# A Qualitative Analysis of Taint-Analysis Results

Linghui Luo
*Paderborn University*
Paderborn, Germany
linghui.luo@upb.de

Eric Bodden
*Paderborn University & Fraunhofer IEM*
Paderborn, Germany
eric.bodden@upb.de

Johannes Späth
*Fraunhofer IEM*
Paderborn, Germany
johannes.spaeth@iem.fraunhofer.de

***Abstract*—In the past, researchers have developed a number of popular taint-analysis approaches, particularly in the context of Android applications. Most previous work in this area has focused on handling Android's unique features, on making analyses scale, or on eliminating false positives by enhancing the precision of static analysis. A promising avenue of increasing precision is to confirm static taint-analysis results through dynamic witnesses.**

**To guide such research, we present the first study that evaluates static Android taint-analysis results at a *qualitative* level. To facilitate this study, we have implemented an extension of the Soot analysis framework called COVA, which computes partial path constraints. These constraints inform about the circumstances under which taint-flows may actually occur in practice.**

**Using COVA, we have conducted a qualitative study on the taint-flows in 1,022 real-world Android applications. Our results reveal two key findings: (1) Many false positives arise due to *inappropriate* source and sink configuration provided by the existing taint-analysis tool. This impacts empirical results obtained for dozens of previously published Android analysis approaches. (2) 21% of the taint-flows are conditioned on user interactions, environment configurations and I/O operations.**

***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 no standalone applications but rather plugins for the Android framework. As such they have a distinct lifecycle, and often dozens if not hundreds of callbacks that respond to various environment stimuli such as button clicks or location changes.
- Android apps often need to operate correctly in different platform versions and devices, which is why they often comprise code that is conditionalized with respect to various environment parameters. In Android it is a common and even recommended practice, for instance, to obtain backward compatibility by probing the platform version.

Existing approaches for static taint analysis in Android have dealt with those challenges in different but always limited ways, always with a strong focus on analysis precision and scalability. While all existing papers proposing those tools do comprise a proper 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.

The lack of such *qualitative* data currently hinders progress in upcoming directions of research. In particular, 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 [7]–[12]. To be efficient and effective, however, such tools must be guided by the taint-flows' path conditions. To guide such research, this paper presents 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*...

- *User interactions*: Previous work [13] has shown that many Android apps leak data as a result of user actions on certain widgets in the apps.
- *Environment settings* (platform versions, country, etc.): Malicious applications can leak data based on environment settings [14], [15].
- *I/O operations*: Leaks that depend on specific inputs are difficult to trigger dynamically [16], [17], 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 inform about the circumstances under which taint-flows may actually occur in practice.

We first conducted a sanity check of the 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 the default configuration of sources and sinks provided by FlowDroid to be inappropriate and causing a huge amount of false positives. Unfortunately, we found this to impact the empirical evaluation of dozens of previously published papers [18]–[31].

After eliminating these false positives in coordination with FlowDroid's authors, we applied COVA to the reported taint-flows and conducted a qualitative study in which we classified the taint-flows based on the three factors mentioned above. Our study reveals that 21% of the flows are conditioned on any of the three factors, whereas 15% depend on user interactions.

In addition, we encountered 5% of taint-flows to be "low-hanging fruits", i.e., taint-flows that are not conditioned on the

three factors above, and are intra-procedural. These taint-flows have a high likelihood of being both correct and actionable to the developers. To summarize, this paper presents the following original contributions:

- COVA, a static analysis tool to compute path constraints.
- A COVA-supported qualitative study of taint-flows from 1,022 Android apps. The insights we gained are:
  - Default source and sink configurations provided by existing taint-analysis tools can be inappropriate and may cause many false positives.
  - Many inter-procedural taint-flows in Android applications are conditioned on user interactions, and less on environment configurations or I/O operations.
  - Reported intra-procedural taint-flows are most likely to be true positives.

## II. A Motivating Example

Figure 1 lists 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 "+491234" (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 not sufficient when one wants to understand *how* 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* [25]. The app leaks the device identifier only when it executes the source and sink statements. Their execution depends on three path conditions [32]. 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 generally applicable to any other client analysis that can be 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`
- `PackageManager.hasSystemFeature(...)`
- `OnClickListener.onClick(...)`

COVA performs a context-, flow-, and field-sensitive data-flow analysis (also a form of taint 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 callgraph reachable invocation of a constraint-API, COVA generates a tainted data-flow fact, simply referred to by *taint*. In Figure 1 for instance, 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 its execution (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 evaluation, 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 method `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 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 the same for the sink. In the example, the resulting

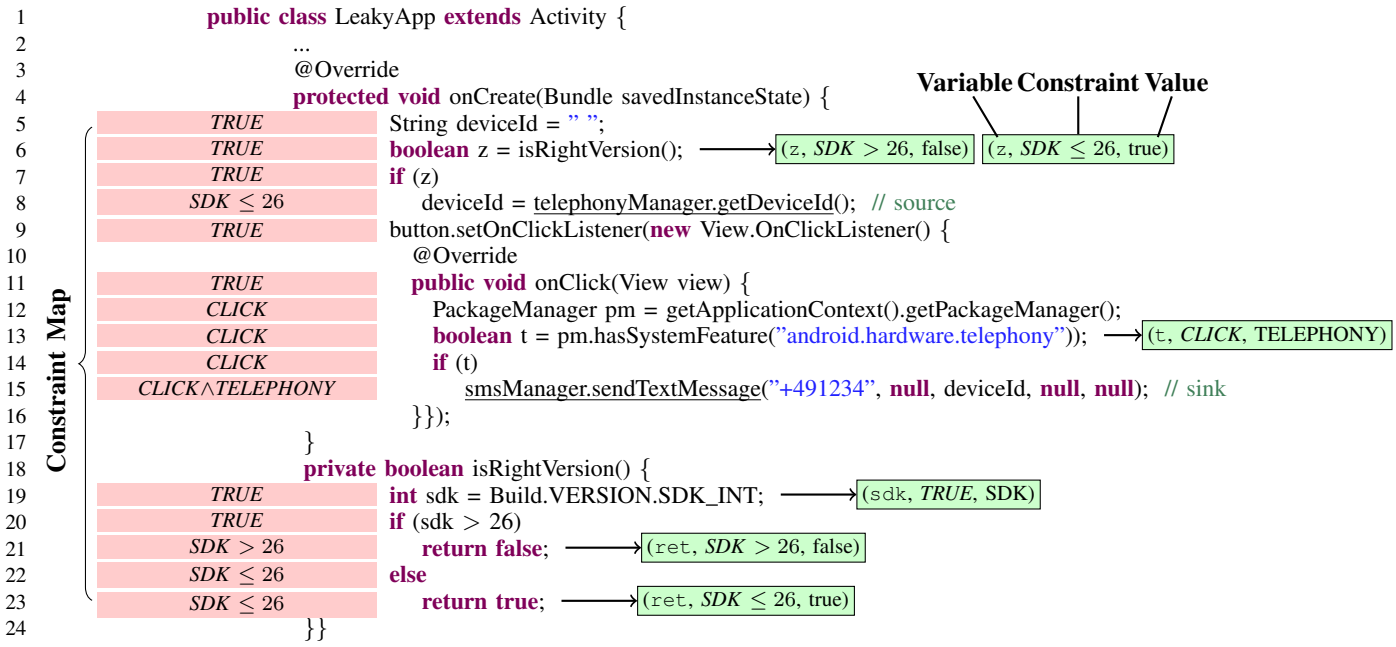


Fig. 1. A motivating example

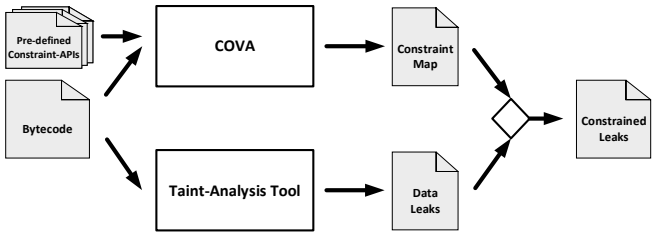


Fig. 2. The workflow of applying COVA to taint-analysis results

leak-constraint is $SDK \leq 26 \wedge CLICK \wedge TELEPHONY$. This leak-constraint is used later in our study to classify the leaks.

## III. Computing the Constraint Map

Computing the constraint map as shown in Figure 1 is non-trivial. The execution of a branch may depend *simultaneously* on two or more values of some constraint-APIs. A static analysis must *jointly* propagate all values, only then it is able to compute the final constraint for a branch. As the values have to be propagated jointly, the analysis is *non-distributive* [33]. Furthermore, each value of a constraint-API must be propagated throughout the whole program, values can flow to fields of objects before the fields are re-accessed elsewhere and aliasing relations must be computed.

We decided to implement the analysis within the data-flow framework VASCO [34], 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 of the program, the *flow functions* define how a data-flow fact changes when it flows from one statement to a successor. VASCO terminates when data-flow facts for all statements reach a fixed point.

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 $d' \in D$. 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. This function typically maps actual method arguments to formal parameters.
- CallExitFlowFunc: handles an inter-procedural flow from a return statement $n$ to the successor $m$ of a call site. 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 $\perp \in D$ 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. For instance, we seed the data-flow propagation with the fact $(TRUE, \emptyset)$ 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 tuple $(a, c, v) \in \mathbb{T}$ and consists of an access path (a local variable followed by a finite sequence of fields [35]). The access path encodes how the value of the constraint-API is heap-referencable 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 $(C_{out}, T_{out})$ 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 [36]. 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 Modifying the Taint Domain

NORMALFLOWFUNC:

This function mostly follows standard access-path based taint tracking data-flow propagation [1], [37]. 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]. In the following 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 [37].

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 $sym$ is a function that generates a unique symbolic value for a given constraint-API. For instance, $(sdk, \textit{TRUE}, \textit{SDK})$ 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 $C_{in} \neq$ *TRUE*, they are used to detect constraints that are indirectly influenced by constraint-APIs.

Jimple also allows return statements $n :$ **return** $z$ with constant values for $z$ and, if constraint $C_{in}$ is not equal to *TRUE*, the flow function returns the set $T_{out} = T^-_{in} \cup \{(\textit{ret}, C_{in}, z)\}$, i.e., COVA also creates a concrete taint $(\textit{ret}, C_{in}, z)$. We use a fake access path *ret* to symbolically represent the returned variable (e.g., $(\textit{ret}, \textit{SDK} \leq 26, true)$ 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 $C_{in} \neq$ *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 $n :$ **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 when an access path is propagated from callee to its caller, COVA also adds the required aliases [37].

### C. Flow Functions Modifying 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

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

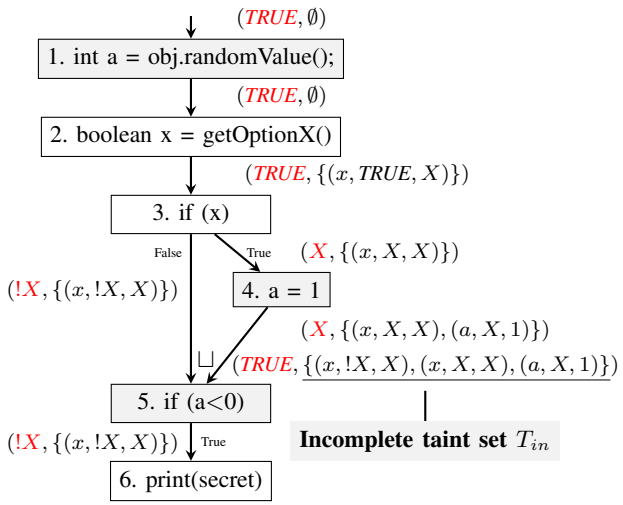


Fig. 3. An example shows an incomplete taint set. Assume getOptionX() is a constrain-API whose return value is represented by the symbolic value $X$.

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, i.e., $C_{out} = C_{in}$.

### D. Obtaining the Constraint Map

The termination of VASCO is guaranteed by the monotonicity of the flow functions and the finiteness of the lattice. The flow functions are monotonic since a variable can either be killed or stay tainted. Based on the taint set, the constraint is either less relaxed or stays the same. Once the taint set achieves the fixed point, the constraints will achieve the fixed point as well. 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 [36] that computes partial path constraints for Java and Android applications. The constraint-APIs are given in configuration files. For Android apps, we construct callgraphs using FlowDroid [1]. We make COVA publicly available and also will subject it to artifact evaluation.

As alias analysis, COVA's taint analysis uses Boomerang [37], 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 [38]. COVA's analysis 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 as explained in the flow functions (see subsection III-B).

To be able to judge the confidence in the results COVA reports, we developed a new micro-benchmark, 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 dependences, 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. We will make the benchmark available along with the source code of COVA's implementation.

## V. Evaluation

We next present the design, setup and results of our study.

### A. Experimental Design

Our experimentation is designed to understand the types of different taint-flows detected by a static taint-analysis tool. We chose FlowDroid [1] as our evaluation tool, since it is well maintained and beats other comparable tools both in accuracy and efficiency according to independent studies [25], [39]. We aim to classify the leaks reported by FlowDroid when applying it to real-world Android apps with the following categories:

- *UI-constrained leaks*: are dependent on UI actions.
- *Configuration-constrained leaks*: are dependent on certain environments, i.e., hardware or software configuration.
- *I/O-constrained leaks*: are dependent on data inputs through I/O streams or file system.

To achieve our goal, 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 generated list, we sorted out the callbacks manually by reading the documentation. Most UI callbacks selected are from the *android.view*, *android.widget* and *android.gesture* packages.
- 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.

In COVA, each constraint-API is assigned a unique symbolic value. An Android app is passed to both FlowDroid and COVA (see the workflow in Figure 2). Whenever a leak is reported by FlowDroid, we conjoin the constraints of the source and the sink statements computed by COVA to obtain the leak-constraint and use it to classify this leak. We aim to answer the following research questions:

- RQ1. Are the leaks reported by FlowDroid in the default configuration reasonable?
- RQ2. Do "low-hanging fruits" exist, and if so, what characteristics do these leaks have?
- RQ3. What types of leaks does FlowDroid report?

### B. Experimental Setup

We downloaded 1,966 Android apps from the AndroZoo dataset [41]. We chose apps that are available in popular app stores (Google Play and Anzhi Market) between year 2016 and 2018. This ensures that we report on the real-world apps from recent years. We used FlowDroid v2.5.1 in its default configuration. In this configuration, FlowDroid lists 47 methods as sources[2] and 122 sinks. To identify leaky apps, we firstly applied FlowDroid to these 1,966 apps alone. FlowDroid reported 1,022 apps to contain leaks. We chose these 1,022 leaky apps as our final dataset and analyzed them with COVA, setting a timeout of 30 minutes per app. COVA terminated its analysis and computed a complete constraint map for 315 apps. 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 Java virtual machine was set to 24GB.

[2]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.

TABLE I
Top source-sink-pairs among reported leaks by FlowDroid

| Source | Sink | #Leaks |
|---|---|---|
| **Intra-procedural Leaks** | | |
| java.net.URL. openConnection | java.net.HttpURLConnection. setRequestProperty | 2193 |
| android.os.Handler. obtainMessage | android.os.Handler. sendMessage | 1410 |
| java.net.HttpURLConnection. getOutputStream | java.io.OutputStream. write | 194 |
| **Inter-procedural Leaks** | | |
| android.database.Cursor. getString | android.app.Activity. startActivityForResult | 1440 |
| java.net.URL. openConnection | java.net.HttpURLConnection. setRequestProperty | 862 |
| android.database.Cursor. getString | android.os.Bundle. putString | 847 |

```
/*** 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. Extracted code patterns using top 3 source-sink-pairs appeared in intra-procedural leaks

### C. Evaluation Results

**RQ1. Are the leaks reported by FlowDroid in the default configuration reasonable?**

With this research question we want to conduct a sanity check to rule out obvious false results of FlowDroid in the default configuration. FlowDroid reported 28,176 leaks for 1,022 apps, which makes it impossible for us to check every single leak in every app. To address this research question nonetheless, we measured which *source-sink-pairs* appeared in the leaks and manually inspected the top 3 source-sink-pairs among intra- and inter-procedural leaks, since these source-

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*] |

sink-pairs dominate a large amount of leaks and among most (88%) of the remaining pairs, each pair only appeared in fewer than 50 leaks.

Table I shows the top 3 source-sink-pairs among intra-procedural leaks and inter-procedural leaks. The source-sink-pair (*URL.openConnection*, *HttpURLConnection.setRequestProperty*) appeared most frequently in both intra- and inter-procedural leaks. Nearly 40% of intra-procedural leaks share this source-sink-pair. While the given source method *URL.openConnection* creates a connection object with a given URL, the sink *HttpURLConnection.setRequestProperty* sets the general properties of a HTTP request. This source-sink-pair combination seems to be *unreasonable* to consider 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 [42]. This result motivated us to take a closer look into the reported leaks with this source-sink-pair.

We sampled a few intra-procedural leaks with the above-mentioned source-sink-pair and decompiled the corresponding apps. Interestingly, we discovered that the reported leaks 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. 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 is a similar case: since *HttpURLConnection.getOutputStream* returns a stream to write data to the connection it does not make sense to consider it as a source. Such cases must be filtered out.

In the end, 2,630 reported leaks matched these three code patterns, i.e., 46% of all intra-procedural leaks. This big fraction cannot be ignored. Thus, we examined all 47 sources by reading the Javadoc carefully together with experienced developers. Altogether, we identified 11 inappropriate sources and 1 inappropriate sink listed in Table II. These inappropriate sources and sink appeared in 7,767 reported leaks, which is 28% of all reported leaks (intra- and inter-procedural).

```
public class MainActivity extends AppCompatActivity {
    private String secret;
    public void caller(){
        ...
        this.secret = cursor.getString(i); // source
        callee();
    }

    public void callee() {
        this.startActivityForResult(new Intent(), 1); // sink
    }}
```

Listing 2. A false-positive code pattern

> 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 leaks being unreasonable.

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.

> Researchers who used FlowDroid in the default configuration may need to re-evaluate their conclusions. In a short investigation, we already found 14 papers in which the respective work was built on top of FlowDroid and inappropriate sources or sinks were used [18]–[31].

After removing the inappropriate sources and sink in Table II, we sampled two source-sink-pairs among the inter-procedural leaks in Table I. The leaks with the source *Cursor.getString* and the sink *Activity.startActivityForResult* appeared in 156 apps. We randomly chose 15 apps for sampling. 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 leaks using such inter-component communication are outside the scope of FlowDroid, our goal for sampling was only to check if this partial flow is reasonable.

Surprisingly 85% of the leaks with this source-sink-pair in these 15 apps are false positives. All these false positives share a similar code pattern as 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. Such over-approximation does not make sense for the sink *Activity.startActivityForResult*. Generally, leaks with taints connecting sources and sinks on the same objects should be filtered. Thus, we re-analyzed the apps with FlowDroid and Boomerang to detect such cases. 330 leaks matched the false-positive pattern in Listing 2. In total, we identified 978 leaks with taints connecting sources and sinks on the same objects. The sinks appearing in these leaks are mainly APIs used for inter-component communication.

Most leaks reported by FlowDroid with taints connecting sources and sinks on the same objects are false positives.

The leaks with the source-sink-pair (*Cursor.getString*, *Bundle.putString*) appeared in 85 apps. The sink *Bundle.putString* is also an API for inter-component communication. Similar as the previous source-sink-pair, we only check if the reported partial flow is reasonable. We sampled 8 apps randomly and none of the reported flows is a false positive.

Altogether we identified 31% (8,745 out of 28,176) leaks reported by FlowDroid in the default configuration to be unreasonable.

At least 31% of leaks reported by FlowDroid in the default configuration are unreasonable.

**Note:** For the following research questions, we eliminated the unreasonable leaks we identified.

**RQ2. Do "low-hanging fruits" exist, and if so, what characteristics do these leaks have?**

We consider as "low-hanging fruits" leaks that are intra-procedural and unconstrained, since we think it highly likely that these leaks are true positives and actionable to be fixed. To answer this question, we classified the leaks based on the leak-constraints computed by COVA. If the leak-constraint is equal to *true*, then the corresponding leak is unconstrained.

Figure 4 shows the distribution of the constrained and unconstrained intra- and inter-procedural leaks. Only 5% of the leaks are "low-hanging fruits". However, this is still the majority of the intra-procedural leaks.

Furthermore, we studied the sources and sinks appearing in these "low-hanging fruits". Among these leaks, data mainly flow into sinks for logging or inter-component communication such as *Bundle.putString*, *SharedPreferences.putString* and *Context.sendBroadcast*. The most common source is *Cursor.getString*, which returns the answer of a database query. It appeared in 45% of the intra-procedural and unconstrained leaks. The most frequently used sinks are the log methods from *android.util.Log*. 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. We randomly sampled 20 leaks with logging methods being a sink—all of these leaks are true positives. Many of these leaks even have source and sink at the same line. In addition, the text that will be logged often specifies what kind of data is being logged. Such leaks can be easily fixed, which is why taint-analysis tools should prioritize these leaks in the report.

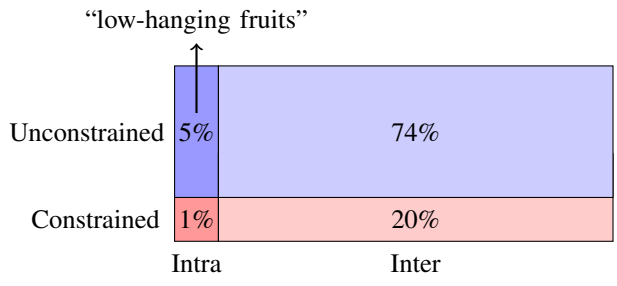


Fig. 4. Contrained and unconstrained intra- and inter-procedural leaks

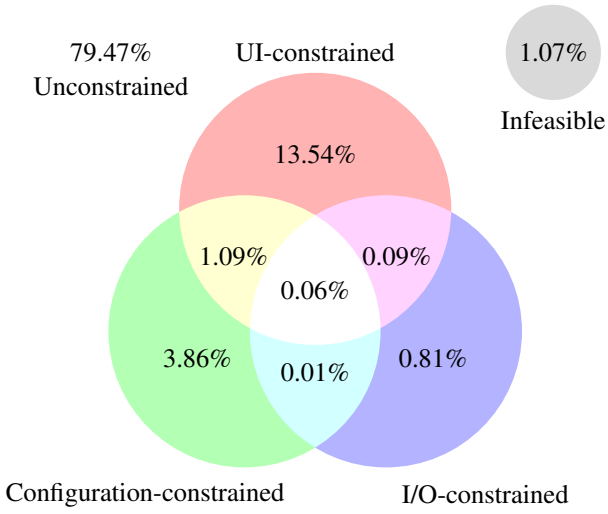


Fig. 5. Different types of leaks

5% of leaks reported by FlowDroid are "low-hanging fruits", which are actionable for developers to fix.

**RQ3. What types of leaks does FlowDroid report?**

In this research question, we want to identify the fraction of leaks that are dependent on the constraint-APIs from the three categories we defined in subsection V-A, i.e., *UI-constrained*, *Configuration-constrained* and *I/O-constrained*. We classify the leaks based on the leak-constraints computed by COVA. For instance, if the leak-constraint contains symbolic values that relate to the constraint-APIs from UI callbacks, then this leak belongs to category "UI-constrained". Certainly, there can be leaks which belong to multiple categories.

The Venn diagram in Figure 5 shows the classification of the leaks. In our experiments, 79.47% of the leaks are not conditionalized by UI actions, nor environment configurations or inputs from I/O operations. However, this is only an upper bound of the unconstrained leaks, since COVA timed out for about two-thirds of the apps in our experimentation and our list of constraint-APIs does not represent the full space of APIs from these three categories. For apps on which COVA timed out, if there is no constraint for the source and sink statements of a leak in the partial constraint map, we assigned this leak with the type "Unconstrained". Further, COVA was able to identify 1.07% (208) leaks that are actually infeasible, since the leak-constraints reported by COVA are unsatisfiable.

Among the 20.53% leaks whose occurrences are dependent on the constraint-APIs in the categories, the majority (13.54%) are in a single category – UI-constrained, which means they only occur when some specific UI actions are performed. 3.86% of the leaks may happen under certain environment configurations, and 0.81% are dependent on inputs from I/O

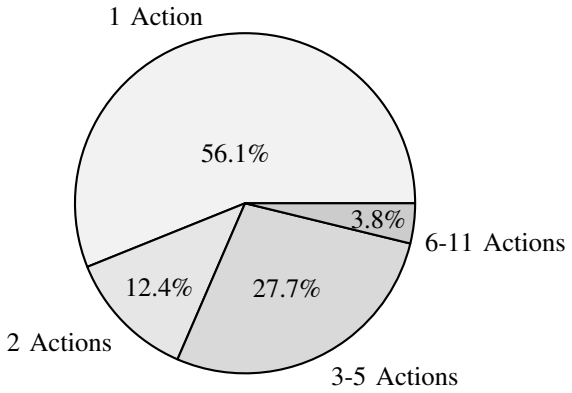


Fig. 6. The complexity of UI-constrained leaks

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.

Certain UI actions are required for triggering the execution of 14.78% leaks.

Moreover, since many leaks only depend on UI actions, we investigated the complexity of the UI actions. Intuitively, leaks triggered by a sequence of user actions should exist. A previous study [7] 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 leak 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.) As shown in Figure 6, 56.1% of leaks happen after a single UI action. There are only 3.8% of leaks that may require 6 or more UI actions. Maximally 11 different actions appeared in a leak-constraint. However, executing the leak 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 leak).

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

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

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

In addition, we evaluated which constraint-APIs from our categories are most frequently used. While click events are relevant to most leaks related to UI actions, the Android SDK version plays a considerable role in environment configurations (see Table III). This is not surprising to us, since the Android operating system remains highly fragmented [43]–[45] and developers are challenged to produce applications that are compatible to multiple platform versions. Our result also confirms the earlier finding of Lillack et al. [46] that SDK and network configuration are the most commonly used environment values in Android apps. But also access to the system location service is often demanded in the context of data leaks. 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)`.

TABLE III
TOP CONSTRAINT-APIS RELATED TO LEAKS

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

Additionally, we observed that for 28% of the 208 infeasible leaks, their source statements were not executable, since the path constraint is *FALSE*. For 76% of the infeasible leaks, their sink statements will never be executed at runtime. Such dead code was probably intentionally built in by developers [47]. During samping, we inspected code used for logging (sinks of leaks) that was disabled with a boolean flag for the released APK, but not removed.

*Discussion:* Our experimental results show that hybrid analysis tools may well be feasible for the case of Android. To confirm static taint flows dynamically, they should focus on modelling 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.

## VI. LIMITATIONS

COVA computes partial path constraints—it only considers control-flow decisions that are dependent on a list of constraint-APIs we collected. If there are other conditions between source and sink that affect the taint-flow, then COVA is not able to identify this. Although COVA supports most language features, some corner cases such as reflection or native calls are not covered [48]. In some cases, the taint set computed by COVA may be incomplete due to unknown return values of method calls such that an over-approximated constraint is computed. For Android applications, we use the callgraph constructed by FlowDroid. This callgraph, however, is partially incomplete for library methods and some UI callbacks [49], [50]—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 reason 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 are from the native code of Z3. In future, we plan to make COVA on-demand such that it only computes a constraint for a given statement instead of computing a constraint map for all reachable statements.

For the qualitative study, although we sanity-checked the leaks reported by FlowDroid in RQ1 (subsection V-C), we do not claim to catch all possible false positives. However, the focus of our research is not identifying false positives, but the types of taint-flows which are conditioned on different factors.

Furthermore, our classification strongly depends on the leaks which FlowDroid reports and the constraint-APIs we collected for the three categories. The results of our study may not generalize beyond that setting.

## VII. Related Work

### A. Path Conditions

Many approaches have considered path conditions to increase the accuracy of their analysis. Snelting [32] has shown how exacting and simplifying path conditions can improve slice accuracy. Taghdiri et al. [51] made information flow analysis more precise by incrementally refining path conditions with witnesses that did not yield an information flow in execution. TASMAN [52] leverages backward symbolic execution as a post-analysis to eliminate false positives in which taint-flows along paths are infeasible at runtime. A major limitation of symbolic execution is that it can not explore executions with path conditions which the underlying SMT solver can not deal with in the given time budget [53], which is also the limitation of our approach. To improve the scalability, modern symbolic execution techniques mix concrete and symbolic execution, i.e., the so-called concolic execution. Anand et al. [54] proposed a concolic execution approach to generate sequences of UI events for Android applications. Schütte et al. [55] also used concolic execution to drive execution to target code. Their approach considers code fragments whose execution may depend not only on user inputs but also on conditions such as execution environment and even communication with remote sites. Based on the results of our study, we also believe considering environmental factors in the analysis can be helpful to disclose more security vulnerabilities.

### B. Data Mining Android Apps

In the past, many researchers have data mined Android applications [13], [22]–[24], [46], [56], [57]. Avdiienko et al. [23] compared the taint-flows in benign apps against malicious apps and used the differences to identify abnormal usage of sensitive data with machine learning. Opposed to COVA, their approach MUDFLOW does not consider path constraints. Keng et al. [13] monitored 220 Android apps with the dynamic taint-analysis tool TaintDroid [58] 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 is also confirmed by our study. Closely related to our approach, Lillack et al. [46] also extended taint analysis to explore the variability of Android apps based on load-time configuration. However, their approach encoded constraint analysis as a distributive problem in the IFDS framework [59]. For our purpose, this is too weak a model, since the execution of a branch may depend simultaneously on two or more configuration options, which IFDS cannot express [33], [60].

### C. Hybrid Analysis

A number of hybrid approaches have been proposed for Android malware detection [7]–[12]. SmartDroid presented by Zheng et al. [7] detects UI interactions paths towards sensitive APIs statically and exposes the malicious behaviors dynamically. Yang et al. [8] proposed 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. [11] demonstrated a tool which generates a reasonably small set of inputs statically to trigger malicious behavior of applications. Their evaluation shows that one only requires to execute a very small part of the application to expose malicious behaviors. Recent work of Rasthofer et al. [10] combined 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. By applying COVA to 1,022 Android apps, we conducted a qualitative study on the leaks reported by FlowDroid. In particular, our study was designed with the intention to gather information about how UI actions, environment configurations and I/O operations are relevant to these leaks. We first evaluated if the leaks reported by FlowDroid are reasonable by studying the top appeared source-sink-pairs. We identified 11 bogus sources and sinks of FlowDroid and 31% false positives. After eliminating these false positives, we classified the leaks based on the constraints computed by COVA. We were able to confirm the fact that many leaks happen under certain environmental conditions, especially the UI actions and configuration. This shows the importance to invest more effort in studying UI-based execution scenarios in Android apps. Our future work will focus on using our results to enhance taint analysis tools and use COVA to assist targeted execution.

## 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. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[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. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660357

[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. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[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. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771803

[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] 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

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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

[13] 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

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

[15] 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

[16] 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

[17] 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.

[18] 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.

[19] 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

[20] 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. [Online]. Available: https://doi.org/10.1109/ISCC.2017.8024568

[21] 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

[22] 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. [Online]. Available: https://doi.org/10.1109/ICSE.2015.50

[23] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818808

[24] 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

[25] 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. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213873

[26] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications," in *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, 2017, pp. 1–9. [Online]. Available: https://doi.org/10.1109/INFOCOM.2017.8057221

[27] 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

[28] 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.

[29] D. Titze and J. Schütte, "Apparecium: Revealing data flows in android applications," in *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, 2015, pp. 579–586. [Online]. Available: https://doi.org/10.1109/AINA.2015.239

[30] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 1036–1046. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568301

[31] 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.

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

[33] 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. [Online]. Available: http://doi.acm.org/10.1145/512927.512945

[34] 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. [Online]. Available: http://doi.acm.org/10.1145/2487568.2487569

[35] 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. [Online]. Available: http://doi.acm.org/10.1145/178243.178263

[36] 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

[37] 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

[38] 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. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[39] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises? - a reproducibility study," in *ESEC/FSE '18: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2018, to appear. [Online]. Available: http://www.bodden.de/pubs/pbw18do.pdf

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

[41] 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. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[42] 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

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

[44] 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

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

[46] 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/

[47] 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

[48] 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

[49] 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

[50] 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. [Online]. Available: http://doi.acm.org/10.1145/2931021.2931026

[51] 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. [Online]. Available: https://doi.org/10.1007/978-3-642-19751-2_5

[52] 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. [Online]. Available: http://doi.acm.org/10.1145/2771284.2771285

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

[54] 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. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[55] 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. [Online]. Available: https://doi.org/10.1109/AINA.2015.238

[56] 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. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624000

[57] 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. [Online]. Available: https://doi.org/10.1109/ICC.2017.7996335

[58] 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.

[59] 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.

[60] 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.