LUNA: Quantifying and Leveraging Uncertainty in Android Malware Analysis through Bayesian Machine Learning

Michael Backes
CISPA, Saarland University /
Max Planck Institute for Software Systems, Germany
backes@mpi-sws.org

Mohammad Nauman
Max Planck Institute for Software Systems, Germany
mnauman@mpi-sws.org

***Abstract*—Android's growing popularity seems to be hindered only by the amount of malware surfacing for this open platform. Machine learning algorithms have been successfully used for detecting the rapidly growing number of malware families appearing on a daily basis. Existing solutions along these lines, however, have a common limitation: they are all based on classical statistical inference and thus ignore the concept of uncertainty invariably involved in any prediction task. In this paper, we show that ignoring this uncertainty leads to incorrect classification of both benign and malicious apps. To reduce these errors, we utilize Bayesian machine learning – an alternative paradigm based on Bayesian statistical inference – which preserves the concept of uncertainty in all steps of calculation. We move from a black-box to a white-box approach to identify the effects different features (such as sensitive resource usage, declared activities, services and intent filters etc.) have on the classification status of an app. We show that incorporating uncertainty in the learning pipeline helps to reduce incorrect decisions, and significantly improves the accuracy of classification. We achieve a false positive rate of 0.2% compared to the previous best of 1%. We present sufficient details to allow the reader to reproduce our results through openly available probabilistic programming tools and to extend our techniques well beyond the boundaries of this paper.**

Keywords: Malware analysis, Uncertainty, Bayesian Machine Learning

1. Introduction

Android is an open source project not only in terms of the source code but also in terms of the whole ecosystem. Creating a developer account is the only prerequisite in the app development and deployment cycle for making an app publicly available. This open nature of Android has facilitated a rapid pace of innovation, but it has also led to creation and widespread deployment of malware [1], [2]. According to the 2015 Internet Security Threat Report from Symantec, which analyzed 6.3 million apps, 1 million of these apps were classified as malware, belonging to a total of 277 families [1]. Fighting such a wide range of malware is an uphill battle for security researchers.

Recently, it has been realized that machine learning can help restore the balance of power by enabling automated classification of apps into benign ones and malicious ones based on different feature types [3], [4]. It has been shown empirically that it is possible to achieve a high detection rate using these features [5]. Previously, a detection rate of 84% has been reported based on the requested permissions alone [6], whereas much higher rates have been achieved by incorporating other features [3].

However, even though these machine learning algorithms work well in general, their black box nature often disguises the underlying reasons of *how* their success is achieved. The general pipeline of this black box approach is that a model is created for the problem at hand – classification of an app as being benign or malicious – and the learning algorithm *fits* the model's parameters to get the best results on the dataset. How the different features affect the outcome has not yet been studied in detail.

A second facet of this issue is that all the machine learning algorithms that have been applied on Android malware datasets follow *classical statistical techniques*, which discard all uncertainty information associated with the analysis. The result of the prediction is always 'benign' or 'malicious' with no level of uncertainty associated with it. We argue that uncertainty is an important aspect of any prediction [7] that should be reported and can help improve analysis.

An alternative to this classical statistical inference approach is the *Bayesian statistical inference*. This school of thought calls for keeping track of uncertainties in all steps of the prediction pipeline and enables us to utilize it to reason with *beliefs* about events. We begin with some *prior* belief [8] about a fact – in our case, the target app being benign or malicious. This "belief" can be one of complete uncertainty, assigning equal probability to the app belonging to either of the two classes. We then look at the facts (provided to us in the form of the dataset of malicious apps) and update our beliefs, thus leading to *posterior* probability about the fact. The computational costs and cognitive load involved in going from the prior to the posterior have been the primary reason for a low adoption rate of Bayesian inference [9]. However, in the recent past, massive advances have been made in the field of *probabilistic programming*, making these analyses not only feasible but quite efficient [10].

DOI 10.1109/EuroSP.2017.24

Our Contributions. In this paper, we present a Bayesian inference-based analysis pipeline – dubbed *LUNA* (Leveraging UNcertainty in Android malware analysis) – for analyzing Android malware datasets through the use of probabilistic programming concepts and tools. We describe an uncertainty-preserving inference model for applying machine learning on a large-scale Android malware dataset and then analyze the results as well as the learned parameters of the model. We outline the tools used and the necessary background to enable the reader to get up to speed with Bayesian inference. Further, we incorporate the uncertainty preserved by Bayesian inference to improve upon the state-of-the-art classification models for Android. In a nutshell, our contributions in this paper can be summarized as follows:

1) We develop a Bayesian machine learning model of Android malware classification that incorporates uncertainty in analysis *and* prediction; our analysis of learned parameters includes interesting and counter-intuitive findings about critical resource usage.
2) We show through empirical results that ignoring uncertainty leads to false positives in the analysis.
3) We improve upon the existing detection accuracy reported by past works and identify some real-world malicious applications which would have been identified *incorrectly* as false positives if not for the application of Bayesian inference.

Paper structure. Below, we first provide some requisite background including Bayesian inference and malware anslysis on Android in Section 2 Our Bayesian model for Android malware classification is presented in Section 3. Incorporation of uncertainty to improve prediction in base classification algorithms is provided in Section 4. Details of our experiments are provided in Section 5 along with a descriptive analysis of the dataset in Section 6. Section 7 reports the improvements we have made over state-of-the-art in malware analysis on Android and Section 8 concludes the paper.

2. Background

2.1. Android Malware Analysis

Android's software stack was designed as a layered architecture. The lowest layer is that of the Linux kernel, and all subsequent layers increase the level of abstraction while also enforcing security policies. The highest layer in the stack is that of applications, including those developed by the manufacturer and carrier as well as by third parties. Applications interact with the *application framework layer* through intents or API calls, which are protected by pre-defined permissions. All sensitive resources such as network access, contacts and the ability to send and receive text messages are protected through different permissions [11].

If an application might need to use a resource at any point during its lifetime, the developer of the app must request the permission associated with that resource at *compile* time. When a user installs the application, she is presented with a list of these permissions and must agree to grant all these permissions in order to be able to install the application. The most recent version of Android (6.0 Marshmallow) allows users to restrict some of these permissions after installation of the app. Regardless, all applications must still declare, at compile time, all the permissions they might need during run-time.

These permissions are quite fine-grained, and as such give insights into the workings of the requesting application. One of the first studies of this phenomenon was Kirin [12], which defined *rules* to match undesired combinations of requested permissions and warn users about them. For instance, if an app requested contacts information *and* the internet permission, the installer would raise a red flag since the app would have the ability to steal the user's contacts data. Another view into the world of requested permissions was provided by Barrera et al. [13]. They used *Self-Organizing Maps* [14] to gain insights into permissions requested by apps belonging to different categories in the Play Store.

Deeper dynamic analyses beyond the requested permissions have also been carried out by several studies including TaintDroid [15], which used taint tracking to discover possible information leakage. Studies along these lines were carried out by DroidScope [16] and FlowDroid [17] among several others [18], [19], [20]. However, these frameworks are difficult to implement and analyze. Moreover, they do not scale well to large-scale datasets and are infeasible to deploy on mobile devices [3].

Machine learning techniques seem to be a more viable solution in this scenario. Several attempts have been made to classify Android apps by training machine learning models on available datasets of malware. One of the first attempts to make such a dataset available was by the Android Malware Genome Project [21]. The researchers in this project spent around a year and a half collecting samples which have since been used in several studies. For instance, Peng et al. [4] introduced risk scores based on several variations of naive Bayes and achieved an area under the curve of 95.3% using requested permissions from this dataset. BayesDroid employs a slightly different approach by using naive Bayes to classify *information flows* of non-malicious apps as legitimate or illegitimate [22]. The word 'naive' in the model's name comes from the assumption of conditional independence among features – a simplification that is seldom true in real-world problems [23]. Similarly, Garcia et al. [5] reported an obfuscation-resilient technique for accurate classification of Android malware. They too base their work on both static analysis along with applications' meta data.

A recent and highly successful variation of these efforts is Drebin [3]. It analyzed different features of Android applications including their requested permissions, accessed network URLs, static analysis reports and several other metrics to fit a *Support Vector Machine (SVM)* classifier. This achieved a detection rate of 94% with an error rate of only 1%. Dash et al. [24] use fine-grained dynamic analysis and virtual machine introspection for placing malware into specific families using SVMs. Drebin and the work by Peng et al. [4] are closest to the concepts presented in this paper. However, both of these, and indeed all the other works using machine learning for classification of Android apps to date, have ignored the uncertainty invariably present in the learned model parameters. We, on the other hand utilize the unique strengths of Bayesian inference by quantifying and leveraging uncertainty in our model. The next section gives a brief review of Bayesian inference and the part played by uncertainty in this framework.

2.2. Bayesian Inference

The primary difference between *Bayesian* inference and *classical* statistical inference is that Bayesian inference preserves uncertainty in calculation. From another perspective, classical models of statistics have the interpretation that probability is the long-term frequency of events, e.g. frequency of cars hitting pedestrians. However, such a view becomes difficult to maintain in cases where the events do not occur more than once. For instance, consider the probability of a team winning the 2018 Soccer World Cup. This event will only occur once, and hence there is no frequency associated with it. The so-called ‘frequentist school of thought’ [23] works around this limitation by defining probability as the frequency that this event will occur from among *all possible realities*.

Bayesian interpretation of probability, on the other hand, is much more intuitive. In this viewpoint, probability is a measure of *belief*. In other words, it is the level of confidence in the occurrence of an event. A probability of 1 means that an individual is certain about that event occurring; one of 0.5 means that there is a lot of uncertainty involved in the belief. The inclusion of an individual in this interpretation means that the individual's prior knowledge about the world also plays a part in the probability – this knowledge is known as the *prior*. After an individual sees more data, she can update her belief to another probability which is known as the *posterior* [23]. Hence, from a Bayesian perspective, every event has an uncertainty associated with it and this uncertainty is carried forward using the rules of probability through all steps of the calculation. We now define this process using formal semantics.

2.2.1. Probability Refresher. We provide a very brief review of some of the concepts of probability below in order to bring the reader up to speed with the terminology. For details, we refer the reader to [23]. A *joint distribution* $p_{X,Y}(x, y)$ gives the probability of random variables X and Y taking on the values x and y respectively. The concept of joint probability is different from that of *conditional probability*, which describes the probability of x happening given that y is already known to have happened. It is defined as: $p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)}$. Re-arranging and expanding this equation leads to *Bayes' rule*:

$$p_{Y|X}(y|x) = \frac{p_{X|Y}(x|y)p_Y(y)}{\sum_y p_{X,Y}(x,y)}$$

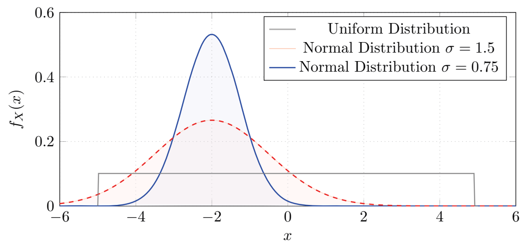


Figure 1: Posterior Distributions from Priors

In order to compute the probabilities of a *continuous* random variable X, we need the *probability density function*, $f_X(x)$, that must satisfy the following property: $f_X(x) \geq 0$, $\int_{-\infty}^{\infty} f_X(x)dx = 1$.

Note that it is not possible to assign a specific value to the probability $p_X(x = a)$ where $x, a \in \mathbb{R}$ since the integral for a specific point is undefined. This will be of importance in our inference later on, as we will not be able to get a *point estimate* for the parameters we are interested in.

In order to describe real-world data and their corresponding classes, we denote the observed values of real-world inputs as D and assume that they are generated by some random process with parameters θ. In our model, the real-world data is represented by discrete variables, whereas θ values are continuous. From a variation of Bayes' rule with input variables as continuous and output variable as discrete, we have:

$$f_{\Theta|D}(\theta|d) = \frac{p_{D|\Theta}(d|\theta)f_\Theta(\theta)}{\int_\theta p_{D|\Theta}(d|\theta)f_\Theta(\theta)d\theta}$$

The definition of a posterior distribution for θ is a unique characteristic of Bayesian inference. In classical machine learning, the model parameter θ is an unknown *constant*. It has a single true value, – computed using the maximum likelihood estimate (MLE) – which is a point estimate without a notion of uncertainty. In Bayesian machine learning, θ is a *random variable* and thus has an uncertainty associated with it. If we have data for specific values of the input, our uncertainty will decrease. In areas where there is insufficient data, we will be uncertain about the values of θ.

It is an evidence of the strength of Bayesian inference that simply by understanding the above equation and its implications, one can perform all sorts of complex analyses on different types of processes. At each iteration, more data is seen and posterior beliefs about the model's parameter can be formed. This then forms the new prior and can be used as a basis of future experiments. In order to get a sense of the result that can be expected from the above, consider the scenario in Figure 1. In the figure, we make no assumption about the theta prior to beginning our experiment. We therefore set the prior to a uniform distribution ($\theta \sim$ *Uniform*$(k = 0.1)$). After having seen some data points, our belief about θ changes to that of a normal distribution with mean -2 and standard deviation of 1.5, i.e. $\theta \sim \mathcal{N}(\mu = -2, \sigma^2 = 2.25)$. Notice that we are carrying the uncertainty within the posterior distribution. This *posterior* then forms our new *prior*. From here on, we can assume that θ is normally distributed and gather further evidence about it. Looking at new data, we update our posterior and conclude that distribution of θ is indeed centered around -2. The new data only reduced our uncertainty, which is depicted by the thinness of the peak. The final value of θ is now $\theta \sim \mathcal{N}(\mu = -2, \sigma^2 = 0.56)$. It is this incremental process through which we can not only preserve uncertainty information but also increase our confidence through repeated experiments with minimal overhead. This is in sharp contrast to the usual 'frequentist' models, where each experiment must incorporate all information anew [8].

2.2.2. Posterior Distributions.

The complication in use of Bayesian machine learning comes from the integral that has to be computed in order to arrive at a solution for the marginal. If θ is a high-dimensional variable, it quickly becomes intractable to compute the integral analytically or numerically [23]. However, owing to the recent advances in computation and in techniques of estimating marginals, using Bayesian inference has become very feasible [25]. Below, we describe one of the most popular and powerful methods of computing posteriors, *Markov Chain Monte Carlo (MCMC)* [26], which has a strong foothold in terms of available tools and precise algorithms.

2.2.3. Markov Chain Monte Carlo (MCMC) Methods.

Monte Carlo methods of simulation are fairly well-known. They are simply the process of drawing *samples* from a distribution. Given a large number of samples, it is possible to generate a practically usable picture of the distribution. However, as mentioned above, there are cases where such sampling isn't feasible (especially in random variables of high dimensions). One way to simulate such sampling is to use *Markov chains* [26].

A Markov chain is a type of process in which the outcome of a given experiment can affect that of the next. The most important property of a Markov chain is that (under certain conditions) it is memory-less, i.e. the probability of the next state depends only on the current state and not on how we arrived at that state [26]. This is given by the *Markov property*:

$$\begin{aligned} &p_{X_{n+1}}(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \ldots, X_0 = i_0) \\ &\quad = p_{X_{n+1}}(X_{n+1} = j | X_n = i) \end{aligned}$$

In other words, the computation does not involve the path to a state, and the starting point does not matter. This greatly improves the performance of computation. This property is used by MCMC methods in cases where it is not possible to draw random samples directly. In general, the way MCMC methods work is to start at a random location and *explore* the space of the random variable's distribution. It moves from the starting point in a random direction. It then computes whether this new point satisfies some conditions, which are specified through the observed data. If the new point matches the observations, it is selected as a sample; otherwise it is discarded. This way, the MCMC algorithm traverses the terrain of the probability density function of the variable being sampled and allows generation of a large number of representative samples without attempting to sample directly from the distribution.

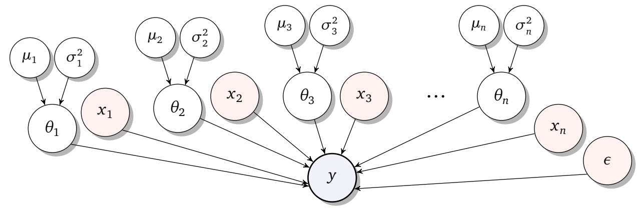


Figure 2: Graphical Model for Android Permission Analysis

Below, we describe how we have used these concepts for applying Bayesian inference to model and analyze the Android malware dataset.

3. Bayesian Machine Learning for Android Permissions

In order to compute and reason with uncertainties associated with classification of Android apps, we begin by defining a simple logistic regression model. This logistic regression model captures the relationship between the features of an application and the label of the application – benign or malicious. In order to keep the focus on the model itself, we begin by formally defining the model first and later turn our attention to how we gathered the real-world data to train the model.

In *generalized linear model (GLM)* [26] formalism, the logistic regression relationship is defined as:

$$\begin{aligned} h_\theta &= \theta X + \epsilon \\ Y &= \delta(h_\theta) \end{aligned}$$

where Y and X are data inputs and class labels respectively, θ terms are called the *weights* or *parameters* of the model, ϵ is the measurement noise – assumed to be normally distributed – and δ is the *activation function*. For activation, we use the `invlogit` function given by: $\delta(z) = e^z(1 + e^z)^{-1}$

The Bayesian reformulation of this model is in terms of random variables and their interplay.

$$Y \sim \textit{Bernoulli}(\delta(h_\theta))$$

Hence, Y is a Bernoulli random variable with probability of $\delta(h_\theta)$. The θ terms are themselves expected to have a normal distribution since they may take on positive or negative values:

$$\theta_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$$

Notice that in this new formulation, we only have a weak assumption about the prior of the model's parameters i.e. that they are normally distributed. The parameters of this distribution itself are treated as unknowns and it is possible to retrieve their posterior probabilities the same way as we would compute the posterior of Y. Assuming that parameter values are normally distributed around 0 acts like a regularizer.

More importantly, our model does not assume independence between the features X_i. The intractability arising from removal of this assumption is taken care of through the use of MCMC.

After having computed the Maximum Apriori Point (MAP) using the efficient Powell's method [27], we initialize our MCMC sampling algorithm (NUTS [25]) using this MAP and sample from the posterior distribution of the output variable Y. We refer the reader to the very interesting original NUTS paper [25] for details. Here, it suffices to say that NUTS is a state-of-the-art and extremely efficient sampling technique. It has few parameters that need tuning and it can find optimum values for these without requiring expert intervention. This makes it an extremely easy-to-use black-box for performing sampling of intricate spaces such as the one required here.

NUTS begins exploring the space of the posterior distribution from the MAP point and collects samples which match the observed values, i.e. the real-world dataset of Android application features and maliciousness labels. As it progresses, it collects samples which have a higher and higher probability of belonging to the posterior distribution, until it 'converges'. The concept of convergence in MCMC means that the samples are within the range of uncertainty threshold dictated by the posterior distribution. This is not a point estimate and as such does not converge to a single point as is the case in traditional machine learning techniques. Instead, a converged MCMC returns a *distribution* that encodes the uncertainty associated with the learned weights of the parameters.

After the collection of posterior distribution samples for the model's parameters, it is possible to make predictions about the class of new data points through a technique similar to the *posterior predictive check* method [23]. The strength of MCMC is that it returns a complete trace of the posterior distribution of the model's parameters. Hence, the posterior of the output variable (before applying a *prediction threshold*) can be computed simply using the form: $s^{(i)} = \delta(\theta^{(i)}X)$ where $\theta^{(i)}$ are different samples from the posterior distribution of θ, and i is the number of samples drawn by NUTS. According to the law of large numbers, as i increases, the distribution represented by $s^{(i)}$ reaches the true posterior distribution of the output variable s. Note that s is normally distributed as opposed to the Bernoulli distributed Y.

The uncertainty in the posterior distribution in Bayesian inference is a much cleaner concept than that of confidence intervals [8] in the frequentist school of thought. Given the posterior distribution, a Bayesian confidence interval of α is simply the range around the mean that gathers $(1-\alpha)$ of the total mass.

The interpretation of this uncertainty is important for our prediction technique. Given a contiguous interval $[\phi_{x1}, \phi_{x2}]$ and an α, the probability that the actual value ϕ_x of the given variable x lies within the range is $(1-\alpha)$.

For the calculation of uncertainty in s, i.e. the values of ϕ_{s1} and ϕ_{s2}, we use *symmetric confidence interval*, which ensures that there is an equal amount of probability mass on either side of the mean, instead of trying to make the two ends of the range equidistant from the mean:

$$\int_{-\infty}^{\phi_{s1}} f_{\Phi_s}(\phi_s|D;\theta)d\phi_s = \int_{\phi_{s2}}^{\infty} f_{\Phi_s}(\phi_s|D;\theta)d\phi_s = \frac{\alpha}{2}$$

The larger the range $[\phi_{s1}, \phi_{s2}]$ – called the *highest probability density (HPD) interval* – the more uncertain we will be about the true value of s. This corresponds to the posterior probability being wider (cf. Figure 1). Through these intervals, we get an easy way to visualize the uncertainty in the model parameters as well as in our predictions of the class of test points. Below, we describe how we have incorporated the concept of uncertainty to improve prediction accuracy as compared to non-Bayesian models.

4. Incorporating Uncertainty for Model Prediction

We defined our model using the linear relationship: $h_\theta(X) = \theta X$ and the logistic activation function: $s = \delta(h_\theta(X))$. We first ran our logistic regression model without the concept of uncertainty to get weights for the parameters. We split our dataset into training and test sets as per the standard practice. The training set was used to learn the parameter weights and the test set was used for evaluation of the model's goodness-of-fit.

The metrics most commonly used for this purpose are *area under the curve (AUC)* of a *receiver operating characteristics (ROC)* curve as well as the triad of *accuracy* (number of correct answers), *precision* (how many of the samples identified by the model as malware were actually malware) and *F score*. F score is the geometric mean of precision and *recall* (of all the malware samples in the dataset, how many were identified as being such). It penalizes a model with either a very low precision or very bad recall. All of these were calculated using the concept of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN), where positive is equivalent to being labeled malicious and negative to benign. Details of the results of this step are provided in Section 7.

In order to extend the model with the concept of uncertainty learned from Bayesian inference, we introduce two new classes of results: *Uncertain Positive (UP)* – where the true class of the data point was positive but the prediction was uncertain about the label – and *Uncertain Negative (UN)* – where the true class was negative but the model failed to provide a label as output.

$$\text{Predicted :}\ \begin{array}{c} \\ \text{P} \\ \text{N} \\ \text{U} \end{array} \begin{array}{cc} \text{Actual Positive} & \text{Actual Negative} \\ \left[\begin{array}{c} \text{TP} \\ \text{FN} \\ \text{UP} \end{array}\right. & \left.\begin{array}{c} \text{FP} \\ \text{TN} \\ \text{UN} \end{array}\right] \end{array}$$

We define the concept of *coverage* as:

$$\text{coverage} = 1 - \frac{\text{UP} + \text{UN}}{\|\text{data points}\|}$$

which is the number of concrete decisions as a ratio of the total number of test points. Ideally, the coverage should be 1, but due to uncertainty in the results, it will fall within the range $[0, 1]$. We propose two methods for augmenting prediction with the concept of uncertainty.

4.1. Naïve Discard (M1)

For the first method – labeled *M1* in the following text – we return the value of 'uncertain' for any data points which have a large uncertainty in their posterior distribution regardless of the value of the point's mean. The rule for prediction is given by:

$$y = \begin{cases} U & |\phi_{s2} - \phi_{s1}| > \lambda \\ P & h \geq \eta \\ N & otherwise \end{cases}$$

where $[\phi_{s1}, \phi_{s2}]$ is the HPD interval for s, λ is the uncertainty threshold and η is the prediction threshold.

4.2. HPD Augmented Prediction Threshold (M2)

As an alternative to the naïve method, we augment the prediction threshold with the upper and lower values of the HPD range. For the sake of simplicity, we refer to this method as *M2*. The rule for prediction in this method is given as:

$$y = \begin{cases} P & (h \geq \eta) \wedge (\eta - \phi_{s1} < \lambda) \\ N & (h < \eta) \wedge (\phi_{s2} - \eta < \lambda) \\ U & otherwise \end{cases}$$

The first rule dictates that the mean of the predicted value must be higher than the prediction threshold η and the lower limit of the HPD range must not be farther away than λ from η. This ensures that the combined effect of the mean and uncertainty will not be too far away to the left of the prediction cut-off point. Similarly, the second rule ensures that in case of a negative prediction, the mean and the upper HPD limit will not be too far off to the right of η. If neither of these conditions are met, the uncertainty is too large to predict either way and we return the decision of 'uncertain'.

For each of these two methods, we find the optimum values for λ and η using *grid search*, i.e. we calculate different goodness-of-fit metrics for different values of λ and η drawn from range $[0, 1]$ and pick the best result. The results of these experiments are discussed in Section 7.

5. Experimental Setup

In order to reason with Android malware, we required a large dataset to get high values of certainty for the probability distributions in different steps. We started with the base collection of 5000 malicious Android applications kindly made available to us by the Drebin project [3]. This set includes the Android Malware Genome project [21] samples as well as others identified as malicious by VirusTotal. We augmented this dataset with about 1800 malicious apps from the Contagio Mobile site [28] bringing the total number of malicious apps to 6737. We picked 7500 benign applications from the Drebin dataset at random bringing the total to 14,237. We intentionally kept benign applications to a small number in order not to skew the final dataset towards the benign class. Moreover, our initial experiments showed that increasing this number significantly hampered the time taken to perform analysis without improving detection rates.

The features of each application included the permissions requested in its manifest as well as those which were found through static analysis to actually have been used by the app code. Other metrics included the intents filtered by the target, the activity list in its manifest, API calls raised in the code and services registered. The Drebin dataset included the URLs opened by the apps as well but we omitted these as they doubled the number of features for each app without improving the detection rate.

We used standard 10-fold cross validation on this dataset to perform training and testing. Posterior probability distributions of the model's parameters were calculated using the training set and posterior predictive checks were carried out on the test set. For the MCMC run, we chose a sample size of 10,000 which led to a convergence of all the unknown variables. Recall, though, that convergence does not mean the removal of all uncertainties from the values. It simply means that the posterior distribution's terrain stopped changing.

5.1. Environment and Tools

In order to run the analysis in an interactive and powerful environment, LUNA uses the popular Python language along with several data science packages. Two of the most important ones were: Theano [29], which generates highly optimized implementations of mathematical functions involving high-dimensional tensors and PyMC3 [9]. This is a Python implementation of probabilistic programming in general and MCMC algorithms in particular and has a very easy to use API for specifying Bayesian models and then performing inference on them. PyMC3 code that can be used to reproduce our experiments can be seen in Figure 3.

Running MCMC simulations is a computation-intensive task. Training the model on our dataset for a large number of samples consumes a lot of time. It was therefore not feasible to run the simulations on a commodity desktop PC. We ran our MCMC sampling algorithms on an Intel Core i7-4770K fourth-generation processor with 8 cores clocked at 3.5GHz and with 32GB of on-board memory. The sampling was done on the NVIDIA Tesla K40c GPU with 2,880 cores thanks to the transparent interface provided by Theano. Note, however, that after the training is done and posterior distributions of our parameters are available, the posterior predictive check that has to be carried out for each individual target application is a fairly straightforward task. It only takes a few microseconds with optimized and vectorized implementations of matrix multiplication easily available in all mainstream languages. These predictive steps can be easily deployed even on a low-end Android smartphone.

```python
import theano.tensor as t
from pymc3 import *

# n_predictors = number of input features/permissions
# predictors   = tensor of feature values
# Y            = observed Y values

def tinvlogit(x):
    return t.exp(x) / (1 + t.exp(x))

model = Model()
with model:      # all model-definitions need to be within the model context

    # Priors for unknown model parameters
    theta   = Uniform('theta', lower=-10, upper=10, shape=(1, n_predictors))
    epsilon = Normal('alpha', mu=0, sd=10, shape=(1, n_predictors))

    # expected probability
    p = tinvlogit(sum(theta * predictors, 1) + epsilon)

    # Likelihood (sampling distribution) of observations
    Y_obs = Bernoulli('Y_obs', p=p, observed=Y)
```

Figure 3: Bayesian Machine Learning Model Represented in `pymc3`

6. Descriptive Analysis

After the MCMC run and the calculation of the posterior probabilities of all parameters, the first result we get is a collection of the density functions of the different model parameters. The takeaway from these densities is that while there is some certainty achieved for several parameters (e.g. applications accessing the network), a lot of them are still very wide and spread out in the $[-10, 10]$ range chosen as the prior. This means that for all the features with a very wide corresponding density function, there is very little certainty about how they contribute to the maliciousness status of the application. On the other hand, the features with very narrow density functions have a high level of certainty as to how they contribute to a target app being labeled malicious.

In our view, this is an important contribution of our work and indeed of Bayesian inference in general. We not only perform machine learning-based prediction of the class labels, but also provide underlying reasons for why the model works and which parts of the inputs to the model contribute to the final decision. This theme will be carried forward throughout the rest of the paper.

6.1. Uncertainties in Weights Associated with Permissions

Since it is not possible to show the detailed view of the posterior probabilities of all the parameters of the model, we have chosen to show a *forest plot* of the most often used permissions in Figure 4. Recall that a permission being used means that static analysis unearthed API calls in app code associated with that permission. We have numbered permissions in the plot so that we may easily refer the reader to them. The numbers are assigned based on the frequency of apps using these permissions. `0:INTERNET` is therefore the most frequently used permission. The values plotted are the means and HPD intervals of the posterior distributions of model weights associated with the respective permission.

One of the most notorious permissions which is vital for violating privacy of a user is access to the internet. Google's latest release of Android (Marshmallow) allows restricting an app's permissions, but excludes the `INTERNET` permission from this choice. Google's stance is that if an application is restricted from accessing any sensitive resources, allowing access to the internet should be harmless. While we do not agree with this line of thought, the data speak against the blacklisting of this permission.

In the forest plot, `0:INTERNET` has a short bar with the mean on the negative side of the weight axis. This means that, according to our model and results, the relationship between using the `INTERNET` permission and being malicious is inversely proportional. Keeping all other factors constant, if an application uses the `INTERNET` permission, it is more likely to be benign than malicious. (Of course, all the other factors are seldom constant. We return to a discussion of permission combinations presently.)

Similarly, `3:READ_PHONE_STATE` has a very high valued mean, which suggests a positive relationship between using this permission and being malicious. We draw a similar conclusion about `6:RECEIVE_BOOT_COMPLETED`, which is a permission commonly used by many spyware-type apps.

`9:SEND_SMS` and `11:READ_SMS` are no surprise, but confirm the intuition that these two contribute substantially (with little uncertainty) to the malicious label.

Results of `12:RECEIVE_SMS` are counter-intuitive, however: it is generally believed that malicious apps try to steal data by snooping on incoming SMSs. However, looking at the bigger picture, it seems that this permission contributes more towards a benign status than a malicious one. We believe the reason might be the existence of several 'good' messaging apps, as well as SMS-based registrations prevalent nowadays.

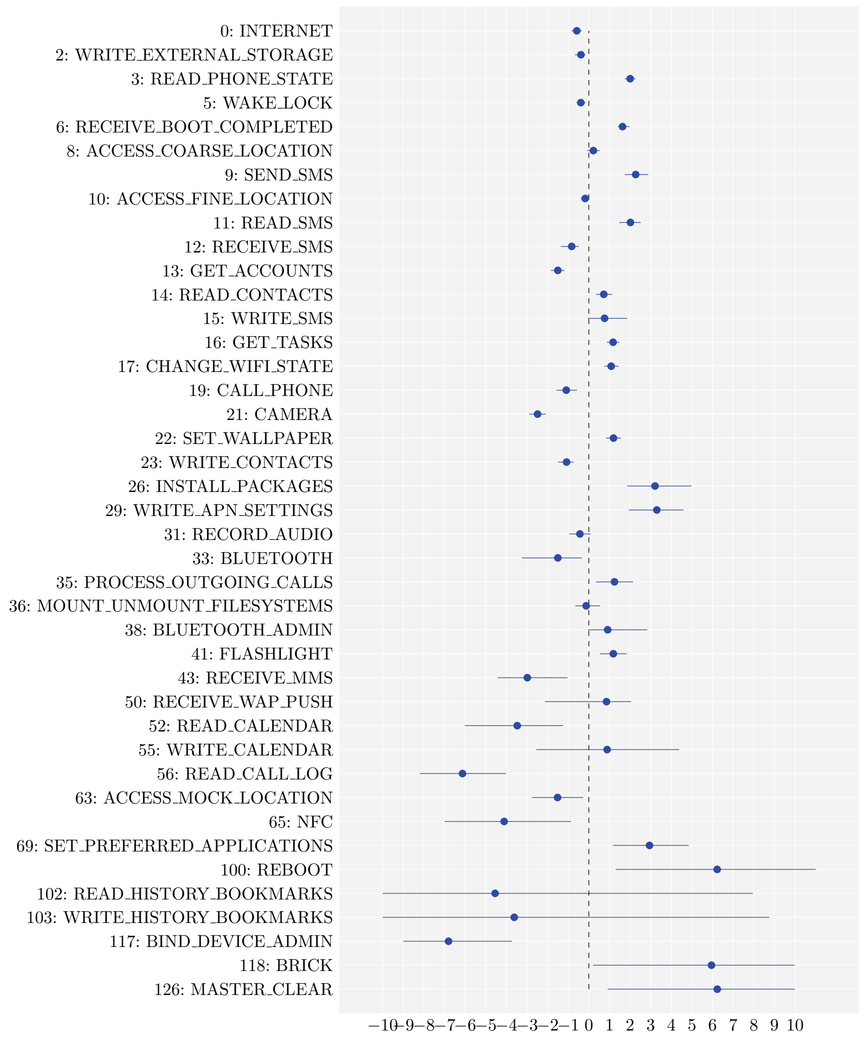


Figure 4: Forest Plot of Interesting θ Values and their Associated Uncertainties. (Right is malicious, left is benign.)

Another interesting result is that of `21:CAMERA`. Malicious apps have little interest in this permission. The mean for the weight associated with this permission is as low as 2.5, with quite a small uncertainty. To us, this seems quite counter-intuitive, especially when coupled with the result for `31:RECORD_AUDIO` since both of these might be a privacy nightmare (or goldmine depending on one's perspective).

Peng et al. [4] identified `26:INSTALL_PACKAGES` as a critical permission in their work, and our results seem to agree with them, given the very high weight of the parameter associated with this permission. However, we note that `29:WRITE_APN_SETTINGS` has an even higher weight and, even correcting for the uncertainty, should be a highly critical metric for identification of an app as being malicious.

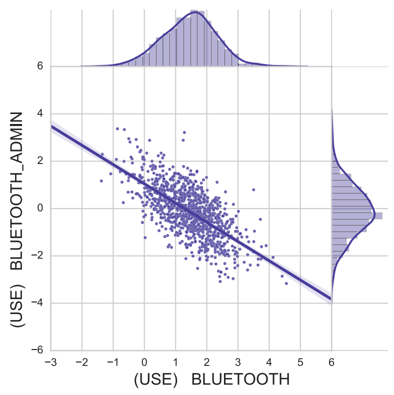
On the flip side, `56:READ_CALL_LOG` and `65:NFC` are two permissions which are highly representative of the benign app set and as such should also be considered when modeling the classes as a function of usage of sensitive resources.

Finally, at the bottom of the figure, we see a few permissions such as `102:READ_HISTORY_BOOKMARKS` and `103:WRITE_HISTORY_BOOKMARKS` that have so much uncertainty associated with them that we cannot make any meaningful decisions based on these.

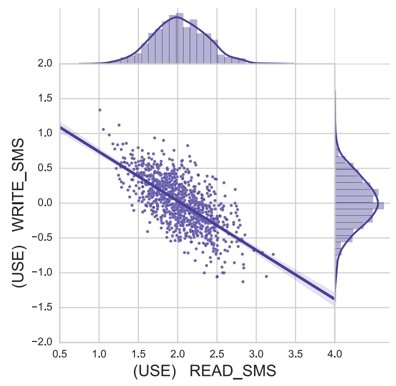
Note that this process is different from typical *feature selection* algorithms. Our methodology not only helps identify which features contribute the most to the malicious label but also gives us a quantified measure of how certain we are about the contribution. For instance, `100: REBOOT` has a high weight and would be picked by a feature selection algorithm as an important feature. However, Bayesian analysis shows that it has a high level of uncertainty and as such might not be a good feature to base the decision on.

6.2. Joint Probabilities: Interplay between Requested and/or Used Permissions

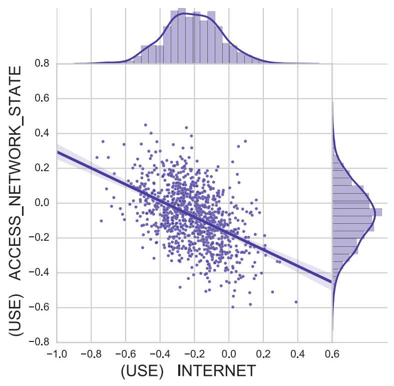
After we analyzed the probability distributions of the different model parameters in detail, we also decided to



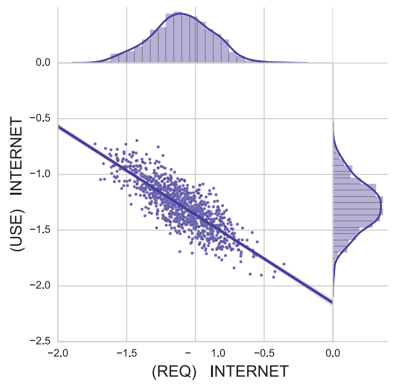
(a)



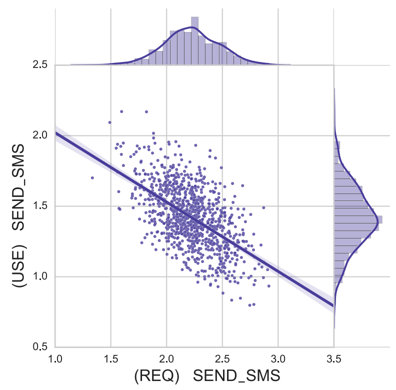
(b)



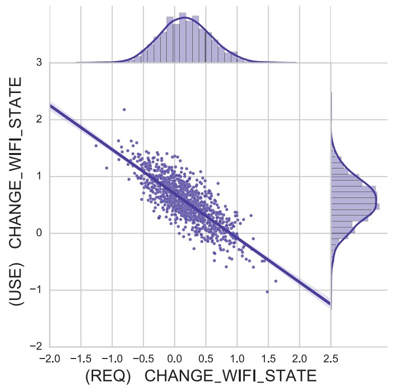
(c)



(d)



(e)



(f)

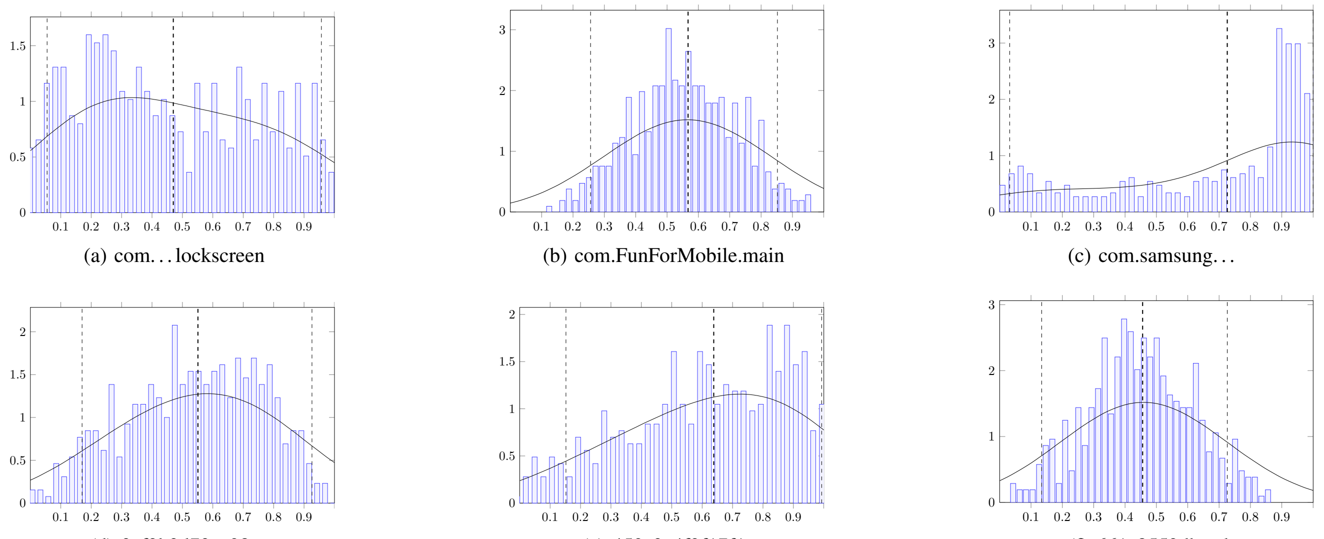
Figure 5: Joint Probability Distributions of θ Values for Different Combinations of Permissions

study their joint probabilities with the hope of uncovering more interesting patterns. For this purpose, we computed the joint probabilities of the requested and used permissions. We then separated the combinations which had a significant correlation and plotted their joint probability distributions in the form of a scatter plot along with a regression line showing their relationships. Some of the more interesting plots are shown in Figure 5. The axes prefixed with (USE) represent permissions that were found to have been actually used by an app whereas (REQUEST) axes mark permissions included in the manifest file.

The plots show that there are a few pairs which have an inverse correlation in terms of their corresponding model parameters. For instance, Figure 5a shows the joint probability plot between `BLUETOOTH_ADMIN` and `BLUETOOTH`. The interpretation of this plot is that in cases where an application uses one of these permissions but not the other, it is much more likely to get assigned a higher maliciousness hypothesis value – thus making it more likely to be malicious. On the other hand, if an application uses both permissions or neither, it is more likely to be a benign app. While this is reminiscent of the rules created by Kirin [12], it is much more concrete as not only do we have a relationship, we also have a way to quantify it.

After looking through the high correlation pairs, we realized that most of them, with the advantage of hindsight, seem intuitive. For instance, it appears that a benign app would use both `WRITE_SMS` and `READ_SMS` (cf. Figure 5b), but a malicious app would be more likely to use only one of them and not the other. We found similar relationships between several other pairs, such as: `WRITE_CONTACTS` and `READ_CONTACTS`, `SEND_SMS` and `RECEIVE_SMS`, `INTERNET` and `ACCESS_NETWORK_STATE`, `READ_SMS` and `READ_CONTACTS`, `FLASHLIGHT` and `CAMERA`, `KILL_BACKGROUND_PROCESSES` and `RESTART_PACKAGES`.

Another interesting finding is the relationship between applications requesting certain permissions and using them. For example, Figure 5d shows that there is a negative correlation between applications requesting the `INTERNET` permission and using it. One interpretation of this is that malicious applications usually don't include this permission in the manifest file. Rather, they rely on root access to bypass Android's permission mechanism to utilize network access. Similar correlations were found in joint probabilities of several other request-use pairs such as `SEND_SMS` (cf. Figure 5e)



(a) com…lockscreen (b) com.FunForMobile.main (c) com.samsung…

(d) 3af9b2d73ec08a (e) 450a3c4f9f17f1 (f) 661e8559dbaeda

Figure 6: False Positives and False Negatives as a Result of Ignoring Uncertainty in the Model

and `CHANGE_WIFI_STATE` (cf. Figure 5f). This can form a strong indicator of the maliciousness of an application. In the future, we aim to analyze such combinations in detail and generalize our study from pairs to n-tuples.

7. Prediction with Uncertainty

Before starting with Bayesian machine learning, we performed learning on our dataset using basic logistic regression and kept track of the evaluation metrics. We also experimented with large-margin classifiers and non-linear hypotheses (including SVMs, decision trees, random forests etc.) but saw little improvements in accuracy of the results. This is owing to the fact that there is already a clear decision boundary between the two classes and adding more complex models does not improve classification. We therefore chose to model the much simpler logistic regression model in the Bayesian framework to reduce the complexity and performance overhead. We achieved an accuracy of 0.951, an F1 score of 0.944 and an AUC of 0.929 for this base model. We kept track of the false positives and false negatives of prediction for later comparison with the Bayesian results.

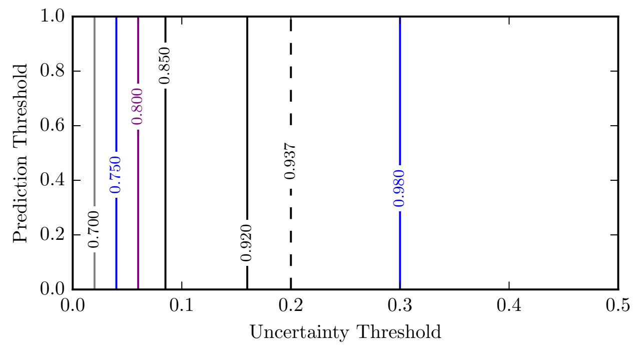
During the run of our Bayesian learning pipeline and after having completed our sampling for the posterior distributions of the model parameters, we ran the posterior predictive checks to calculate the posterior probability distributions of the output variable in our test set. We then compared the results of our base logistic regression model and manually analyzed the plots of posterior distributions of the test points for which we had a false positive or false negative in the basic non-Bayesian logistic regression.

Figure 6 shows some of the interesting posterior distributions from this experiment. Figures 6a and 6c show *false positives* of the base logistic regression model. As can be seen, the mean is near the threshold of 0.5 but the confidence interval is very high (denoted by the vertical dashed lines near the edges of the plot). This means that there is so much uncertainty in the results that we should not be able to make a prediction based on this result. In Figure 6c, the mean is far higher than the threshold but again, there is so much uncertainty that the mean cannot reasonably be used to make any prediction. However, since in the non-Bayesian view of machine learning, the uncertainty in prediction is lost, this information is not taken into consideration, thus leading to an incorrect classification of these benign apps to the malicious set. Similarly, Figures 6e and 6f show examples from the *false negative* set. These malicious apps also had a large margin of uncertainty, but ignoring this margin led to them being classified incorrectly.

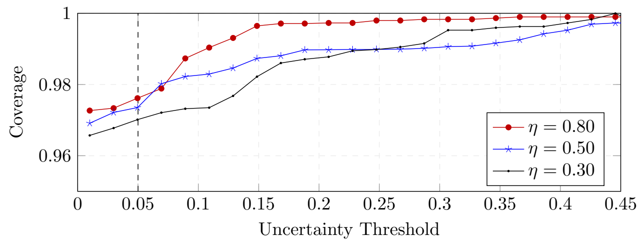
7.1. Uncertainty-Augmented Prediction

Learning from these and several other examples, we proposed the use of uncertainty during model prediction. In the rest of this section, we provide details of the results achieved as a result of the enhancements proposed to the basic logistic regression model for classification of Android applications as benign or malicious. The model itself was detailed in Section 4. We ran grid search for finding the best values of model *hyperparameters* λ and η with both values ranging from 0.0 to 1.0 in 100 steps each and calculated different metrics for each run.

Recall that the metric measuring the ratio of concrete prediction to the number of total test points was defined as *coverage* with the rest of the points being marked as uncertain. Figure 7a shows the contour plot of the coverage for M1 (cf. Section 4). Note that since the uncertainty decision in this method does not depend on the prediction threshold, the contour lines are parallel to the y-axis. For M2, the picture

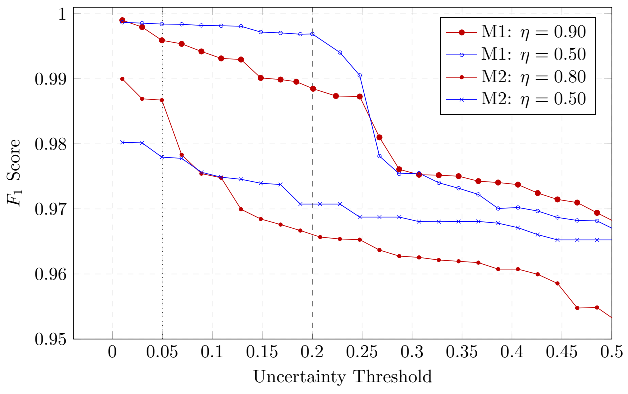


(a) M1

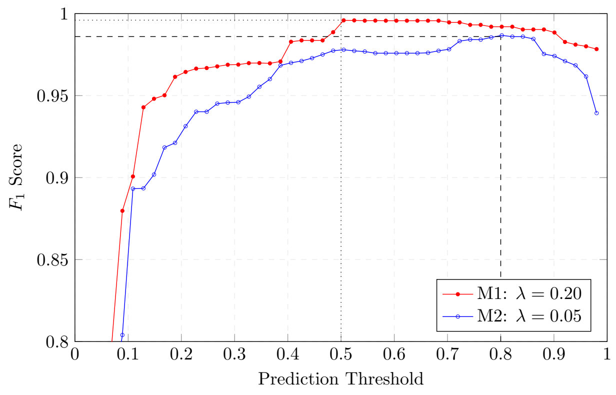


(b) M2

Figure 7: Coverage Achieved with Different Uncertainty Threshold Values



(a) F1 Scores as a Function of Uncertainty and Prediction Threshold



(b) Model Metrics as a Function of Prediction Threshold

Figure 8: Grid Search based on Uncertainty and Prediction Thresholds

is slightly more complicated. This is depicted in Figure 7b. Here, the coverage changes as a result of both uncertainty threshold changes and prediction threshold perturbations. Based on these plots and further results described below, we identified the sweet spot as corresponding to the uncertainty threshold of 0.2 for M1 and 0.05 for M2. This allows us to achieve a coverage of 93.7% for M1 and 97.6% for M2 while improving upon the results of the existing efforts in terms of accuracy, precision and AUC.

We also used grid search to evaluate our model with different values of the hyperparameters and measured the accuracy, precision and F1 Scores. Note that, for a particular threshold value (i.e. domain specific results), it is better to report the F1 scores instead of the AUC since the ROC metric leaves it up to the user to make a judgment about the threshold value. We do however report the AUC values later when comparing to previous work in order to enable the reader to evaluate our work in light of existing literature. Moreover, accuracy alone is not a sufficient measure of goodness-of-fit of an algorithm for skewed classes. For instance, it is possible to achieve an accuracy of 98% in cases where 98% of the real-wold data points belong to the malicious class simply by predicting all data points as malicious. The F1 score metric counters this issue by incorporating precision into the calculation as well. Hence, we begin our discussion by reporting the F1 scores of our model at different levels of uncertainty and prediction thresholds.

These results are plotted in Figure 8a. An uncertainty threshold of 1 means that any uncertainty level is acceptable and thus equates our model with the non-Bayesian logistic regression. As we reduce the level of acceptable uncertainty in prediction, we note that the number of false positives and false negatives decreases, thus improving our F1 score. At our chosen uncertainty level of 0.2 for M1, we see that the F1 score is 0.996 for the prediction threshold of 0.5. For M2, the optimum uncertainty threshold of 0.05 and prediction threshold of 0.8 gives us an F1 score of 0.986. Note also that, we are able to further improve our F1 score by reducing the uncertainty threshold. However, reading this plot in conjunction with Figures 7a and 7b, we conclude that this would also decrease the *coverage* to an unacceptable extent.

Finally, to provide a fine-grained view of our two models' F1 scores as compared to each other, we plot the F1 score of both as a function of the prediction threshold as shown in Figure 8b. For these plots, we kept the uncertainty threshold of M1 fixed at 0.2 and that of M2 at 0.05. The results support our view that the prediction threshold of 0.5 performs the best at the uncertainty level of 0.2 for M1, and the prediction threshold of 0.8 gives the optimum results for M2 at the uncertainty level of 0.05.

TABLE 1: Comparison of Results with Existing Techniques

Metric	Peng et al. [4]	Drebin [3]	Reveal Droid [5]	LUNA with M1	LUNA with M2
Accuracy	-	-	0.858	0.998	0.989
Precision	-	-	0.892	0.993	0.982
F1 Score	-	-	0.874	**0.996**	0.986
Coverage	1.0	1.0	1.0	0.937	**0.976**
FPR	-	0.01	-	0.002	0.002
Detec. Rate	-	0.94	0.969	0.991	0.981
AUC	0.953	-	-	**0.993**	0.987

7.2. Comparison with Existing Techniques

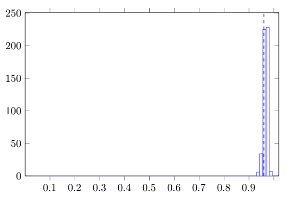
In order to put our model into perspective, we compare the results of both M1 and M2 with those previously reported in the literature. We compare our results with similar models presented by Peng et al. [4], Drebin [3] and RevealDroid [5]. Table 1 shows a concise overview of this comparison. Values marked as '-' are not reported by the authors of the original paper.

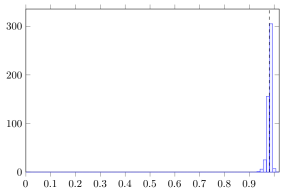
For instance, Peng et al. [4] achieved an AUC of 0.953 with their best model of HMNB. We improve upon their results by achieving an AUC of 0.987 with a high coverage using M2. Our M2 detection rate of 0.981 is also an improvement over Drebin. Moreover, we reduce the false positive rate to 0.2% from Drebin's 1% owing to the incorporation of uncertainty in our results. We also improve significantly upon the F1 score of RevealDroid [5] with both M1 and M2. From our two models, we note that while the F1 score achieved by M2 is slightly lower, its coverage is a major improvement over M1. Hence, we believe that of the two, M2 would be a better solution for a production environment.

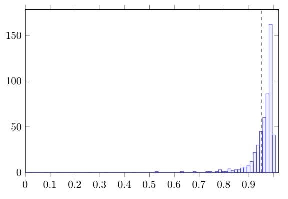
7.3. False False-Positives

Last but not least, we discuss an interesting finding that came out as a result of our analysis of the Bayesian model's accuracy. As mentioned above, there was a lot of uncertainty in the predictions of quite a few false positives. However, we noted a few points which were false positives but had a very high level of *certainty* associated with them. This meant that not only was our model classifying them as malicious, it was also quite certain about its prediction. So, we decided to manually scrutinize these applications. Posterior distributions of four such apps are shown in Figure 9. We provide a brief overview of these below:

Figure 9a shows the posterior distribution for the class of an app titled, "Start Earning Free Money Now". As can be seen, it is predicted as being malicious with a high level of certainty. We scraped this application from the Play Store around June 2015 but it has since been removed from the store, leading us to believe that Google's Bouncer probably also found some maliciousness in this app. Figures 9b and 9c show posterior distributions associated with "State Bank Freedom" and the self-proclaimed "Default Android 4.4 (KitKat) Messaging App" respectively; both of these have been removed recently from the Play store. Finally, in Figure 9d, we have the posterior probability of a flashlight app that has not been removed from the Play Store (yet). However, we did find an interesting review from a user there asking, "Why does it require permission to access phone ID and call information?"


(a) com.sefmn1021


(b) com…banking


(c) com…mms

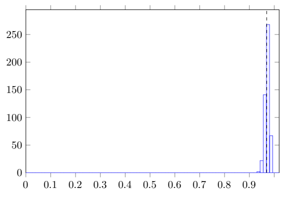

(d) com…fastlight

Figure 9: False False-Positives Identified through Bayesian Machine Learning

We found several other applications along these lines (e.g. `us.tv.remote.control .all4fun`) but since an analysis of these apps requires manual labor, we were not able to do it for a large set of apps. We do recognize that not all false positives belonged to this class, since there will always be noise in real-world datasets leading to inaccuracies in prediction. However, with a cursory search, we were able to find quite a few of these cases which leads us to believe that a detailed analysis along these lines might produce interesting results.

At the very least, this result shows that, even though outlier detection was not one of our objectives, LUNA has turned out to be robust to imperfections in training data.

8. Conclusion and Extensions

Android's ecosystem has a very open nature. Anyone with a developer account can upload applications to the official

Play Store. The Play Store then relies, for the most part, on crowd-sourcing to filter out bad apps. However, members of the crowd mostly only evaluate applications based on their utility. They are unable and ill-equipped to decide if an app is malicious. There have been several attempts to instead use machine learning techniques to find malicious apps from within the Play Store through learning from the available datasets. However, all the existing techniques have used statistical methods that discard uncertainty from their predictions.

In this paper, we have presented the application of Bayesian machine learning to Android malware analysis to take into consideration the uncertainty involved in prediction. Our contributions in this paper have been to introduce Bayesian machine learning and explore the model weights to gain insights into how critical features actually affect predictions. We quantified uncertainty associated with different features and explored their interplay for classifying an app as malicious. We also introduced an uncertainty-incorporating prediction model that was able to best the existing techniques for malware prediction.

We note a limitation of our approach: in order to improve the model, we reduced our coverage to 97.6%. This means that for 2.4% of apps, we would not be able to predict a class at all. We envision the incorporation of our model in a larger setting where such apps would be flagged as suspicious and would be analyzed in more detail through, say, offline dynamic analysis to classify them concretely.

We also note that while we have presented our findings related specifically to Android, the techniques and exploration methods presented here can be expanded to other smartphone domains and even to broader areas such as traditional malware analysis on the desktop systems and in social networks.

9. Acknowledgements

We would like to thank the maintainers of Drebin [3], Android Malware Genome Project [21] and the Contagio Mobile site [28] for making their datasets available to us. The computation-intensive MCMC sampling was made possible by the generous contribution of the Tesla K40c GPU by NVIDIA Corporation.

References

[1] Symantec, "Internet security threat report, volume 20," Accessed: Jul 15, 2016. [Online]. Available: http://www.symantec.com/security_response/publications/threatreport.jsp

[2] GData, "Mobile malware report: Q2/2015," Accessed: July 15, 2016. [Online]. Available: https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[4] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 241–252.

[5] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," George Mason University, Tech. Rep., 2015.

[6] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*. ACM, 2012, pp. 13–22.

[7] G. E. Box and G. C. Tiao, *Bayesian inference in statistical analysis*. John Wiley & Sons, 2011, vol. 40.

[8] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, *Bayesian data analysis*. Taylor & Francis, 2014, vol. 2.

[9] A. Patil, D. Huard, and C. J. Fonnesbeck, "PyMC: Bayesian stochastic modelling in python," *Journal of statistical software*, vol. 35, no. 4, p. 1, 2010.

[10] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using PyMC," Accessed: July 15, 2016. [Online]. Available: http://pymc-devs.github.io/pymc3

[11] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *Security & Privacy, IEEE*, vol. 7, no. 1, pp. 50–57, 2009.

[12] ——, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, 2009, pp. 235–245.

[13] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 73–84.

[14] T. Kohonen, *Self-organizing maps*. Springer, 2001, vol. 30, p. 105.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[16] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[18] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.

[19] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in DroidSafe," in *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[20] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the android framework," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[21] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[22] O. Tripp and J. Rubin, “A bayesian approach to privacy enforcement in smartphones,” in *USENIX Security*, 2014.

[23] D. Barber, *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[24] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, “Droidscribe: Classifying android malware based on runtime behavior,” *Mobile Security Technologies (MoST 2016)*, vol. 7148, pp. 1–12, 2016.

[25] M. D. Homan and A. Gelman, “The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1593–1623, 2014.

[26] W. K. Hastings, “Monte carlo sampling methods using markov chains and their applications,” *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[27] M. J. Powell, “A fast algorithm for nonlinearly constrained optimization calculations,” in *Numerical analysis*. Springer, 1978, pp. 144–157.

[28] Mila, “Contagio Mobile,” Accessed: July 15, 2016. [Online]. Available: http://contagiominidump.blogspot.com/

[29] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, “Theano: new features and speed improvements,” *arXiv preprint arXiv:1211.5590*, 2012.