A Bayesian Approach to Privacy Enforcement in Smartphones

Omer Tripp
IBM Research, USA

Julia Rubin
IBM Research, Israel

Abstract

Mobile apps often require access to private data, such as the device ID or location. At the same time, popular platforms like Android and iOS have limited support for user privacy. This frequently leads to unauthorized disclosure of private information by mobile apps, e.g. for advertising and analytics purposes. This paper addresses the problem of privacy enforcement in mobile systems, which we formulate as a classification problem: When arriving at a privacy sink (e.g., database update or outgoing web message), the runtime system must classify the sink's behavior as either legitimate or illegitimate. The traditional approach of information-flow (or taint) tracking applies "binary" classification, whereby information release is legitimate iff there is no data flow from a privacy source to sink arguments. While this is a useful heuristic, it also leads to false alarms.

We propose to address privacy enforcement as a learning problem, relaxing binary judgments into a quantitative/probabilistic mode of reasoning. Specifically, we propose a Bayesian notion of statistical classification, which conditions the judgment whether a release point is legitimate on the evidence arising at that point. In our concrete approach, implemented as the BAYESDROID system that is soon to be featured in a commercial product, the evidence refers to the similarity between the data values about to be released and the private data stored on the device. Compared to TaintDroid, a state-of-the-art taint-based tool for privacy enforcement, BAYESDROID is substantially more accurate. Applied to 54 top-popular Google Play apps, BAYESDROID is able to detect 27 privacy violations with only 1 false alarm.

1 Introduction

Mobile apps frequently demand access to private information. This includes unique device and user identifiers, such as the phone number or IMEI number (identifying the physical device); social and contacts data; the user's location; audio (microphone) and video (camera) data; etc. While private information often serves the core functionality of an app, it may also serve other purposes, such as advertising, analytics or cross-application profiling [9]. From the outside, the user is typically unable to distinguish legitimate usage of their private information from illegitimate scenarios, such as sending of the IMEI number to a remote advertising website to create a persistent profile of the user.

Existing platforms provide limited protection against privacy threats. Both the Android and the iOS platforms mediate access to private information via a permission model. Each permission is mapped to a designated resource, and holds per all application behaviors and resource accesses. In Android, permissions are given or denied at installation time. In iOS, permissions are granted or revoked upon first access to the respective resource. Hence, both platforms cannot disambiguate legitimate from illegitimate usage of a resource once an app is granted the corresponding permission [8].

Threat Model In this paper, we address privacy threats due to authentic (as opposed to malicious) mobile applications [4, 18]. Contrary to malware, such applications execute their declared functionality, though they may still expose the user to unnecessary threats by incorporating extraneous behaviors — neither required by their core business logic nor approved by the user [11] — such as analytics, advertising, cross-application profiling, social computing, etc. We consider unauthorized release of private information that (almost) unambiguously identifies the user as a privacy threat. Henceforth, we dub such threats *illegitimate*.

While in general there is no bullet-proof solution for privacy enforcement that can deal with any type of covert channel, implicit flow or application-specific data transformation, and even conservative enforcement approaches can easily be bypassed [19], there is strong evidence that authentic apps rarely exhibit these challenges.

According to a recent study [9], and also our empirical data (presented in Section 5), private information is normally sent to independent third-party servers. Consequently, data items are released in clear form, or at most following well-known encoding/encryption transformations (like Base64 or MD5), to meet the requirement of a standard and general client/server interface.

The challenge, in this setting, is to determine whether the app has taken sufficient means to protect user privacy. Release of private information, even without user authorization, is still legitimate if only a small amount of information has been released. As an example, if an application obtains the full location of the user, but then releases to an analytics server only coarse information like the country or continent, then in most cases this would be perceived as legitimate.

```
String mImsi = ...; // source
// 6 digits <= IMSI (MCC+MNC+MSIN) <= 15 (usually 15)
if (mImsi != null &&
    (mImsi.length() < 6 || mImsi.length() > 15)) {
  loge(" invalid ␣IMSI␣" + mImsi); // sink
  mImsi = null; }
log("IMSI:␣" + mImsi.substring(0, 6) + "xxxxxxxxx"); // sink
```

Figure 1: Fragment from an internal Android library, `com.android.internal.telephony.cdma.RuimRecords`, where a prefix of the mobile device's IMSI number flows into the standard log file

Privacy Enforcement via Taint Analysis The shortcomings of mobile platforms in ensuring user privacy have led to a surge of research on realtime privacy monitoring. The foundational technique grounding this research is *information-flow tracking*, often in the form of *taint analysis* [23, 15]: Private data, obtained via privacy *sources* (e.g. `TelephonyManager.getSubscriberId()`, which reads the device's IMSI), is labeled with a taint tag denoting its source. The tag is then propagated along data-flow paths within the code. Any such path that ends up in a release point, or privacy *sink* (e.g. `WebView.loadUrl(...)`, which sends out an HTTP request), triggers a leakage alarm.

The tainting approach effectively reduces leakage judgments to boolean reachability queries. This can potentially lead to false reports, as the real-world example shown in Figure 1 illustrates. This code fragment, extracted from a core library in the Android platform, reads the device's IMSI number, and then either (ii) persists the full number to an error log if the number is invalid (the `loge(...)` call), or (ii) writes a prefix of the IMSI (of length 6) to the standard log while carefully masking away the suffix (of length 9) as `'x'` characters. Importantly, data flow into the `log(...)` sink is not a privacy problem, because the first 6 digits merely carry model and origin information. Distinctions of this sort are beyond the discriminative power of taint analysis [26].

Quantitative extensions of the core tainting approach have been proposed to address this limitation. A notable example is McCamant and Ernst's [13] information-flow tracking system, which quantities flow of secret information by dynamically tracking taint labels at the bit level. Other approaches — based e.g. on distinguishability between secrets [1], the rate of data transmission [12] or the influence inputs have on output values [14] — have also been proposed. While these systems are useful as offline analyses, it is highly unlikely that any of them can be engineered to meet the performance requirements of a realtime monitoring solution due to the high complexity of their underlying algorithms. As an example, McCamant and Ernst report on a workload on which their analysis spent over an hour.

Our Approach We formulate data leakage as a classification problem, which generalizes the source/sink reachability judgment enforced by standard information-flow analysis, permitting richer and more relaxed judgments in the form of statistical classification. The motivating observation is that reasoning about information release is fuzzy in nature. While there are clear examples of legitimate versus illegitimate information release, there are also less obvious cases (e.g., a variant of the example in Figure 1 with a 10- rather than 6-character prefix). A statistical approach, accounting for multiple factors and based on rich data sets, is better able to address these subtleties.

Concretely, we propose Bayesian classification. To label a release point as either legitimate or illegitimate, the Bayesian classifier refers to the "evidence" at that point, and computes the likelihood of each label given the evidence. The evidence consists of feature/value pairs. There are many ways of defining the evidence. In this study, we concentrate on the data arguments flowing into release operations, though we intend to consider other classes of features in the future. (See Section 7.)

Specifically, we induce features over the private values stored on the device, and evaluate these features according to the level of similarity between the private values and those arising at release points. This distinguishes instances where data that is dependent on private values flows into a release point, but its structural and/or quantitative characteristics make it eligible for release, from illegitimate behaviors. Failure to make such distinctions is a common source of false alarms suffered by the tainting approach [4].

To illustrate this notion of features, we return to the example in Figure 1. Because the IMSI number is consid-

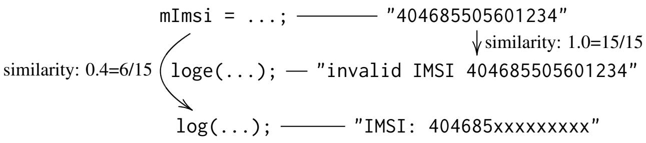

Figure 2: Similarity analysis applied to the code in Figure 1

ered private, we define a respective feature *IMSI*. Assume that the concrete IMSI value is “404685505601234”. Then the value arising at the `log(...)` release point is “IMSI: 404685xxxxxxxxx”. The quantitative similarity between these two values serves as evidence for the decision whether or not `log(...)` is behaving legitimately. This style of reasoning is depicted in Figure 2.

Evaluation To evaluate our approach, we have implemented the BayesDroid system for privacy enforcement. We report on two sets of experiments over BayesDroid.

First, to measure the accuracy gain thanks to Bayesian analysis, we compared BayesDroid with the TaintDroid system [4], a highly popular and mature implementation of the tainting approach that is considered both efficient (with average overhead of approximately 10%) and accurate. We applied both BayesDroid and TaintDroid to the DroidBench suite,[1] which comprises the most mature and comprehensive set of privacy benchmarks currently available. The results suggest dramatic improvement in accuracy thanks to Bayesian elimination of false reports, yielding accuracy scores of 0.96 for BayesDroid versus 0.66 for TaintDroid.

The second experiment examines the practical value of BayesDroid by applying it to 54 top-popular mobile apps from Google Play. We evaluate two variants of BayesDroid, one of which is able to detect a total of 27 distinct instances of illegitimate information release across 15 of the applications with only 1 false alarm.

Contributions This paper makes the following principal contributions:

1. Novel approach to leakage detection (Section 2): We present a Bayesian classification alternative to the classic tainting approach. Our approach is more flexible than taint tracking by permitting statistical weighting of different features as the basis for privacy judgments.
2. Similarity-based reasoning (Section 3): We instantiate the Bayesian approach by applying quantitative similarity judgments over private values and values about to be released. This enables consideration of actual data, rather than only data flow, as evidence for privacy judgments.
3. Implementation and evaluation (Sections 4–5): We have instantiated our approach as the BayesDroid system, which is about to be featured in an IBM cloud-based security service. We report on two sets of experiments, whose results (i) demonstrate substantial accuracy gain thanks to Bayesian reasoning, and (ii) substantiate the overall effectiveness of BayesDroid when applied to real-world apps. All the leakage reports by BayesDroid are publicly available for scrutiny.[2]

2 The Bayesian Setting

Our starting point is to treat privacy enforcement as a classification problem, being the decision whether or not a given release point is legitimate. The events, or instances, to be classified are (runtime) release points. The labels are *legitimate* and *illegitimate*. Misclassification either yields a false alarm (mistaking benign information release as a privacy threat) or a missed data leak (failing to intercept illegitimate information release).

2.1 Bayes and Naive Bayes

Our general approach is to base the classification on the *evidence* arising at the release point. Items of evidence may refer to qualitative facts, such as source/sink data-flow reachability, as well as quantitative measures, such as the degree of similarity between private values and values about to be released. These latter criteria are essential in going beyond the question of *whether* private information is released to also reason about the *amount* and *form* of private information about to be released.

A popular classification method, representing this mode of reasoning, is based on Bayes’ theorem (or rule). Given events X and Y, Bayes’ theorem states the following equality:

$$\Pr(Y|X) = \frac{\Pr(X|Y) \cdot \Pr(Y)}{\Pr(X)} \quad (1)$$

where $\Pr(Y|X)$ is the conditional probability of Y given X (i.e., the probability for Y to occur given that X has occurred). X is referred to as the *evidence*. Given evidence X, Bayesian classifiers compute the conditional likelihood of each label (in our case, *legitimate* and *illegitimate*).

We begin with the formal background by stating Equation 1 more rigorously. Assume that Y is a discrete-valued random variable, and let $X = [X_1, \ldots, X_n]$ be a

[1] http://sseblog.ec-spride.de/tools/droidbench/

[2] researcher.ibm.com/researcher/files/us-otripp/Artifacts.zip

vector of n discrete or real-valued attributes X_i. Then

$$\Pr(Y = y_k | X_1 \dots X_n) = \frac{\Pr(Y = y_k) \cdot \Pr(X_1 \dots X_n | Y = y_k)}{\Sigma_j \Pr(Y = y_j) \cdot \Pr(X_1 \dots X_n | Y = y_j)} \quad (2)$$

As Equation 2 hints, training a Bayesian classifier is, in general, impractical. Even in the simple case where the evidence X is a vector of n boolean attributes and Y is boolean, we are still required to estimate a set

$$\theta_{ij} = \Pr(X = x_i | Y = y_j)$$

of parameters, where i assumes 2^n values and j assumes 2 values for a total of $2 \cdot (2^n - 1)$ independent parameters.

Naive Bayes deals with the intractable sample complexity by introducing the assumption of conditional independence, as stated in Definition 2.1 below, which reduces the number of independent parameters sharply to $2n$. Intuitively, conditional independence prescribes that events X and Y are independent given knowledge that event Z has occurred.

Definition 2.1 (Conditional Independence)**.** *Given random variables X, Y and Z, we say that X is* conditionally independent *of Y given Z iff the probability distribution governing X is independent of the value of Y given Z. That is,*

$$\forall i, j, k.\ \Pr(X = x_i | Y = y_j, Z = z_k) = \Pr(X = x_i | Z = z_k)$$

Under the assumption of conditional independence, we obtain the following equality:

$$\Pr(X_1 \dots X_n | Y) = \Pi_{i=1}^n \Pr(X_i | Y) \quad (3)$$

Therefore,

$$\Pr(Y = y_k | X_1 \dots X_n) = \frac{\Pr(Y = y_k) \cdot \Pi_i \Pr(X_i | Y = y_k)}{\Sigma_j \Pr(Y = y_j) \cdot \Pi_i \Pr(X_i | Y = y_j)} \quad (4)$$

2.2 Bayesian Reasoning about Leakage

For leakage detection, conditional independence translates into the requirement that at a release point st, the "weight" of evidence e_1 is not affected by the "weight" of evidence e_2 knowing that st is legitimate/illegitimate. As an example, assuming the evidence is computed as the similarity between private and released values, if st is known to be a statement sending private data to the network, then the similarity between the IMSI number and respective values about to be released is assumed to be independent of the similarity between location coordinates and respective values about to be released.

The assumption of conditional independence induces a "modular" mode of reasoning, whereby the privacy features comprising the evidence are evaluated independently. This simplifies the problem of classifying a release point according to the Bayesian method into two quantities that we need to clarify and estimate: (i) the likelihood of legitimate/illegitimate release ($\Pr(Y = y_k)$) and (ii) the conditional probabilities $\Pr(X_i | Y = y_k)$.

3 Privacy Features

In this section we develop, based on the mathematical background in Section 2, an algorithm to compute the conditional likelihood of legitimate versus illegitimate data release given privacy features F_i. With such an algorithm in place, given values v_i for the features F_i, we obtain

$$v_{leg} = \Pr(legitimate | [F_1 = v_1, \dots, F_n = v_n])$$
$$v_{illeg} = \Pr(illegitimate | [F_1 = v_1, \dots, F_n = v_n])$$

Bayesian classification then reduces to comparing between v_{leg} and v_{illeg}, where the label corresponding to the greater of these values is the classification result.

3.1 Feature Extraction

The first challenge that arises is how to define the features (denoted with italicized font: *F*) corresponding to the private values (denoted with regular font: F). This requires simultaneous consideration of both the actual private value and the "relevant" values arising at the sink statement (or release point). We apply the following computation:

1. Reference value: We refer to the actual private value as the *reference value*, denoting the value of private item F as $[\![\mathrm{F}]\!]$. For the example in Figures 1–2, the reference value, $[\![\mathrm{IMSI}]\!]$, of the *IMSI* feature would be the device's IMSI number: $[\![\mathrm{IMSI}]\!]$ = "404685505601234".
2. Relevant value: We refer to value v about to be released by the sink statement as *relevant* with respect to feature F if there is data-flow connectivity between a source statement reading the value $[\![\mathrm{F}]\!]$ of F and v. Relevant values can thus be computed via information-flow tracking by propagating a unique tag (or label) per each private value, as tools like TaintDroid already do. Note that for a given feature F, multiple different relevant values may arise at a given sink statement (if the private item F flows into more than one sink argument).
3. Feature value: Finally, given the reference value $[\![\mathrm{F}]\!]$ and a set $\{v_1, \dots, v_k\}$ of relevant values for feature F, the value we assign to F (roughly) reflects the highest degree of pairwise similarity (i.e., minimal

distance) between $[\![\mathrm{F}]\!]$ and the values v_i. Formally, we assume a distance metric d. Given d, we define:

$$[\![F]\!] \equiv \min_{1 \leq i \leq k} \{ d([\![\mathrm{F}]\!], v_i) \}$$

We leave the distance metric $d(\ldots)$ unspecified for now, and return to its instantiation in Section 3.2.

According to our description above, feature values are unbounded in principle, as they represent the distance between the reference value and any data-dependent sink values. In practice, however, assuming (i) the distance metric $d(\ldots)$ satisfies $d(x,y) \leq \max\{|x|,|y|\}$, (ii) $\exists c \in \mathbb{N}.\ |[\![\mathrm{F}]\!]| \leq c$ (as with the IMEI, IMSI, location, etc.), and (iii) $[\![\mathrm{F}]\!]$ is not compared with values larger than it, we can bound $[\![F]\!]$ by c. In general, any feature can be made finite, with (at most) $n+1$ possible values, by introducing a privileged "$\geq n$" value, which denotes that the distance between the reference and relevant values is at least n.

3.2 Measuring Distance between Values

To compute a quantitative measure of similarity between data values, we exploit the fact that private data often manifests as strings of ASCII characters [4, 9, 27]. These include e.g. device identifiers (like the IMEI and IMSI numbers), GPS coordinates, inter-application communication (IPC) parameters, etc. This lets us quantify distance between values in terms of string metrics.

Many string metrics have been proposed to date [17]. Two simple and popular metrics, which we have experimented with and satisfy the requirement that $d(x,y) \leq \max\{|x|,|y|\}$, are the following:

Hamming Distance This metric assumes that the strings are of equal length. The Hamming distance between two strings is equal to the number of positions at which the corresponding symbols are different (as indicated by the indicator function $\delta_{c_1 \neq c_2}(\ldots)$):

$$\mathsf{ham}(a,b) = \Sigma_{0 \leq i < |a|} \delta_{c_1 \neq c_2}(a(i), b(i))$$

In another view, Hamming distance measures the number of substitutions required to change one string into the other.

Levenshtein Distance The Levenshtein string metric computes the distance between strings a and b as $\mathsf{lev}_{a,b}(|a|,|b|)$ (abbreviated as $\mathsf{lev}(|a|,|b|)$), where

$$\mathsf{lev}(i,j) = \begin{cases} \max(i,j) & \textit{if } \min(i,j) = 0 \\ \min \begin{pmatrix} \mathsf{lev}(i-1,j)+1 \\ \mathsf{lev}(i,j-1)+1 \\ \mathsf{lev}(i-1,j-1)+\delta_{a_i \neq b_j} \end{pmatrix} & \textit{otherwise} \end{cases}$$

Informally, $\mathsf{lev}(|a|,|b|)$ is the minimum number of single-character edits — either insertion or deletion or substitution — needed to transform one string into the other. An efficient algorithm for computing the Levenshtein distance is bottom-up dynamic programming [24]. The asymptotic complexity is $O(|a| \cdot |b|)$.

```
Data: Strings $u$ and $v$
Data: Distance metric $d$
begin
    $x \longleftarrow |u| < |v|$ ? $u$ : $v$ // min
    $y \longleftarrow |u| \geq |v|$ ? $u$ : $v$ // max
    $r \longleftarrow y$
    for $i = 0$ to $|y| - |x|$ do
        $y' \longleftarrow y[i, i+|x|-1]$
        if $d(x,y') < r$ then
            $r \longleftarrow d(x,y')$
        end
    end
    return $r$
end
```

Algorithm 1: The BAYESDROID distance measurement algorithm

Given string metric $d(x,y)$ and pair (u,v) of reference value u and relevant value v, BAYESDROID computes their distance according to the following steps:

1. BAYESDROID ensures that both u and v are `String` objects by either (i) invoking `toString()` on reference types or (ii) converting primitive types into `String`s (via `String.valueOf(...)`), if the argument is not already of type `String`.
2. To meet the conservative requirement that $|x| = |y|$ (i.e., x and y are of equal length), BAYESDROID applies Algorithm 1. This algorithm induces a sliding window over the longer of the two strings, whose width is equal to the length of the shorter string. The shorter string is then compared to contiguous segments of the longer string that have the same length. The output is the minimum across all comparisons.

To ensure that comparisons are still meaningful under length adjustment, we decompose private values into indivisible *information units*. These are components of the private value that cannot be broken further, and so comparing them with a shorter value mandates that the shorter value be padded. In our specification, the phone, IMEI and IMSI numbers consist of only one unit of information. The `Location` object is an example of a data structure that consists of several distinct information units. These include the integral and fractional parts of the longitude and latitude values, etc. BAYESDROID handles objects that decompose into multiple information units by treating each unit as a separate object and applying the steps above to each unit in turn. The notion of information units guards BAYESDROID against ill-founded judgments, such as treating release of a single IMEI digit as strong evidence for leakage.

3.3 Estimating Probabilities

The remaining challenge, having clarified what the features are and how their values are computed, is to estimate the probabilities appearing in Equation 4:

- We need to estimate the probability of the *legitimate* event, $\Pr(legitimate)$, where *illegitimate* is the complementary event and thus $\Pr(illegitimate) = 1 - \Pr(legitimate)$.
- We need to estimate the conditional probabilities $\Pr(F = u|legitimate)$ and $\Pr(F = u|illegitimate)$ for all features F and respective values u.

$\Pr(legitimate)$ can be approximated straightforwardly based on available statistics on the frequency of data leaks in the wild. For the conditional probabilities, assuming feature X_i is discrete valued with j distinct values (per the discussion in Section 3.1 above), we would naively compute the estimated conditional probability θ_{ijk} according to the following equation:

$$\theta_{ijk} = \widehat{\Pr}(X_i = x_{ij}|Y = y_k) \quad = \quad \frac{\#D\{X_i{=}x_{ij} \wedge Y{=}y_k\}}{\#D\{Y{=}y_k\}} \tag{5}$$

The danger, however, is that this equation would produce estimates of zero if the data happens not to contain any training examples satisfying the condition in the numerator. To fix this, we modify Equation 5 as follows:

$$\theta_{ijk} = \widehat{\Pr}(X_i = x_{ij}|Y = y_k) \quad = \quad \frac{\#D\{X_i{=}x_{ij} \wedge Y{=}y_k\}+l}{\#D\{Y{=}y_k\}+l \cdot J} \tag{6}$$

where l is a factor that "smoothens" the estimate by adding in a number of "hallucinated" examples that are assumed to be spread evenly across the J possible values of X_i. In Section 5.1, we provide concrete detail on the data sets and parameter values we used for our estimates.

4 The BayesDroid Algorithm

In this section, we describe the complete BayesDroid algorithm. We then discuss enhancements of the core algorithm.

4.1 Pseudocode Description

Algorithm 2 summarizes the main steps of BayesDroid. For simplicity, the description in Algorithm 2 assumes that source statements serve private data as their return value, though the BayesDroid implementation also supports other sources (e.g. callbacks like `onLocationChanged(...)`, where the private `Location` object is passed as a parameter). We also assume that each source maps to a unique privacy feature. Hence, when a source is invoked (i.e., the `OnSourceStatement` event fires), we obtain the unique tag corresponding to its respective feature via the `GetFeature(...)` function. We then attach the tag to the return value r. Normal data flow obeys the standard rules of tag propagation, which are provided e.g. by Enck et al. [4]. (See Table 1 there.)

```
Input: S // privacy specification

begin
    while true do
        OnSourceStatement $r$ := src $\overline{p}$ :
            // map source to feature
            $f \longleftarrow$ GetFeature src
            attach tag $f$ to $r$
        OnNormalStatement $r$ := nrm $\overline{p}$ :
            propagate feature tags according to data flow
        OnSinkStatement $r$ := snk $\overline{p}$ :
            // map feat.s to param.s with resp.  tag
            $\{f \mapsto \overline{p_f}\} \longleftarrow$ ExtractTags $\overline{p}$
            foreach $f \to \overline{p_f} \in \{f \to \overline{p_f}\}$ do
                $u \longleftarrow$ ref $f$
                $\delta \longleftarrow \min\{d(u, [\![p]\!])\}_{p \in \overline{p_f}}$
                $f \longleftarrow \delta \geq c_f \; ? \; \text{“} \geq_{c_f} \text{”} : \delta$
            end
            if IsLeakageClassification $\{f\}$ then
                Alarm snk $\overline{p}$
            end
    end
end
```

Algorithm 2: Outline of the core BayesDroid algorithm

When an `OnSinkStatement` event is triggered, the arguments flowing into the sink `snk` are searched for privacy tags, and a mapping from features f to parameters p_f carrying the respective tag is built. The value of f is then computed as the minimal pairwise distance between the parameters $p \in p_f$ and `ref` f. If this value is greater than some constant c_f defined for f, then the privileged value "$\geq c_f$" is assigned to f. (See Section 3.1.) Finally, the judgment `IsLeakageClassification` is applied over the features whose tags have reached the sink `snk`. This judgment is executed according to Equation 4.

We illustrate the BayesDroid algorithm with reference to Figure 3, which demonstrates a real leakage instance in `com.g6677.android.princesshs`, a popular gaming application. In this example, two different private items flow into the sink statement: both the IMEI, read via `getDeviceId()`, and the Android ID, read via `getString(...)`.

At sink statement `URL.openConnection(...)`, the respective tags *IMEI* and *AndroidID* are extracted. Values are assigned to these features according to the description in Section 3, where we utilize training data, as discussed later in Section 5.1, for Equation 6:

$$\begin{array}{llll} \Pr(IMEI \geq 5|leg) = & 0.071 & \Pr(AndID \geq 5|leg) = & 0.047 \\ \Pr(IMEI \geq 5|ilg) = & 0.809 & \Pr(AndID \geq 5|ilg) = & 0.833 \end{array}$$

```
source : private value
   TelephonyManager.getDeviceId() : 000000000000000
   Settings$Secure.getString (...)  : cdf15124ea4c7ad5

sink : arguments
 URL.openConnection(...) : app_id=2aec0559c930 ... &
 android_id =cdf15124ea4c7ad5 \& udid= ... &
  serial_id = ... & ... &
  publisher_user_id =000000000000000
```

Figure 3: True leakage detected by BAYESDROID in `com.g6677.android.princesshs`

We then compute Equation 4, where the denominator is the same for both *leg* and *illeg*, and so it suffices to evaluate the nominator (denoted with $\tilde{\Pr}(...)$ rather than $\Pr(...)$):

$$\tilde{\Pr}(leg|IMEI \geq 5, AndID \geq 5) =$$
$$\Pr(leg) \times \Pr(IMEI \geq 5|leg) \times \Pr(AndID \geq 5|leg) =$$
$$0.66 \times 0.071 \times 0.047 = 0.002$$
$$\tilde{\Pr}(ilg|IMEI \geq 5, AndID \geq 5) =$$
$$\Pr(ilg) \times \Pr(IMEI \geq 5|ilg) \times \Pr(AndID \geq 5|ilg) =$$
$$0.33 \times 0.809 \times 0.833 = 0.222$$

Our estimates of 0.66 for $\Pr(leg)$ and 0.33 for $\Pr(ilg)$ are again based on training data as explained in Section 5.1. The obtained conditional measure of 0.222 for *ilg* is (far) greater than 0.002 for *leg*, and so BAYESDROID resolves the release instance in Figure 3 as a privacy threat, which is indeed the correct judgment.

4.2 Enhancements

We conclude our description of BAYESDROID by highlighting two extensions of the core algorithm.

Beyond Plain Text While many instances of illegitimate information release involve plain text, and can be handled by the machinery in Section 3.1, there are also more challenging scenarios. Two notable challenges are (i) data transformations, whereby data is released following an encoding, encryption or hashing transformation; and (ii) high-volume binary data, such as camera or microphone output. We have extended BAYESDROID to address both of these cases.

We begin with data transformations. As noted earlier, in Section 1, private information is sometimes released following standard hashing/encoding transformations, such as the Base64 scheme. This situation, illustrated in Figure 4, can distort feature values, thereby leading BAYESDROID to erroneous judgments. Fortunately, the transformations that commonly manifest in leakage scenarios are all standard, and there is a small number of such transformations [9].

```
TelephonyManager tm =
    getSystemService(TELEPHONY_SERVICE);
String imei = tm.getDeviceId(); // source
String encodedIMEI = Base64Encoder.encode(imei);
Log.i(encodedIMEI); // sink
```

Figure 4: Adaptation of the DroidBench `Loop1` benchmark, which releases the device ID following Base64 encoding

To account for these transformations, BAYESDROID applies each of them to the value obtained at a source statement, thereby exploding the private value into multiple representations. This is done lazily, once a sink is reached, for performance. This enhancement is specified in pseudocode form in Algorithm 3. The main change is the introduction of a loop that traverses the transformations $\tau \in T$, where the identity transformation, $\lambda x.\ x$, is included to preserve the (non-transformed) value read at the source. The value assigned to feature f is then the minimum with respect to all transformed values.

Binary data — originating from the microphone, camera or bluetooth adapter — also requires special handling because of the binary versus ASCII representation and, more significantly, its high volume. Our solution is guided by the assumption that such data is largely treated as "uninterpreted" and immutable by application code due to its form and format. This leads to a simple yet effective strategy for similarity measurement, whereby a fixed-length prefix is truncated out of the binary content. Truncation is also applied to sink arguments consisting of binary data.

Heuristic Detection of Relevant Values So far, our description of the BAYESDROID algorithm has relied on tag propagation to identify relevant values at the sink statement. While this is a robust mechanism to drive feature computation, flowing tags throughout the code also has its costs, incurring runtime overheads of $\geq 10\%$ and affecting the stability of the application due to intrusive instrumentation [4].

These weaknesses of the tainting approach have led us to investigate an alternative method of detecting relevant values. A straightforward relaxation of data-flow tracking is bounded ("brute-force") traversal of the reachable values from the arguments to a sink statement up to some depth bound k: All values pointed-to by a sink argument or reachable from a sink argument via a sequence of $\leq k$ field dereferences are deemed relevant. Though in theory

```
Input: $T \equiv \{\lambda x.\, x, \tau_1, \ldots, \tau_n\}$ // std. transformations

begin
    ...
    OnSinkStatement r := snk $\overline{p}$ :
        $\{f \mapsto \overline{p_f}\} \longleftarrow$ ExtractTags $\overline{p}$
        foreach $f \rightarrow \overline{p_f} \in \{f \rightarrow \overline{p_f}\}$ do
            foreach $\tau \in T$ do
                $u \longleftarrow \tau$ (ref $f$)
                $\delta \longleftarrow \min\{d(u, [\![p]\!])\}_{p \in \overline{p_f}}$
                $f \longleftarrow \min\{[\![f]\!], \delta \geq c_f \;?\; \text{“} \geq_{c_f} \text{”} : \delta\}$
            end
        end
    ...
end
```

Algorithm 3: BAYESDROID support for standard data transformations

this might introduce both false positives (due to irrelevant values that are incidentally similar to the reference value) and false negatives (if k is too small, blocking relevant values from view), in practice both are unlikely, as we confirmed experimentally. (See Section 5.)

For false positives, private values are often unique, and so incidental similarity to irrelevant values is improbable. For false negatives, the arguments flowing into privacy sinks are typically either `String` objects or simple data structures. Also, because the number of privacy sinks is relatively small, and the number of complex data structures accepted by such sinks is even smaller, it is possible to specify relevant values manually for such data structures. We have encountered only a handful of data structures (e.g. the `android.content.Intent` class) that motivate a specification.

5 Experimental Evaluation

In this section, we describe the BAYESDROID implementation, and present two sets of experiments that we have conducted to evaluate our approach.

5.1 The BAYESDROID System

Implementation Similarly to existing tools like TaintDroid, BAYESDROID is implemented as an instrumented version of the Android SDK. Specifically, we have instrumented version 4.1.1_r6 of the SDK, which was chosen intentionally to match the most recent version of TaintDroid.[3] The experimental data we present indeed utilizes TaintDroid for tag propagation (as required for accurate resolution for relevant values).

Beyond the TaintDroid instrumentation scheme, the BAYESDROID scheme specifies additional behaviors for sources and sinks within the SDK. At source points, a hook is added to record the private value read by the source statement (which acts as a reference value). At sink points, a hook is installed to apply Bayesian reasoning regarding the legitimacy of the sink.

Analogously to TaintDroid, BAYESDROID performs privacy monitoring over APIs for file-system access and manipulation, inter-application and socket communication, reading the phone's state and location, and sending of text messages. BAYESDROID also monitors the HTTP interface, camera, microphone, bluetooth and contacts. As explained in Section 4.1, each of the privacy sources monitored by BAYESDROID is mirrored by a tag/feature. The full list of features is as follows: *IMEI*, *IMSI*, *AndroidID*, *Location*, *Microphone*, *Bluetooth*, *Camera*, *Contacts* and *FileSystem*.

The BAYESDROID implementation is configurable, enabling the user to switch between distance metrics as well as enable/disable information-flow tracking for precise/heuristic determination of relevant values. (See Section 4.2.) In our experiments, we tried both the Levenshtein and the Hamming metrics, but found no observable differences, and so we report the results only once. Our reasoning for why the metrics are indistinguishable is because we apply both to equal-length strings (see Section 3.2), and have made sure to apply the same metric both offline and online, and so both metrics achieve a very similar effect in the Bayesian setting.

Training To instantiate BAYESDROID with the required estimates, as explained in Section 3.3, we applied the following methodology: First, to estimate $\Pr(\mathit{legitimate})$, we relied on (i) an extensive study by Hornyack et al. spanning 1,100 top-popular free Android apps [9], as well as (ii) a similarly comprehensive study by Enck et al. [5], which also spans a set of 1,100 free apps. According to the data presented in these studies, approximately one out of three release points is illegitimate, and thus $\widehat{\Pr}(\mathit{legitimate}) = 0.66$ and complementarily $\widehat{\Pr}(\mathit{illegitimate}) = 1 - 0.66 \approx 0.33$.

For the conditional probabilities $\widehat{\Pr}(X_i = x_{ij} | Y = y_k)$, we queried Google Play for the 100 most popular apps (across all domains) in the geography of one of the authors. We then selected at random 35 of these apps, and analyzed their information-release behavior using debug breakpoints (which we inserted via the `adb` tool that is distributed as part of the Android SDK).

Illegitimate leaks that we detected offline mainly involved (i) location information and (ii) device and user identifiers, which is consistent with the findings reported by past studies [9, 5]. We confirmed that illegitimate leaks are largely correlated with high similarity between

[3] http://appanalysis.org/download.html

private data and sink arguments, and so we fixed six distance levels for each private item: $[0,4]$ and "≥ 5". (See Section 3.1.) Finally, to avoid zero estimates for conditional probabilities while also minimizing data perturbation, we set the "smoothing" factor l in Equation 6 at 1, where the illegitimate flows we detected were in the order of several dozens per private item.

5.2 Experimental Hypotheses

In our experimental evaluation of BAYESDROID, we tested two hypotheses:

1. H1: Accuracy. Bayesian reasoning, as implemented in BAYESDROID, yields a significant improvement in leakage-detection accuracy compared to the baseline of information-flow tracking.
2. H2: Applicability. For real-life applications, BAYESDROID remains effective under relaxation of the tag-based method for detection of relevant values and its stability improves.

5.3 H1: Accuracy

To assess the accuracy of BAYESDROID, we compared it to that of TaintDroid, a state-of-the-art information-flow tracking tool for privacy enforcement. Our experimental settings and results are described below.

Subjects We applied both TaintDroid and BAYESDROID to DroidBench, an independent and publicly available collection of benchmarks serving as testing ground for both static and dynamic privacy enforcement algorithms. DroidBench models a large set of realistic challenges in leakage detection, including precise tracking of sensitive data through containers, handling of callbacks, field and object sensitivity, lifecycle modeling, inter-app communication, reflection and implicit flows.

The DroidBench suite consists of 50 cases. We excluded from our experiment (i) 8 benchmarks that crash at startup, as well as (ii) 5 benchmarks that leak data via callbacks that we did not manage to trigger (e.g., `onLowMemory()`), as both TaintDroid and BAYESDROID were naturally unable to detect leakages in these two cases. The complete list of benchmarks that we used can be found in Table 4 of Appendix B.

Methodology For each benchmark, we measured the number of true positive (TP), false positive (FP) and false negative (FN) results. We then summarized the results and calculated the overall *precision* and *recall* of each tool using the formulas below:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

	TPs	FPs	FNs	Precision	Recall	F-measure
TaintDroid	31	17	0	0.64	1.00	0.78
BAYESDROID	29	1	2	0.96	0.93	0.94

Table 1: Accuracy of BAYESDROID and TaintDroid on DroidBench

High precision implies that a technique returns few irrelevant results, whereas high recall implies that it misses only few relevant ones.

Since ideal techniques have both high recall and high precision, the F-measure is commonly used to combine both precision and recall into a single measure. The F-measure is defined as the harmonic mean of precision and recall, and is calculated as follows:

$$F\text{-}Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The value of F-measure is high only when both precision and recall are high. We thus use the F-measure for accuracy evaluation.

Results The results obtained for both TaintDroid and BAYESDROID on version 1.1 of DroidBench are summarized in Table 1 and presented in detail in Table 4. The findings reported by BAYESDROID are also publicly available.[4]

Overall, TaintDroid detects 31 true leakages while also reporting 17 false positives, whereas BAYESDROID suffers from 2 false negatives, discovering 29 of the true leakages while flagging only 1 false alarm. The recall of both TaintDroid and BAYESDROID is high (1 and 0.93, respectively) due to a low number of false-negative results. Yet the precision of TaintDroid is much lower than that of BAYESDROID (0.64 vs. 0.96), due to a high number of false positives. The overall F-measure is thus lower for TaintDroid than for BAYESDROID (0.78 vs. 0.94).

The results mark BAYESDROID as visibly more accurate than TaintDroid. To further confirm this result, we performed a two-tail McNemar test, considering 48 observations for each tool. These observations correspond to findings reported in Table 4: 31 true positives and 17 classified as false alarms. Each observation is a boolean value that represents the accuracy of the tool and is assumed to be from a Bernoulli distribution. We then checked whether the difference in accuracy is statistically significant by testing the null hypothesis that the set of 48 observations from TaintDroid are sampled from the same Bernoulli distribution as the set of 48 observations from BAYESDROID.

[4] See archive file researcher.ibm.com/researcher/files/us-otripp/droidbench.zip.

```
TelephonyManager tm =
    getSystemService(TELEPHONY_SERVICE);
String imei = tm.getDeviceId(); //source
String obfuscatedIMEI = obfuscateIMEI(imei); ...;
Log.i(imei); // sink

private String obfuscateIMEI(String imei) {
  String result = "";
  for (char c : imei.toCharArray()) {
    switch(c) {
      case '0': result += 'a'; break;
      case '1': result += 'b'; break;
      case '2': result += 'c'; break; ...; } }
```

Figure 5: Fragment from the DroidBench `ImplicitFlow1` benchmark, which applies a custom transformation to private data

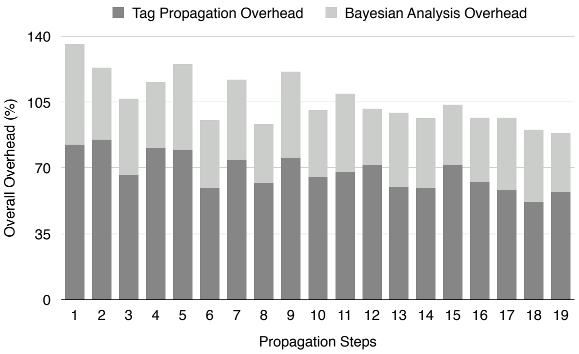

Figure 6: Overhead breakdown into tag propagation and Bayesian analysis at sink

We found that TaintDroid was accurate in 31 out of 48 cases, and BAYESDROID was accurate in 45 out of 48 cases. We built the 2×2 contingency table showing when each tool was correct and applied a two-tail McNemar test. We found a p-value of 0.001, which rejects the null hypothesis that the observations come from the same underlying distribution and provides evidence that BAYESDROID is more accurate than TaintDroid, thereby confirming H1.

Discussion Analysis of the per-benchmark findings reveals the following: First, the 2 false negatives of BAYESDROID on `ImplicitFlow1` are both due to custom (i.e., non-standard) data transformations, which are outside the current scope of BAYESDROID. An illustrative fragment from the `ImplicitFlow1` code is shown in Figure 5. The `obfuscateIMEI(...)` transformation maps IMEI digits to English letters, which is a non-standard behavior that is unlikely to arise in an authentic app.

The false positive reported by BAYESDROID, in common with TaintDroid, is on release of sensitive data to the file system, albeit using the `MODE_PRIVATE` flag, which does not constitute a leakage problem in itself. This can be resolved by performing Bayesian reasoning not only over argument values, but also over properties of the sink API (in this case, the storage location mapped to a file handle). We intend to implement this enhancement.

Beyond the false alarm in common with BAYESDROID, TaintDroid has multiple other sources of imprecision. The main reasons for its false positives are

- coarse modeling of containers, mapping their entire contents to a single taint bit, which accounts e.g. for the false alarms on `ArrayAccess{1,2}` and `HashMapAccess1`;
- field and object insensitivity, resulting in false alarms on `FieldSensitivity{2,4}` and `ObjectSensitivity{1,2}`; and more fundamentally,
- ignoring of data values, which causes TaintDroid to issue false warnings on `LocationLeak{1,2}` even when location reads fail, yielding a `Location` object without any meaningful information.

The fundamental reason for these imprecisions is to constrain the overhead of TaintDroid, such that it can meet the performance demands of online privacy enforcement. BAYESDROID is able to accommodate such optimizations while still ensuring high accuracy.

5.4 H2: Applicability

The second aspect of the evaluation compared between two versions of BAYESDROID, whose sole difference lies in the method used for detecting relevant values: In one configuration (T-BD), relevant values are detected via tag propagation. The other configuration (H-BD) uses the heuristic detailed in Section 4.2 of treating all values reachable from sink arguments (either directly or via the heap graph) up to a depth bound of *k* as relevant, which places more responsibility on Bayesian reasoning. We set *k* at 3 based on manual review of the data structures flowing into privacy sinks.

We designed a parametric benchmark application to quantify the overhead reduction imposed by the H-BD variant of BAYESDROID. The application consists of a simple loop that flows the device IMEI into a log file. Loop iterations perform intermediate data propagation steps. We then performed a series of experiments — over the range of 1 to 19 propagation steps — to quantify the relative overhead of tag propagation versus Bayesian analysis.

The results, presented in Figure 6, suggest that the overhead of tag propagation is more dominant than that of Bayesian analysis (with a ratio of roughly 2:1), even when the set of relevant values is naively over approximated. Discussion of the methodology underlying this experiment is provided in Appendix A.

In general, H-BD trades overhead reduction for accuracy. H2 then asserts that, *in practice*, the tradeoff posed by H-BD is effective. Below, we discuss our empirical evaluation of this hypothesis over real-life subjects.

Subjects To avoid evaluators' bias, we applied the following selection process: We started from the 65 Google Play apps not chosen for the training phase. We then excluded 8 apps that do not have permission to access sensitive data and/or perform release operations (i.e., their manifest does not declare sufficient permissions out of INTERNET, READ_PHONE_STATE, SEND_SMS, etc), as well as 3 apps that we did not manage to install properly, resulting in 54 apps that installed successfully and exercise privacy sources and sinks.

The complete list of the application we used is given in Table 5 of Appendix B. A subset of the applications, for which at least one leakage was detected, is also listed in Table 3.

Methodology We deployed the apps under the two BayesDroid configurations. Each execution was done from a clean starting state. The third column of both Tables 3 and 5 denotes whether our exploration of the app was exhaustive. By that we mean exercising all the UI points exposed by the app in a sensible order. Ideally we would do so for all apps. However, (i) some of the apps, and in particular gaming apps, had stability issues, and (ii) certain apps require SMS-validated sign in, which we did not perform. We did, however, create Facebook, Gmail and Dropbox accounts to log into apps that demand such information yet do not ask for SMS validation. We were also careful to execute the exact same crawling scenario under both the T-BD and H-BD configurations. We comment, from our experience, that most data leaks happen when an app launches, and initializes advertising/analytics functionality, and so for apps for which deep crawling was not possible the results are still largely meaningful.

For comparability between the H-BD and T-BD configurations, we counted different dynamic reports involving the same pair of source/sink APIs as a single leakage instance. We manually classified the findings into true positives and false positives. For this classification, we scrutinized the reports by the two configurations, and also — in cases of uncertainty — decompiled and/or reran the app to examine its behavior more closely. As in the experiment described in Section 5.3, we then calculated the precision, recall and F-measure for each of the tools.

	TPs	FPs	FNs	Precision	Recall	F-measure	Crashes
H-BD	27	1	0	0.96	1.00	0.98	12
T-BD	14	0	10	1.00	0.58	0.73	22

Table 2: Accuracy of H-BD and T-BD BayesDroid configurations

Results The results obtained for H-BD and T-BD are summarized in Table 2. Table 3 summarizes the findings reported by both H-BD and T-BD at the granularity of privacy items: the device number, identifier and location, while Table 5 provides a detailed description of the results across all benchmarks including those on which no leakages were detected. The warnings reported by the H-BD configuration are also publicly available for review.[5]

As Table 2 indicates, the H-BD variant is more accurate than the T-BD variant overall (F-measure of 0.98 vs. 0.73). As in the experiment described in Section 5.3, we further performed a two-tail McNemar test, considering 67 observations for each tool: 27 that correspond to true positives, 1 to the false positive due to H-BD and 39 to cases where no leakages were found.

We found that H-BD was accurate in 66 out of 67 cases, and T-DB was accurate in 54 out of 67 cases. Building the 2×2 contingency table and applying the two-tail McNemar test showed that the difference between the tools in accuracy is significant (with a p-value of 0.001 to reject the null hypothesis that the accuracy observations for both tools come from the same Bernoulli distribution). Moreover, H-BD has a lower number of crashes and lower runtime overhead, which confirms H2.

Discussion To give the reader a taste of the findings, we present in Figures 7–8 two examples of potential leakages that BayesDroid (both the H-BD and the T-BD configurations) deemed legitimate. The instance in Figure 7 reflects the common scenario of obtaining the current (or last known) location, converting it into one or more addresses, and then releasing only the country or zip code. In the second instance, in Figure 8, the 64-bit Android ID — generated when the user first sets up the device — is read via a call to Settings$Secure.getString(ANDROID_ID). At the release point, into the file system, only a prefix of the Android ID consisting of the first 12 digits is published.

As Table 3 makes apparent, the findings by H-BD are more complete: It detects 18 leakages (versus 8 reports by T-BD), with no false negative results and only one false positive. We attribute that to (i) the intrusive instru-

[5] See archive file researcher.ibm.com/researcher/files/us-otripp/realworldapps.zip.

App	Domain	Deep crawl?	H-BD			T-BD		
			number	dev. ID	location	number	dev. ID	location
atsoft.games.smgame	games/arcade	✓		✓	✓		✓	✓
com.antivirus	communication	✓		✓			✓	
com.appershopper.ios7lockscreen	**personalization**		✓	✓	✓			✓
com.bestcoolfungames.antsmasher	games/arcade	✓			✓			✓
com.bitfitlabs.fingerprint.lockscreen	**games/casual**			✓				
com.cleanmaster.mguard	tools	✓		✓			✓	
com.coolfish.cathairsalon	**games/casual**	✓		✓				
com.coolfish.snipershooting	**games/action**	✓		✓				
com.digisoft.TransparentScreen	entertainment	✓			✓			✓
com.g6677.android.cbaby	games/casual			✓				
com.g6677.android.chospital	games/casual			✓				
com.g6677.android.design	games/casual			✓				
com.g6677.android.pnailspa	games/casual			✓				
com.g6677.android.princesshs	games/casual			✓				
com.goldtouch.mako	news	✓		✓			✓	
15		8	1	13	4	0	4	4

Table 3: Warnings by the H-BD and T-BD BAYESDROID configurations on 15/54 top-popular mobile apps

source : private value

GeoCoder.getFromLocation(...) : [Lat: ..., Long: ...,
Alt: ..., Bearing: ..., ..., **IL**]

sink : arguments

WebView.loadUrl(...) : http://*linux.appwiz.com/*
profile/72/72_exitad.html?
p1=RnVsbCtBbmRyb2lkK29uK0VtdWxhdG9y&
p2=Y2RmMTUxMjRlYTRjN2FkNQ%3d%3d&
... LOCATION=**IL**& ...
MOBILE_COUNTRY_CODE=&
NETWORK=WIFI

Figure 7: Suppressed warning on ios7lockscreen

source : private value

Settings$Secure.getString (...) : **cdf15124ea4c7ad5**

sink : arguments

FileOutputStream.write (...) :
<?xml version='1.0' encoding='utf−8'
standalone='yes'
?><map><string
name="openudid">**cdf15124ea4c**

Figure 8: Suppressed warning on fruitninjafree

mentation required for tag propagation, which can cause instabilities, and (ii) inability to track tags through native code, as discussed below.

The T-BD variant introduces significantly more instability than the H-BD variant, causing illegal application behaviors in 21 cases compared to only 12 under H-BD. We have investigated this large gap between the H-BD and T-BD configurations, including by decompiling the subject apps. Our analysis links the vast majority of illegal behaviors to limitations that TaintDroid casts on loading of third-party libraries. For this reason, certain functionality is not executed, also leading to exceptional app states, which both inhibit certain data leaks.[6]

A secondary reason why H-BD is able to detect more leakages, e.g. in the lockscreen app, is that this benchmark makes use of the `mobileCore` module,[7] which is a highly optimized and obfuscated library. We suspect that data-flow tracking breaks within this library, though we could not fully confirm this.

At the same time, the loss in accuracy due to heuristic identification of relevant values is negligible, as suggested by the discussion in Section 4.2. H-BD triggers only one false alarm, on ios7lockscreen, which is due to overlap between irrelevant values: extra information on the `Location` object returned by a call to `LocationManager.getLastKnownLocation(...)` and unrelated metadata passed into a `ContextWrapper.startService(...)` request. Finally, as expected, H-BD does not incur false negatives.

6 Related Work

As most of the research on privacy monitoring builds on the tainting approach, we survey related research mainly in this space. We also mention several specific studies in other areas.

[6] For a technical explanation, see forum comment by William Enck, the TaintDroid moderator, at https://groups.google.com/forum/#!topic/android-security-discuss/U1fteEX26bk.

[7] https://www.mobilecore.com/sdk/

Realtime Techniques The state-of-the-art system for realtime privacy monitoring is TaintDroid [4]. TaintDroid features tolerable runtime overhead of about 10%, and can track taint flow not only through variables and methods but also through files and messages passed between apps. TaintDroid has been used, extended and customized by several follow-up research projects. Jung et al. [10] enhance TaintDroid to track additional sources (including contacts, camera, microphone, etc). They used the enhanced version in a field study, which revealed 129 of the 223 apps they studied as vulnerable. 30 out of 257 alarms were judged as false positives. The Kynoid system [20] extends TaintDroid with user-defined security policies, which include e.g. temporal constraints on data processing as well as restrictions on destinations to which data is released.

The main difference between BAYESDROID and the approaches above, which all apply information-flow tracking, is that BAYESDROID exercises "fuzzy" reasoning, in the form of statistical classification, rather than enforcing a clear-cut criterion. As part of this, BAYESDROID factors into the privacy judgment the data values flowing into the sink statement, which provides additional evidence beyond data flow.

Quantitative Approaches Different approaches have been proposed for quantitative information-flow analysis, all unified by the observation that data leakage is a quantitative rather than boolean judgment. McCamant and Ernst [13] present an offline dynamic analysis that measures the amount of secret information that can be inferred from a program's outputs, where the text of the program is considered public. Their approach relies on taint analysis at the bit level. Newsome et al. [14] develop complementary techniques to bound a program's *channel capacity* using decision procedures (SAT and #SAT solvers). They apply these techniques to the problem of false positives in dynamic taint analysis. Backes et al. [1] measure leakage in terms of indistinguishability, or equivalence, between outputs due to different secret artifacts. Their characterization of equivalence relations builds on the information-theoretic notion of entropy. Budi et al. [2] propose *kb*-anonymity, a model inspired by *k*-anonymity that replaces certain information in the original data for privacy preservation, but beyond that also ensures that the replaced data does not lead to divergent program behaviors.

While these proposals have all been shown useful, none of these approaches has been shown to be efficient enough to meet realtime constraints. The algorithmic complexity of computing the information-theoretic measures introduced by these works seriously limits their applicability in a realtime setting. Our approach, instead, enables a quantitative/probabilistic mode of reasoning that is simultaneously lightweight, and therefore acceptable for online monitoring, by focusing on relevant features that are efficiently computable.

Techniques for Protecting Web Applications There exist numerous static and dynamic approaches for preventing attacks on web applications, e.g., [23, 22, 7]. Most relevant to our work are Sekar's taint-inference technique for deducing taint propagation by comparing inputs and outputs of a protected server-side application [21] and a similar browser-resident technique developed in a subsequent study [16]. While BAYESDROID shares ideas with these approaches, it is explicitly designed for mobile devices and applications. Curtsinger et al. [3] apply a Bayesian classifier to identify JavaScript syntax elements that are highly predictive of malware. The proposed system, ZOZZLE, analyzes the application's code statically, while BAYESDROID operates dynamically and focuses on data values.

7 Conclusion and Future Work

In this paper, we articulated the problem of privacy enforcement in mobile systems as a classification problem. We explored an alternative to the traditional approach of information-flow tracking, based on statistical reasoning, which addresses more effectively the inherent fuzziness in leakage judgements. We have instantiated our approach as the BAYESDROID system. Our experimental data establishes the high accuracy of BAYESDROID as well as its applicability to real-world mobile apps.

Moving forward, we have two main objectives. The first is to extend BAYESDROID with additional feature types. Specifically, we would like to account for (i) sink properties, such as file access modes (private vs public), the target URL of HTTP communication (same domain or third party), etc; as well as (ii) the history of privacy-relevant API invocations up to the release point (checking e.g. if/which declassification operations were invoked). Our second objective is to optimize our flow-based method for detecting relevant values (see Section 3.1) by applying (offline) static taint analysis to the subject program, e.g. using the FlowDroid tool [6].

References

[1] M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *S&P*, pages 141–153, 2009.

[2] A. Budi, D. Lo, L. Jiang, and Lucia. kb-anonymity: a model for anonymized behaviour-preserving test and debugging data. In *PLDI*, pages 447–457, 2011.

[3] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security*, pages 33–48, 2011.

[4] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[5] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, pages 21–21, 2011.

[6] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, 2014.

[7] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, pages 177–187, 2011.

[8] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow permissions for android. In *ASE*, pages 652–657, 2013.

[9] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, pages 639–652, 2011.

[10] J. Jung, S. Han, and D. Wetherall. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *SPSM*, pages 45–50, 2012.

[11] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *USENIX Security*, pages 113–130, 2013.

[12] G. Lowe. Quantifying information flow. In *CSFW*, pages 18–31, 2002.

[13] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.

[14] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85, 2009.

[15] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[16] Riccardo Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *ASIACCS*, pages 5–5, 2012.

[17] J. Piskorski and M. Sydow. String distance metrics for reference matching and search query correction. In *BIS*, pages 353–365, 2007.

[18] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODAPSY*, pages 209–220, 2013.

[19] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[20] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *WISTP*, pages 208–223, 2012.

[21] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[22] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

[23] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[24] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[25] Bernard L Welch. The generalization of student's problem when several different population variances are involved. *Biometrika*, 34(1–2):28–35, 1947.

[26] D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang. Privacy revelations for web and mobile apps. In *HotOS*, pages 21–21, 2011.

[27] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, pages 1043–1054, 2013.

Benchmark	Algorithm	TPs	FPs	FNs
ActivityCommunication1	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle1	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle2	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle4	BayesDroid	1	0	0
	TaintDroid	1	0	0
Library2	BayesDroid	1	0	0
	TaintDroid	1	0	0
Obfuscation1	BayesDroid	1	0	0
	TaintDroid	1	0	0
PrivateDataLeak3	BayesDroid	1	1	0
	TaintDroid	1	1	0
AnonymousClass1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ArrayAccess1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ArrayAccess2	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
HashMapAccess1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
Button1	BayesDroid	1	0	0
	TaintDroid	1	0	0
Button3	BayesDroid	2	0	0
	TaintDroid	2	0	0
Ordering1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
RegisterGlobal1	BayesDroid	1	0	0
	TaintDroid	1	0	0
DirectLeak1	BayesDroid	1	0	0
	TaintDroid	1	0	0
FieldSensitivity2	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
FieldSensitivity3	BayesDroid	**1**	**0**	**0**
	TaintDroid	**1**	**0**	**0**
FieldSensitivity4	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ImplicitFlow1	BayesDroid	**0**	**0**	**2**
	TaintDroid	**2**	**0**	**0**
InheritedObjects1	BayesDroid	1	0	0
	TaintDroid	1	0	0
ListAccess1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
LocationLeak1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
LocationLeak2	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
Loop1	BayesDroid	1	0	0
	TaintDroid	1	0	0
Loop2	BayesDroid	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle1	BayesDroid	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle3	BayesDroid	1	0	0
	TaintDroid	1	0	0
MethodOverride1	BayesDroid	1	0	0
	TaintDroid	1	0	0
ObjectSensitivity1	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ObjectSensitivity2	BayesDroid	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
Reflection1	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection2	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection3	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection4	BayesDroid	1	0	0
	TaintDroid	1	0	0
SourceCodeSpecific1	BayesDroid	5	0	0
	TaintDroid	5	0	0
StaticInitialization1	BayesDroid	1	0	0
	TaintDroid	1	0	0
Total	BayesDroid	29	1	2
	TaintDroid	31	17	0

Table 4: Detailed summary of the results of the H1 experiment described in Section 5.3

A Overhead Measurement: Methodology

To complete the description in Section 5.4, we now detail the methodology governing our overhead measurements. The behavior of the benchmark app is governed by two user-controlled values: (i) the length ℓ of the source/sink data-flow path (which is proportional to the number of loop iterations) and (ii) the number m of values reachable from sink arguments.

Based on our actual benchmarks, as well as data reported in past studies [23], we defined the ranges $1 \leq \ell \leq 19$ and $1 \leq m \leq 13 = \Sigma_{n=0}^{2} 3^n$. We then ran the parametric app atop a "hybrid" configuration of BayesDroid that simultaneously propagates tags and treats all the values flowing into a sink as relevant. For each value of ℓ, we executed the app 51 times, picking a value from the range $[0,2]$ for n uniformly at random in each of the 51 runs. We then computed the average overhead over the runs, excluding the first (cold) run to remove unrelated initialization costs. The stacked columns in Figure 6 each correspond to a unique value of ℓ.

B Detailed Results

Table 4 summarizes the results of the H1 experiment described in Section 5.3. For each of the benchmarks, it specifies the number of true-positive, false-positive and false-negative findings for the compared tools, BayesDroid and TaintDroid. The benchmarks on which the tools differ are highlighted for convenience.

Similarly, Table 5 summarizes the results of the H2 experiment described in Section 5.4. The first two columns of Table 5 list the applications and their respective domain, and the third column denotes whether crawling was exhaustive. Then, the number of crashes, true-positive, false-positive and false-negative findings are reported for both the H-BD and the T-BD variants of BayesDroid.

In Section 5.4, we describe an experiment designed to evaluate our Bayesian analysis in "pure" form, i.e. without the support of information-flow tracking to detect relevant values. To make our description of this experiment complete, we include Table 5, which provides a detailed summary of the results of this experiment across all benchmarks (including ones on which no leakages were detected). For comparability between the H-BD and T-BD configurations, we count different dynamic reports involving the same pair of source/sink APIs as a single leakage instance.

App	Domain	Deep crawl?	H-BD				T-BD			
			Crashes	TPs	FPs	FNs	Crashes	TPs	FPs	FNs
air.au.com.metro.DumbWaysToDie	games/casual		✓	0	0	0	✓	0	0	0
at.nerbrothers.SuperJump	games/arcade			0	0	0	✓	0	0	0
atsoft.games.smgame	**games/arcade**	✓		**4**	**0**	**0**		**4**	**0**	**0**
com.antivirus	**communication**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.appershopper.ios7lockscreen	**personalization**			**5**	**1**	**0**		**3**	**0**	**3**
com.applicaster.il.hotvod	entertainment	✓		0	0	0		0	0	0
com.appstar.callrecorder	tools			0	0	0		0	0	0
com.awesomecargames.mountainclimbrace_1	games/racing		✓	0	0	0	✓	0	0	0
com.bestcoolfungames.antsmasher	**games/arcade**	✓		**2**	**0**	**0**		**2**	**0**	**0**
com.bigduckgames.flow	games/puzzles			0	0	0	✓	0	0	0
com.bitfitlabs.fingerprint.lockscreen	**games/casual**			**2**	**0**	**0**		**0**	**0**	**1**
com.channel2.mobile.ui	news	✓		0	0	0		0	0	0
com.chillingo.parkingmaniafree.android.rowgplay	games/racing		✓	0	0	0	✓	0	0	0
com.cleanmaster.mguard	**tools**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.coolfish.cathairsalon	**games/casual**	✓		**2**	**0**	**0**	✓	**0**	**0**	**1**
com.coolfish.snipershooting	**games/action**	✓		**2**	**0**	**0**	✓	**0**	**0**	**1**
com.cube.gdpc.isr	health & fitness		✓	0	0	0	✓	0	0	0
com.cyworld.camera	photography			0	0	0		0	0	0
com.devuni.flashlight	tools	✓		0	0	0		0	0	0
com.digisoft.TransparentScreen	**entertainment**	✓		**2**	**0**	**0**		**2**	**0**	**0**
com.domobile.applock	tools	✓		0	0	0		0	0	0
com.dropbox.android	productivity	✓		0	0	0	✓	0	0	0
com.ea.game.fifa14_row	games/sports			0	0	0	✓	0	0	0
com.ebay.mobile	shopping			0	0	0		0	0	0
com.facebook.katana	social	✓		0	0	0		0	0	0
com.facebook.orca	communication			0	0	0		0	0	0
com.g6677.android.cbaby	**games/casual**			**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.chospital	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.design	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.pnailspa	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.princesshs	**games/casual**			**1**	**0**	**0**	✓	**0**	**0**	**1**
com.gameclassic.towerblock	games/puzzles	✓		0	0	0	✓	0	0	0
com.gameloft.android.ANMP.GloftDMHM	games/casual			0	0	0		0	0	0
com.game.fruitlegendsaga	games/puzzles			0	0	0		0	0	0
com.gau.go.launcherex	personalization			0	0	0		0	0	0
com.glu.deerhunt2	games/arcade		✓	0	0	0	✓	0	0	0
com.goldtouch.mako	**news**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.goldtouch.ynet	news	✓		0	0	0		0	0	0
com.google.android.apps.docs	productivity			0	0	0		0	0	0
com.google.android.apps.translate	tools			0	0	0		0	0	0
com.google.android.youtube	media & video			0	0	0		0	0	0
com.google.earth	travel & local		✓	0	0	0	✓	0	0	0
com.halfbrick.fruitninjafree	games/arcade			0	0	0		0	0	0
com.halfbrick.jetpackjoyride	games/arcade	✓		0	0	0		0	0	0
com.icloudzone.AsphaltMoto2	games/racing			0	0	0	✓	0	0	0
com.ideomobile.hapoalim	finance			0	0	0		0	0	0
com.imangi.templerun2	games/arcade		✓	0	0	0	✓	0	0	0
com.kiloo.subwaysurf	games/arcade		✓	0	0	0	✓	0	0	0
com.king.candycrushsaga	games/arcade		✓	0	0	0	✓	0	0	0
com.sgiggle.production	social			0	0	0		0	0	0
com.skype.raider	communication			0	0	0		0	0	0
com.UBI.A90.WW	games/arcade			0	0	0		0	0	0
com.viber.voip	communication			0	0	0		0	0	0
com.whatsapp	communication			0	0	0		0	0	0
Total		17	12	27	1	0	22	14	0	10

Table 5: Detailed summary of the results of the H2 experiment described in Section 5.4