides. Attackers find new ways to hide their malicious behavior while defenders develop more and better analyses to detect attacks. However, the user – unwillingly being a third party in this race – typically cannot sufficiently benefit from the advances of the defenders and thus cannot catch up. Reasons for this are various, including the missing knowledge about the existence of up-to-date tools, their usage and ways of combining them, the unavailability of tools or of resources to run tools.

This exactly characterizes the present situation in the area of analyses for *data leaks* in Android apps. Data leaks permit the leakage of private data in smartphones (like contact data or – more critical – banking or health data) to the outside. Conceptually, a data leak is a flow of data from a private *source* to a public *sink*. Data leaks might arise out of accidental coding mistakes, but might also be intentionally placed in apps by malicious attackers. In the past, researchers have come up with various techniques and tools for leak detection, ranging from static [2, 8, 12, 15, 18, 20, 22, 27, 31] over dynamic [14, 32] and hybrid [1] analyses to methods built on logical reasoning [9]. Tools furthermore often specialize to certain programming features in apps which complicate the analysis (e.g. reflection [7, 23], inter-app communication [24, 33]). Moreover, such tools require varying amounts of resources (computation time, power and memory) – often exceeding a normal user's capacities –, different operating systems and/or require specialized additional software or hardware. Hence, it is only the absolute expert who can fully leverage the plethora of existing tools.

In this paper, we thus propose CoDiDroid, a framework for automatic <u>co</u>operative (and <u>di</u>stributed) An<u>droid</u> app analysis. For a given (set of) app(s) and a given analysis question, CoDiDroid automatically divides the analysis task into several subtasks, distributes subtasks onto tools, dispatches them and at the end merges tool answers into a final answer to the initial question. As language for formulating questions and for interfacing between different tools, we employ the domain-specific language AQL (Android App Analysis Query Language) [28, 37]. We extend the AQL in several ways, in particular to enable novel combinations of tools. For the unexperienced user, CoDiDroid for instance offers this cooperative analysis on a website with a default analysis question; for a user knowledgeable in AQL, CoDiDroid allows to specify the selection of tools and subtasks in varying degrees of detail.

The power of CoDiDroid lays in the principle of *cooperation*: While previous approaches only use very limited, hardcoded forms of tool combinations, CoDiDroid can in principle compose arbitrary tools extracting information about flows in apps. This flexibility in the combination allows for an easy integration of a new tool, upgrade to new tool versions or exchange of tools. Cooperative app analysis furthermore cannot only be employed to *solve* an analysis question, but also to *crosscheck* the correctness of an analysis answer of one tool by another.

While a number of basic building blocks for a cooperative analysis already exist, we also identified gaps where tools are either lacking, do not exist as black box standalone tools or only compute (too) imprecise information. As a further contribution of this work we have closed two such gaps. The first development concerns the implementation of a new tool for the inference of inter-component communication links (*ICC links*) called PIM. PIM precisely computes links between intent-sinks and intent-sources as collected by IC3 [25] or Epicc [26], and thereby contributes to a rigorous analysis of inter-component communication. Our second contribution is NOAH for the analysis of calls to native libraries (via the Java Native Interface) in apps. Current tools typically ignore native calls while NOAH also identifies sources and sinks in native methods.

We have carried out a number of experiments on the standard benchmark DroidBench [2] (plus additional inter-app communication cases) to test the feasibility, precision and scalability of CoDiDroid. The experiments show that cooperative analysis is effective in that it can (a) increase the precision, (b) have lower runtimes than other approaches, (c) increase the scalability and (d) enable the detection of previously undetectable data leaks.

To summarize, this paper makes the following contributions:

- We propose the novel framework CoDiDroid for cooperative Android app analysis,
- we develop and implement a technique for the automated distribution and dispatch of analysis tasks,
- we design and implement two new analysis tools (PIM and NOAH) readily usable within CoDiDroid,
- we provide associated new benchmark cases for inter-app communication scenarios, and
- we show the effectiveness and efficiency of CoDiDroid in large-scale experiments.

The paper is structured as follows. The next section introduces foundations of taint analyses, tools and the analysis query language AQL. Section 3 presents the general approach of cooperative Android app analysis. In Sections 4 and 5 we present our experimental studies and discuss their results. Related work is presented in Section 6 and Section 7 finally concludes.

2 FOUNDATIONS

In this section we start by explaining basic concepts, present the tools which are currently part of CoDiDroid and discuss the domain-specific query language AQL.

2.1 Taint Analyses

The detection of data leaks in Android apps is predominantly carried out by *taint analyses*. Taint analyses track information from sources to sinks by "tainting" elements (objects, fields, ...) and tracking their flows through the program. A source in this context is a statement extracting sensitive information (e.g. contact data, location information, device identifiers). A sink in contrast is a statement that leaks data to the outside, e.g. via SMS or the Internet. Information flows between sources and sinks detected by taint analyses are called *taint flows*. Some of these sources and sinks require a *permission* to be granted by the Android system or the user. Any possibly required permission is specified in a so-called manifest.

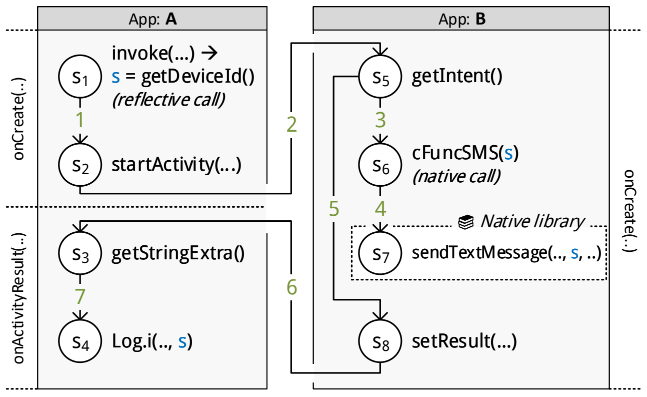


Figure 1: Running example: Flows in 2 apps

In order to successfully detect taint flows, a number of challenges have to be mastered by an analysis tool, some of which are given below.

Sensitivities A tainted element might be a whole object or just a field. Any flow accessing a tainted element might be constrained by its own context and or certain conditions. Ideally, these and similar aspects should be taken into account. Precise taint analyses should at the best be flow, context, field, object, path and thread sensitive.

Native calls The apps might contain calls to native libraries via the Java Native Interface (JNI). When the analysis does not inspect library code, it might miss flows.

Inter-component communication Android employs a specific concept of inter-component and inter-app communication (ICC/IAC). This communication is realized through *intents*: apps can send intents which are received by other apps when they match the receiver's predefined intent-filters. A precise analysis has to track flows via intents, and in particular needs to be able to detect matching pairs of intents and intent-filters.

Reflection Attackers might use the Java reflection mechanism to hide e.g. accesses to sources. The analysis has to be able to resolve reflective calls.

A further challenge is efficiency. To decrease runtime and memory consumption, (static) analysis tools often compute over-approximations of taint flows, resulting in a large number of false positives.

Figure 1 introduces our running example of two apps exhibiting several of the above listed challenges. The figure depicts the taint flows an analysis tool should be able to detect. App A in statement s_1 contains a use of reflection extracting data from a private source (the device id). This tainted value is flowing to statement s_2 (flow 1). In s_2, app A sends out an intent the result of which is collected in s_3. This result flows into the sink in the logging statement in s_4 (flow 7). The intent matches the intent-filter of app B (flows 2 and 6) in which the data transferred via the intent flows into both statements s_8 and s_6 (flows 5 and 3). Finally, the library code for the native call in statement s_6 contains the sink `sendTextMessage(...)` and hence introduces flow 4. The two expected taint flows hidden in the running example thus start in `getDeviceId()` (s_1) and end in `sendTextMessage(...)` (s_7) and in `Log.i(...)` (s_4), respectively.

Currently, no publicly available Android app analysis tool alone is able to detect both flows.

2.2 Analysis Tools

The objective of our work is to increase the effectiveness of Android app analysis by cooperation. As we will see later, a cooperative analysis of the example in Figure 1 can detect both flows. To this end, our framework CoDiDroid employs a number of existing tools which we briefly describe in the following. The choice of tools is guided by our overall objective of cooperation: we require tools which provide basic building blocks of a complex analysis and which can be flexibly combined. Furthermore our choice is driven by recent studies, e.g. [29, 30], which give insight into a tool's performance.

- DroidRA [23] is a tool for resolving reflection. To do so, it identifies the signature of classes and methods that are instantiated or called through reflection via a *static constant propagation analysis*.
- FlowDroid [2] is a tool for the detection of intra-component taint flows[1]. It considers real sources and sinks as well as intent-sources and -sinks. The identification of sinks and sources is followed by a *static* taint analysis.
- HornDroid [9] is a static analyzer based on logical reasoning. It uses Horn clauses to decide which sinks are able to leak sensitive data. Due to the use of logic, it is able to distinguish definite from only potential leaks.
- IC3 is a tool for detecting intent-sinks and -sources via a *static constant propagation analysis*. It has been developed by Octeau et al. [25] and updated by us in order to work with up-to-date Android APIs[2]. It computes precise attribute-sets containing information about the action, category and data of intents and intent-filters and maps each of them to a statement.

To enable a flexible combination of tools in a taint analysis, we are lacking a standalone tool for (a) the identification of sinks and sources in native code, and (b) the precise matching of intents with intent-filters, i.e. the comparison of attribute sets as computed by IC3. Our first contribution towards cooperative Android app analysis is thus the development of two new tools.

- NOAH (Native Over-Approximation Handler) is a tool for the detection of Android sources or sinks called *internally*, i.e. within code of a native library. With this information in place it can detect flows ending in and starting at native method calls.
- PIM (Precise Intent Matcher) is a tool computing connections between intent-sinks and -sources by comparing the mapped attribute-sets as delivered by IC3. To do so, it uses an Android device or virtual device (emulator) and asks the Android system installed if a certain pair of intent and intent-filter fits. It thus performs a *dynamic* analysis.

[1] The up-to-date version 2.7.1 of FlowDroid also incoporates inter-app flows, but for the purpose of cooperative analysis we prefer the base version.

[2] Additionally, we reported this issue on github (https://github.com/siis/ic3/issues/28).

2.3 Android App Analysis Query Language

To realize our CoDiDroid framework, we use the domain-specific language *Android App Analysis Query Language* (AQL), developed and maintained by us [28, 37]. This language allows us to formulate queries (AQL-*Queries*) composed of questions (AQL-*Questions*). Results to such queries are given in the form of AQL-*Answers*. AQL is used in the communication with the user as well as for communication between tools and our framework CoDiDroid.

An AQL-Question mainly consists of two parts, the property of interest (`<poi>`) and the target (`<target>`). With these two basic elements we can formulate queries to ask for e.g. flows, permissions, intents or intent-filters inside a whole app or parts of it such as a certain class, method or statement. An unknowledgeable user might always want to use standard queries, like the one below to get all flows inside some app `A` stored in file `A.apk`:

$$\overbrace{\texttt{Flows}}^{\texttt{<poi>}}\ \textbf{IN}\ \overbrace{\texttt{App('A.apk')}}^{\texttt{<target>}}\ \textbf{?}$$

or to get all flows from `A` to `B`:

```
Flows FROM App('A.apk') TO App('B.apk') ?
```

A more knowledgeable user might ask for more details, like

```
Permissions IN
Method('onCreate(...)')->App('A.apk') ?
```

to find all permissions required by a statement inside `onCreate(..)`. Considering the running example this would return the permissions `READ_PHONE_STATE` required by `getDeviceId()` inside the `onCreate(...)` method.

The AQL also allows to further process the answers given to certain questions. To this end, a number of *operations* on queries (or rather, their answers) are already available (and can further be extended via configuration). Note that – conceptually – answers are *sets* of data about flows, permissions, intents, intent-filters, intent-sinks and intent-sources.

- The `UNIFY`-operator can be used to merge multiple answers into a single one by applying a union operator on the answer sets. In doing so, it implicitly removes redundant elements.
- The `FILTER`-operator is used to extract specific data from an answer set.
- The `CONNECT`-operator first applies `UNIFY` and then computes a *closure* on the data, i.e. constructs the transitive closure of flows and connects intent-sinks with -sources using some simple ICC link computation.
- The `MATCH`-operator is a more sophisticated version of `CONNECT`, currently implemented using PIM for accurate ICC link computation.
- Finally, the `CHECK`-operator can be used to have another tool crosscheck an answer to a query.

In addition, the AQL embodies facilities for (1) specifying the preprocessing of app code

```
Flows IN App('A.apk' | 'DEOBFUSCATE') ?,
```

(2) selecting certain tools when specific features are given

```
Flows IN App('A.apk') FEATURING 'NATIVE' ?,
```

or (3) using particular tools to answer a query

```
Flows IN App('A.apk') USING 'NOAH' ?.
```

```
<answer>
  <flows>
    <flow>
      <reference type="from">
        <statement> getDeviceId() </statement>
        <method>onCreate(...)</method>
        <classname>MainActivity</classname>
        <app>
          <file>.../AppA.apk</file>
          <hashes>...</hashes>
        </app>
      </reference>
      <reference type="to">
        <statement>sendTextMessage(...)</statement>
        <app>.../AppB.apk</app>
      </reference>
    </flow>
    <flow>
      <reference type="from"> getDeviceId() </reference>
      <reference type="to"> i(...) </reference>
    </flow>
  </flows>

  <intentsinks>
    <intentsink>
      <target>
        <action>...codidroid.codirunex.SMS</action>
      </target>
      <reference type="from">
        startActivityForResult(...)
      </reference>
    </intentsink>
  </intentsinks>

  <intentsources>
     <intentsource>
      <target>
        <action>...codidroid.codirunex.SMS</action>
        <category>android.intent.category.DEFAULT</category>
      </target>
      <reference type="to"> getIntent() </reference>
    </intentsource>
  </intentsources>
</answer>
```

Listing 1: Shortened AQL-Answer for the running example

Technically, AQL-Answers are encoded in XML format. The specific structure is determined by an XML Schema definition (XSD) file [28, 37]. An answer can hold information about all types of properties of interest. As an example, Listing 1 shows the two taint-flows contained inside our running example as well as the intent-sink and intent-source required to be able to find both flows. The first reference to the reflectively called `getDeviceId()` statement is almost fully given. For all other references that are mentioned we reduced the amount of information provided to increase readability. The denoted intent-sink and -source both come with one action element. In both cases it contains the action string `codidroid.codirunex.SMS`. The action string is one element that has to match in order to infer a link between intent-sink and intent-source. Note that the full answer contains all flows depicted in Figure 1.

3 APPROACH

To build CoDiDroid we constructed an environment for cooperative analyses that allows us to configure arbitrary networks of analysis tools. An instance of such a network consists of a number of so-called AQL-Systems configured to access tools realizing basic functionality of app analyses. Any AQL-System in such a network can be *frontend* or *backend* which communicates with other systems in the network through the AQL. Once a query is processed by a frontend the following procedure is started: i) check which tools are available in the network and which queries can be answered by them, ii) run a lightweight static analysis to compute the *features* (app characteristics) used in the targeted apps, iii) based on this information split the query according to the configured *strategies*, and iv) decide which tool or tool combination has the highest priority to answer each part of the divided query.

In order to execute these four steps every system needs to be configured. A configuration fixes (1) the set of available tools and their functionality (i.e., ability to answer queries and priority for certain features), (2) technical information about running tools, (3) converters to AQL from tool outputs, and (4) the configurable strategies. Hence, to add a new or updated tool to the whole network only two adaptions need to take place considering (2) and (3): Add the required technical information to the associated configuration files and provide a matching converter[3].

The AQL-Systems which can be used in CoDiDroid come in different forms:

AQL-System [37] is simply the system itself accessible via command line or a graphical user interface. It is also usable as a library in other applications such as the following.

AQL-WebServices [38] wrap an AQL-System into a webservice lifting its availability from local to network level. These webservices can be accessed by other AQL-Systems of any kind.

AQL-Online [36] is a website implementation representing an online user interface for arbitrary AQL-WebServices.

BREW [39] is an automatic benchmarking system using an AQL-System to analyze apps in benchmark sets.

The up-to-date CoDiDroid instance of such an AQL-System network is depicted in Figure 2. It is split into one frontend system (**1.**) and two backends (here, **2.** and **3.**). All backends run AQL-WebServices. The frontend distributes the analysis tasks onto the backends and to this end knows which queries they can process. Backend **2.** runs all static analysis tools in a virtual machine under Linux. The other backend runs all dynamic analyses, in particular the tool PIM on an Android device or emulator.

AQL-Systems store all the analysis queries they process together with the obtained answers. This allows for a *re-use* of answers and can significantly speed-up analysis for later cases.

The key ingredients of AQL-Systems enabling cooperation are *strategies* for dividing analysis tasks into subtasks, or – when viewed on the level of AQL– for splitting queries into questions and operators.

[3]Options to add converters can be found in the following tutorial: https://github.com/FoelliX/AQL-System/wiki/Add_tools (The average size of all available converters is about 150 LOC)

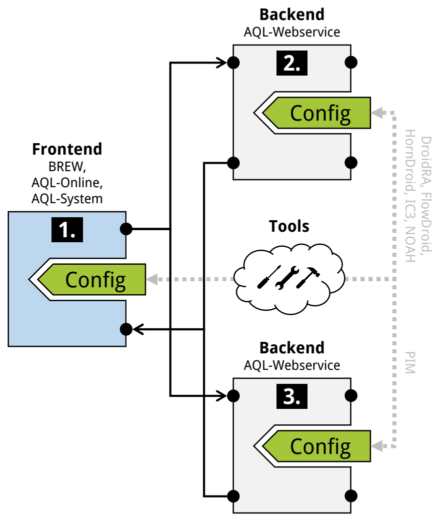


Figure 2: Structure of CoDiDroid

Strategy I: Inter-Component and Inter-App Communication. This strategy is employed whenever the AQL-System detects that more than one component or app is involved. The subtasks to be generated then concern (a) the detection of intra-component taint flows in every involved component, (b) the collection of intent sources and sinks in these components, and (c) the matching of intent-sinks and -sources and – with this at hand – the computation of closures of flows.

For an example, suppose that the query involves apps `A.apk` and `B.apk`. The AQL-System then constructs the following more detailed query specifying the subtasks.

```
MATCH [
    Flows IN App('A.apk') ?,
    Flows IN App('B.apk') ?,
    IntentSources IN App('A.apk') ?,
    IntentSinks IN App('A.apk') ?,
    IntentSources IN App('B.apk') ?,
    IntentSinks IN App('B.apk') ?
]
```

Depending on the configuration, the AQL-System afterwards selects tools being able to answer the questions contained in the query and runs them. Similarly, it chooses a tool for the AQL-Operator `MATCH`. The current configuration of our AQL-Systems employs FlowDroid for the first two questions, IC3 for the last four and PIM for the matching operator.

Strategy II: Reflection. Strategy II is employed when the apps involved in the query use reflection, i.e. the parsing of the `.apk`-file detects the use of `invoke` (or similar methods). In this case, every question about intra-component flows in an app `A.apk` is replaced by

```
Flows IN App('A.apk' | 'DEOBFUSCATE') ?
```

The keyword `DEOBFUSCATE` next to the path identifying the app specifies that the app needs to be preprocessed before the analysis. The current configuration maps this preprocessing task to the tool DroidRA.

Strategy III: Native Code. Strategy III is applied when the app(s) contain calls to native methods via the JNI interface. In this case, the analysis has to inspect the code of the native methods as well. Two subtasks have to be done for this: (a) it needs to be detected whether the native call can be considered to be a source or sink (so that the intra-app taint analysis can take this into account), and (b) the taint flows inside the native method need to be computed. Every question about intra-app flows is thus replaced by

```
CONNECT [
    Flows IN App('A.apk' | 'UNCOVER') ?,
    Flows IN App('A.apk' | 'UNCOVER')
          FEATURING 'NATIVE' ?
]
```

Here, the keyword `UNCOVER` stands for a specific preprocessor deriving additional sources and sinks. The second question explicitly specifies the flow analysis to be done by a tool capable of analyzing native methods. Finally, the `CONNECT` operator unites the two answers and computes the closure. Note that we do not require `MATCH` here since this is an intra-component analysis.

Our current configuration selects NOAH both for the `UNCOVER` preprocessing and the second question. The first question (after preprocessing) is delegated to FlowDroid and the `CONNECT` operator's default implementation is hardcoded in the AQL-System.

Whenever we detect the necessity of using strategy II or III, the preprocessing step (like `UNCOVER` or `DEOBFUSCATE`) needs to be added to all questions about this app.

Strategy IV: False Positives. Strategy IV is only employed when a high level of precision is required, i.e., when the analysis should not deliver so many false positives. As taint analysis tools typically compute overapproximations of taint flows, this can be achieved by having a second tool crosscheck the results of the first. In case that the second tool cannot confirm flows of the first, these are discarded.

For the queries, we currently have to explicitly specify the crosschecking tool (to override the configuration specifying priorities of tools for certain questions and features). As an example, consider the following query crosschecking computed intra-app taint flows.

```
CHECK [
    Flows IN App('A.apk') ?,
    Flows IN App('A.apk') USING 'HornDroid' ?
]
```

The `CHECK` operator takes the answer of the first question and checks whether the taint flows in there also occur in the second answer. For this, it is enough when only parts of a taint flow are found in the second answer.

In the example query, CoDiDroid makes use of HornDroid to compute the second answer. Since HornDroid itself is not able to compute *complete* taint flows but identifies leaking sinks with high accuracy, it is a perfect candidate for this purpose. Hence, CoDiDroid is able to combine completely orthogonal approaches as to get the best of all worlds.

Except for strategy IV, all strategies are standardly employed in a cooperative analysis. For our running example of Figure 1, the query

```
Flows FROM App('A.apk') TO App('B.apk') ?
```

thus gets divided into

Table 1: Tools used in CoDiDroid

Tool	**Date** (Version/Commit ID)
DroidRA [45]	April 2017 (b766a32)
FlowDroid [47]	April 2017 (Nightly)*
	January 2019 (2.7.1)
HornDroid [48]	October 2018 (cd52ba4)
IC3 [49]	March 2018 (196d68b)
NOAH [51]	February 2019 (tba)
PIM [52]	February 2019 (tba)
Amandroid [34]	November 2017 (3.1.2)
DIALDroid [41]	April 2017 (5df5734)
DidFail [43]	March 2015 (latest)
DroidSafe [46]	June 2016 (final)
IccTA [50]	February 2016 (831afaa)
ApkCombiner [35]	March 2018 (196d68b)

* used for comparability in RQ1.

```
MATCH [
   Flows IN App('A.apk' | 'DEOBFUSCATE') ?,
   CONNECT [
     Flows IN App('B.apk' | 'UNCOVER') ?,
     Flows IN App('B.apk' | 'UNCOVER')
           FEATURING 'NATIVE' ?
   ],
   IntentSources IN App('A.apk' | 'DEOBFUSCATE') ?,
   IntentSinks IN App('A.apk' | 'DEOBFUSCATE') ?,
   IntentSources IN App('B.apk' | 'UNCOVER') ?,
   IntentSinks IN App('B.apk' | 'UNCOVER') ?
]
```

CoDiDroid automatically distributes the subtasks onto the two backends. At the end, the analysis returns the two taint flows from `getDeviceId()` to `sendTextMessage(...)` and `getDeviceId()` to `Log.i(...)`. None of the existing (publicly available) taint analysis tools is able to compute both flows.

4 EXPERIMENT DESIGN

To evaluate our approach for cooperative Android app analysis, we carried out a number of experiments. Our first question concerned the *feasibility* of cooperative analysis, in particular the possibility of using CoDiDroid as a black box which automatically generates and distributes analysis tasks as well as executes them. To this end, we implemented our approach and employed it in the experiments detailed below. With this, the general feasibility of the approach is demonstrated. The implementation as well as all experiment resources and in particular the results are publicly available[4]. The following research questions target more specialized properties of the approach than plain feasibility.

4.1 Research Questions and Experiments

Besides feasibility, we were interested in the following research questions.

RQ1 How does CoDiDroid compare to other analysis approaches with respect to precision?

RQ2 How well does (the inter-app analyis of) CoDiDroid scale to higher numbers of apps?

[4]https://FoelliX.github.io/CoDiDroid [40]

RQ3 How well does CoDiDroid scale to larger (real-world) apps?

Another aspect of evaluation could have been the execution time of analysis tools. Distributing the execution of analysis tools to various machines allows users to access resources that, for example, may never become available locally. Thereby, analysis time becomes less dependent on the locally available resources. Furthermore, our framework allows for the parallel execution of analyses on subtasks, and thus in general speeds up analysis. Hence we did not include an explicit research question about runtime. We only inspect runtime as one aspect of the first scalability research question.

Benchmarks. For the experiments, we employed the standard benchmarks of DroidBench [44] (version 3.0) plus some recent extensions developed in [29]. DroidBench partitions the benchmarks into categories (according to specific features occurring in the apps). Each benchmark case can furthermore contain one or more apps, and each app can have one or more components with one starting app. For RQ2 we furthermore created two additional benchmarks scenarios (see below), for RQ3 we employed benchmarks from DIALDroid-Bench [42]. Finally, we constructed two apps exhibiting the flows of our running example in Figure 1.

Tools. For the comparison with other approaches, we employed some of the tools which are already part of CoDiDroid, but this time as standalone tools. In addition, we used the state-of-the-art tools ApkCombiner [21] (merging two or more apps into one), IccTA [22], DIALDroid [8], DidFail [20] (all three are different combinations of FlowDroid and IC3), Amandroid [31], and DroidSafe [18]. These tools have been part of the reproducibility study of [29] which allows us to directly use the study's F-measure results, and the tools are furthermore all available for own additional experiments. For these, we employed the tools in the versions listed in Table 1.

Benchmarking. The benchmarking itself is carried out with Brew, using an AQL-System to run tools. Brew generates an initial query by inspecting the apps in the benchmark case: in case of a single app, the query is

```
Flows IN App('A.apk') ?
```

in case of two or more apps, Brew generates the following query for every pair of apps (direction depending on the sources of the starting app and sinks):

```
Flows FROM App('A.apk') TO App('B.apk') ?
```

While our own approach CoDiDroid splits up these queries, the other tools only get the overall query converted to their input formats and then run their analysis. At the end, Brew collects all answers.

For **RQ1**, we ran CoDiDroid on the extended DroidBench benchmark and calculated the F-measure scores. Due to the existing reproducibility study on exactly this benchmark set, there was no need to recompute the F-measures of other tools. We simply used the results of the study. For evaluation of our new tool PIM for precise intent matching, we used the 224 benchmark cases of the extension of DroidBench in [29]. They just contain *inter-component* flows and hence can be specifically used for evaluating intent matching.

For **RQ2**, we were interested in the scalability of CoDiDroid for benchmark cases requiring inter-app analysis. Furthermore,

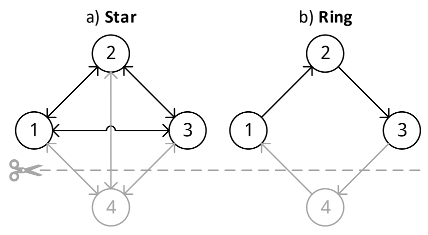


Figure 3: Star & Ring Benchmark Scenarios

we wanted to compare CoDiDroid with approaches specifically designed to support inter-component analysis (Amandroid and IccTA). IccTA is based on FlowDroid and uses IC3 to lift up FlowDroid's analysis to inter-component level. Nevertheless, since it is based on FlowDroid it can only analyze one app at a time. In consequence, before analyzing multiple apps, these have to be *merged*. Typically, the tool ApkCombiner is used for this purpose. Similarly, when running Amandroid we needed to preprocess the apps with ApkCombiner. Thus, both Amandroid and IccTA require the use of ApkCombiner to reach inter-app level.

As benchmarks for this research question, we used the category "InterAppCommunication" of DroidBench. Since the cases in there only involve two apps each, we furthermore constructed additional benchmarks. Figure 3a and 3b, respectively, depict the flows in the *Star* and *Ring* benchmarks. We vary the number of apps participating in a Ring (3 or 4) and a Star (3, 4, 5, 6 or 10). Note that the analysis should also return flows arising from the transitive closure of links, like from (a source in) app 1 to (a sink in) app 3 in case of Ring. We ran two versions of the experiments: (1) running Ring($n+1$) and Star($n+1$) *after* the analyses for Ring(n) and Star(n), respectively, to evaluate whether the re-use feature of CoDiDroid actually pays off, and (2) running them *without* prior analyses of the n-app versions. This first idea is depicted by the scissors in Figure 3.

For **RQ3**, we wanted to evaluate the scalability of CoDiDroid to large, real-world apps. When dealing with real-world apps experiments become more difficult, mainly because of missing information about the ground truth. In other studies, the corpus of evaluation often consists of the 10 to 1.000 most downloaded, best rated apps on Google's PlayStore[5]. Based on this corpus, either the sole number of findings (taint flows) is reported or tools are compared based on the number of findings. However, neither is the expected number of finding known nor the exact flows belonging to these findings. Moreover, most often neither the concrete apps having been analyzed nor the detected flows are being stored somewhere for replication.

To put our experiments on more solid grounds, the DIALDroidBench [42] benchmark is employed to answer RQ3. It comprises 30 real world apps for which parts of the ground truth have been specified in [29]. We detailedly inspect the type of flows found by CoDiDroid in these 30 apps as to see whether cooperation pays off to find flows deeply hidden in real-world apps by means of e.g. reflection or native code.

[5]https://play.google.com

Table 2: F-Measure Scores

ID	Category	FlowDroid	Best	CoDiDroid	Difference to Best	Difference to FlowDroid
1	Aliasing	0.667	0.667	0.667	0.000	0.000
2	AndroidSpecific	0.900	0.900	0.900	0.000	0.000
3	ArraysAndLists	0.615	0.667	0.615	0.052	0.000
4	Callbacks	0.897	0.897	0.897	0.000	0.000
5	DynamicLoading	0.000	0.500	0.000	0.500	0.000
6	EmulatorDetection	0.966	0.966	0.966	0.000	0.000
7	FieldAndObjectSensitivity	1.000	1.000	1.000	0.000	0.000
8	GeneralJava	0.810	0.810	0.810	0.000	0.000
9	ImplicitFlows	0.000	0.000	0.000	0.000	0.000
10	InterAppCommunication	0.000	0.625	0.625	0.000	-0.625
11	InterComponentCommunication	0.348	0.750	0.690	0.060	-0.342
12	Lifecycle	0.769	0.933	0.769	0.164	0.000
13	Native	0.000	0.333	0.889	-0.556	-0.889
14	Reflection	0.095	0.333	0.800	-0.467	-0.705
15	Reflection_ICC	0.000	0.000	0.000	0.000	0.000
16	SelfModification	0.000	0.000	0.000	0.000	0.000
17	Threading	1.000	1.000	1.000	0.000	0.000
18	UnreachableCode	1.000	1.000	1.000	0.000	0.000
	Ø	0.504	0.632	0.646	-0.014	-0.142

4.2 Execution Environment

For our experiments we setup the CoDiDroid framework as shown in Figure 2. AQL-WebService **2.** was executed on a Debian (Jessie) virtual machine with Java 8 (1.8.0_191) installed. During the time of our experiments it had two cores of an Intel® Xeon® CPU (E5-2695 v3 @ 2.30GHz) and 32 GB memory assigned. 30 GB were dedicatedly used for tool execution.

The second AQL-WebService (**3.**) was executed on a different machine, namely a Windows 10 laptop which had Java 8 (1.8.0_181) installed. This machine was used for the frontend (**1.**) execution as well. It is running on an Intel® Core® CPU (i7-5600U @ 2.60GHz) and 16 GB memory of which 1 and 6 GB were assigned to the frontend and backend, respectively. The rest of the memory was left for the operating system and the Android emulator used to run PIM. The emulated Android virtual device was setup to use API 26.

5 EVALUATION RESULTS

In this section the results of our experiments are described. In doing so, the research questions formulated above are answered.

5.1 RQ1: How does CoDiDroid compare to other analysis approaches with respect to precision?

Table 2 shows the results of our experiments comparing tools on the DroidBench benchmarks. The first two columns identify the category associated with each row. The remaining 5 columns show F-measure scores for different participants in this experiment. The column labelled FlowDroid contains the scores achieved by FlowDroid (possibly in combination with ApkCombiner to merge apps for inter-app benchmark cases). We separately list FlowDroid as it performed best in the reproducibility study of [29]. The next column shows the score achieved by the tool performing best in the reproducibility study *in that category*. Note that these are different tools in different categories; there is no single tool performing best

Table 3: Absolute numbers of flows (DroidBench 3.0)

	Cases			FlowDroid				Best				CoDiDroid			
Category	**Positive**	**Negative**	**Sum**	**TP**	**FP**	**TN**	**FN**	**TP**	**FP**	**TN**	**FN**	**TP**	**FP**	**TN**	**FN**
ArraysAndLists	4	6	10	4	5	1	0	4	4	2	0	4	5	1	0
DynamicLoading	3	0	3	0	0	0	3	1	0	0	2	0	0	0	3
Lifecycle	24	0	24	15	0	0	9	20	0	0	4	15	0	0	9
Native	5	0	5	0	0	0	5	1	0	0	4	4	0	0	1
Reflection	9	0	9	1	0	0	8	4	0	0	5	6	0	0	3
InterComponentCommunication	19	9	28	4	0	9	15	12	1	8	7	10	0	9	9
InterAppCommunication	11	0	11	0	0	0	11	5	0	0	6	5	0	0	6
Sum	75	15	90	24	5	10	51	47	5	10	28	44	5	10	31
DroidBench 3.0 (all categories)	163	41	204	58	2	24	30	81	2	24	7	78	2	24	10

in all categories. For having CoDiDroid select the best tool, we would need to tie its tool selection to category. Beside in benchmarks, such a strict categorization of apps is, however, not possible since apps can belong to several such categories.

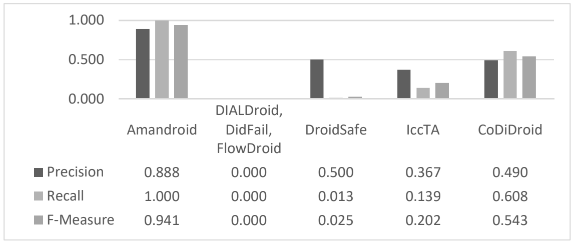
The last two columns show the difference between CoDiDroid and the best tool per category and between CoDiDroid and FlowDroid. Negative values show that CoDiDroid scored better than the object of comparison. Those values are highlighted with green background. The darker the background is, the larger is the difference. The analogous color scheme is used for the positive numbers highlighted in red.

We see that on average CoDiDroid outperforms both FlowDroid as well as the hypothetical "best" tool (not existing as a single tool). Compared to FlowDroid, CoDiDroid is 14% more precise according to the computed F-measure values. Clearly visible in the table, the reason for this advantage is based in the categories where we used tools in cooperation which are IAC (62%), ICC (34%), Native (89%) and Reflection (70%). We also see that there is one category ("DynamicLaoding") where the "best" tool outperforms CoDiDroid by a larger difference in F-measure, i.e. 0.5. The best tool in this category is Amandroid. Looking closer at the detected flows, the difference in F-measure is, however, due to a single flow which Amandroid detects, but CoDiDroid does not. Since Amandroid did not show any effect on the remaining cases[6], we decided to not embed it as a dedicated tool for dynamic loading cases.

We also tried to eliminate false positives by employing HornDroid to crosscheck the results produced by FlowDroid (Strategy IV, see Section 3). Here, cooperation did not work out because HornDroid often times out regarding the maximal execution time of 10 minutes which we imposed on the experiments. Moreover, even with a higher maximal execution time HornDroid often reports leaks as "POTENTIAL LEAK" only, which is not sufficient to discard flows found by FlowDroid. To be able to better leverage the crosschecking capabilities of CoDiDroid, we either need to allow for more time or need faster tools.

Startled by the reported difference in F-measure of 0.5 due to a single flow (cf. Table 3: Row 2), we furthermore decided to look at the absolute numbers of flows in experiments. Table 3 shows them for the seven categories of DroidBench where a difference in F-measure unequal to 0 became visible. The first column refers

[6]The "DynamicLaoding" benchmark category contains 5 apps with 3 flows to be detected out of which Amandroid finds one.



	Amandroid	DIALDroid, DidFail, FlowDroid	DroidSafe	IccTA	CoDiDroid
■ Precision	0.888	0.000	0.500	0.367	0.490
■ Recall	1.000	0.000	0.013	0.139	0.608
■ F-Measure	0.941	0.000	0.025	0.202	0.543

Figure 4: Intent-Matching: Precision, Recall, F-Measure

to the category; the last two rows in this first column to the sum over all listed categories and over all DroidBench categories, respectively. The other columns are arranged as 4 boxes labeled with Cases, FlowDroid, Best (again the tool performing best in this category) and CoDiDroid. The first box summarizes the expected results, more precisely, how many *positive* and *negative* benchmark cases exist. A positive benchmark case refers to a taint flow which is expected to be found. The contrary holds for negative cases, the associated taint flows should indeed not be found. The remaining boxes report the findings of FlowDroid, the "best" tool and CoDiDroid in terms of true and false positives (TP/FP) as well as true and false negatives (TN/FN). Highlighted in green are all the values for which CoDiDroid performs best.

We see that CoDiDroid performs well on the categories which our current strategies target, in particular it significantly outperforms FlowDroid. Moreover, the difference to the "best" tool (again, not existing as a single tool) lays in 3 flows only. Still, no approach could detect all leaks. Note that this is, however, *not* a weakness of CoDiDroid as a framework for cooperative analysis, but due to the lack of existing tools which could calculate these flows and which we could include in one of our strategies.

In addition, we more detailedly looked at the precision obtained for *intent matching* using the 224 benchmark cases of the extension of DroidBench. They just contain inter-component flows. The result for these cases can be found in Figure 4. The figure depicts precision, recall and F-measure. The depicted values show that CoDiDroid with an F-measure of 54% outperforms all tools except Amandroid. Since Amandroid is able to outperform CoDiDroid here, it seems that Amandroid would be an excellent candidate for inclusion into CoDiDroid. However, the ICC links which Amandroid infers are largely due to a coarse overapproximation. This pays off for the Intent-Matching benchmarks, but not in general as we see below.

Table 4: Timings and Results for inter-app benchmark cases

Benchmark case (number of apps)	Cooperation					
	ApkCombiner	ApkCombiner + IccTA	ApkCombiner + Amandroid	CoDiDroid	CoDiDroid*	CoDiDroid* (Amandroid)
Star (3)	25s	★ 58s	✪ 42s	✪ 37s	✪ 39s	× 109s
Star (4)	31s	★ 64s	✪ 49s	✪ 49s	✪ 19s	× 33s
Star (5)	38s	★ 72s	✪ 56s	✪ 59s	✪ 19s	× 32s
Star (6)	44s	★ 78s	✪ 62s	✪ 70s	✪ 21s	× 34s
Star (10)	67s	★ 102s	✪ 87s	✪ 133s	✪ 59s	× 138s
Ring (3)	25s	× 60s	× 47s	✪ 37s	✪ 37s	× 63s
Ring (4)	31s	★ 68s	✪ 53s	✪ 53s	✪ 22s	× 28s

* incremental × wrong result, ★ correct result, ◯ complete result

Therefore, we have refrained from including Amandroid into the tool set of CoDiDroid.

The results also show that intent matching in CoDiDroid is not yet as good as we would like it to be. A manual inspection of some failing cases shows that the conversion of IC3 results into intents and intent-filters while applying PIM is not always perfectly accurate. Unfortunately, in many cases the information provided by IC3 is incomplete making an improvement impossible. Hence, it seems that we rather need an improved version of IC3 than of PIM.

In conclusion, the results demonstrate that cooperative analysis cannot only simplify the usage of diverse analysis tools, but also achieves a gain in terms of precision. In particular, adding cooperating tools to assist an analysis tool such as FlowDroid results in a higher precision and lifts up its analysis from intra-app level to inter-app level.

5.2 RQ2: How well does (the inter-app analyis of) CoDiDroid scale to higher numbers of apps?

The results of the experiments involving the category "InterApp-Communication" have already been given in Tables 2 and 3. We see that CoDiDroid is improving over FlowDroid and equals the F-measure of the best tool. Table 4 furthermore shows the outcome for the two new benchmark scenarios Star and Ring. Each cell in the first column identifies the benchmark case and number of involved apps associated with each row. In the first row the approach (tool) used is shown. The *-symbol behind the approach's label refers to the fact that this approach is *re-using* the results reported in row n to compute the result in row $n + 1$ (native feature of CoDiDroid). Two types of results are reported in each cell. First, the symbols ✪, ★ and × give information about the correctness and completeness of the obtained result. Second, the analysis time in seconds is given. As a means for comparison, we list the runtime of the merging procedure of ApkCombiner in the second column.

Taking a look at the correctness, only CoDiDroid always computes the correct result. All expected and no unexpected flows are found. In contrast, IccTA misses some flows. In all scenarios except for Ring (3), all apps contain flows from their own sinks to their own sources *via other apps*. All these remain undetected by IccTA. For benchmark case Ring (3) on the other hand, there is no flow from app 2 and 3 to 1 (as app 4 is not present in Ring (3)). Still IccTA reports these flows. This latter incorrect result is also reported by ApkCombiner plus Amandroid.

The last column labeled with CoDiDroid* (Amandroid) refers to a version of CoDiDroid using Amandroid instead of FlowDroid for intra-app flow questions. This is an alternative version of cooperation which we wanted to evaluate. However, without prior merging of apps via ApkCombiner, it becomes visible that Amandroid reports one incorrect flow in every case, namely a direct connection between each source and sink within an app, i.e. without involving inter-app communication. Explained on our running example, Amandroid reports a connection from statement s_2 to s_3 while only looking at app A. For this reason, we did not include Amandroid in CoDiDroid.

In summary, considering the correctness and completeness CoDiDroid performs best.

Next, let us take a look at the time required by each approach. For the creation of a combined app, ApkCombiner itself requires about 8 seconds per app to combine. Amandroid is slightly faster than IccTA but in combination with ApkCombiner the whole analysis takes more than 40 seconds in any scenario. CoDiDroid on the contrary is faster when analyzing three apps without having stored any previously computed results, because it does not require ApkCombiner to be executed beforehand. The biggest advantage becomes visible in those cases where ApkCombiner together with Amandroid and IccTA take longest, namely the scenarios dealing with $n + 1 > 3$ apps. This time, CoDiDroid* already knows what has been computed before (for n apps) and is able to reuse this information, leading to an analysis time of 19 to 21 seconds per app. To conclude, CoDiDroid is on average 51% faster than any approach requiring ApkCombiner. Other approaches not relying on ApkCombiner (e.g. DIALDroid [8] storing intermediate results as well) will show a similar scalability. With CoDiDroid, however, any analysis tool able to analyze inter-component scenarios can gain this property.

Note furthermore that ApkCombiner is not able to combine up-to-date apps built with Android Studio. Hence, additional effort was spent to create Star and Ring in a version that is ready to be combined. We also tried to use the up-to-date version of FlowDroid, which includes IccTA, in combination with ApkCombiner. The analysis time was lower than the time required by Amandroid and IccTA. However, none of the inter-app flows were found, accordingly we did not report the results.

In the end, the results show that it is beneficial to use CoDiDroid for inter-app analysis, since it is faster and more precise than all other approaches in our scope.

5.3 RQ3: How well does CoDiDroid scale to larger (real-world) apps?

For RQ3 we used the benchmarks of DIALDroid-Bench. The 30 real world apps belonging to DIALDroid-Bench come with a partially defined ground truth of 26 flows. We ran CoDiDroid on these 30 apps. All of the 26 flows are found. This shows that CoDiDroid is able to process (a) large real-world apps and (b) large numbers of apps. The latter in particular proves the power of cooperation in inter-app analysis. Approaches relying on merging via ApkCombiner directly fail on 30 apps.

Table 5: Findings for DIALDroidBench (30 Apps)

Type	Flows found by	Number
Native	CoDiDroid: Strategy III	629 (+0 /+0)
Reflection	CoDiDroid: Strategy II	636 (+7 /+12)
InterApp- / InterComponent-Communication	CoDiDroid: Strategy I	660 (+31 /+14)

CoDiDroid in fact finds several hundreds more than the 26 flows. Only a manual inspection could reveal whether these are false or true positives, which is beyond the scope of this paper. Still we wanted to see whether cooperation can bring us any *new* potential flows not detectable without cooperation. To this end, we compared the flows with those found by FlowDroid. Note that it is our usage of the AQL-System storing answers (in particular, precise flows) which allows for such a comparison. Overall FlowDroid detects 629 flows which are intra-app flows only.

Table 5 shows the results computed with CoDiDroid. The first two columns refer to the type of flow and the strategy of CoDiDroid which was employed as singelton strategy to specifically detect this sort of flow. The last column shows the findings. We also list in brackets (a) how the overall number of findings have changed and (b) how many new taint flows associated with this type of flow have been found – always in comparison to FlowDroid alone.

4 out of 30 DIALDroid-Bench apps use native libraries. None of these uses are involved in a taint flow. Thus, CoDiDroid still only finds 629 candidates for taint flows when employing Strategy III. Taking the cooperation of FlowDroid and DroidRA (Strategy II) into account it gets more interesting. 21 of the 30 considered apps employ reflection. The results show on the one hand, 12 yet undetected taint flows that involve reflection. On the other hand, 5 taint flow candidates could be eliminated since the reflective statement did not refer to a source or sink of any kind. Thus, we ended up with 636 (+7) flows in total.

Furthermore, the cooperation of FlowDroid, IC3 and PIM (Strategy I) detected overall 660 flows. To find these, 1966 intent-sinks and 1215 intent-sources, found by IC3, were taken into account. 41 of these intent-sinks and 156 of these intent-sources represent one end of the overall 660 reported flows. Among these 660 flows, 14 inter-app taint flows transfer data from a real source in one app to a real sink in another app. These flows have not been found by any other approach, yet.

In summary, the detection of 26 new taint flows (12 involving reflection and 14 inter-app flows) shows that the cooperation of different analysis tools can be beneficial in the context of large real world apps. Moreover, the experiments show that cooperative analysis is able to scale to real-world apps.

5.4 Threats to Validity

The main threat to the validity of our results is the fact that most experiments are carried out on artificially constructed benchmarks. The main reason for doing so is the need for knowing the ground truth to compare the detected flows against. Without a ground truth, there is no basis for comparing tools and for computing their accuracy. For real-world apps, the construction of the ground truth requires a major manual effort. Ground truth computation can of course be assisted by analysis tools. However, as these are at the same time the targets of our evaluation, this is not ideal.

Another threat to validity is the informative value of metrics like precision, recall and F-measure. As we have seen in our experiments, a difference in F-measure of 0.5 can be due to a difference in the detection of just a single flow. We nevertheless employ these measures here since they are very commonly used and thus allow for a comparison of our results with the results of other researchers.

6 RELATED WORK

The idea of *combining* different analysis techniques and tools as to improve performance of an analysis has already been investigated in a number of works. The combinations can be differentiated into *tight* and *loose* integrations. Tight combinations deeply integrate two or more analysis techniques into a new tool. Typical examples are tools performing an under- and overapproximation of the state space at the same time (like [3, 11, 16, 17]). Considering the area of app analysis, several tight combinations have already been mentioned. DIALDroid [8], DidFail [20] and IccTA [22] are tight combinations of IC3 [25] and FlowDroid [2]. The techniques and instruments used to build the combination is what distinguishes them from each other. DIALDroid is most closely related to CoDiDroid. Its approach to tackle the challenge of inter-app analysis is similar, thereby they achieve a comparable scalability. However, this is the only category which is tackled and their analysis does not allow to incorporate other tools which handle, for example, native or reflexive method calls.

Tight combinations usually improve over singleton techniques in terms of precision. The disadvantage is that a new tool has to be written for every newly available technique which has proven to be worthwhile to be integrated.

Loose combinations on the other hand take existing tools as black boxes and combine them, often by running tools in sequence (like [5]). *Cooperative* analyses are taking this principle to the extreme. Cooperative approaches divide the analysis work onto different tools and let the tools communicate on computed results by exchange of information. Such information can take the form of (residual) programs [6], program annotations [10], error paths [13] and correctness or error witnesses [4, 19]. For Android app analysis, no cooperative technique has been existing so far.

7 CONCLUSION

In this paper, we have introduced the novel concept of *cooperative* Android app analysis. Our approach allows for a flexible combination of existing tools via an automatic splitting of analysis tasks into subtasks and their distribution onto tools. The framework thereby does not only significantly simplify the usage of app analysis tools (which has to be evaluated in future work) but also improves over singleton (non-cooperative) approaches in terms of precision and scalability.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

REFERENCES

[1] Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, and Francesco Bergadano. 2017. TeICC: targeted execution of inter-component communications in Android. In *Proceedings of SAC, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1747–1752.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of PLDI, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269.

[3] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. 2010. Proofs from Tests. *IEEE Trans. Software Eng.* 4 (2010), 495–508.

[4] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. 2016. Correctness witnesses: exchanging verification results between verifiers. In *Proceedings of the 24th FSE, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 326–337.

[5] Dirk Beyer and Marie-Christine Jakobs. 2019. CoVeriTest: Cooperative Verifier-Based Testing. In *FASE (Lecture Notes in Computer Science)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.), Vol. 11424. Springer, 389–408. https://doi.org/10.1007/978-3-030-16722-6_23

[6] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. 2018. Reducer-based construction of conditional verifiers. In *Proceedings of the 40th ICSE, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1182–1193.

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd ICSE, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 241–250.

[8] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of AsiaCCS, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 71–85.

[9] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *Proceedings of EuroS&P, 2016*. IEEE, 47–62.

[10] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th ICSE, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 144–155.

[11] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.* 2 (2008), 8:1–8:37.

[12] Xingmin Cui, Jingxuan Wang, Lucas Chi Kwong Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. 2015. WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps. In *Proceedings of the 8th Conference on Security & Privacy in Wireless and Mobile Networks, 2015.* ACM, 25:1–25:12.

[13] Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. 2016. Abstraction-driven Concolic Testing. In *Proceedings of the 17th VMCAI, 2016*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 328–347.

[14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th OSDI, 2010*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 393–407.

[15] Yu Feng, Isil Dillig, Saswat Anand, and Alex Aiken. 2014. Apposcopy: automated detection of Android malware (invited talk). In *Proceedings of the 2nd DeMobile, 2014*, Aharon Abadi, Rafael Prikladnicki, and Yael Dubinsky (Eds.). ACM, 13–14.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of PLDI, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223.

[17] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th POPL, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 43–56.

[18] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the 22nd NDSS, 2015.*

[19] Marie-Christine Jakobs and Heike Wehrheim. 2017. Compact Proof Witnesses. In *Proceedings of the 9th NASA Formal Methods, 2017*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). Springer, 389–403.

[20] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd SOAP, 2014*, Steven Arzt and Raúl A. Santelices (Eds.). ACM, 5:1–5:6.

[21] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of SEC, 2015*, Hannes Federrath and Dieter Gollmann (Eds.). Springer, 513–527.

[22] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th ICSE, 2015*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE, 280–291.

[23] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. In *Proceedings of the 25th ISSTA, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 318–329.

[24] Damien Octeau, Somesh Jha, Matthew Dering, Patrick D. McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proceedings of the 43rd POPL, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 469–484.

[25] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick D. McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of ICSE, 2015*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE, 77–88.

[26] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Security Symposium, 2013*, Samuel T. King (Ed.). USENIX Association, 543–558.

[27] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. *SCanDroid: Automated security certification of Android applications.* Technical Report. University of Maryland.

[28] Felix Pauck. 2017. *Cooperative static analysis of Android applications.* Master's thesis. Paderborn University, Germany.

[29] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools keep their promises?. In *Proceedings of the 26th ESEC/FSE, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 331–341.

[30] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ISSTA, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 176–186.

[31] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the Conference on Computer and Communications Security, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1329–1341. https://doi.org/10.1145/2660267.2660357

[32] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. 2016. IacDroid: Preventing Inter-App Communication capability leaks in Android. In *Proceedings of ISCC, 2016*. IEEE, 443–449.

[33] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. In *Proceedings of the 26th ESEC/FSE, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 342–353.

[34] 2017. Amandroid. Retrieved 02/16/2019 from https://bintray.com/arguslab/maven/argus-saf/3.1.2

[35] 2014. ApkCombiner. Retrieved 02/16/2019 from https://github.com/lilicoding/ApkCombiner

[36] 2019. AQL-Online. Retrieved 06/18/2019 from https://FoelliX.github.io/AQL-Online

[37] 2018. AQL-System. Retrieved 02/16/2019 from https://FoelliX.github.io/AQL-System

[38] 2019. AQL-WebService. Retrieved 06/18/2019 from https://github.com/FoelliX/AQL-WebService

[39] 2018. BREW. Retrieved 02/16/2019 from https://FoelliX.github.io/BREW

[40] 2019. CoDiDroid. Retrieved 06/18/2019 from https://FoelliX.github.io/CoDiDroid

[41] 2017. DIALDroid. Retrieved 02/16/2019 from https://github.com/dialdroid-android/DIALDroid

[42] 2016. DIALDroidBench. Retrieved 02/16/2019 from https://github.com/amiangshu/dialdroid-bench

[43] 2015. DidFail. Retrieved 02/16/2019 from https://www.cert.org/secure-coding/tools/didfail.cfm

[44] 2016. DroidBench 3.0. Retrieved 02/16/2019 from https://github.com/secure-software-engineering/DroidBench/tree/develop

[45] 2017. DroidRA. Retrieved 02/16/2019 from https://github.com/serval-snt-uni-lu/DroidRA

[46] 2016. DroidSafe. Retrieved 02/16/2019 from https://mit-pac.github.io/droidsafe-src/

[47] 2017. FlowDroid. Retrieved 02/16/2019 from https://github.com/secure-software-engineering/soot-infoflow-android/wiki

[48] 2018. HornDroid. Retrieved 02/16/2019 from https://github.com/ylya/horndroid

[49] 2019. IC3. Retrieved 06/18/2019 from https://github.com/FoelliX/ic3

[50] 2016. IccTA. Retrieved 02/16/2019 from https://sites.google.com/site/icctawebpage/source-and-usage

[51] 2019. NOAH. Retrieved 06/18/2019 from https://github.com/FoelliX/NOAH

[52] 2019. PIM. Retrieved 06/18/2019 from https://github.com/FoelliX/PIM