Algebraic-datatype Taint Tracking, with Applications to Understanding Android Identifier Leaks

Anonymous Author(s)

ABSTRACT

Current taint analyses track flow from sources to sinks, and report the results simply as source $\longrightarrow$ sink pairs, or flows. This is imprecise and ineffective in many real-world scenarios; examples include taint sources that are mutually exclusive, or flows that combine sources (e.g., IMEI and MAC Address are concatenated, hashed, leaked vs. IMEI and MAC Address hashed separately and leaked separately). These shortcomings are particularly acute in the context of Android, where sensitive identifiers can be combined, processed, and then leaked, in complicated ways. To address these issues, we introduce a novel, *algebraic-datatype* taint analysis that generates rich yet concise taint signatures involving AND, XOR, hashing – akin to algebraic, product and sum, types. We implemented our approach as a static analysis for Android that derives app leak signatures – an algebraic representation of how, and where, hardware/software identifiers are manipulated before being exfiltrated to the network. We perform six empirical studies of algebraic-datatype taint tracking on 1,000 top apps from Google Play and their embedded libraries, including: discerning between "raw" and hashed flows which eliminates a source of imprecision in current analyses; finding apps and libraries that go against Google Play's guidelines by (ab)using hardware identifiers; showing that third-party code, rather than app code, is the predominant source of leaks; exposing potential de-anonymization practices; and quantifying how apps have become more privacy-friendly over the past two years.

1 INTRODUCTION

Current taint analyses (for Android, this includes FlowDroid [35], TaintDroid [41], Amandroid [56]) track whether data from a privacy-sensitive *source* flows to an insecure *sink*. The analysis result for a sink (e.g., a leak to the network) is a list of sources (optionally, flows) that reach that sink. First, this is ineffective: the same leak report can be too alarmist, or not alarmist enough, depending on the context. Second, this is inadequate as it does not capture the relationship between sources, e.g., mutually exclusive sources; neither does it capture source processing or the severity of the leak, e.g., hashing can be applied to each source in turn or hashing can be applied to a concatenation of sources. Finally, rich, convoluted "leak signatures" are ubiquitous in Android apps, but cannot be exposed by existing Android taint trackers. Our approach produces expressive leak signatures; rather than simply enumerating sources, e.g.,

$$IMEI, MAC\ Address, AndroidID$$

as in current taint trackers, our approach uses formulas indicating relations between, and operations on, sources, e.g.,

$$IMEI \oplus (hash(AndroidID) \wedge hash(MAC\ Adresss))$$

Consider for instance the com.appsee mobile analytics library, which leaks the hashed phone identifier IMEI onto the network vs. the io.fabric library which leaks the raw IMEI onto the network. While a conventional taint analysis would consider these leaks as equivalent, they are not: the raw IMEI leak is dangerous, as it allows the server to identify the manufacturer and phone model; the hashed leak is less dangerous since the phone information cannot be identified (at least not directly). Our analysis distinguishes these two flows, encoding the former as e,[1] and the latter as $h(e)$, capturing the difference. In fact, our study on top Android apps shows that 23% of identifier leaks are hashed leaks, not raw leaks.

As another example, consider the Likes + Followers Instagram app that leaks the hashed IMEI and AndroidID (signature: $h(e) \wedge h(a)$), which is different from app JCPenney that leaks the raw IMEI and AndroidID (signature: $e \wedge a$), which is different from app AARP Now that leaks either the IMEI or the AndroidID, but not both (signature: $e \oplus a$); and finally app Moon Calendar with signature $h(e \wedge a)$, note the crucial difference from $h(e) \wedge h(a)$, especially relevant in the context of homomorphic encryption. Current flow analyses would conflate all these cases declaring "IMEI and AndroidID are leaked". Our study (Section 6.2) quantifies this conflation on 1,000 top Google Play apps: a conventional analysis would conflate 100 apps, whose signature is $h(a) \wedge g$, with 70 apps whose signature is $a \wedge g$, simply declaring that all 170 apps leak the AndroidID and the GUID.

Our taint analysis produces a leak signature that corresponds to an algebraic, finite (non-recursive) datatype: our logical AND ('$\wedge$') and XOR ('$\oplus$') connectives encode product and sum types, respectively. This algebraic-datatype encoding allows us to define *subsumption* – what it means for an app to leak less than another app – rigorously, via subset semantics.

In Android apps, leak patterns such as: "leak [a hardware ID] if available, otherwise, leak [a software ID]" abound; we capture this via XOR. Consider the CBS News app which leaks either the IMEI, or the AndroidID, but not both. Traditional analyses would report this app's signature as $e \wedge a$, i.e., both identifiers are leaked, which is imprecise. Our analysis constructs the precise signature $e \oplus a$. More generally, using our approach, developers can check properties such as: "the app only leaks the IMEI on a real device, otherwise it leaks the AndroidID". Precisely capturing taint as logically-connected raw or hashed identifiers is crucial: our study shows that two-thirds of leaks are due to leaking identifier combinations (65% of flows leak combinations of IDs, and only 35% of flows leak individual IDs). Our algebraic representation is defined in Section 3.

Section 4 presents the design of our static analysis. We use a novel dataflow-centric call graph to produce a sound over-approximation of app flows, followed by two refinement phases – separating raw leaks from hashed leaks, and app's own flows from third-party flows – finally yielding an algebraic leak representation aka signature.

In Section 5 we evaluate our approach. The main dataset consisted of 1,000 top apps from Google Play, chosen from 19 categories, and the 821 libraries these apps embed. We found that the approach is effective. First, when compared to the state-of-the-art FlowDroid

[1]For conciseness, we use shorthands for Android unique identifiers: e for the IMEI, a for the AndroidID, g for GUID, etc; Section 3.1 defines the full list.

static analyzer, our approach discovers 2.1x as many leaks, while shedding light on the nature and location of the leak. Second, when compared with ground truth, our approach attains 72.6% precision and 100% recall. The approach is also efficient, with a median analysis time per app of 347 seconds.

In Section 6, we conduct a series of empirical studies to examine (a) the landscape and evolution of the 1,000 top Google Play apps in terms of privacy leaks involving non-resettable "hardware" and resettable "software" IDs, and (b) their embedded libraries, with a focus on top-20 libraries (financial, advertising, analytics).

An analysis of worst-case leaks, i.e., hardware identifiers sent raw, has found that 57% of the apps leak at least one of the raw hardware IDs (IMEI, IMSI, Serial#, MAC Address), which is at odds with Google's ID usage guidelines. In fact, 7 out of the top-10 most common signatures involve hardware IDs.

We found that out of top-25 leakiest apps, 13 apps leak hardware IDs with no financial justification; among top-20 leakiest libraries, only 3 are financial (where Google guidelines permit hardware IDs).

Certain identifiers, e.g., IMEI, tend to be leaked by libraries, whereas others, e.g., MAC Address, by own code. Overall, 59% of leaks are due to third-party libraries and 41% leaks are due to apps' own code. This has important implications because leaks due to third-party code can get an app removed from Google Play, even though the developer (app's own code) has followed the guidelines.

Finally, by comparing apps' versions from 2018 with their 2020 counterparts we found a decrease in the use of raw/hardware identifiers, indicating that apps have become more privacy-friendly.

Our work makes several contributions:

- A novel, algebraic-datatype taint representation, enabling expressive yet concise leak signatures and leak analysis.
- An implementation of algebraic taint tracking as a static analysis for Android.
- A six-part empirical study, characterizing identifier (ab)use in top Android apps and libraries. The study demonstrates the flexibility and precision of our algebraic representation.

2 MOTIVATION AND DESIGN CHOICES

Taint analyses determine whether data from a privacy-sensitive source (e.g., MAC Address) flows to an insecure sink (e.g., Internet). Current analyses' imprecision affects their usability. For example, applying a standard Android taint analysis produces the same result (IMEI⟶Internet), in three different scenarios:

(1) the app exfiltrates the IMEI to a third-party server (e.g., Blink Health RX [9]); this practice is discouraged or forbidden by Google Play guidelines, depending on the nature of the app.
(2) the app (e.g., CareZone [1]) links with a financial library that uses the IMEI for payment fraud prevention; this is allowed by the guidelines.
(3) the app (e.g., Spectrum TV [29]) concatenates the IMEI with another identifier, e.g., AndroidID, hashes the result, and uses this hash value for customer identification. Since the actual IMEI cannot be reverse-engineered, the privacy loss is lower compared to the first and second scenarios.

Conflating these three use cases is problematic, as they are very different in terms of guidelines compliance and privacy implications.

Table 1: Identifiers considered and their semantics.

	ID	Semantics
Hardware	IMEI/MEID	15-digit; identifies mobile phone
	IMSI	15-digit; identifies SIM/network subscriber
	MAC Address	48-bit; identifies the network card -actual value reported by Android <6 -"02:00:00:00:00:00" reported by Android 6−9 -randomized value reported by Android ≥10
	Serial#	Manufacturer-assigned; identifies device
Resettable	AndroidID	Android ≥8: unique for an app or app group Android <8: unique user&device combination
	GUID	identifies app instance
	AdvertisingID	identifies user for ad tracking purposes

We combine a precise static analysis with an algebraic representation that can distinguish between these, and other, scenarios.

Android identifiers. Our analysis considers the seven popular Android IDs described in Table 1. The first four are "hardware" identifiers, i.e., tied to the specific phone hardware, and cannot be reset/changed in software; the remaining three are resettable identifiers. Google/Android developer guidelines have specific policies designed to protect user privacy [7] by discouraging, or even forbidding, access to hardware identifiers, such as:

- "Avoid using hardware identifiers"
- "Only use an Advertising ID for user profiling or ads"
- "Use an Instance ID or GUID whenever possible for all other use cases, except for payment fraud prevention and telephony"
- "By its nature, fraud prevention requires proprietary signals"

To sum up, the only acceptable use of hardware identifiers is financial/fraud detection; all other scenarios, e.g., advertising or analytics, require the use of resettable identifiers. Many apps violate these guidelines; to counter this abuse, as shown in Table 1's "MAC Address" line, the Android platform's recent versions took increasingly stringent measures, first reporting a constant MAC Address, and then a randomized one. Android version 10 (used by 8.2% of Android devices as of February 25th, 2021, per Android Studio) restricts access to hardware identifiers to privileged (e.g., system, vendor) apps; this does not affect the generality of our approach.

3 ALGEBRAIC-DATATYPE REPRESENTATION FOR SIGNATURES

In type theory, a product type is the type of an n-ary tuple, e.g., in OCaml the tuple (1,3.14,"foo") has type int * float * string. A sum type is the type of a union, e.g., in C, union u {int i; float f;} or the OCaml [48] variant type number = Int of int | Float of float have product type int ⊕ float (either an int or a float, but not both, inhabit the variant). Our core insight is an *algebraic-datatype* definition of taint: identifiers are base types and leak signatures are finite (non-recursive) algebraic types over base types.

3.1 Definitions

We define these shorthands for the Android identifiers: e for the IMEI, s for the IMSI, a for the AndroidID, r for Serial, m for MAC Address, v for AdvertisingID, g for GUID.

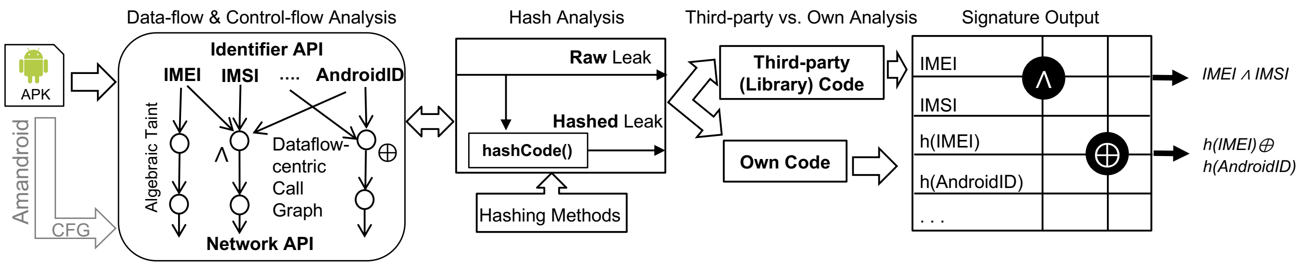


Figure 1: Overview.

Signatures can be identifiers; hashes; or combinations thereof introduced via AND or XOR. We use S and T as metavariables for signatures. Hence our signature grammar is defined simply as:

$$\boxed{\begin{array}{lll} \mathit{Identifier} \quad i & ::= & e \mid s \mid a \mid r \mid m \mid v \mid g \\ \mathit{Signature} \quad S & ::= & i \mid h(S) \mid S \oplus S \mid S \wedge S \end{array}}$$

AND, denoted $S \wedge T$, indicates that *both* S and T (which can be identifiers or signatures), are used. This corresponds to a product type in type theory, and Cartesian product in set theory. Note that we deliberately use '$\wedge$' instead of the standard type theory symbol '$\times$' as it is more suggestive in our context.

XOR, denoted $S \oplus T$, indicates that either S or T is used, but not both. This corresponds to a sum type in type theory,[2] and disjoint set union in set theory. We use '$\oplus$' instead of the standard '+' from type theory as it is more suggestive in our context.

Hash. The hash, denoted $h(S)$, indicates that an identifier (or identifier combination) is leaked, not the actual "raw" identifier(s); e.g., $h(e)$ could be computed via `IMEI.hashCode()` or `md.digest(IMEI.getBytes())`.

3.2 Properties

Defining formally what it means for a program to *leak less* than another program is key. For this purpose we introduce the *subsumption* relation, '<:', induced by subset semantics. Informally, app A leaks less than app B, aka B subsumes A, if the set of all possible values leaked by A is a subset of the set of all possible values leaked by B. We now define subsumption for the algebraic representation.

Subsumption (AND). An app whose signature is S leaks less than an app whose signature is $S \wedge T$; this is denoted $S <: S \wedge T$. Similarly, an app whose signature is T leaks less than an app whose signature is $S \wedge T$; this is denoted $T <: S \wedge T$.

Subsumption (XOR). An app with signature $S \oplus T$ leaks less than an app whose signature is $S \wedge T$.

Subsumption (hash). An app with signature $h(S)$ leaks less[3] than an app with signature S; this is denoted $h(S) <: S$.

First, note how subsumption introduces a partial order (in certain cases, a total order) on apps' leaking properties: its power becomes apparent in Section 6 when we use it to check whether a signature subsumes another (i.e., an app leaks more than another app, or more than a different version of the same app). Second, the algebraic representation naturally induces *equivalence classes*: apps with the same signature will leak the same identifiers (and semantically, the identifiers are manipulated in the same way, e.g., hashed).

[2]Technically, in the Curry-Howard isomorphism [44], sum types correspond to OR in logic, not to XOR. However OR is not our intended semantics, since $a = TRUE$ in $a \vee b = TRUE$ does not force b to be $FALSE$ whereas in our semantics it does (mutual exclusion); a longer explanation is available here [50]. Our semantics is readily apparent in the Church Boolean [39] function XOR, i.e., $\lambda a.\lambda b.a\ (not\ b)\ b$, where if a reduces to TRUE, $(not\ b)$ must reduce to FALSE for the XOR to reduce to TRUE.
[3]"Leaks less" in a privacy/cryptographic sense, rather than strictly $h(S) \subseteq S$.

```
String getTelephonyDeviceId(Context context) {
    String deviceIMEI = ((TelephonyManager)
        context.getSystemService("phone")).getDeviceId();
    return deviceIMEI; }
String getAndroidId(Context context) {
    String androidId = Secure.getString(context.getContentResolver(),
        "android_id");
    return androidId; }
String getWifiMacAddress(Context context) {...
    String mac=wifiManager.getConnectionInfo().getMacAddress();
    return mac; }
String getUniqueDeviceID(Context context) {
    return generateDeviceId(getTelephonyDeviceId(context),
        getWifiMacAddress(context), getAndroidId(context));
}
String generateDeviceId(String str, String str2, String str3) {
    if (! TextUtils.isEmpty(str)) { // str3 → {a}
        str3=str; // str3 → {e}
    }
    else if (! TextUtils.isEmpty(str2) && !TextUtils.isEmpty(str3)) {//
        str3 → {a}
        str3 = new UUID((long) str3.hashCode(), (long)
            str2.hashCode()).toString(); // str3 → {h(m) ∧ h(a)}
    } else if ( TextUtils.isEmpty(str3)) {// str3 → {a}
        str3=UUID.randomUUID().toString();// str3 → {g}
    }
    return str3; // str3 → {e ⊕ a ⊕ h(m) ∧ h(a) ⊕ g }
void SendDeviceinfo() {...
    httpParaMap.put("deviceID", getUniqueDeviceID(context).toString());
   ...}
```

Figure 2: UUID generation in the Audiobooks.com app.

4 APPROACH

The architecture of our system is shown in Figure 1. Given an Android app (APK file), we perform a chain of analyses to construct the app's leak signature: control- and data-flow analyses that compute and propagate algebraic taint; a secondary analysis to detect hashing; and a third-party vs. own analysis. The initial control-flow graph is produced by the Amandroid static analyzer [56] (shown in gray; not a contribution of this work).

4.1 Motivating Example

We illustrate our approach, and contrast it with prior taint analyses, on the Audiobooks.com [11] app. The relevant source code is shown

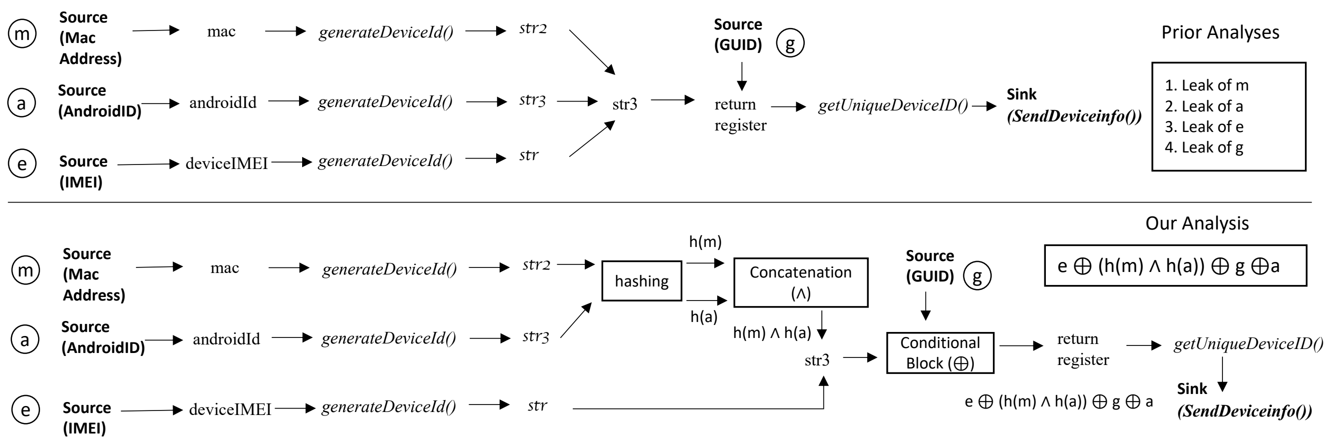


Figure 3: Prior taint approaches (top) vs. our approach (bottom).

in Figure 2. The app attempts to leak an unique device ID, aka UUID onto the network via SendDeviceInfo() (lines 23–25). The UUID is: the IMEI, if available (retrieved on lines 1–3); if not available, the hashes of MAC Address and AndroidID if they are available (lines 4–12); otherwise the GUID, if available (line 20); finally, if none of these conditions are met, AndroidID is the UUID.

Traditional taint analyzers, e.g., FlowDroid or Amandroid, perform taint analysis of each source separately and report the leaks separately. For the aforementioned code snippet, such tools perform four tainted paths calculations for four different sources (IMEI, MAC Address, AndroidID, and GUID), as illustrated in Figure 3 (top). Eventually they produce a report stating that all four identifiers are leaked. This, however, is imprecise for two reasons. First, the identifiers' *hashed* values are leaked, which is less dangerous than *raw* leaks. Second, the tools fail to report the aggregation: actually MAC address and AndroidID are used *together*, exclusive of IMEI and GUID – i.e., a signature, whereas the tools report separate leaks.

In contrast, our analysis produces the correct signature. The high-level view is shown in Figure 3 (bottom); lower-level dataflow analysis will be discussed shortly. Instead of simple taint propagation, we propagate algebraic taint. For the example shown in the figure, instead of four different leaks, we report one precise signature, the leak actually present here, that is:

$$e \oplus (h(m) \wedge h(a)) \oplus g \oplus a$$

4.2 Dataflow-centric Call Graph Construction and Analysis

While other static taint analyzers [35, 56] perform interprocedural control- and data-flow analyses (as we do), their taint facts and propagation are both imprecise and insufficient for our purposes. We address this via a series of analyses, the first of which is *call graph extraction*, as explained next.

4.2.1 Call Graph Extraction. We start our analysis from the Amandroid-generated control-flow graph, and soundly extract the sub-graph where the algebraic taint-relevant data propagates. We illustrate our approach in Figure 4: the top shows the source code while the bottom shows our dataflow-centric call graph (the grey edges/vertices depict the parts of the control flow graph that can be soundly

```
1   public class PersistentUUID {
2   JSONObject jsonObject = new JSONObject();
3   private static final String UUID_KEY = "nr_uuid";
4    ...
5       private void generateUniqueID(Context context)  {...
6         TelephonyManager tm = (TelephonyManager)
                 context.getSystemService("phone");
7         hardwareDeviceId = tm.getDeviceId () ;
8         putUUID(hardwareDeviceId);...
9       }
10      protected void putUUID(String uuid)  {...
11        jsonObject.put(UUID_KEY, uuid);   ...
12      }
13      public String getPersistentUUID()  {...
14        uuid = jsonObject. getString (UUID_KEY);...
15        return uuid;
16      }
17  }
18  public class AndroidAgentImpl {
19      public void sendDeviceInformation()  {...
20        hashMap.put("Model",Build.MODEL);
21        hashMap.put("deviceID",persistentUUID.getPersistentUUID() ) ;...
```

Dataflow-centric Call Graph

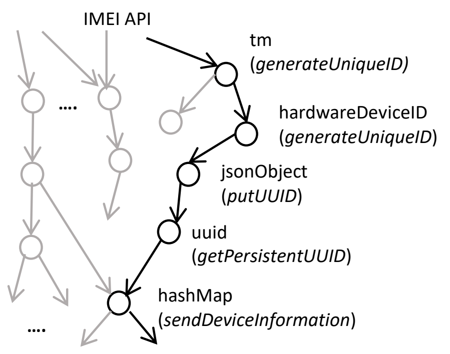


Figure 4: Source code and its dataflow-centric call graph.

abstracted away). In the source code, the IMEI is obtained via the Android API on lines 6 and 7, and stored in hardwareDeviceId. On line 8, the IMEI flows into the putUUID() method as a parameter; the IMEI is saved into a JSONObject on line 11. Our graph captures this dataflow. We can see a dataflow edge between the hardwareDeviceID and jsonObject variables where the IMEI data is saved as key-value JSON data. Next, if we follow the dataflow of the jsonObject currently holding the IMEI data, the data is saved into a new variable uuid on line 14 in the getPersistentUUID() method. This is captured by the edge between jsonObject and uuid. Finally, on line 21, the hashMap creates an entry (key-value pair) with deviceID as key and IMEI as value, resulting in an edge from uuid to hashMap in the graph. Next, our dataflow analysis (forward/may – a variant of reaching definitions [34]) propagates algebraic taint on top of the graph.

4.2.2 Dataflow Analysis. We illustrate our dataflow analysis on the program in Figure 2, method generateDeviceID. A simplified-for-legibility version of the $Out(s)$ sets limited to variable str3 are shown as comments on the right side of the code. At the beginning the set contains the value {a}, i.e., AndroidID. At line 15, str3 is assigned a new value which contains the IMEI, {e}. The $Out(s)$ set for the statement at line 18 contains the combined hash value signature, (h(m)∧ h(a)); str3 is then assigned {g}, GUID, at line 20. For each statement, the union of the $Out(s')$ of all the predecessors s' of s gives the $In(s)$ value [34, 46]. In our example, line 22's predecessor set is lines {14,15,18,20} (of course, in the actual analysis, conditional statements are more fine-grained hence we typically performs two-way joins rather than a four-way join). Hence at program joint point (line 22), the *In* set, in this case identical to the *Out* set, which represents the UUID signature, is:

$$e \oplus (h(m) \wedge h(a)) \oplus g \oplus a$$

A key factor that informed our analysis design, and helps keep the analysis precise, was our observation (drawn from manual taint analysis, Section 5.1.2) that apps' code for constructing the signature, such as the code discussed above, tends to lack back edges, which helps contain dataflow sets size.

4.3 Hash Analysis

As illustrated in Figure 1, hashed leaks are leaks that flow through hashing methods, e.g., hashCode(). We detect such flows by setting up another flow analysis as follows. First, we set the entry of hash methods as *sinks*. Next, we set the return of hash methods as *sources* and network API methods as sinks. As a result, we separate the underlying identifiers leaks into *raw leaks* and *hashed leaks*. Note that our analysis takes a "Hashing Methods" list as input; we constructed this list based on an exhaustive analysis of hashing functions/practices available in Java and practices used by manually-analyzed Android apps. For example, some common hashing Java API methods include hashCode() and nameUUIDFromBytes(), or Java classes such as MessageDigest. This list is user-configurable hence easily extended.

4.4 Third-party vs. Own Code Analysis

To separate own leaks from third-party leaks we used a predefined list of common third-party libraries as reference,[4] along with flow partitioning. Specifically, if an identifier's entire flow involves only third-party library methods, we tag that leak as *third-party* leak; otherwise we tag it as *own* code leak. We have not found any cross-flow between third-party code and own code in our examined apps.

[4]The same library can appear under different names in different apps due to obfuscation; we mapped obfuscated libraries' names to a unique name, common across all apps, for that library.

5 EVALUATION

We evaluated our approach, and performed six studies, on 1,000 top apps from Google Play. The 1,000 apps span 19 popular categories from Google Play. The distribution – number of apps per category – is shown in Table 2. The number of apps varied slightly across categories as we favored popular apps. For the evolution study only (Section 6.6), we compared apps' year 2018 versions with their year 2020 counterparts, i.e., 2,000 APKs.

5.1 Effectiveness

We first evaluate the effectiveness of our approach by comparing, on the 1,000 apps, with state-of-the-art FlowDroid; next, we compare with ground truth on 64 apps where flows were tracked manually.

5.1.1 Comparison with FlowDroid. We ran the July 2020 version of FlowDroid from its official GitHub page [2] on our 1,000-app dataset. We configured FlowDroid to match our configuration: we enabled implicit flow analysis and context sensitivity. Dataflow analysis, callback collection during call graph construction, and result collection time limits were set to 1000 seconds, 1000 seconds, and 500 seconds respectively. As a point of reference, our analysis' median time per app was 347 seconds (Section 5.2), so we believe the aforementioned time limits are reasonable. We directed FlowDroid to use sources and sinks that match ours. As sources, we used the API methods responsible for retrieving the 7 identifiers we track (Table 1). For sinks, we used the SuSi list [3], i.e., all possible sinks under NETWORK_INFORMATION category, as we are only interested in exfiltration to the network. Note that the API methods that read the 'Serial' and 'AdvertisingID' cannot be expressed in FlowDroid's taint source format, so we marked those as 'n/s'.

We show the results in Table 3: the number of apps where leaks were found, by FlowDroid and our approach, respectively. We make three observations. First, FlowDroid misses a substantial number of leaks, as it reports 46% of the leaks we report, 1083 vs. 2335 (for those five IDs we could run FlowDroid on); prior work suggests that false negatives' root causes in FlowDroid/SuSi include inter-component communication and imprecise sink/source lists [54]. Second, FlowDroid cannot distinguish between raw and hashed leaks, as our approach does (third and fourth rows show the raw/hashed split). Third, our approach has some false negatives compared to FlowDroid (i.e., we miss leaks that FlowDroid does not miss), as depicted in the last row. We found that false negatives originate in the CFG provided by Amandroid – when Amandroid missed some control-flow edges, our approach missed those edges as well.

5.1.2 Comparison with Ground Truth. We measured the False Positives (FP) and False Negatives (FN) by comparing the results of our static analysis with ground truth – known flows found in prior work via a manual analysis on 64 apps;[5] these "ground truth flows" are not a contribution of this work. The confusion matrix is:

True Positives: 186	**False Positives**: 70
False Negatives: 0	**True Negatives**: 512

These figures, a 72% precision and 100% recall, are par for the course for a static analysis, indicating that our approach is effective.

[5]The manual flow analysis was exhaustive, e.g., went so far as capturing and rewriting network packets.

Table 2: App distribution across categories.

Category	Finance	Books	Education	Ent'ment.	Event	Food	Health	Home	Lifestyle	Maps	Medical	Music	News	Shopping	Social	Sports	Tools	Travel	Weather
#Apps	52	60	55	55	42	60	54	53	53	54	55	53	50	49	50	47	58	54	54

Table 3: Number of apps where FlowDroid, and our approach respectively, found leaks.

	IMEI	**IMSI**	**Serial**	**MAC**	**And.ID**	**AdvID**	**GUID**
FlowDroid	104	23	n/s	101	336	n/s	519
Our Approach							
Total	405	108	316	372	722	455	728
Raw	334	43	235	324	695	455	728
Hashed	145	79	142	83	297	0	0
False Negatives	3	0	n/a	7	0	n/a	11

Table 4: Efficiency results.

Analysis time (seconds)				**Bytecode size (MB)**			
min	max	median	mean	min	max	median	mean
140	47,651	347	411	0.04	103.4	16.6	15

5.2 Efficiency

We conducted the experiments on a MacBook Pro (3.5 GHz dual-core Intel Core i7 with 16GB RAM), running Mac OS X 10.14.6. We show statistics (computed across the entire app dataset) of analysis running time, along with app bytecode size, in Table 4. A typical app took about 6 minutes to analyze – median 347 seconds, geometric mean 411 seconds – which is efficient for a static analysis; the longest analysis time was 13 hours, which we believe can be reduced substantially with more engineering. The app bytecode statistics – median 16.6MB, geometric mean 15MB, maximum 103MB – show that our approach is capable of analyzing large apps.

Table 5: Identifiers stats.

	IMEI	**IMSI**	**Serial**	**MAC**	**And.ID**	**AdvID**	**GUID**
Apps (%)	51	21	40	47	91	58	92
Raw (%)	83	38	75	88	96	100	100
Hashed (%)	37	74	46	23	41	0	0
Raw & Hashed (%)	20	12	21	11	37	0	0

6 APPLICATIONS

We now present six studies that provide evidence for the expressiveness and effectiveness of algebraic-datatype taint tracking.

6.1 What IDs Are Leaked, and in What Form?

We first studied the frequency and nature of identifier leaks. In Table 5 we show the percentage of apps that leak that identifier, and the form of the leak. Three critical identifiers, IMEI/Serial/MAC Address, are leaked by 40–51% of the apps, which is the first reason for concern. The second reason for concern is that identifiers are leaked *raw* by 75–88% of the apps that leak them; 23–46% of apps leak these IDs hashed – in lieu of, or in addition to, the raw leak. On a more positive note, the IMSI is leaked to a lesser extent, only 21% of the apps, and mostly hashed (74%).

For the remaining three, resettable identifiers, we found that the AndroidID and GUID are leaked routinely: by 91% and 92% of the apps, respectively. The Advertising ID is seeing a reduced leak rate (58% of the apps). Raw leaks are the norm for these identifiers: 96–100% of the leaks are in raw form.

We observed that certain apps leak *both* the raw and hashed ID (last row of Table 5). Note that for IMEI, IMSI, Serial, and AndroidID, this figure is quite high, 12–37% of the apps. We believe this practice to be particularly pernicious, because such apps essentially have the $h(ID) \rightarrow ID$ mapping. If these apps communicate the mapping to other apps that only have $h(ID)$, then the raw ID value, unique to the device, *can be de-anonymized.*

6.2 Multiple-identifier Leaks

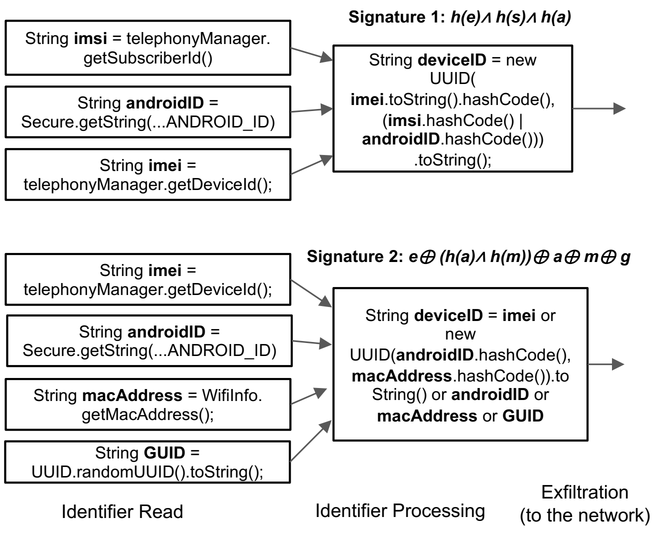
We now study cases where multiple identifiers are leaked by a single app. We present the most frequent signatures in Table 6. On a positive note, 3 out of top-10 most common signatures are $h(a) \wedge g$, $a \wedge g$, and $a \wedge v$, that is, resettable identifiers (10%, 7%, and 3.1%, respectively). The flip side is that the other 7 out of top-10 use hardware identifiers: we have $h(e) \wedge h(r) \wedge h(a)$, then $h(m) \wedge h(a)$, then $e \wedge a$, at 4% and above. We have $h(r) \wedge h(a)$, then $h(e) \wedge h(s)$, then $h(e) \wedge h(m)$, then $h(e) \wedge h(a) \wedge g$, at 2.6% and above.

Note how these findings underline the effectiveness of our approach. A standard taint analysis would conflate the 100 apps whose signature is $h(a) \wedge g$ with the 70 apps whose signature is $a \wedge g$; and would conflate the 24 apps using $h(e) \wedge h(s) \wedge h(r) \wedge h(m) \wedge h(a)$ with the 4 apps using $e \wedge s \wedge r \wedge m \wedge a$.

Examples: complex yet common signatures. Our prior work on manual taint analysis (Section 5.1.2) has revealed groups of apps with common signatures – apps use the same mechanism for constructing a unique "DeviceID". Our analysis can group apps into equivalence classes induced by app signatures; this has a variety of applications, from finding groups of apps with common behavior [43] to groups of apps with common developers, etc. We show two such examples in Figure 5. The left side (first stage) of the figure lists all the identifiers involved in signature construction. The second stage shows how those identifiers are combined or processed to generate a hashed unique DeviceID, which is then exfiltrated.

Table 6: Most common multi-ID leaks; R=raw, H=hashed.

IMEI		IMSI		Serial		MAC		And.ID		ADvID	GUID	#Apps
R	H	R	H	R	H	R	H	R	H	R	R	
									✓		✓	100
								✓			✓	70
	✓				✓				✓		✓	55
							✓		✓			41
✓								✓				40
					✓				✓			33
								✓		✓		31
	✓		✓									28
	✓						✓					27
	✓								✓		✓	26
	✓		✓		✓		✓		✓			24
✓						✓		✓				24
	✓				✓				✓			17
	✓		✓						✓			14
					✓				✓		✓	13
✓											✓	13
✓		✓										11
✓				✓								11
									✓	✓		10


Signature 1: h(e)∧ h(s)∧ h(a)
Signature 2: e⊕ (h(a)∧ h(m))⊕ a⊕ m⊕ g
Identifier Read
Identifier Processing
Exfiltration
(to the network)

```
String imsi = telephonyManager.
getSubscriberId()

String androidID =
Secure.getString(...ANDROID_ID)

String imei =
telephonyManager.getDeviceId();

String deviceID = new
UUID(
imei.toString().hashCode(),
(imsi.hashCode() |
androidID.hashCode()))
.toString();
```

```
String imei =
telephonyManager.getDeviceId();

String androidID =
Secure.getString(...ANDROID_ID)

String macAddress = WifiInfo.
getMacAddress();

String GUID =
UUID.randomUUID().toString();

String deviceID = imei or
new
UUID(androidID.hashCode(),
macAddress.hashCode()).to
String() or androidID or
macAddress or GUID
```

Figure 5: DeviceID signatures.

Signature 1 (app: Texas Roadhouse Mobile [30]) creates a DeviceID from the combination of IMEI, IMSI, and AndroidID. Since all identifiers are used, the signature uses ANDs:

$$h(e) \wedge h(s) \wedge h(a)$$

Signature 2 (library io.intercom) is quite complex, as the DeviceID is exactly one of: either the IMEI, or the AndroidID, or the MAC Address, or the GUID, or the AND of hashed Android ID and hashed MAC Address. Our representation captures this effectively:

$$e \oplus (h(a) \wedge h(m)) \oplus a \oplus m \oplus g$$

Table 7: Third-party vs. Own code statistics: number and percentage of leaks (T=third-party, O=own code).

	IMEI		IMSI		Serial		MAC		And.ID	
	T	O	T	O	T	O	T	O	T	O
R	183	208	28	16	128	120	223	145	586	418
H	97	63	56	25	120	33	61	23	198	144
R(%)	33	38	22	13	32	30	49	32	44	30
H(%)	18	11	45	20	30	8	14	5	15	11

6.3 Library Leaks vs. App's Own Leaks

We motivate this analysis via two scenarios. In the first scenario, a developer submits an app for publishing onto Google Play, and the app is rejected for violating guidelines, e.g., a raw hardware leak in a non-financial app. Even though the developer has used no IDs, the app is linked with a "leaky" advertising library that causes the ID leak. The developer should be able to extract the library's signature and the app's signature to determine the leak's cause and course of action. In the second scenario, the Google Play marketplace itself tries to determine whether a raw hardware leak is allowable or not, prior to publishing an app. If the app uses a payments services library, the leak would be allowed in the name of fraud prevention. Hence it is essential to find whether a leak is caused by a library or the app itself. Our analysis isolates the source of the leak (Figure 1) and attributes it to either *third-party* (library) code or *own* code.

In Table 7 we present the results of leak attribution in our examined apps. For each ID, we show the number of third-party (T) vs. own (O) leaks, whether the leak is raw or hashed, as well as the percentage distribution. All the hardware identifiers – IMEI, IMSI, Serial, and MAC Address – as well as the AndroidID, are leaked more by libraries than own code (51%, 67%, 62%, 63%, 59%, respectively). For identifiers AdvertisingID and GUID (omitted from the table for space), leaks were substantial but balanced: 297 third-party vs. 291 own for AdvertisingID and 639 vs. 631 for GUID.

This finding – hardware ID leaks are attributable more to third-party code than own code – is important, because it shows that apps could *unwittingly* be the source of problematic leaks, e.g., due to "leaky" libraries, and could be unfairly blamed for leaks that apps' own developers did not introduce, or were not even aware of.

6.4 Leakiest Libraries

As mentioned previously, libraries are a significant source of leaks. Summarizing leaks in libraries is non-trivial, however, because of *context-sensitivity*: a leak would materialize (or not) depending on how an app invokes the library. We illustrate this in Figure 6, on library com.threatmetrix. When the library is invoked from the JCPenney app (top), the IMEI is leaked *hashed*: on line 3 the IMEI is read and its hash (digest) added to hashMap. However, when the library is invoked from the Dunkin app (bottom), the IMEI is leaked *raw*: on line 10 the IMEI is read and added, raw, to httpParameterMap.

Therefore, in Table 8 we present the results of our library analysis; of the 821 libraries used in our apps, we show the top-20

Table 8: Third party libraries: the number of methods leaking each ID, and the form of the leak (H=hashed, R=raw). Raw hardware leaks in non-financial libraries shown in red.

Library	IMEI		IMSI		Serial		MAC		AndroidID		AdvID	GUID	Purpose
	H	R	H	R	H	R	H	R	H	R	R	R	
com.paypal	35	8	35	4	35	4	35	6	35	4	30	142	**finance**
io.fabric	0	38	0	0	0	30	0	35	0	543	0	285	analytics
net.hockey.app	0	0	0	0	0	0	0	0	114	69	0	266	analytics
com.apps.flyer	0	2	0	0	0	0	0	2	7	91	95	106	ads
com.kochava	0	3	0	0	0	0	0	3	0	36	34	33	ads
com.threat.metrix	2	24	0	6	3	32	0	0	2	12	0	37	analytics
com.google	0	0	0	0	0	0	0	0	3	22	164	16	ads
io.intercom	0	9	0	0	0	0	0	8	0	9	0	106	analytics
io.branch	0	0	0	0	0	0	0	0	0	65	0	66	analytics
com.appsee	15	0	15	0	0	15	0	0	0	0	0	60	analytics
bo.app	0	0	0	0	0	7	0	0	0	0	0	60	analytics
com.tune	0	0	0	0	0	0	0	0	0	12	0	107	**finance**
com.segment	0	24	0	0	0	24	0	0	0	24	3	76	analytics
com.adjust	0	0	0	0	0	0	0	55	0	55	0	57	**finance**
com.leanplum	0	0	0	0	0	0	12	0	1	11	0	11	analytics
com.nielsen	0	0	0	0	0	4	0	1	0	26	6	4	analytics
com.iovation	0	9	0	8	0	0	0	9	0	9	0	7	analytics
com.startapp	0	0	0	0	0	0	0	0	0	32	32	99	ads
com.newrelic	34	0	0	0	34	0	0	0	34	0	0	174	analytics
com.mobvista	0	22	0	0	0	0	0	22	27	22	22	65	analytics

```
// JCPenney app
static String g(Context context) {...
    hashMap.put("di",(telephonyManager.getDeviceId()).digest());
    ...
    jSONObject.put("di", hashMap.get("di")) ;...
}

// Dunkin app
final HttpParameterMap getRiskBodyParameterMap(){...
    httpParameterMap.add("imei", (TelephonyManager)
            context.getSystemService("phone")).getDeviceId(), true) ;...
    return httpParameterMap;
}
```

Figure 6: Hashed leak (top) vs. raw leak (bottom) in the same, com.threatmetrix library.

"leakiest"; for each library and each ID, we show the number of library methods that leak the raw ID and the number of library methods that leak the hashed ID. For example, library com.paypal has 35 methods that leak the hashed IMEI, 8 methods that leak the raw IMEI, 35 methods that leak the hashed IMSI, etc.

For each library, we also present the library's purpose, as indicated on the library's website or GitHub page. Note that only three libraries are financial: com.paypal, com.tune, com.adjust; hardware ID leaks are expected, and allowed, in these libraries. However, the analysis shows that most leaks are in non-financial libraries, the overwhelming majority of which are advertising and analytics.

Table 8 paints a grim picture of the Android library landscape when it comes to privacy: advertising and analytics libraries make heavy use of hardware IDs, but this use appears aimed at identifying users and devices rather than preventing fraud. Ironically, financial libraries com.tune and com.adjust are among the most privacy-friendly libraries (least intensive users of hardware IDs).

6.5 Leakiest Apps

We examined the "leakiest" apps in light of the Google guidelines for acceptable use of hardware IDs. We focus on the top-25 apps that manage to leak *all hardware identifiers, raw*. Moreover, many of these apps also leak hashed versions of hardware identifiers; leaking both raw and hashed versions is a concern for de-anonymization. We show the results in Table 9. For each app we show the popularity (the floor of the number of installs, as indicated on Google Play on February 25th, 2021), the app category, and the list of leaks.

We identified those apps that have a legitimate financial reason to use hardware IDs as follows: apps that are in the Finance category, or apps that link with a financial library, and the leaks are due to the library (third-party) code rather than the app code. The apps that did not meet these conditions, shown in red in the table, potentially violate ID usage guidelines. Our approach distinguishes between raw and hashed, and between third-party vs. own leaks, helping spot potential violations. In contrast, an approach that misses these nuances might flag a substantial number of benign, policy-abiding apps as problematic (i.e., a high rate of false positives).

Altogether, our dataset had 190 apps that either use a financial library, or the app itself is in the Finance category. These apps might need hardware identifier information for fraud & abuse checking purposes, so leaks from these apps can be accepted. However, 47 out of these 190 apps leak at least one raw hardware ID via a non-financial third-party library, which is a concern.

Table 9: "Leakiest" apps. Non-financial apps with no financial libraries shown in red; R=raw, H=hashed.

App	#Installs	Category	IMEI		IMSI		Serial		MAC		AndrID		AdvID	GUID
	(million)		R	H	R	H	R	H	R	H	R	H	R	R
Spectrum TV [29]	10	Entertainment	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
CGTN [13]	5	News & Magazines	✓		✓		✓	✓	✓		✓		✓	✓
GPS Navigation System [20]	10	Maps & Navigation	✓		✓		✓		✓		✓		✓	✓
WiFi Map [19]	50	Productivity	✓	✓	✓	✓	✓		✓		✓		✓	✓
Bitcoin, Crypto News [8]	1	Finance	✓		✓		✓		✓		✓		✓	✓
CheapOair [14]	1	Travel & Local	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
Greyhound Lines [21]	1	Travel & Local	✓		✓		✓		✓		✓			✓
JCPenney [23]	5	Shopping	✓		✓		✓		✓		✓			✓
CBS News [12]	1	News & Magazines	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓
Wendy's [31]	5	Food & Drink	✓		✓		✓	✓	✓		✓	✓	✓	✓
Zipcar [33]	1	Maps & Navigation	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓
Lyft Rideshare [25]	10	Maps & Navigation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Western Union [32]	5	Finance	✓	✓	✓		✓	✓	✓		✓	✓		✓
NJ TRANSIT [26]	1	Maps & Navigation	✓		✓		✓	✓	✓	✓	✓			✓
Curb The Taxi App [15]	1	Maps & Navigation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
Apartments.com [4]	5	House & Home	✓		✓		✓		✓		✓			✓
CareZone [1]	1	Medical	✓		✓		✓		✓		✓	✓		✓
BURGER KING [10]	10	Food & Drinks	✓		✓		✓		✓		✓		✓	✓
Fox Now [18]	10	Entertainment	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
One Dollar [27]	0.5	Shopping	✓		✓		✓		✓		✓	✓	✓	✓
Sam's Club [28]	1	Shopping	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓
Aaptiv [5]	1	Health & Fitness	✓		✓		✓		✓		✓		✓	✓
Letgo [24]	100	Shopping	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Amber Weather [6]	1	Weather	✓		✓		✓	✓	✓		✓	✓	✓	✓
Blink Health Rx [9]	0.1	Medical	✓		✓		✓		✓		✓			✓

Table 10: Identifier-centric study results: 2018 → 2020 changes in identifier use.

		IMEI		IMSI		Serial		MAC		AndroidID		AdvID		GUID	
		TP	O	TP	O	TP	O	TP	O	TP	O	TP	O	TP	O
Raw		+8	+19	+1	+8	+14	+23	+8	+10	+41	+44	+24	+44	+47	+52
		-45	-59	-8	-2	-28	-25	-48	-28	-75	-52	-48	-23	-45	-27
	Net	*-37*	*-40*	*-7*	*6*	*-14*	*-2*	*-40*	*-18*	*-34*	*-8*	*-24*	*21*	*2*	*25*
Hashed		+12	+8	+5	+5	+15	+10	+5	+5	+33	+26	+0	+0	+0	+0
		-19	-16	-10	-8	-21	-5	-12	-2	-34	-18	-0	-0	-0	-0
	Net	*-7*	*-8*	*-5*	*-3*	*-6*	*5*	*-7*	*3*	*-1*	*8*	*0*	*0*	*0*	*0*

6.6 Longitudinal Study: 2018 vs. 2020

To investigate whether apps are becoming more guidelines-compliant and privacy-friendly, we conducted a longitudinal study, comparing the 2018 versions of 1,000-app dataset with their 2020 counterparts.

6.6.1 Identifier-centric Study. We first investigate how the prevalence/use of a certain identifier has changed over two years. We tabulate the findings in Table 10. For each ID, each code location (third-party (TP) or own (O)), and each leak type (hashed or raw) we show the number of apps that added that ID leak with '+' and the number of apps that removed that ID leak with '-'. For example, for raw IMEI we have: in third-party code, 8 apps have added this leak and 45 apps have removed this leak, yielding a net change of *-37*; whereas in own code, 19 apps have added this leak and 59 apps have removed this leak, yielding a net change of *-40*. The results reveal several trends. First, the use of raw IDs has decreased across the board: notice the negative net figures for IMEI, Serial, MAC Address, AndroidID. Two groups saw an increase: AdvertisingID and GUID,[6] especially in own code, as well as hashed own code (AndroidID, Serial, MAC). These results, also corroborated by the app-centric study in Section 6.6.2, indicate (1) a move away form hardware identifiers and toward resettable identifiers, and (2) replacing raw with hashed values, which is encouraging.

6.6.2 App-centric Study. The second part of our study is app-centric. Assuming the signature of an app in 2018 was S_{2018} while in 2020

[6]We keep the '0' values for AdvertisingID and GUID in the table for uniformity; since these IDs were not used in the hashed form to begin with, there was no change.

Table 11: App-centric study results: subsumption kind, informal definition, and # of apps exhibiting subsumption.

Kind	Subsumption Definition	#Apps
AND	hardware ID leaks decreased	108
	hardware ID leaks decreased, software ID leaks increased	11
	raw IMEI leak removed	87
	raw MAC Address leak removed	71
	raw Serial leak removed	49
	raw IMSI leak removed	12
	raw AndroidID leak removed	107
Hash	raw hardware ID leak ⟶ hashed hw. ID leak	26
	raw IMEI ⟶ hashed IMEI	14
	raw IMSI ⟶ hashed IMSI	3
	raw MAC Address ⟶ hashed MAC Address	6
	raw Serial ⟶ hashed Serial	3
	raw AndroidID ⟶ hashed AndroidID	20

the signature is S_{2020}, we check whether $S_{2020} <: S_{2018}$. We show how our notion of subsumption allows for flexible definitions, hence we can gauge, along several dimensions, whether the apps have become more privacy-friendly.

We show the results in Table 11. We start with *AND subsumption*, e.g., $e \wedge s <: e \wedge s \wedge r$ indicates a reduction in hardware identifiers; we found that 108 apps exhibit this condition, which is encouraging as it means dropping the use of one or more hardware identifiers. When relaxing the subsumption notion to allow for increases in software IDs, we found a further 11 apps that exhibit this condition, which is still positive, as the use of software IDs is preferred to the use of hardware IDs. We also show the number of apps that drop each ID; IMEI and MAC Address are the most-dropped hardware identifiers (87 and 71 apps, respectively), while AndroidID was dropped by 107 apps.

Hash subsumption, e.g., $e \wedge h(ID) <: e \wedge ID$, indicates that the app has switched from leaking the raw ID to leaking the hashed ID. While few apps exhibit this subsumption (26 for hardware IDs, 20 for AndroidID), it is nevertheless a privacy gain.

Finally, 35 apps were in the undesirable "reverse subsumption" situation: at least one hardware ID leak was added in the 2020 version. To conclude, the longitudinal analysis reveals an overall move away from usage/leaks of hardware IDs, toward resettable IDs; and to smaller extent, a move toward hashed hardware IDs.

7 RELATED WORK

Many static flow analyzers for Android have been developed, including Amandroid [56], DIALDroid [16], DidFail [17], DroidSafe [42], FlowDroid [35] and IccTA [22]. A prior study [54] found that a typical analyzer takes on average 6 minutes per app, on par with our approach. Most of these tools use the predefined sources and sinks list from Susi [3], with the binary goal of deciding whether a source flows to a sink; this renders the results quite imprecise, limiting tools' usability.

DroidInfer [45] uses a context-sensitive information flow type system to improve static analysis precision and scalability, and supports analysis of libraries; their focus is on sensitive data leaks (to network or logs). Evaluation on top Google Play apps (144) shows high precision (FP=15%). DroidInfer's goal is intuitive source→sink tracking rather than algebraic signatures and ID (ab)use studies.

Horndroid [37] focused on improving the precision of existing static analyses by determining whether a sink will be reached by tainted flows, and refining branch conditions to avoid false positives. However, Horndroid does not allow naming sources (as we do with the seven IDs), so their approach is not directly comparable to ours.

Myers and Liskov's label model (Jif/DLM) [52] describes "unions" of labels: set union, i.e., AND in our model. Our XORs, not supported in Jif, would be set disjoint union. While we do not support label polymorphism as Jif does, polymorphism would only help if there was cross first-party to third-party flow which we did not find (Section 4.4). Stefan et al.'s disjunction category labels [55] are defined as conjunctions and disjunctions on principals; "can-flow" as logical implication governs safe information flow. They implement `dclabel-static`, a prototype information flow control in Haskell, but no evaluation is provided. Montagu et al. [51] introduced label algebras – a set of labels that form a pre-lattice, i.e., with a pre-order (the term "algebraic" in our work, from algebraic data types, refers to the product and sum operations on types). The focus of these approaches was the formalism/flow model. In contrast, for us, the algebraic taint representation is a conduit to implementing a static analysis for Android and conducting six studies on 1,000 top apps.

MAPS [57] distinguishes between first-party and third-party ID leaks by the call site of sensitive API methods but does not perform taint tracking or static analysis – understandable for the scale (1,035,853 apps). This is prone to false positives, e.g., ID-retrieving calls in dead code, or IDs which are read but not used/leaked.

The Taintdroid [41] dynamic taint tracker has exposed that location and phone information are routinely leaked to advertising and content servers. Taintdroid's focus is on efficiently tracking taint within an app and the Android OS, whereas we perform static tracking, and within the confines of the app only. TaintDroid does not distinguish between raw and hashed leaks.

Dynamic taint analysis [40, 41, 47] has different goals compared to us: reduce false positives or track which servers packets go to (which is impossible with our approach). Our static approach aims to reduce false negatives and allows analysis at scale. Dynamic analysis, in general, needs to overcome two issues (1) low coverage [36, 38, 49, 53], and (2) signing-in successfully – this is problematic in cases such as the Western Union banking app, which requires a Western Union customer account (as do other apps in Table 9).

8 CONCLUSIONS

We introduce an algebraic taint representation that solves a key problem with existing taint analyses: distinguishing between programs that leak data in ways that are similar on the surface, but very different underneath. We implemented algebraic taint tracking as a static analysis for Android, and demonstrate its effectiveness through six studies on identifier (ab)use in top Android apps and libraries. We found that being able to capture subtle yet critical differences is key for understanding app behavior w.r.t. user privacy or abiding by developer guidelines. Our longitudinal study shows that over the past two years, apps have become more privacy-friendly.

REFERENCES

[1] 2020. CareZone. https://carezone.com/.

[2] 2020. FlowDroid. https://github.com/secure-software-engineering/FlowDroid.

[3] 2020. SuSi. https://github.com/secure-software-engineering/SuSi.

[4] 2021. Apartments.com Rental Search. https://play.google.com/store/apps/details?id=com.apartments.mobile.android.

[5] 2021. Aaptiv: 1 Audio Fitness App. https://play.google.com/store/apps/details?id=com.aaptiv.android.

[6] 2021. Amber WeatherRadar Free. https://play.google.com/store/apps/details?id=com.amber.weather.

[7] 2021. Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids#java. Accessed: 2021-02-25.

[8] 2021. Bitcoin, Ethereum, IOTA Ripple Price,Crypto News. https://play.google.com/store/apps/details?id=com.crypto.currency.

[9] 2021. Blink Health Rx - Best Discount Pharmacy Prices. https://play.google.com/store/apps/details?id=com.blinkhealth.blinkandroid.

[10] 2021. BURGER KING. https://play.google.com/store/apps/details?id=com.emn8.mobilem8.nativeapp.bk.

[11] 2021. CBS News - Live Breaking News. https://play.google.com/store/apps/details?id=com.audiobooks.androidapp&hl=en_US&gl=US.

[12] 2021. CBS News - Live Breaking News. https://play.google.com/store/apps/details?id=com.treemolabs.apps.cbsnews.

[13] 2021. CGTN – China Global TV Network. https://play.google.com/store/apps/details?id=com.imib.cctv.

[14] 2021. CheapOair: Cheap Flights, Cheap Hotels Booking App. https://play.google.com/store/apps/details?id=com.fp.cheapoair.

[15] 2021. Curb - The Taxi App. https://play.google.com/store/apps/details?id=com.ridecharge.android.taximagic.

[16] 2021. DIALDroid. https://github.com/dialdroid-android/DIALDroid.

[17] 2021. DidFail. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=508078.

[18] 2021. FOX NOW: Watch Live On Demand TV Stream Sports. https://play.google.com/store/apps/details?id=com.fox.now.

[19] 2021. Free WiFi Passwords Internet Hotspot - WiFi Map. https://play.google.com/store/apps/details?id=io.wifimap.wifimap.

[20] 2021. GPS Navigation System, Traffic Maps by Karta. https://play.google.com/store/apps/details?id=com.kartatech.karta.gps.

[21] 2021. Greyhound Lines. https://play.google.com/store/apps/details?id=com.greyhound.mobile.consumer.

[22] 2021. IccTA. https://sites.google.com/site/icctawebpage/source-and-usage.

[23] 2021. JCPenney – Shopping Deals. https://play.google.com/store/apps/details?id=com.jcp.

[24] 2021. letgo: Buy Sell Used Stuff, Cars, Furniture. https://play.google.com/store/apps/details?id=com.abtnprojects.ambatana.

[25] 2021. Lyft - Rideshare, Bikes, Scooters Transit. https://play.google.com/store/apps/details?id=me.lyft.android.

[26] 2021. NJ TRANSIT Mobile App. https://play.google.com/store/apps/details?id=com.njtransit.njtapp.

[27] 2021. One Dollar - Tap To Win. https://apk.support/app/com.giinger.onedollar.

[28] 2021. Sam's Club Scan Go: Wholesale Shopping Savings. https://apk.support/app/com.samsclub.sng.

[29] 2021. Spectrum TV. https://play.google.com/store/apps/details?id=com.TWCableTV.

[30] 2021. Texas Roadhouse Mobile. https://play.google.com/store/apps/details?id=com.relevantmobile.texasroadhouse.

[31] 2021. Wendy's – Earn Rewards, Order Food Score Offers. https://play.google.com/store/apps/details?id=com.wendys.nutritiontool.

[32] 2021. Western Union International: Send Money Transfer. https://play.google.com/store/apps/details?id=com.westernunion.moneytransferr3app.eu.

[33] 2021. Zipcar. https://play.google.com/store/apps/details?id=com.zc.android.

[34] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[35] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. SIGPLAN Not. 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[36] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications (Indianapolis, Indiana, USA) (OOPSLA '13). Association for Computing Machinery, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[37] S. Calzavara, I. Grishchenko, and M. Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In 2016 IEEE European Symposium on Security and Privacy (EuroS P). 47–62.

[38] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, USA, 429–440. https://doi.org/10.1109/ASE.2015.89

[39] ALONZO CHURCH. 1941. The Calculi of Lambda Conversion. (AM-6). Princeton University Press. http://www.jstor.org/stable/j.ctt1b9x12d

[40] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure Password Tracking for Android. In 23rd USENIX Security Symposium (USENIX Security 14). USENIX Association, San Diego, CA, 481–494. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (Vancouver, BC, Canada) (OSDI'10). USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[42] Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. https://doi.org/10.14722/ndss.2015.23089

[43] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking App Behavior against App Descriptions. In Proceedings of the 36th International Conference on Software Engineering (Hyderabad, India) (ICSE 2014). Association for Computing Machinery, New York, NY, USA, 1025–1035. https://doi.org/10.1145/2568225.2568276

[44] William A Howard. 1980. The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism 44 (1980), 479–490.

[45] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In Proceedings of the 2015 International Symposium on Software Testing and Analysis. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

[46] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Boston, Massachusetts) (POPL '73). Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[47] H. Kuzuno and S. Tonami. 2013. Signature generation for sensitive information leakage in android applications. In 2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW). 112–119. https://doi.org/10.1109/ICDEW.2013.6547438

[48] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system release 4.11 Documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/index.html

[49] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (Saint Petersburg, Russia) (ESEC/FSE 2013). Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[50] Bartosz Milewski et al. 2020. If Either can be either Left or Right but not both, then why does it correspond to OR instead of XOR in Curry-Howard correspondence? https://stackoverflow.com/questions/64394487/if-either-can-be-either-left-or-right-but-not-both-then-why-does-it-correspond

[51] Benoît Montagu, B. Pierce, and R. Pollack. 2013. A Theory of Information-Flow Labels. 2013 IEEE 26th Computer Security Foundations Symposium (2013), 3–17.

[52] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. ACM Trans. Softw. Eng. Methodol. 9, 4 (Oct. 2000), 410–442. https://doi.org/10.1145/363516.363526

[53] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the Effectiveness of Random Testing for Android: Or How i Learned to Stop Worrying and Love the Monkey. In Proceedings of the 13th International Workshop on Automation of Software Test (Gothenburg, Sweden) (AST '18). Association for Computing Machinery, New York, NY, USA, 34–37. https://doi.org/10.1145/3194733.3194742

[54] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. 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 (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

[55] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. 2011. Disjunction category labels. In Nordic conference on secure IT systems. Springer, 223–239.

[56] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In Proceedings of the 2014 ACM SIGSAC

Conference on Computer and Communications Security (Scottsdale, Arizona, USA) (CCS '14). Association for Computing Machinery, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[57] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel Reidenberg, N. Russell, and Norman Sadeh. 2019. MAPS: Scaling Privacy Compliance Analysis to a Million Apps. Proceedings on Privacy Enhancing Technologies 2019 (07 2019), 66–86. https://doi.org/10.2478/popets-2019-0037