A Qualitative Analysis of Taint-Analysis Results

Anonymous authors

***Abstract*—In the past, researchers have developed a number of popular taint-analysis approaches, particularly in the context of Android applications. Numerous studies have shown that automated code analyses are adopted by developers only if they yield a good "signal to noise ratio", i.e., high precision. Many previous studies have reported analysis precision quantitatively, but this gives little insight into what can and should be done to increase precision further.**

To guide future research on increasing precision, we present a comprehensive study that evaluates static Android taint-analysis results on a *qualitative* level. To unravel the exact nature of taint flows, we have designed and implemented COVA, an analysis tool to compute partial path constraints that inform about the circumstances under which taint flows may actually occur in practice.

We have conducted a qualitative study on the taint flows in 1,022 real-world Android applications. Our results reveal several key findings: Many taint flows occur only under specific conditions, e.g., environment settings, user interaction, I/O. Program analyses should consider the application context to discern such situations. COVA shows that few taint flows are guarded by multiple different kinds of conditions simultaneously, so tools that seek to confirm true positives dynamically can concentrate on one kind at a time, e.g., only simulating user interactions. Lastly, many false positives arise due to a too liberal source/sink configuration. Thus, program analyses must be more carefully configured, and their configuration could benefit from better tool assistance.

***Index Terms*—taint analysis, path conditions, Android**

I. Introduction

The past few years have brought to light a wealth of diverse taint-analysis approaches, most of them static and for the Android platform [1]–[6]. Static taint analysis is not a trivial task, due to the static abstractions and sometimes approximations it requires. Static taint analysis for Android is particularly challenging since one must consider some rather unique features of Android: Android apps are not standalone applications but rather plugins for the Android framework. As such they have a distinct life cycle, and often numerous callbacks that respond to various environmental stimuli such as button clicks or location changes. Android apps often need to operate correctly for different platform versions and devices, which is why they often contain code that is conditionalized with respect to various environment parameters. In Android it is a common and recommended practice, for instance, to obtain backward compatibility by probing the platform version.

Existing studies show that code analysis tools are most likely to be adopted if they yield high precision, i.e., a low rate of false positives [7], [8], which is hard given the challenges named above. While all existing papers proposing those tools do comprise an insightful evaluation, often even on a large scale, these evaluations are virtually all entirely *quantitative*. As such, they do permit one to conclude *how many* taint-flow warnings a given taint-analysis tool reports on a given benchmark set, but *not of which nature* those taint flows exactly are, i.e., under which conditions they can occur at runtime, and what could be done further to discard potentially remaining false positives within those taint flows. The lack of such *qualitative* data currently hinders progress in upcoming directions of research. Researchers are investigating novel "hybrid" combinations of static with dynamic analysis which, for instance, seek to dynamically confirm static-analysis findings by computing an actual *witness path* exposing the taint flow at runtime [9]–[14]. To guide such research, we present the first study that evaluates static Android taint-analysis results at a *qualitative* level. In particular, this study seeks to identify how taint flows are *conditioned on…*

- *Environment settings* (platform versions, country, etc.): Malicious applications can leak data based on environment settings [15], [16].
- *User interactions*: Previous work [17] has shown that many Android apps leak data as a result of user actions on certain widgets in the apps.
- *I/O operations*: Leaks that depend on specific inputs are difficult to trigger dynamically [12], [18], [19], since the search space is often very large.

To facilitate this study, we have implemented COVA, a static analysis tool that computes partial path constraints. COVA can be configured to track information about the three factors named above, and thus the circumstances under which taint flows may actually occur in practice.

We conducted a case study of the most common taint flows (data leaks) reported by the static taint-analysis tool FlowDroid [1] from 1,022 real-world Android apps. During manual inspection of the sampled taint flows, we observed some default sources and sinks provided by FlowDroid to be inappropriate: they were causing only false positives, and in large quantities. Unfortunately, we found this to impact the empirical evaluation of many previously published papers [20]–[28].

After having eliminated these false positives, we classified the reported taint flows with COVA, based on the three factors mentioned above. Our study reveals that at least 14% of the flows are conditioned on at least one of the three factors (environment, interaction, I/O), but also that few flows are conditioned on multiple factors at the same time. This means that hybrid analysis tools must be able to deal with those factors, but for each factor one can likely build specialized support.

In addition, we encountered 3.5% of taint flows to be "low-hanging fruits", i.e., taint flows that are not conditioned on the three factors above and also are intra-procedural. These taint flows can be identified purely statically, and are likely to be both correct and actionable to the developers. Static analysis tools should thus aim to prioritize their reporting. To summarize, this paper presents the following original contributions:

- COVA, a static analysis tool to compute path constraints.
- A micro-benchmark to assess COVA and similar tools.
- A COVA-supported qualitative study of taint flows from 1,022 Android apps.

The insights drawn from the study are:

- Source/sink configurations are essential to precision. Previous bad defaults have caused vast amounts of false positives.
- Many inter-procedural taint flows in Android applications are conditioned on user interactions, and fewer on environment configurations or I/O operations. The three categories are almost disjoint. This is good news: one can build effective hybrid analysis tools that specialize on either category.
- Reported intra-procedural taint flows are most likely to be true positives. One should thus prioritize their reporting.

We make COVA and the Android apps we analyzed publicly available at:

https://github.com/covaanalyst/cova-root.

The rest of this paper is organized as follows: we first motivate the need for the tool COVA, then explain its design, and lastly our experimental study.

II. A Motivating Example

Figure 1 shows an `Activity` of an Android application that contains a data leak—a simplified example. The activity first reads the unique device identifier (Line 8), stores it into variable `deviceId` before method `onClick` uses the variable and sends an SMS containing the identifier to the phone number "+1234" (Line 15).

State-of-the-art static taint-analysis tools for Android, e.g., FlowDroid [1], AmanDroid [2] or DroidSafe [3], are capable of detecting such leaks with a high precision. The tools deliver highly precise context-, field-, and flow-sensitive results. However, as we observed during our study, these precision dimensions are insufficient when trying to understand *how* and *when* apps leak data.

While any of the mentioned taint-analysis tools reports the leak in Figure 1, none of the tools reports that the leak can only occur under a specific execution path. The tools are not *path-sensitive* [29]. The app leaks the device identifier only when it executes the source and sink statements. Their execution depends on three path conditions [30]. First, the app must run the correct Android SDK version (Line 7), second, the user must trigger the app to execute the `onClick` callback by pressing a button (Line 11), and third, a special system feature has to be enabled on the execution device (Line 14).

For an automatic qualitative evaluation of the path conditions of data leaks reported by the taint-analysis tools, we implemented the static analysis tool COVA. COVA computes a *constraint map* which associates with each statement of a program the path conditions required to execute the statement.

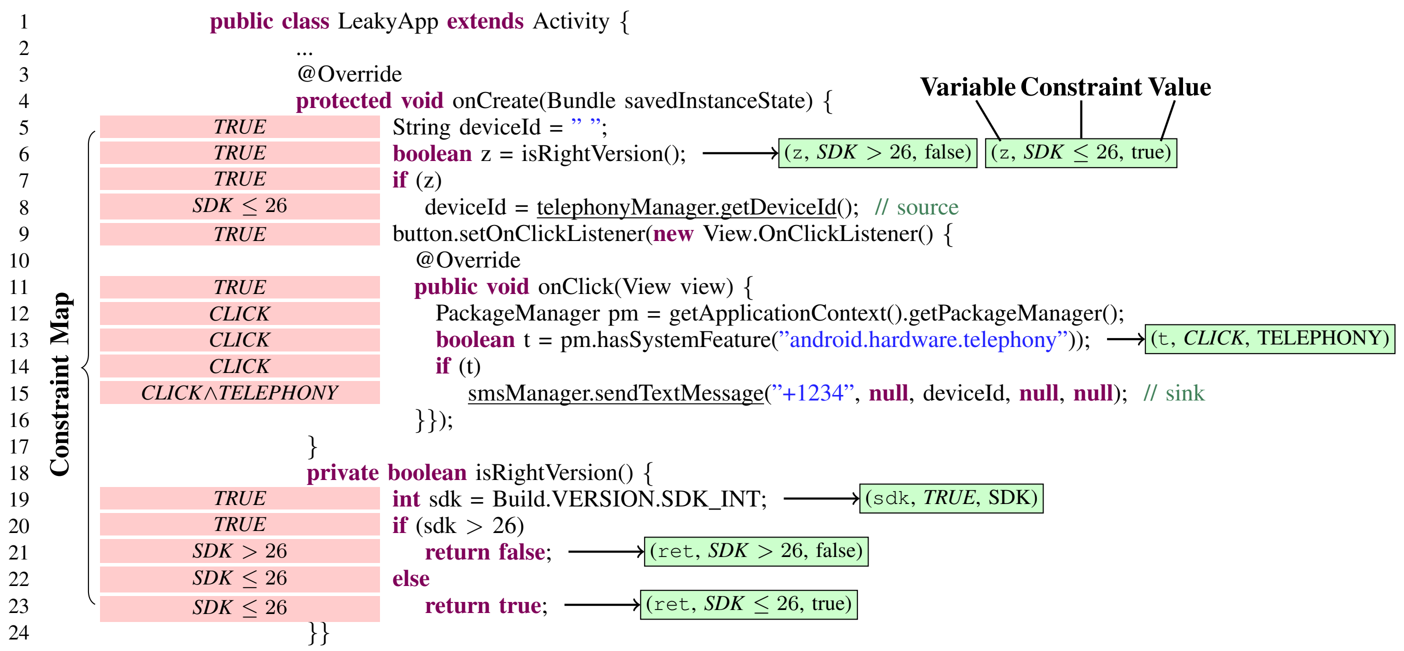
Figure 2 describes the workflow of COVA when used with a taint-analysis tool. COVA accepts as input an Android application in bytecode format and a set of *pre-defined constraint-APIs*. COVA then computes the path conditions, i.e., the constraint map, which depend on values from the constraint-APIs. Instead of computing all path conditions for the program (which is practically infeasible), COVA focuses on propagating values of the constraint-APIs and path conditions dependent on these. The constraint map computed by COVA can be used to refine the data leaks reported by an existing taint-analysis tool, i.e., leaks can be reported with path constraints. Although we applied COVA to taint analysis, COVA is applicable to any other client analysis that can benefit from path information.

To understand, for instance, the leak in Figure 1, it suffices to track the following constraint-APIs: `Build.VERSION.SDK_INT`, `OnClickListener.onClick` and `PackageManager.hasSystemFeature`. COVA performs a context-, flow-, and field-sensitive data-flow analysis starting from the entry point of the program. In Figure 1, the entry point of the activity is `onCreate` and it is always executable, thus the statement at Line 5 has the initial constraint *TRUE*. At each reachable invocation of a constraint-API, COVA generates a tainted data-flow fact, simply referred to by *taint*. In Figure 1, COVA propagates the taint (`sdk`, *TRUE*, SDK) starting from Line 19. The symbol `sdk` is the variable containing the value returned from the constraint-API; the second entry *TRUE* is the constraint under which the data-flow fact at the statement is reachable; SDK stands for the symbolic value of the static field `Build.VERSON.SDK_INT`. COVA then propagates the taints along the inter-procedural control-flow graph (ICFG) of the program and creates constraints over the symbolic values of taints whenever taints are used in conditional statements.

For instance, the return value of the method `isRightVersion()` is `true` when the SDK version is at most 26. The Android app further branches (indirectly) based on the version at the if-statement in Line 7. The constraint map of COVA captures these path conditions. COVA computes the taint (`ret`, $SDK \leq 26$, true) for the statement in Line 23; the taint encodes that the return value `ret` equals `true` when the version is $SDK \leq 26$. This taint propagates back to the call site in Line 6 as taint (`z`, $SDK \leq 26$, true).

In early results of our qualitative study, we observed many data leaks to depend on callbacks of user interfaces which motivated us to symbolically represent them in COVA. Technically, COVA creates a constraint *CLICK* at the callback `OnClickListener.onClick` and propagates this constraint to all statements reachable from this method. *CLICK* is a symbolic value representing a button click, only when a user clicks the button the statements become reachable.

COVA propagates all taints from all constraint-APIs and



```
public class LeakyApp extends Activity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        String deviceId = " ";
        boolean z = isRightVersion();
        if (z)
            deviceId = telephonyManager.getDeviceId(); // source
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                PackageManager pm = getApplicationContext().getPackageManager();
                boolean t = pm.hasSystemFeature("android.hardware.telephony"));
                if (t)
                    smsManager.sendTextMessage("+1234", null, deviceId, null, null); // sink
            }});
    }
    private boolean isRightVersion() {
        int sdk = Build.VERSION.SDK_INT;
        if (sdk > 26)
            return false;
        else
            return true;
    }}
```

Fig. 1. A motivating example

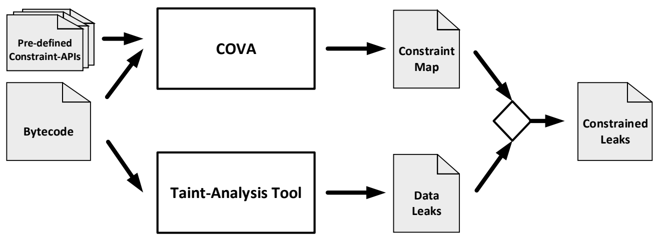


Fig. 2. The workflow of applying COVA to taint-analysis results

simultaneously computes a constraint for each reachable statement based on available taints at the current statement. Once the data-flow propagation is completed, the constraint map is also computed.

In this paper, we enrich the leak-reports by FlowDroid with the constraints computed by COVA. We compute the constraint of a leak, its *leak-constraint*, according to the logical formula $C_{source} \wedge C_{sink}$, in which C_{source} denotes the constraint under which the source statement may be executed, and C_{sink} the same for the sink. In the example, the resulting leak-constraint is *SDK* $\leq 26 \wedge$ *CLICK* $\wedge$ *TELEPHONY*. This leak-constraint is used later in our study to classify the taint flows.

III. Computing the Constraint Map

Computing the constraint map as shown in Figure 1 is non-trivial. The execution of a branch may *simultaneously* depend on two or more values of some constraint-APIs. A static analysis must *jointly* propagate all values to compute the final constraint for a branch. As the values have to be propagated jointly, the analysis is *non-distributive* [31]. Furthermore, each value of a constraint-API must be propagated throughout the whole program, since values can flow to fields of objects before the fields are re-accessed elsewhere and aliasing relations must be computed.

We implemented the analysis within the data-flow framework VASCO [32], which solves non-distributive inter-procedural data-flow problems in a highly precise (context- and flow-sensitive) manner. VASCO propagates data-flow facts (elements of a data-flow domain D) from statement to statement along the control-flow graph of the program. The *flow functions* define how a data-flow fact changes when it flows from one statement to a successor. The flow functions are generic arguments to VASCO as they define the analysis problems and define which information to maintain, generate or kill for each control-flow edge. The flow functions accept a data-flow fact $d \in D$ and a control-flow edge $\langle n, m \rangle$ of the ICFG as input, and output a new value $d' \in D$. Similarly to IFDS [33], VASCO differentiates between the following four kinds of functions:

- NormalFlowFunc: handles intra-procedural flows where n is not a call site.
- CallLocalFlowFunc: handles intra-procedural flows where n is a call site. It propagates the values of local variables not used at the call site.
- CallEntryFlowFunc: handles an inter-procedural flow from call site n to the first statement m of a callee. It typically maps actual method arguments to formal parameters.
- CallExitFlowFunc: it is the inverse of CallEntryFlowFunc and maps parameters back to the call site's arguments.

A. Analysis Domain

The domain D for our analysis in VASCO is two-dimensional: $\mathbb{C} \times 2^{\mathbb{T}}$, where $\mathbb{C}$ is a constraint domain and $\mathbb{T}$ is a taint domain. We use $\bot \in D$ to denote an unknown fact. Consider data-flow fact $(C, T) \in \mathbb{C} \times 2^{\mathbb{T}}$ to hold at statement n, then C is the constraint under which statement n is reachable. We seed the data-flow propagation with the fact $(\mathit{TRUE}, \emptyset)$ at the entry point of the application. The constraint is *TRUE*, as the entry point statement is always reachable. At the entry point, the set T is the empty set as no constraint-API call has been encountered.

In general, T is the set of taints generated at constraint-APIs reaching statement n. Each taint is a triple $(a, c, v) \in \mathbb{T}$ and consists of an access path (a local variable followed by a finite sequence of fields [34]). The access path a encodes how the value of the constraint-API is heap-referenceable at statement n. Value v holds the actual value of a. In the case it is the return value of a constraint-API, it is represented symbolically, if possible concrete values of primitive types are traced. The constraint c describes under which conditions a has the value v. Note, at a statement n the constraint c of a taint within the set T and the constraint C are not necessarily equal (e.g., Line 6 in Figure 1).

The meet operator $\sqcup$ is the logical disjunction $\vee$ for the constraint domain and set union $\cup$ for the taint domain, i.e., $(C_1, T_1) \sqcup (C_2, T_2) = (C_1 \vee C_2, T_1 \cup T_2)$ for two data-flow facts (C_1, T_1) and (C_2, T_2). For any data-flow fact (C, T), we define $(C, T) \sqcup \bot = (C, T)$. Since the data-flow domain is two-dimensional, in the following we separate the flow functions into two parts: the flow functions of the taint analysis and of the constraint analysis. Let $\langle n, m \rangle$ be a control-flow edge and let (C_{in}, T_{in}) refer to the data-flow fact before n and (C_{out}, T_{out}) denote the fact before m, then we describe the flow function F in form of the result set $(C_{out}, T_{out}) = F(C_{in}, T_{in})$. The analysis operates on an intermediate representation, called Jimple [35]. Jimple is a three-addressed code reconstructed from Java bytecode. We define the analysis based on the statements affecting either C or T of a data-flow fact (C, T).

B. Flow Functions of the Taint Domain

NORMALFLOWFUNC:

This flow function mostly follows standard access-path based taint tracking data-flow propagation [1], [36]. For instance, as a field-sensitive analysis, COVA kills any tainted access path with local variable x at an assignment statement $n : x = \star$ [1]. Let $T_{in}^{-} = T_{in} \setminus \{(x, \star, \star)\}$. For an assignment statement $n : x = y$ it is $T_{out} = T_{in}^{-} \cup \{(x, c, v)\}$ if there is a taint $(y, c, v) \in T_{in}$, i.e., if any incoming access path matches the right side, an access path for the left side is added to the out set. For a field-store assignment statement, i.e., $n : x.a = y$, an access-path based analysis has to add the *indirectly aliasing access paths* of x [1] and COVA relies on an on-demand alias analysis [36].

[1]The symbol $\star$ is a placeholder representing an irrelevant argument.

In the following we discuss some corner cases that deviate from standard taint-tracking flow functions. For an assignment statement $n : x = A.b$ where the right side is a static field and the field is labeled as a constraint-API, COVA generates a *source taint* $(x, C_{in}, sym(A.b))$ and $T_{out} = T_{in}^{-} \cup \{(x, C_{in}, sym(A.b))\}$. Hereby C_{in} is the constraint that reaches statement n and sym For instance, $(sdk, \mathit{TRUE}, \mathit{SDK})$ is a source taint created at Line 19 in Figure 1.

For an assignment statement $n : x = z$ for which z has a constant value, COVA creates a *concrete taint* (x, C_{in}, z), i.e., $T_{out} = T_{in}^{-} \cup \{(x, C_{in}, z)\}$, if the constraint C_{in} is not equal to *TRUE*. As these concrete taints are only created when $C_{in} \neq \mathit{TRUE}$, they are used to detect constraints that are indirectly influenced by constraint-APIs.

Jimple also allows return statements $n : \textbf{return}\ z$ with constant values for z and, if constraint C_{in} is not equal to *TRUE*, the flow function returns the set $T_{out} = T_{in}^{-} \cup \{(ret, C_{in}, z)\}$, i.e., COVA also creates a concrete taint (ret, C_{in}, z). We use a fake access path *ret* to symbolically represent the returned variable (e.g., $(\mathit{ret}, \mathit{SDK} \leq 26, true)$ at Line 23 in Figure 1).

For an assignment statement $n : x = y \oplus z$ of a binary operator $\oplus$, COVA creates an *imprecise taint*, if $(y, \star, v_1) \in T_{in}$ and $(z, \star, v_2) \in T_{in}$, and $T_{out} = T_{in}^{-} \cup \{(x, C_{in}, im(v_1, v_2)\}$. The symbolic value $im(v_1, v_2)$ means the value of x is affected by v_1 and v_2. In case the taint of one variable is missing, for example, $(y, \star, v) \in T_{in}$ and $(z, \star, \star) \notin T_{in}$, COVA creates an imprecise taint $(x, C_{in}, im(v))$.

For an assignment statement $n : x = \ominus y$ of a unary operator $\ominus$ the flow function is analog.

CALLLOCALFLOWFUNC:

Apart from killing taints that start in the overwritten variable r, at a call statement $n : r = o.f(a_1, ..., a_k)$, COVA adds the source taint $(r, C_{in}, sym(f))$ to T_{out}, in case method f is configured to be a constraint-API. If $(o, \star, v) \in T_{in}$, COVA creates an imprecise taint $(r, C_{in}, im(v))$.

CALLENTRYFLOWFUNC:

For a call statement $n : o.f(a_1, ..., a_k)$, any taint in T_{in}, whose access path's local variable is an argument of the call $(a_1, ..., a_k)$, is mapped to the respective parameter access path within the callee. For non-static call sites, the arguments include the receiver variable (o) of the call. Access paths representing static data-flow facts, i.e., static fields, are mapped to the callee, as the callee may change their values. If there exists a_i being constant and $C_{in} \neq \mathit{TRUE}$, COVA creates a concrete taint (p_i, C_{in}, a_i) for the respective parameter p_i within the callee.

CALLEXITFLOWFUNC:

At a return statement $n : \textbf{return}\ s$ of a callee that returns to a call site $m : r = o.f(a_1, ..., a_k)$, the access path with a local variable which is a parameter of the callee is mapped to the respective argument in $a_1, ..., a_k$ (the inverse of CALLENTRYFLOWFUNC). Additionally, all taints with the local variable s are propagated to the caller where the returned variable s is replaced by the assigned variable r. At the scope

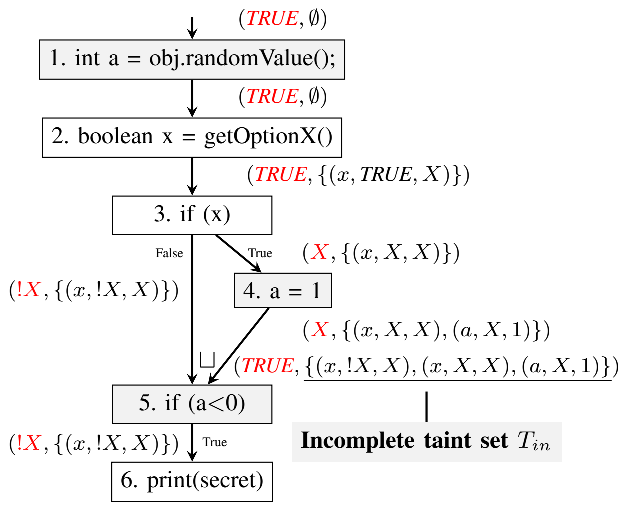

(TRUE, ∅)
1. int a = obj.randomValue();
(TRUE, ∅)
2. boolean x = getOptionX()
(TRUE, {(x, TRUE, X)})
3. if (x)
False
True
(X, {(x, X, X)})
(!X, {(x, !X, X)})
4. a = 1
(X, {(x, X, X), (a, X, 1)})
⊔
(TRUE, {(x, !X, X), (x, X, X), (a, X, 1)})
5. if (a<0)
(!X, {(x, !X, X)})
True
Incomplete taint set T_in
6. print(secret)

Fig. 3. An example shows an incomplete taint set. Assume getOptionX() is a constrain-API whose return value is represented by the symbolic value X.

when an access path is propagated from callee to its caller, COVA also adds the required aliases [36].

C. Flow Functions of the Constraint Domain

To compute C_{out}, COVA conjoins the constraint C_{in} with an extending constraint C_{new} which is created at conditional statements or UI callbacks, i.e., $C_{out} = C_{in} \wedge C_{new}$. The constraint of each taint in T_{in} will also be extended with C_{new}. Moreover, COVA only propagates taints whose constraints are not equal to *FALSE*. Thus, $T_{out} = \{(x, c \wedge C_{new}, v) | (x, c, v) \in T_{in}^{-}$ and $(c \wedge C_{new} \neq \mathit{FALSE})\}$. In the following we focus on introducing how C_{new} is computed.

NORMALFLOWFUNC:

For an if-statement n : **if** $(a \oplus b)$ with a comparison operator $\oplus$, COVA creates C_{new} based on the available taints in T_{in}. The following cases are considered:

(i) T_{in} contains only taints for variable a (it is analog for b). Assume there are k taints $(a, c_i, v_i) \in T_{in}$ with $i \in \{1, ..., k\}$. If b is a constant, COVA creates a constraint e_i by substituting the variable a in the formula $a \oplus b$ with its value v_i and conjoining the result with c_i for each taint (a, c_i, v_i), i.e., $e_i = (v_i \oplus b) \wedge c_i$ if the successor statement m is in the *TRUE* branch of n. For the case m is in the *FALSE* branch $e_i = \neg(v_i \oplus b) \wedge c_i$. If b is an untracked variable, the formula $v_i \oplus b$ is replaced by the imprecise constraint $im(v_i)$ in e_i. COVA then computes $C_{new} = (\bigvee_{i=1}^{k} e_i) \vee c_{miss}$. We explain the constraint c_{miss} in the following.

If COVA would track all values used in an *if*-statement, the taints $(a, c_i, v_i) \in T_{in}$ share the following invariant $\bigvee_{i=1}^{k} c_i = C_{in}$. In practice, we often have an *incomplete* taint set T_{in}, which means the value of a for some constraint is present, but not for the other constraints and the invariant is violated. Figure 3 illustrates such a case. The taint set before the statement `if(a<0)` indicates a to hold the value 1 under the constraint X. The statement `if(a<0)` has constraint $C_{in} = \mathit{TRUE}$ (in red) and is unconditionally reachable. The taint set for a is incomplete, because COVA cannot propagate a taint for a under the constraint $!X$ as a holds an *unknown* return value of the method call `obj.randomValue()`. For variable a the constraint $!X$ is the missing-constraint c_{miss}.

(ii) T_{in} contains taints for both a and b. Assume there are k taints $(a, c_i, v_i) \in T_{in}$ and q taints $(b, d_j, w_j) \in T_{in}$. COVA computes e_{ij} by substituting a and b analogously as in the previous case. $e_{ij} = (v_i \oplus w_j) \wedge c_i \wedge d_j$ for the *TRUE* branch and $e_{ij} = \neg(v_i \oplus w_j) \wedge c_i \wedge d_j$ for the *FALSE* branch. Let c_{miss} and d_{miss} be the missing-constraints for variable a and b respectively, $C_{new} = (\bigvee_{ij} e_{ij}) \vee c_{miss} \vee d_{miss}$.

For a switch-statement, the flow function is analog.

CALLENTRYFLOWFUNC:

For a call statement $n : o.f(a_1, ..., a_k)$, $C_{new} = sym(f)$ if f is a UI callback from the constraint-APIs.

CALLLOCALFLOWFUNC: This function is the same as CALLENTRYFLOWFUNC.

CALLEXITFLOWFUNC: The constraint stays unchanged.

D. Obtaining the Constraint Map

VASCO terminates once a fixed point is reached. The result computed by VASCO is a map from $(ctx, n) \in \mathit{Context} \times \mathit{Statement}$ to data-flow facts in $\mathbb{C} \times 2^{\mathbb{T}}$. The C_{in} values before each statement are used to extract the constraint map. Since a statement n can be in multiple contexts, COVA merges the C_{in} values of n from different contexts by logical disjunction.

IV. IMPLEMENTATION

We implemented COVA as an extension to Soot [35] that computes partial path constraints for Java and Android applications. The constraint-APIs are given in configuration files. For Android apps, we construct call graphs using FlowDroid [1]. For alias analysis, COVA's taint analysis uses Boomerang [36], a demand-driven flow- and context-sensitive pointer analysis. To simplify and evaluate the constraints during the constraint analysis, we rely on the theorem prover Z3 [37]. COVA only propagates taints with satisfiable constraints. The current implementation fully supports constraints in boolean propositional logic, equality logic and linear arithmetic logic. To increase scalability we did not model string operations precisely, but instead only use imprecise symbolic values to express them.

To be able to judge the confidence in the results COVA reports, we developed a new micro-benchmark (publicly available with COVA), comprising 92 specially crafted test programs (e.g., primitives or heap objects used in conditional statements, nested conditional statements, intra- and inter-procedural conditional dependencies, callback invocations, indirect conditional dependencies, etc.). On this benchmark, COVA achieved a precision of 100% and a recall of 95%, which gives us a reasonable confidence of the results COVA computes.

V. EVALUATION

Our evaluation is designed to understand the nature of taint flows detected by a static Android taint-analysis tool, and potential avenues to eliminating false positives among those

TABLE I
STRATIFIED SAMPLING THE TOP SOURCE-SINK-PAIRS AMONG THE TAINT FLOWS REPORTED BY FLOWDROID

Gr.	Source	Sink	#Taint Flows	#Apps	#Sampled Apps
		Intra-procedural			
A	java.net.URL.openConnection	java.net.HttpURLConnection.setRequestProperty	2,193	535	54
B	android.os.Handler.obtainMessage	android.os.Handler.sendMessage	1,410	199	20
C	java.net.HttpURLConnection.getOutputStream	java.io.OutputStream.write	194	166	17
		Inter-procedural			
D	android.database.Cursor.getString	android.app.Activity.startActivityForResult	1,440	156	16
E	java.net.URL.openConnection	java.net.HttpURLConnection.setRequestProperty	862	291	30
F	android.database.Cursor.getString	android.os.Bundle.putString	847	85	9

taint flows. We chose FlowDroid [1] as our evaluation tool, since it is well maintained and, according to previous studies [29], [38], beats other comparable tools both in accuracy and efficiency. The evaluation intends to answer the following research questions:

- RQ1. What types of taint flows does FlowDroid report? How common is each type?
- RQ2. Do "low-hanging fruits" exist, and if so, what characteristics do these taint flows have?

We next address both questions one after the other.

RQ1. What types of taint flows does FlowDroid report? How common is each type?:

a) Methodology: We randomly sampled 2,000 Android apps from the AndroZoo dataset [39]. All sampled apps were available in popular app stores (Google Play and Anzhi Market) between year 2016 and 2018. These criteria ensure that we report on the real-world apps from recent years. The apps can be downloaded from this link[2]. We used FlowDroid v2.5.1 in its default configuration. In this configuration, FlowDroid lists 47 methods as sources[3] and 122 as sinks. We applied FlowDroid to these 2,000 apps and it reported 1,022 apps to contain data leaks. FlowDroid reported 28,176 taint flows for these 1,022 apps, which makes it intractable to study every single taint flow in every app. Thus, our methodology follows these two steps:

(1) we measured which *source-sink-pairs* appeared in the taint flows and chose the top 3 source-sink-pairs among intra- and inter-procedural taint flows (see Table I) for our case study, since these source-sink-pairs dominate a large amount of taint flows, and among most (88%) of the remaining pairs each pair only appeared in fewer than 50 taint flows (out of 28,176 in total). To determine apps for our case study, we applied *stratified random sampling*: the apps with taint flows using these 6 source-sink-pairs are divided into 6 groups, one for each pair, which we here label with A to F. Due to large amount of reverse engineering and manual work involved in the inspection, we only sampled 10% of the apps of each group. The manual inspection was done in pair by two of the authors.

(2) we conducted an experiment in which we applied both FlowDroid and COVA to the apps in our dataset. An app is passed to both FlowDroid and COVA (see Figure 2). The experiment was designed to classify the taint flows with the following *types*:

- *UI-constrained taint flows* are dependent on UI actions.
- *Configuration-constrained taint flows* are dependent on hardware/software configuration.
- *I/O-constrained taint flows* are dependent on data inputs through streams or file system.

Whenever a taint flow is reported by FlowDroid, we conjoin the constraints of the source and the sink computed by COVA to obtain the leak-constraint and use it to classify this taint flow. For instance, the leak in our motivating example (see Figure 1) will be classified to both UI-constrained and Configuration-constrained, since the leak-constraint $SDK \leq 26 \wedge CLICK \wedge TELEPHONY$ contains symbolic values which stand for configuration (*SDK* and *TELEPHONY*) and UI action (CLICK) at the same time. We collected a list of constraint-APIs from the Android Platform (API level 27) that COVA ought to track:

- 335 APIs for UI actions, which are UI callbacks. We first scanned the whole Android platform with gestural keywords such as click, scroll, etc., to extract a list of possible UI callbacks. Based on this list, callbacks were manually selected.
- 448 APIs for hardware and software configuration. We collected the APIs based on the official Android guide of device compatibility [40].
- 120 APIs for data input via I/O streams or file system, which are mainly from the *java.io* package.

The selection of the APIs was done by pair-reviewing by two researchers. The list is publicly available with COVA. We set a timeout of 30 minutes per app for COVA. COVA terminated its analysis and computed a complete constraint map for 315 apps. (In cases in which analysis times out, this was most often due to slow constraint solving in Z3, see section VI.) For the remaining 707 apps, COVA only computed partial constraint maps. The experiment was conducted on a virtual machine with an Intel Xeon CPU running on Debian GNU/Linux 9 with Oracle's Java Runtime version 1.8 (64 bit). The maximal heap size of the JVM was set to 24 GB.

b) Results: Figure 4 shows the different types of taint flows and their fractions in our study. While the false positives were all identified in step (1), the fractions of other types (UI-constrained, Configuration-constrained, I/O-constrained, Infeasible, Unconstrained and intersections) were computed in

[2]https://www.kaggle.com/covaanalyst1/cova-dataset

[3]46 sources are listed in the configuration file `SourcesAndSinks.txt` and 1 source *android.app.Activity.findViewById(int)* is treated specially by only considering password input fields.

```
/*** code pattern 1 ***/
HttpURLConnection c = (HttpURLConnection) new
    URL("http...").openConnection(); //source
c.setDoInput(true);
c.setRequestProperty("User-Agent", "Mozilla/5.0");
    //sink

/*** code pattern 2 ***/
Message m = handler.obtainMessage(); //source
handler.sendMessage(m); //sink

/*** code pattern 3 ***/
HttpURLConnection c = (HttpURLConnection) new
    URL("http...").openConnection();
c.setDoOutput(true);
OutputStream s = c.getOutputStream(); //source
s.write(data); //sink
```

Listing 1. Code patterns from group A, B, C

```
public class MainActivity extends Activity {
  private String secret;
  public void caller(){
    ...
    this.secret = cursor.getString(i); //source
    callee();
  }
  public void callee() {
    this.startActivityForResult(new Intent(), ...);
        //sink
}}
```

Listing 2. Code pattern from group D

step (2). The infeasible taint flows are those with unsatisfiable leak-constraints reported by COVA. The fraction of the unconstrained taint flows is only an upper bound. For apps on which COVA timed out, if there is no constraint for the source and sink statements of a taint flow in the partial constraint map, we assigned this taint flow with the type "Unconstrained".

In step (1), we studied the most common taint flows while keeping the following questions in mind: Is this taint flow feasible, i.e., could it be a leak? Do some code patterns with the same source-sink-pair exist in the taint flows? To assess the feasibility, we used the data-flow path (in Jimple) between source and sink of each taint flow reported by FlowDroid and the decompiled code of the apps.

Intra-procedural taint flows, Groups A-C: As shown in Table I, the source-sink-pair (*URL.openConnection*, *HttpURLConnection.setRequestProperty*) appeared most frequently in both intra- and inter-procedural taint flows (group A and E). While the given source method creates a connection object with a given URL, the sink sets the general properties of a HTTP request. This source-sink-pair combination apparently does not constitute a leak, since the connection is not even opened when only calling *URL.openConnection*. One instead still has to call *URLConnection.connect* or equivalent methods (e.g., *URLConnection.getInputStream*) to initiate the communication [41]. During the inspection for the above-mentioned source-sink-pair in group A, we discovered that the reported taint flows share some common patterns. Code pattern 1 in Listing 1 shows an example usage of this source-sink-pair, which is a common way to set up the header of a HTTP request. This is no leak.

Code pattern 2 in Listing 1 is another common pattern we identified in group B. The factory method *Handler.obtainMessage* is regarded as a source by FlowDroid. This method *creates* a new empty message instance. It does *not* poll a message from the message queue of the Android handler. This method should thus be excluded from the list of sources. Code pattern 3 from group C is a similar case.

In summary, taint flows which fall into these code patterns are false positives. To determine how many taint flows match these code patterns, we extended FlowDroid to detect these patterns, and re-analyzed the apps in groups A, B and C. In the end, 2,630 (46%) reported intra-procedural taint flows matched these three code patterns. As shown in these code patterns, the root cause of these false positives is that their sources, which FlowDroid uses in its default configuration, are actually inappropriate, i.e., they do not return sensitive data.

Such a big fraction of false positives caused by this reason cannot be ignored. Thus, the first author examined *all* 47 sources by reading the Javadoc carefully together with one developer with more than 5 years experience in Java. Altogether, they identified 11 APIs that were mistakenly made source/sink (see Table II). These inappropriate sources and sinks resulted in 7,767 reported taint flows, which is 28% of all reported taint flows (intra- and inter-procedural).

TABLE II
INAPPROPRIATE SOURCES AND SINKS USED BY FLOWDROID

Signature
android.os.Handler.obtainMessage()
android.os.Handler.obtainMessage(int,int,int)
android.os.Handler.obtainMessage(int,int,int,Object)
android.os.Handler.obtainMessage(int)
android.os.Handler.obtainMessage(int,Object)
android.app.PendingIntent.getActivity(Context,int,Intent,int)
android.app.PendingIntent.getActivity(Context,int,Intent,int,Bundle)
android.app.PendingIntent.getBroadcast(Context,int,Intent,int)
android.app.PendingIntent.getService(Context,int,Intent,int)
java.net.URLConnection.getOutputStream()
java.net.URL.openConnection() *[*regarded as both source and sink*]

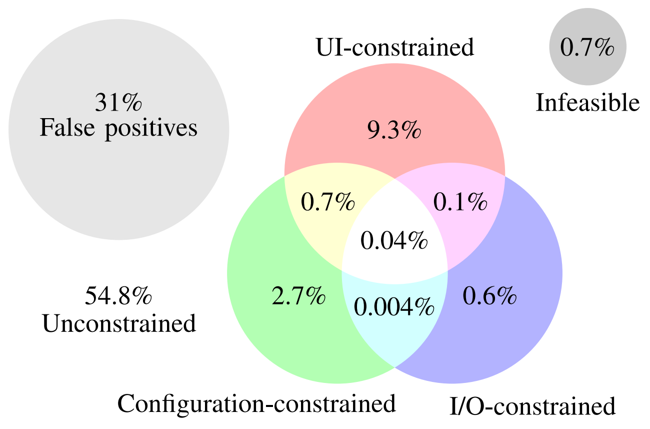


Fig. 4. Different types of taint flows

About a quarter (11 out of 47) of default sources provided by FlowDroid are inappropriate and cause more than a quarter (28%) of all reported taint flows being false positives.

After a discussion with FlowDroid's maintainers, they confirmed the mistake and removed the inappropriate sources and sinks from the default list in FlowDroid's GitHub repository.[4] This affects the pairs A, B, C and E.

Inter-procedural taint flows, Groups D-F: Because the source of group E is inappropriate, the manual inspection of this group was unnecessary. We next describe the results of the manual inspection of the remaining groups D and F. The taint flows from group D use the source-sink-pair (*Cursor.getString*, *Activity.startActivityForResult*). Taint-flows with this source-sink-pair could be part of a leak when the intent passed to *Activity.startActivityForResult* contains data reading from *Cusor.getString* and the second activity passes this received data to an untrusted sink. Since taint flows using such inter-component communication are outside the scope of FlowDroid, our goal for inspection was only to check if this partial data-flow is feasible. Surprisingly 88% of the taint flows from Group D proved to be false positives. All these false positives share a similar code pattern, shown in Listing 2. In this example, FlowDroid taints `this.secret` and reports a leak when the sink method is called on the base object of the taint `this.secret`, which is the `this` object. However, there is no tainted data that flows into the intent passed for the sink method. Such over-approximation in FlowDroid's analysis logic, while sometimes useful, is too approximative for the sink *Activity.startActivityForResult*.

Generally, taint flows with taints connecting sources and sinks on the same objects should be filtered. Thus, we extended FlowDroid with a static analysis that detects such cases and re-analyzed the relevant apps. 330 taint flows matched the false positive pattern in Listing 2. In total, we identified 978 taint flows with taints connecting sources and sinks on the same objects. The sinks appearing in these taint flows are mainly APIs used for inter-component communication. The remaining sinks (e.g. *HttpResponse.execute(HttpUriRequest)*) only make sense when the right parameter was tainted. However, FlowDroid reported these taint flows when the base object was tainted.

In our study, all taint flows reported by FlowDroid with taints connecting sources and sinks on the same objects are false positives.

The sink *Bundle.putString* used in taint flows from group F is also an API for inter-component communication. Similar to group D, we checked if the reported partial flow is feasible. We found out that while 89% of the reported flows *are* feasible, the false positives all happened in one app and the flows were just for putting the name of the app into the sink. However, the fact that the partial flows are feasible does not mean they are a part of true leaks, since one does not know how the sensitive data stored in *Bundle* were used in other activities, which was not reported by FlowDroid.

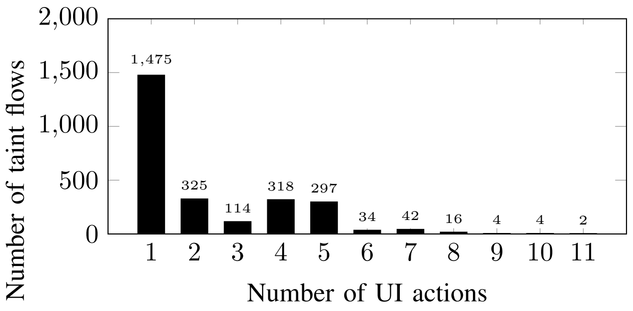


Fig. 5. The distribution of UI-constrained taint flows

At least one third (31%) of taint flows reported by FlowDroid in the default configuration are false positives.

In step (2), we classified[5] taint flows that are dependent on the constraint-APIs with COVA. For instance, if the leak-constraint contains symbolic values that relate to the constraint-APIs from UI callbacks, then this taint flow belongs to category "UI-constrained". Certainly, there can be taint flows which belong to multiple categories.

As shown in Figure 4, among the 14.2% taint flows whose occurrences are dependent on the constraint-APIs in the categories, the majority are in a single category – UI-constrained, which means they only occur when some specific UI actions are performed. 2.7% of the taint flows may happen under certain environment configurations, and 0.6% are dependent on inputs from I/O operations. The numbers in the Venn diagram's intersections of different categories indicate that interactions between values read from APIs in different categories are rare but do exist.

Taint flows are seldom conditioned by combinations of UI interactions, environment configurations and I/O operations. Most taints could thus be dynamically confirmed by *different* tools that specialize on the respective category.

Because complex UI dependencies may require a test harness to drive the application with the needed sequences of events, we investigated the complexity of the UI actions. Intuitively, taint flows triggered by a sequence of user actions should exist. A previous study [9] has found malicious applications in which a user needs to click a series of buttons to trigger the display of a widget which leaks the data. To estimate the complexity, we calculated how many different UI actions are involved in a UI-constrained taint flow by counting the number of symbolic values for UI actions used in the leak-constraint. (Note that our constraint encoding is able to distinguish different UI actions.) Figure 5 shows the distribution of UI-constrained taint flows. 56.1% (1,475) of

[4]The link to the commit: https://github.com/secure-software-engineering/FlowDroid/commit/211b73e32a0ade1ded021f2fc30b0aa647be5862

[5]**Note:** For the classification, we excluded the false positives we identified in step (1).

TABLE III
TOP CONSTRAINT-APIS RELATED TO TAINT FLOWS

UI Callback	**#Flows**
android.view.View.OnClickListener.onClick	2088
android.widget.AdapterView.OnItemClickListener.onItemClick	623
android.content.DialogInterface.OnClickListener.onClick	595
Configuration	**#Flows**
android.os.Build.VERSION.SDK_INT	255
android.content.Context.getSystemService("connectivity")	246
android.content.Context.getSystemService("location")	224
I/O Operation	**#Flows**
java.io.InputStream.read	158
java.io.BufferedReader.readLine	16
java.io.ObjectInputStream.readObject	10

taint flows happen after a single UI action. There are only 3.8% of taint flows that may require 6 or more UI actions. Maximally 11 different actions appeared in a leak-constraint. However, executing the taint flow does not require all 11 actions at the same time, since there are disjunctions in the leak-constraint (e.g., $A \vee B$ contains two actions A and B, but one action is sufficient to execute the taint flow).

Despite the existence of sophisticated sequences of UI actions, our results indicate the dynamic exploration of most UI action-related taint flows could be easier than expected.

Among the configuration-constrained taint flows, the distribution is even simpler: the largest number of taint flows (85.6%) require a single configuration option and 13.9% of them are dependent on two options. Only 5 taint flows happen under a complex configuration with more than two options.

The necessary configuration-based conditions for exposing taint flows are easy to be satisfied in the majority of cases.

Table III shows which constraint-APIs from our categories are most frequently used. While click events are relevant to most taint flows related to UI actions, the Android SDK version plays a considerable role in environment configurations. This is not surprising to us, since the Android operating system remains highly fragmented [42]–[44] and developers are challenged to produce applications that are compatible to multiple platform versions. However, the importance of taint flows which only occur in obsolete versions of Android may be limited in practice. Constraints based on I/O operations are mostly checking if the end of a data stream has been reached, e.g., `if(inputStream.read()!=-1)`.

Additionally, we observed that for 28% of the 208 infeasible taint flows, their source statements were not executable, since the path constraint is *FALSE*. For 76% of the infeasible taint flows, their sink statements will never be executed at runtime. Such dead code was probably intentionally built in by developers [45], e.g., during sampling, we inspected code used for logging (sinks of taint flows) that was disabled with a boolean flag for the released APK, but not removed.

RQ2. Do "low-hanging fruits" exist, and if so, what characteristics do these taint flows have?:

c) Methodology: We consider taint flows that are intra-procedural and unconstrained, i.e., the leak-constraint computed by COVA is equal to *true*, as *"low-hanging fruits"*. To acquire the characteristics of the "low-hanging fruits", we again applied stratified random sampling with proportion 10% to the taint flows with top source-sink-pairs in Table IV.

d) Results: In our study, only 3.5% of the taint flows are "low-hanging fruits" (intra-procedural and unconstrained). However, "low-hanging fruits" are still the majority of the intra-procedural taint flows and they exist in 32% (329) of the apps in our dataset. During the manual inspection for taint flows with top source-sink-pairs in Table IV, we found that taint flows with source-sink-pair of group X cannot usually be interpreted as leaks. Listing 3 shows a simplified taint flow using this source-sink-pair. The private field `this.secret` is first tainted. FlowDroid taints the return value of `this.getContentResolver()` since the base object is the prefix of the tainted `this.secret`. Finally, the taint flow is then reported when the sink is called on the tainted `this.getContentResolver()`. This is again an over-approximation FlowDroid uses similar to the one in Listing 2. However, such taint flow *could* be a leak, since the implementation of *Context.getContentResolver* and *ContentResolver.query* could be overridden maliciously.

In comparison to group X, taint flows of group Y and Z are straightforward: they log data from databases. Actually, the log methods from *android.util.Log* are the most frequently used sinks. About half (46%) of the "low-hanging fruits" are leaks in which sensitive information such as data from databases, location information, device ID, the MAC addresses or even passwords are logged. Many of these leaks even have source and sink at the same line of code. In addition, the text that will be logged often specifies what kind of data is being logged.

Besides log methods, sinks for inter-component communication such as *Bundle.putString*, *SharedPreferences.putString* and *Context.sendBroadcast* are also popular among the "low-hanging fruits". They appeared in 20% of the taint flows. To determine if these taint flows are malicious, additional context must be provided, since benign applications often use these methods for accessing and modifying preference data between activities.

Discussion: First, our results show important ways in which taint-analysis tools can and should be improved. On the one hand, the sources and sinks configured for the tools should be checked more carefully, since an inappropriate source can cause a large amount of false positives, as we determined for FlowDroid in RQ1. Researchers who used FlowDroid in the default configuration may need to re-evaluate their conclusions. Even in just a short investigation, we already found 9 papers in which the respective work was built on top of FlowDroid and inappropriate sources or sinks were used [20]–[28]. In none of these papers did the authors mention that they have manually checked for false positives that would have been caused by the inappropriate source/sink configurations. Hence, while it is possible that such manual checks were conducted without mentioning them, it is equally possible that

```
public class MainActivity extends Activity {
  private String secret;
  public void foo(){
       this.secret = cursor.getString(i); //source
       ...
       this.getContentResolver().query(...); //sink
}}
```

Listing 3. Code pattern from group X

TABLE IV
TOP SOURCE-SINK-PAIRS AMONG "LOW-HANGING FRUITS"

Gr.	Source	Sink	#Sampled/#Total Taint Flows
X	android.database. Cursor.getString	android.content. ContentResolver.query	14/137
Y	android.database. Cursor.getString	android.util.Log.e	10/96
Z	android.database. Cursor.getString	android.util.Log.i	7/70

the papers report results that are distorted by the presence of those false positives. Given the over 1,000 citations of the FlowDroid paper, many more such works are likely to exist. On the other hand, some rules used in taint analysis may be not suitable for all sources and sinks, as we have seen in the case shown in Listing 2 for FlowDroid, a taint flow was reported when the base object calling the sink was tainted. However, here the correct way to report a taint flow is when the actual argument (intent) of the sink is tainted. Such cases could be handled easily without increasing analysis complexity. Although we only studied the results reported by FlowDroid, problems we discovered for such a widely used tool may not be a single case among numerous taint-analysis tools.

Second, hybrid analysis tools may well be feasible for the case of Android. The results of RQ1 show that to confirm static taint flows dynamically, they should focus on modeling UI actions, but in some cases must be able to set correct environment options as well, and must deal with stream-I/O to some limited extent. Luckily, the overlap between those three classes is small, so that one can probably go a long way even by designing specific, decoupled analysis tools for all three situations. Our tool COVA can further aid the implementation of such hybrid analysis tools: the path constraints it computes can guide dynamic analyses, even in situations where flows are not conditioned on external stimuli at all.

Third, "low-hanging fruits" are quite common—they exist in 32% of all apps as we show in RQ2. Many of such leaks can be easily fixed, and so even purely static taint-analysis tools can and should prioritize these leaks in the report.

VI. Limitations

COVA computes partial path constraints—it only considers control-flow decisions that are dependent on a list of constraint-APIs we collected. However, as we discuss in section V, we feel that by the way this list was collected, it is comprehensive. Although COVA supports most language features, some corner cases such as reflection or native calls are not covered [46]. In some cases, the taint set computed by COVA may be incomplete due to unknown return values of API method calls such that an over-approximated constraint is computed. For Android applications, we use the call graph constructed by FlowDroid. This call graph, however, is partially incomplete for library methods and some UI callbacks [47], [48]—a known limitation of FlowDroid.

Since COVA uses Z3 for constraint-solving, the limitations of Z3 are inherited by COVA. In our experiments, an average of 49% percent of the analysis time was occupied by Z3. In fact, this is also one of the main reasons why COVA failed to analyze some apps within the given time budget. In the worst case, 98% of the analysis time for an app was spent for constraint-solving. Increasing the time budget may not help, since Z3 can hit memory pressure and throw exceptions when solving large formulas, which happened in our preliminary experiments. Such exceptions cannot be evaded by increasing the JVM heap size, since they originate from the native code of Z3. In the future, we plan to turn COVA into an on-demand analysis such that it only computes a constraint for a given statement instead of computing a constraint map for all reachable statements.

VII. Related Work

We discuss how our approach relates to previous work in the areas of taint analysis, path conditions, as well as hybrid analysis approaches.

Studies Involving Taint Flows: Many researchers have studied Android applications from various perspectives [17], [24], [25], [49]–[52]. Avdiienko et al. [24] compared the taint flows in benign apps against those in malicious apps, and used machine learning to identify the differences in usage of sensitive data. Unlike COVA, their approach MUDFLOW does not consider path constraints. Keng et al. [17] monitored 220 Android apps with the dynamic taint-analysis tool TaintDroid [53] to study the correlation between user actions and leaks. However, their results are limited to the leaks they observed during the runtime. Their results show that many apps leak data due to user actions on certain GUI widgets, which we were able to show statically. Closely related to our approach, Lillack et al. [51] also extended taint analysis to explore the variability of Android apps based on load-time configuration. However, their approach encodes constraint analysis as a distributive problem in the IFDS framework [33]. For our purpose, this model is insufficient, since the execution of a branch may depend simultaneously on two or more configuration options, which IFDS cannot express [31], [54].

Path Conditions: Many approaches have considered path conditions to increase the accuracy of their analysis. Snelting [30] has shown how exacting and simplifying path conditions can improve slice accuracy. Taghdiri et al. [55] made information flow analysis more precise by incrementally refining path conditions with witnesses that did not yield an information flow in execution. TASMAN [56] leverages backward symbolic execution as a post-analysis to eliminate false positives in which taint flows along paths are infeasible at runtime. TASMAN is based on the distributive IFDS framework, but constraint computation is not a distributive problem. TASMAN thus needs to approximate in places which COVA can handle precisely. In result, COVA's computation is more expensive but COVA's path expressions are also more precise. A general major limitation of symbolic execution is that it cannot explore executions with path conditions which the underlying SMT solver cannot deal with in the given time budget [57]. This limitation is shared with COVA. To improve scalability, modern symbolic execution techniques mix concrete and symbolic execution in so-called concolic execution. Anand et al. [58] propose a concolic execution approach to generate sequences of UI events for Android applications. Schütte et al. [59] also use concolic execution to drive execution to cover target code. They claim that their approach is not limited to any specific kind of conditions, i.e., can handle all kinds of condition (user input, environmental setting and even remote site input). Yet their prototype ConDroid was only designed and evaluated for one specific vulnerability. COVA was evaluated on a wide range of taint flows. The results of our study indicate that tools which seek to expose taint flow dynamically could concentrate on one kind of condition at a time, which is important for scalability.

Hybrid Analysis: A number of hybrid approaches, i.e., combinations of static and dynamic analysis, have been proposed for Android malware detection [9]–[14]. SmartDroid by Zheng et al. [9] statically detects UI interaction sequences that lead to sensitive API calls, and it exposes those behaviors dynamically. Yang et al. [10] propose a hybrid approach in which they first identify the possible attack-critical path with static mining algorithms based on sensitive APIs and existing malware patterns, then execute the program in a focused scope under dynamic taint analysis. Wong et al. [13] demonstrate IntelliDroid, a tool which generates a reasonably small set of inputs statically to trigger malicious behavior of applications. Their evaluation shows that one only needs to execute a very small part of the application to expose malicious behaviors. Recent work of Rasthofer et al. [12] combines a set of static and dynamic analyses with fuzzing to generate execution environments to expose hidden malicious behaviors efficiently.

VIII. Conclusion

In this paper, we introduced COVA, a new tool for tracking user-defined APIs through the program and computing path constraints based on these APIs. We conducted a COVA-supported study which gathers information about the nature of static taint-analysis results. Our study shows important ways how static taint-analysis tools can be improved and how information about taint-flows conditioned on different factors can be used for future taint-analysis research, particularly with the aim of further eliminating false positives.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 259–269.

[2] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 1329–1341.

[3] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[4] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 280–291.

[5] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015, pp. 106–117.

[6] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, "Static analysis of android apps: A systematic literature review," *Information & Software Technology*, vol. 88, pp. 67–95, 2017.

[7] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," 2016, pp. 332–343.

[8] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" 2013, pp. 672–681.

[9] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

[10] T. Yang, K. Qian, L. Li, D. C. Lo, and L. Tao, "Static mining and dynamic taint for mobile security threats analysis," in *2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016*, 2016, pp. 234–240. [Online]. Available: https://doi.org/10.1109/SmartCloud.2016.43

[11] J. C. J. Keng, "Automated testing and notification of mobile app privacy leak-cause behaviours," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 880–883. [Online]. Available: http://doi.acm.org/10.1145/2970276.2975935

[12] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: targeted fuzzing of android execution environments," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 300–311. [Online]. Available: https://doi.org/10.1109/ICSE.2017.35

[13] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[14] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 899–914. [Online]. Available: https://doi.org/10.1109/SP.2015.60

[15] J. Oberheide and M. Charlie, "Dissecting the Android Bouncer," accessed 2018-07-20. [Online]. Available: https://jon.oberheide.org/files/summercon12-bouncer.pdf

[16] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, 2014, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590325

[17] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan, “The case for mobile forensics of private data leaks: towards large-scale user-oriented privacy protection,” in *Asia-Pacific Workshop on Systems, APSys ’13, Singapore, Singapore, July 29-30, 2013*, 2013, pp. 6:1–6:7. [Online]. Available: http://doi.acm.org/10.1145/2500727.2500733

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 317–331. [Online]. Available: https://doi.org/10.1109/SP.2010.26

[19] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “DTA++: dynamic taint analysis with targeted control-flow propagation,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[20] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, “I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis,” *CoRR*, vol. abs/1404.7431, 2014.

[21] O. Mirzaei, G. Suarez-Tangil, J. E. Tapiador, and J. M. de Fuentes, “Triflow: Triaging android applications using speculative information flows,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, 2017, pp. 640–651. [Online]. Available: http://doi.acm.org/10.1145/3052973.3053001

[22] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, “Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data,” in *2017 IEEE Symposium on Computers and Communications, ISCC 2017, Heraklion, Greece, July 3-6, 2017*, 2017, pp. 438–443.

[23] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla, “What did really change with the new release of the app?” in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 142–152. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196449

[24] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, “Mining apps for abnormal usage of sensitive data,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 426–436.

[25] L. Sinha, S. Bhandari, P. Faruki, M. S. Gaur, V. Laxmi, and M. Conti, “Flowmine: Android app analysis via data flow,” in *13th IEEE Annual Consumer Communications & Networking Conference, CCNC 2016, Las Vegas, NV, USA, January 9-12, 2016*, 2016, pp. 435–441. [Online]. Available: https://doi.org/10.1109/CCNC.2016.7444819

[26] K. Tian, G. Tan, D. D. Yao, and B. G. Ryder, “Redroid: Prioritizing data flows and sinks for app security transformation,” in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, 2017, pp. 35–41. [Online]. Available: http://doi.acm.org/10.1145/3141235.3141239

[27] S. Rasthofer, S. Arzt, and E. Bodden, “A machine-learning approach for classifying and categorizing android sources and sinks,” in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[28] S. Wu, P. Wang, X. Li, and Y. Zhang, “Effective detection of android malware based on the usage of data flow apis and machine learning,” *Information & Software Technology*, vol. 75, pp. 17–25, 2016.

[29] L. Qiu, Y. Wang, and J. Rubin, “Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 176–186.

[30] G. Snelting, “Combining slicing and constraint solving for validation of measurement software,” *Static Analysis SE - 23*, vol. 1145, no. Springer, pp. 332–348, 1996.

[31] G. A. Kildall, “A unified approach to global program optimization,” in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, 1973, pp. 194–206.

[32] R. Padhye and U. P. Khedker, “Interprocedural data flow analysis in soot using value contexts,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis, SOAP 2013, Seattle, WA, USA, June 20, 2013*, 2013, pp. 31–36.

[33] T. W. Reps, S. Horwitz, and S. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, 1995, pp. 49–61.

[34] A. Deutsch, “Interprocedural may-alias analysis for pointers: Beyond *k*-limiting,” in *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, 1994, pp. 230–241.

[35] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, “The Soot framework for Java program analysis: a retrospective,” *Cetus ’11*, 2011. [Online]. Available: https://sable.github.io/soot/resources/lblh11soot.pdf

[36] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, “Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java,” in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6116

[37] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340.

[38] F. Pauck, E. Bodden, and H. Wehrheim, “Do android taint analysis tools keep their promises?” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 331–341.

[39] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, “Androzoo: collecting millions of android apps for the research community,” in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 468–471.

[40] “Device compatibility,” accessed 2018-08-20. [Online]. Available: https://developer.android.com/guide/practices/compatibility

[41] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, “Effective typestate verification in the presence of aliasing,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, 2006, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146254

[42] “Android Fragmentation,” accessed 2018-07-19. [Online]. Available: https://opensignal.com/reports/2015/08/android-fragmentation/

[43] P. Mutchler, Y. Safaei, A. Doupé, and J. C. Mitchell, “Target fragmentation in android apps,” in *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 204–213. [Online]. Available: https://doi.org/10.1109/SPW.2016.31

[44] “Platform Versions,” accessed 2018-07-19. [Online]. Available: https://developer.android.com/about/dashboards/

[45] M. Eichberg, B. Hermann, M. Mezini, and L. Glanz, “Hidden truths in dead software paths,” in *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich*, 2016, pp. 63–64. [Online]. Available: https://dl.gi.de/20.500.12116/723

[46] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, “In defense of soundiness: a manifesto,” *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[47] S. Arzt, “Static Data Flow Analysis for Android Applications,” Ph.D. dissertation, Technische Universität Darmstadt, Dec 2016. [Online]. Available: http://bodden.de/pubs/phd-arzt.pdf

[48] Y. Wang, H. Zhang, and A. Rountev, “On the unsoundness of static analysis for android guis,” in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*, 2016, pp. 18–23.

[49] R. Stevens, J. Ganz, V. Filkov, P. T. Devanbu, and H. Chen, “Asking for (and about) permissions used by android apps,” in *Proceedings of the*

10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013, 2013, pp. 31–40.

[50] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 303–313.

[51] M. Lillack, C. Kastner, and E. Bodden, "Tracking Load-time Configuration Options," *IEEE Transactions on Software Engineering*, vol. 5589, no. c, pp. 1–1, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8049300/

[52] H. Chen, H. Leung, B. Han, and J. Su, "Automatic privacy leakage detection for massive android apps via a novel hybrid approach," in *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, 2017, pp. 1–7.

[53] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014.

[54] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Lecture Notes in Computer Science*, vol. 915, pp. 651–665, 1995.

[55] M. Taghdiri, G. Snelting, and C. Sinz, "Information flow analysis via path condition refinement," in *Formal Aspects of Security and Trust - 7th International Workshop, FAST 2010, Pisa, Italy, September 16-17, 2010. Revised Selected Papers*, 2010, pp. 65–79.

[56] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, 2015, pp. 1–6.

[57] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[58] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 59.

[59] J. Schütte, R. Fedler, and D. Titze, "Condroid: Targeted dynamic analysis of android applications," in *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, 2015, pp. 571–578.