Power-aware Anomaly Detection in Smartphones: An Analysis of On-Platform versus Externalized Operation

Guillermo Suarez-Tangil[a,*], Juan E. Tapiador[a], Pedro Peris-Lopez[a], Sergio Pastrana[a]

[a] *COSEC – Computer Security Lab*
Department of Computer Science, Universidad Carlos III de Madrid
Avda. Universidad 30, 28911, Leganés, Madrid, Spain

Abstract

Many security problems in smartphones and other smart devices are approached from an anomaly detection perspective in which the main goal reduces to identifying anomalous activity patterns. Since machine learning algorithms are generally used to build such detectors, one major challenge is adapting these techniques to battery-powered devices. Many recent works simply assume that on-platform detection is prohibitive and suggest using offloaded (i.e., cloud-based) engines. Such a strategy seeks to save battery life by exchanging computation and communication costs, but it still remains unclear whether this is optimal or not in all circumstances. In this paper, we evaluate different strategies for offloading certain functional tasks in machine learning based detection systems. Our experimental results confirm the intuition that outsourced computation is clearly the best option in terms of power consumption, outweighing on-platform strategies in, essentially, all practical scenarios. Our findings also point out noticeable differences among different machine learning algorithms, and we provide separate consumption models for functional blocks (data preprocessing, training, test, and communications) that can be used to obtain power consumption estimates and compare detectors.

Keywords: Smartphone security, anomaly detection, outsourced security, power consumption.

[*]Corresponding author. Tel: +34 91 624 6260, Fax: +34 91 624 9429
Email addresses: `guillermo.suarez.tangil@uc3m.es` (Guillermo Suarez-Tangil), `jestevez@inf.uc3m.es` (Juan E. Tapiador), `pperis@inf.uc3m.es` (Pedro Peris-Lopez), `spastran@inf.uc3m.es` (Sergio Pastrana)

1. Introduction

The past few years have witnessed a rapid proliferation of portable "smart" devices with increasingly powerful computing, networking and sensing capabilities. One of the most successful examples of such devices so far are smartphones and tablets, but new appliances are appearing at a steady pace, including watches, glasses, and other wearable systems. One key difference between such smart devices and traditional, non-smart platforms is that they offer the possibility to easily incorporate third-party applications ("apps" for short) through online markets. The popularity of smartphones has been recurrently corroborated by commercial surveys, showing that they will very soon outsell the number of PCs worldwide [8] and that users are already spending nearly as much time on smartphone applications as on the Web (73% vs. 81%) [26].

In many respects, devices such as smartphones present greater security and privacy risks to users than traditional computing platforms. One key reason is the presence in the device of numerous sensors that could leak highly sensitive information about the user's behavioral patterns (e.g., location, gestures, moves and other physical activities) [21], as well as recording audio, pictures and video from their surroundings. As a consequence, the development of smartphone technologies and its widespread user acceptance have come hand in hand with a similar increase in the number and sophistication of threats tailored to these platforms. For example, recent surveys have warned about the alarming volume of smartphone malware distributed through alternative markets [35] and the spread of new forms of fraud, identity theft, sabotage, and other security threats.

1.1. Anomaly Detection in Smart Devices

Many security issues can be essentially reduced to the problem of separating malicious from non-malicious activities. Such a reformulation has turned out to be valuable for many classic computer security problems, including detecting network intrusions, filtering out spam messages, or identifying fraudulent transactions. But, in general, defining in a precise and computationally useful way what is harmless or what is offensive is often too complex. To overcome these difficulties, many solutions to such problems have traditionally adopted a machine learning approach, notably through the use of classifiers to automatically derive models of good and/or bad behavior that could be later used to identify the occurrence of malicious activities.

Anomaly-based detection strategies have proven particularly suitable for scenarios where the main goal is to separate “self” (i.e., normal, presumably harmless behavior) from “non-self” (i.e., anomalous and, therefore, potentially hostile activities). In this setting, one often uses a dataset of self instances to obtain a model of normal behavior. In detection mode, each sample that does not fit the model is labelled as anomalous. This notion has been thoroughly explored over the last two decades and applied to multiple domains in the security arena [4, 12, 15].

More recently, many security problems related to smartphone platforms have been approached with anomaly-based schemes (see, e.g., [33, 13, 32, 10, 3]). One illustrative example is found in the field of continuous –or implicit– authentication through behavioral biometrics [19, 34, 7]. The key idea here is to equip the device with the capability of continuously authenticate the user by monitoring a number of behavioral features, such as for example the gait –measured through the built-in accelerometer and gyroscope–, the keystroke dynamics, the usage patterns of apps, etc. These schemes rely on a model learned from user behaviors to identify anomalies that, for example, could mean that the device is mislaid, in which case it should lock itself and request a password.

Proposals for detecting malware in smartphones have also made extensive use of anomaly detection approaches. Most schemes are built upon the hypothesis that malicious apps somehow behave differently from goodware. The common practice consists of monitoring a number of features for non-malicious apps, such as for example the amount of CPU used, network traffic generated, system/API calls made, permissions requested, etc. These traces are then used to train models of normality that, again, can be used to spot suspicious behavior. Modelling app behavior in this way is particularly useful in two scenarios. The first one is related to the problem of repackaged apps, which constitutes one of the most common distribution strategies for smartphone malware. In this case, the malicious payload is piggybacked into a popular app and distributed through alternative markets. Detecting repackaged apps is a challenging problem, in particular when the payload is obfuscated or dynamically retrieved at runtime. The second problem is thwarting the so-called grayware, i.e., apps that are not fully malicious but that entail security and/or privacy risks of which the user may not be fully aware. For instance, an increasingly number of apps access user-sensitive information such as locations frequently visited, contacts, etc. and send it out of the phone for obscure purposes [21]. As users find it difficult to define their privacy preferences in a precise way, automatic methods to tell apart good from bad activities constitute a promising approach.

1.2. Motivation

Essentially all machine learning-based anomaly detection solutions can be broken down into the following functional blocks:

- *Data acquisition.* Activity traces are required both for (re-)training the model of normality and in detection mode. The nature of the data collected varies across applications and may include events such as system calls, network activities, user-generated inputs, etc.
- *Feature extraction.* Machine learning algorithms require data to be expressed in particular formats, commonly in the form of feature vectors. A number of features are extracted from the acquired activity traces during a preprocessing stage. The complexity of such preprocessing depends on the problem and ranges from computationally straightforward procedures (e.g., obtaining simple statistics from the data) to more resource intensive transformations.
- *Training.* A representative set of feature vectors is used to train a model that captures the underlying notion of normality. This process may be done offline, in which case periodic re-trainings are often necessary in order to adapt the model to drifts in behavioral patterns, or else constantly as new data arrives.
- *Detection.* Once a behavioral model is available, it is used along with a similarity function to obtain an anomaly score for each observed feature vector. This process is often carried out in real time and requires constant data acquisition and feature extraction.

All the functions described above can be quite demanding –particularly if they must operate constantly– and it is debatable whether they can be afforded in energy-constrained devices with limited computational capabilities. As a consequence, a number of recent works (see, e.g., [29, 42]) have suggested externalizing some of these tasks to dedicated servers in the cloud or to other mobile devices nearby [39]. Although off-loading computation seems intuitively advantageous, such a strategy has an implicit trade-off between the energy savings resulting from not performing on-platform computations and the costs involved in data exchanges over the network. Intermediate strategies are also possible, such as for example off-loading the training stage only and performing detection locally, or externalizing everything but the data acquisition and preprocessing stages. Additionally, each plausible placement strategy has consequences in aspects other than power consumption. For example, off-loaded detection may result in delays in detecting anomalous events, or even malfunctions if network connectivity is unavailable.

Intuition suggests that intensive monitoring is prohibitive for platforms such as the current generation of smartphones [31]. However, the power consumption trade-offs among the various on-platform and externalized computation strategies are unclear, and to the best of our knowledge no analysis on this has been carried out yet.

1.3. Overview and Organization

In this paper, we address the problem discussed above and assess the power-consumption trade-offs among different strategies for off-loading, or not, functional tasks in machine learning based anomaly detection systems. Our analysis is motivated by, and hence strongly biased towards, security applications of anomaly detectors, such as for example malware detection or behavioral authentication. Nevertheless, the majority of our experimental setting, results and conclusions are general and may be of interest to other domains where smartphone-based anomaly detectors are used (e.g., health monitoring applications [21]).

In summary, our results confirm the intuition that externalized computation is, by far, the best option energy-wise. However, one rather surprising finding is that it is several orders of magnitude cheaper than on-platform computations, which suggests that networking is much more optimized than computation in such platforms. Furthermore, we have noticed substantial differences among the machine learning algorithms tested. Since some of them appear not to scale well for large feature vectors and/or datasets, developers should make careful choices when opting for one algorithm or another. In addition, anomaly detectors are found to consume considerably more power than popular apps such as games or online social networks, which motivates the need for more lightweight machine learning algorithms.

The rest of the paper is organized as follows. Section 2 describes the experimental setting used in our work, including the platform used, the anomaly detectors tested and the experiments carried out. Empirical results are discussed in Section 3, and power-consumption linear models are numerically derived for each separate function. Such models are used in Section 4 to analyze various off-loading strategies and provide a comparative discussion. In Section 5 we illustrate the main findings discussed throughout the paper using an anomaly-based detector of repackaged malware. An overview of related work is given in Section 6, and Section 7 concludes the paper by summarizing our contributions and main conclusions.

2. Experimental Setting

In this section, we describe the experimental framework used for evaluating power consumption in Android devices, including the machine learning algorithms evaluated, the tests carried out, and the tools and operational procedures used to measure power consumption.

2.1. Machine Learning Algorithms

We have tested three machine learning algorithms that can be used as anomaly detectors. Our choosing of these particular schemes is motivated by the different computational approaches followed by each one of them, and also because they are representative of broad classes of machine learning strategies: decision trees [30], clustering [14], and probabilistic approaches [18]. For completeness, we next provide an overview of each algorithm's working principles.

- *J48* is a Java implementation of the classic C4.5 algorithm [18]. The procedure builds a decision tree from a labelled training dataset using information gain (entropy) as a criterion to choose attributes. The algorithm starts with an empty tree and progressively grows nodes by choosing those attributes that most effectively split the dataset into subsets where one class dominates. This procedure is recursively repeated until reaching nodes where all instances belong to the same class [18].

  The resulting tree can be used as a classifier that outputs the class of future observations based on their attributes. The binary setting (i.e., two classes: normal and anomalous) is commonly used in anomaly detection problems, although it is perfectly possible to train a classifier with more a complex class structure.

- *K-means* is a clustering algorithm that groups data into k clusters and returns the geometric centroid of each one of them. Given a dataset composed of feature vectors $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the algorithm searchs for a partition of $\mathcal{D}$ into k clusters $\{C_1, \ldots, C_k\}$ such that the within-cluster sum of squares

  $$\sum_{i=1}^{k} \sum_{\mathbf{x_j} \in C_i} \| \mathbf{x}_j - \mu_i \|^2 \qquad (1)$$

  is minimized, where μ_i is the geometric mean of the vectors in C_i.

When used in a supervised training setting, each centroid μ_i receives a class label derived from the labels of the samples associated with the corresponding cluster. Labelled centroids can be then used, together with a nearest neighbor classifier, to determine the class of an observation by simply assigning it to a cluster according to some distance. Clustering algorithms have been extensively used in anomaly detection, particularly in one-class settings where only normal training instances are available. In such cases, a sample is often labelled as anomalous if its sufficiently far away from its nearest centroid.

- *OCNB (One Class Naïve Bayes)* [18] is a supervised learning algorithm that has been successfully used in a wide range of applications. OCNB is often a very attractive solution because of its simplicity, efficiency and excellent performance. It uses the Bayes rule to estimate the probability that an instance $x = (x_1, \ldots, x_m)$ belongs to class y as

$$P(y|x) = \frac{P(y)}{P(x)} P(x|y) = \frac{P(y)}{P(x)} \prod_{i=1}^{m} P(x_i|y) \tag{2}$$

so the class with highest $P(y|x)$ is predicted. (Note that $P(x)$ is independent of the class and therefore can be omitted.) The naïvety comes from the assumption that in the underlying probabilistic model all the features are independent, and hence $P(x|y) = \prod_{i=1}^{m} P(x_i|y)$. The probabilities $P(x_i|y)$ are derived from a training set consisting of labelled instances for all possible classes. This is done by a simple counting procedure, often using some smoothing scheme to ensure that all terms appear with non-zero probability. The priors $P(y)$ are often ignored.

In a one-class (OC) setting the training set consists exclusively of normal data. Since a profile of non-self behavior is not required, the detection is performed by simply comparing the probability of a sample being normal (or, equivalently, the anomaly score) to a threshold. Such a threshold can be adjusted to control the false and true positive rates, and the resulting ROC (Receiver Operating Characteristic) curve provides a way of measuring the detection quality.

2.2. Instrumentation

The experiments have been conducted in a Google Nexus One smartphone. Power consumption has been measured by applying a battery of tests involving both computation and communication capabilities. Each test is an app containing some of the functionality present in a given anomaly detector, such as for example the training process or the detection stage. The app is loaded into the device and repeatedly executed using some provided configuration. The process is sequential, so only one execution is run at a time.

The device was previously instrumented with AppScope [38], an energy metering framework based on monitoring kernel activity for Android. AppScope collects usage information from the monitored device and estimates the consumption of each running application using an energy model given by DevScope [20]. AppScope provides the amount of energy consumed by an app in the form of several time series, each one associated with a component of the device (CPU, Wi-Fi, cellular, touchscreen, etc.). We restrict our measures to CPU for computations and Wi-Fi for communications, as our tests do not have a graphical user interface, do not require user interaction and, therefore, do not use any other component.

2.3. Power Consumption Tests

The power consumption tests were independently carried out over the four functional tasks described in Section 1.2 in order to obtain a separate consumption model for each anomaly detection component. With this aim in mind, we designed the following four families of tests:

1. *Data preprocessing.* The underlying machine learning algorithm takes as input a dataset of behavioral patterns encoded in some specific format, often in the form of feature vectors. Obtaining such patterns may involve non-negligible computations, such as for example computing histograms, obtaining statistics, applying data transformations, etc. In our case, this stage consisted of processing a trace file where an ordered list of system calls executed by a monitored app was provided. The trace is sequentially read using a sliding window and a feature vector is computed for each window. The vector is then written into an Attribute-Relation File Format (ARFF) file, which will be later used for training or detection purposes. Overall, the preprocessing requires some on-platform computations and also reading and writing files. We used generic I/O Java components for this task, such as FileInputStream and BufferedReader.
2. *Training.* The training process reads an ARFF dataset and builds a model of normal behavior according to some machine learning algorithm. We prepared three different subtests, one for each algorithm discussed above. We used an stripped version of the well known Weka

[16] library for Android devices, as this implementation is reasonably optimized. Training involves a number of parameters that may influence the algorithm's running time. In our case, each algorithm was provided with the configuration yielding optimal detection results as discussed in the previous section.

3. *Detection.* This tests measures the amount of power consumed by a constantly running detector. Again, we prepared one subtest for each machine learning algorithm and implemented the detector using the stripped version of Weka. Each detector is assumed to have the behavioral model already loaded, so the test only measures power consumption associated with loading a test vector and deciding its class (normal or anomalous).
4. *Communications.* In this test we measured the amount of power consumed by sending and receiving data over a Wi-Fi connection. As the amount of data exchanged and the frequency of such exchanges may vary across operational scenarios, we focused on obtaining a model of power consumed per exchanged byte. We identified three subtests here, depending on whether a secure (encrypted and authenticated) channel is necessary or not. The tests were implemented using standard Java libraries, such as HttpURLConnect and HttpsURLConnect for insecure and secure communications, respectively. Besides, we tested two different networking scenarios. In the first one, the detector communicates with a locally reachable device, which implies low network latency. For these cases we tested both open and WPA-protected Wi-Fi networks. In this case, the time required for a packet to travel from the device to the server and back (Round-Trip Time, RTT) is about 0.6 ms. In the second scenario, we assumed that the detector communicates with a device located reasonably far away in terms of network latency, such as for example in a cloud service accessible via Internet. In our experimental setting, the server is accessed via Internet using a WPA-protected Wi-Fi network with a network latency of around 31 ms.

As indicated above, each test is a separate app that is installed on the device, executed, measured with AppScope, and finally uninstalled. Each test was executed 30 times with different input parameters, such as the length and number of feature vectors in the training dataset and the frequency of sending and receiving data over the network. We elaborate on this later when discussing the experimental results.

The test suite is summarized in Table 1.

Test	**Subtest**	**No. Executions**
Data preprocessing	Preprocessing	30
Training	Training.J48	30
	Training.K-means	30
	Training.OCNB	30
Detection	Detection.J48	30
	Detection.K-means	30
	Detection.OCNB	30
Comms	Comms.LoLat.Open.HTTP	30
	Comms.LoLat.Open.HTTPS	30
	Comms.LoLat.WPA.HTTP	30
	Comms.LoLat.WPA.HTTPS	30
	Comms.HiLat.WPA.HTTP	30
	Comms.HiLat.WPA.HTTPS	30

Table 1: Power consumption tests executed.

3. Power Consumption of Anomaly Detection Components

We next present the experimental results obtained after running the tests described in the preceding section. We group the results into two separate categories: computation and communications. The first one includes data preprocessing, training, and detection, while the second focuses on data exchange over the network. We finally obtain and discuss linear regression models for each algorithm and functional task.

3.1. Computation

We experimentally found that power consumption related to preprocessing, training, and detection tasks depends on:

- The length $|v|$ of the feature vectors, measured as the number of attributes that each vector has.
- The size $|\mathcal{D}|$ of the dataset, measured as the number of vectors to be processed, i.e., generated during preprocessing, used for training, or evaluated during detection.

We executed all the preprocessing, training and detection tests with values of $|v| = 10, 100, 200, 300$, and 400. These lengths are representative of the feature vectors used in most security applications of machine learning.

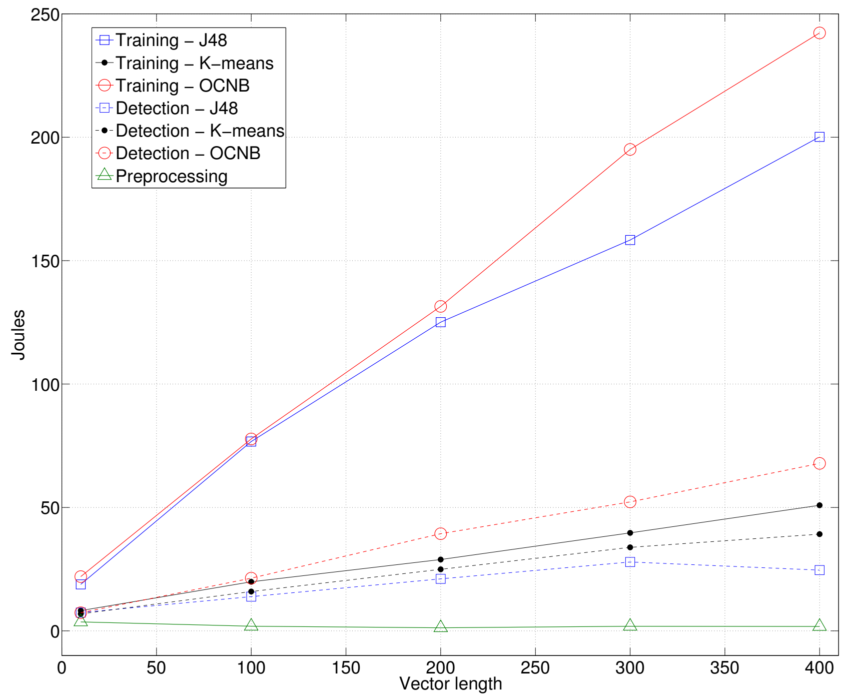

Figure 1: Power consumption results in Joules per vector for different vector lengths for the preprocessing, training and detection tests.

On the other hand, for each vector length we generated datasets of sizes $|\mathcal{D}| = 10, 50, 100, 200, 500$, and 1000, and then computed the average power consumption per vector. The average power consumption in Joules (J) per vector for each vector length is shown in Figure 1. Several conclusions can be drawn from these results:

1. Data preprocessing consumes very little power when compared to detection and training. This cannot be easily generalized, as it strongly depends on the sort of preprocessing applied. In our case data preprocessing is quite straightforward (computing histograms) and consumes less than 10 J/vector.
2. For a given algorithm, detection is significantly cheaper than training in terms of power consumption, but there are exceptions. For example, for both J48 and OCNB and vectors of length 100 training requires around 50 J/vector more than detection. This difference increases to

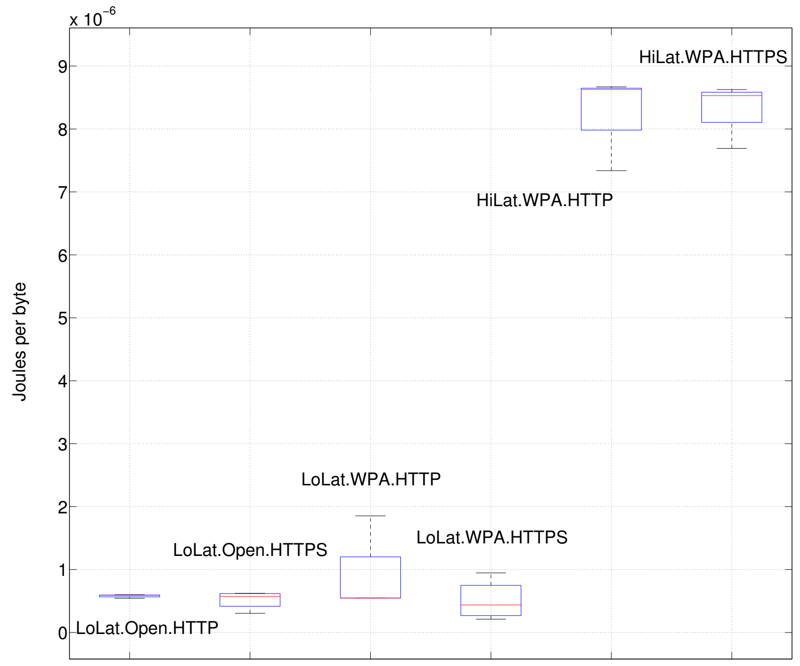

Figure 2: Power consumption results in Joules per byte exchanged (sent or received) for the communication test.

more than 100 J/vector for lengths greater than 300. K-means is an exception, with training and detection consuming approximately the same power.

3. The algorithm matters: K-means consumes far less than J48 and OCNB. In turn, OCNB is more expensive power-wise than J48, both in training and detection.
4. For the three tasks, power consumption increases approximately linearly in $|v|$.

3.2. Communications

Each communication test consists of the app sending and receiving 10 large files to/from a server, using both HTTP and HTTPS. After each test, the total power consumed is divided by the number of bytes sent or received to obtain a normalized measure in Joules per byte. Each test was repeated 30 times, resulting in the boxplots shown in Figure 2.

The results are quite surprising. On the one hand, we found no significant difference between using HTTP or HTTPS. In other words, key establishment plus encryption/decryption for each packet sent/received seems to be extremely efficient in terms of power consumption. One possible explanation for these figures might be related to the granularity used by AppScope to measure energy and compute the attribution of consumption. AppScope uses application-specific energy consumption data for each hardware component. However, authors argue that the "system" consumes a certain amount of power when communications are used. It may be the case that AppScope is not attributing the consumption of crypto operations to the app using HTTPS.

Apart from the observation above, our results suggest that network latency has a clear influence power consumption. In our experiments, increasing latency from 0.6 ms to 31 ms resulted in 8 times more power consumption. This may just be a consequence of the app execution taking more time to transmit the data.

3.3. Linear Models

We used the figures obtained above to derive linear power consumption models that could be later used to determine the best deployment strategy for each function depending on aspects such as the remaining power available on the device or the detection architecture. To do this, we applied a simple linear regression analysis using least squares over the power consumption data.

In the case of the computation functions, each model has the form:

$$P_f(|v|) = \alpha_f \cdot |v| + \beta_f \tag{3}$$

where $f \in \{\texttt{pre}, \texttt{tra}, \texttt{det}\}$, i.e., preprocessing, training, and detection, respectively. Similarly, power consumption incurred by communications is estimated by a linear model:

$$P_{\texttt{comms}}(s) = \gamma \cdot s \tag{4}$$

where s is the number of bytes to be sent or received, and γ is the average power consumption of the network configuration used by the device.

The coefficients thus estimated are provided in Table 2 and confirm the conclusions drawn above. For example the slope α of the three training algorithms reveals the difference between K-means, which introduces a multiplying factor of 0.11 J per additional attribute in the vector, and J48/OCNB, for which such a factor is 0.45 J and 0.57 J, respectively. Similarly, OCNB is clearly much more costly in terms of detection, with a 0.15 J factor per additional vector attribute against 0.05 and 0.08 for J48 and K-means, respectively.

Function		**Model**	
Computation		α_f	β_f
Preprocessing	–	0.00	2.82
Training	Training.J48	0.45	24.45
	Training.K-means	0.11	7.85
	Training.OCNB	0.57	18.78
Detection	Detection.J48	0.05	9.04
	Detection.K-means	0.08	7.16
	Detection.OCNB	0.15	6.34
Communications		γ	
Comms	Comms.LoLat.Open.HTTP	$8.74 \cdot 10^{-7}$	
	Comms.LoLat.Open.HTTPS	$5.09 \cdot 10^{-7}$	
	Comms.LoLat.WPA.HTTP	$5.81 \cdot 10^{-7}$	
	Comms.LoLat.WPA.HTTPS	$5.18 \cdot 10^{-7}$	
	Comms.HiLat.WPA.HTTP	$8.31 \cdot 10^{-6}$	
	Comms.HiLat.WPA.HTTPS	$8.34 \cdot 10^{-6}$	

Table 2: Regression coefficients for the linear power consumption models for computation and communication tasks.

4. Deployment Strategies and Trade-offs

Based on the findings presented in the previous section, we next discuss different deployment strategies for the various functions composing an anomaly detection system and analyze the associated power consumption costs.

4.1. Energy Consumption Strategies

We make two assumptions in our subsequent analysis. Firstly, data acquisition is executed in the device by means of some instrumentation procedure, e.g., through the system API to get access to activity traces. This would not be strictly true for some recently proposed approaches based on keeping a synchronized clone of the device in the cloud [29, 5, 42]. We believe, however, that the overhead incurred by such approaches may be equivalent to that of directly monitoring the device, although this issue needs further investigation. Secondly, our envisioned applications require relatively straightforward data preprocessing (see Table 2) that can easily be incorporated into the data acquisition module. As a result, both acquiring the data and preparing the feature vectors incur a constant overhead for all discussed strategies and will be left out of our analysis.

The two remaining functional blocks are training and detection. Each one, or both, of them can be placed locally in the device (L) or off-loaded to a remote server (R). This gives rise to four possible strategies that will denoted by LL, LR, RL, and RR. In all cases, power consumption is a linear function:

$$P_{i,j}(t) = \pi_{i,j} \cdot t \tag{5}$$

with $i, j \in \{L, R\}$, where $\pi_{i,j}$ is determined by each strategy.

In what follows $|v|$ represents the length in bytes of each feature vector; $|D|$ is the size of the dataset used for training, measured in number of vectors; $|M|$ is the size in bytes of the normality model returned by the training process; and ω_t and ω_d represent the frequencies at which training and detection take place, respectively.

- *Local Training, Local Detection (LL).* In this case the entire operation of the detector is executed locally in the device. The power consumption factor π_{LL} is composed of two terms: $P_t(|v|)$ Joules per vector in the dataset during training, plus $P_d(|v|)$ Joules per vector for each detection event. Overall, we have:

$$\pi_{LL} = \omega_t |D| P_t(|v|) + \omega_d P_d(|v|) \tag{6}$$

- *Local Training, Remote Detection (LR).* In this scenario training takes place in the device but detection is off-loaded. During training, power consumption is equivalent to the corresponding term in (6) plus the cost of sending the model M to the cloud ($P_d(|M|)$). In detection mode, every vector must be also sent out for analysis. We consider here that receiving the result has a negligible cost, as it may just be 1 bit (normal/anomalous). In summary:

$$\pi_{LR} = \omega_t \Bigg(|D| P_t(|v|) + P_c(|M|) \Bigg) + \omega_d P_c(|v|) \tag{7}$$

- *Remote Training, Local Detection (RL).* This strategy captures the idea of off-loading the model training stage while performing detection locally. To do this, every time that a (re-)training event is triggered the entire dataset must be sent out for analysis and, subsequently, the model must be received. In detection mode, power consumption for each analyzed vector is ascribed to the device, resulting in:

$$\pi_{RL} = \omega_t \Big(|D| P_c(|v|) + P_c(|M|) \Big) + \omega_d P_d(|v|) \tag{8}$$

- *Remote Training, Remote Detection (RR).* Finally, this strategy considers the possibility of externalizing all functions to a remote server. Consequently, the only power consumption attributed to the device is that related to sending and receiving feature vectors both for training and detection. Thus:

$$\pi_{RR} = \omega_t |D| P_c(|v|) + \omega_d P_c(|v|) \tag{9}$$

We then discuss the tradeoffs between these four possibilities. In particular, we compare the LL strategy with the other three to understand the potential gains from off-loading training, detection, or both.

4.2. LL vs LR

The LL strategy is preferred to LR if:

$$\begin{aligned} \pi_{LL} &\leq \pi_{LR} \\ \omega_t |D| P_t(|v|) + \omega_d P_d(|v|) &\leq \omega_t \Big(|D| P_t(|v|) + P_c(|M|) \Big) + \omega_d P_c(|v|) \\ \omega_d P_d(|v|) &\leq \omega_t P_c(|M|) + \omega_d P_c(|v|) \\ \omega_d P_d(|v|) &\leq (\omega_t + \omega_d) P_c(|M| + |v|) \\ P_d(|v|) &\leq \frac{\omega_t + \omega_d}{\omega_d} P_c(|M| + |v|) \end{aligned} \tag{10}$$

Note that, in general, $\omega_d \gg \omega_t$, in which case the term $\frac{\omega_t+\omega_d}{\omega_d} \approx 1$. Alternatively, in the extreme case of training being done for each incoming vector, we have $\omega_d = \omega_t$ and $\frac{\omega_t+\omega_d}{\omega_d} = 2$. Renaming this term as

$$k = \frac{\omega_t + \omega_d}{\omega_d} \in [1, 2] \tag{11}$$

and using the linear forms of P_d and P_c we can rewrite the inequatlity above as:

$$\begin{aligned} \alpha |v| + \beta &\leq k\gamma(|v| + |M|) \\ (\alpha - k\gamma)|v| &\leq \gamma |M| - \beta \\ |v| &\leq k \frac{\gamma |M| - \beta}{\alpha - k\gamma} \end{aligned} \tag{12}$$

A simple analysis of the orders of magnitude of the quantities involved in (12) provides some insights. Recall that $\alpha \approx 10^{-2}$, $\beta \approx 10$ and $\gamma \approx 10^{-7}$ (see Table 2). Replacing these values in (12), and ignoring the factor k, we get

$$|v| \leq \frac{10^{-7}|M| - 10}{10^{-2} - 10^{-7}} \approx 10^{-5}|M| \tag{13}$$

Consequently, the right-hand term in (12) will be negative unless $|M|$ is of the order of 10^6 or greater. However, almost all machine learning algorithms produce models that rarely exceed a few hundred kilobytes.

The main conclusion that can be drawn is that the LL strategy is worse energy-wise than the LR unless the model is so large and the vectors tiny enough so that the power consumed by sending both the model and the vectors to the cloud outweighs the power of performing detection locally.

4.3. LL vs RL

In this case we have:

$$\begin{aligned}
\pi_{LL} &\leq \pi_{RL} \\
\omega_t|D|P_t(|v|) + \omega_d P_d(|v|) &\leq \omega_t\Big(|D|P_c(|v|) + P_c(|M|)\Big) + \omega_d P_d(|v|) \\
\omega_t|D|P_t(|v|) &\leq \omega_t\Big(|D|P_c(|v|) + P_c(|M|)\Big) \\
|D|P_t(|v|) &\leq |D|P_c(|v|) + P_c(|M|) \\
|D|P_t(|v|) &\leq P_c(|D||v| + |M|) \\
|D|(\alpha|v| + \beta) &\leq \gamma(|D||v| + |M|) \\
|D|\alpha|v| + |D|\beta &\leq |D|\gamma|v| + \gamma|M| \\
|D|(\alpha - \gamma)|v| &\leq \gamma|M| - |D|\beta \\
|v| &\leq \frac{\gamma|M| - |D|\beta}{|D|(\alpha - \gamma)} \qquad (14)
\end{aligned}$$

Expression (14) presents a trade-off somewhat similar to that discussed in the previous section, but more acute. The fact that training takes place remotely factors in the size of the dataset in the inequality, which must be transferred for the remote server to build up the model. The overall consequence is however similar: the RL strategy consumes less than LL unless the model is sufficiently large with respect to the size of the dataset. Since the factor $-|D|\beta$ appears in the numerator of (14), the model size must now be even greater than in the previous case.

In summary, outsourcing the training stage is consistently better than performing it locally unless the datasets to be sent for analysis and the models received are massive.

4.4. LL vs RR

Local training and detection consumes less than a fully off-loaded operation if:

$$\begin{aligned} \pi_{LL} &\leq \pi_{RR} \\ \omega_t|D|P_t(|v|) + \omega_d P_d(|v|) &\leq \omega_t|D|P_c(|v|) + \omega_d P_c(|v|) \\ \omega_t|D|\Big(P_t(|v|) - P_c(|v|)\Big) &\leq \omega_d\Big(P_c(|v|) - P_d(|v|)\Big) \end{aligned} \tag{15}$$

Note that in (15) the various power consumption functions are applied to inputs of the same length $|v|$. However, communications are several orders of magnitude cheaper than training and detection, so

$$P_t(|v|) - P_c(|v|) \approx P_t(|v|) \tag{16}$$

and

$$P_c(|v|) - P_d(|v|) \approx -P_d(|v|) \tag{17}$$

Replacing this in (15) we get

$$\omega_t|D|P_t(|v|) \leq -\omega_d P_d(|v|) \tag{18}$$

which never holds. The conclusion is clear and, in a sense, rather expected from the findings discussed in the two previous sections: off-loading the entire operation of the detector is always better in terms of power consumption than operating locally in the device.

Taking another look at (15), the only scenario where LL may be competitive against RR arises when $P_c(|v|) \geq P_d(|v|)$. This situation may correspond to extremely lightweight detectors in which computing the anomaly score takes less power than sending the vector over the network. In such a case, (15) can be reduced to:

$$|D|\frac{P_t(|v|)}{P_c(|v|) - P_d(|v|)} \leq \frac{\omega_d}{\omega_t} \tag{19}$$

which essentially establishes that local operation pays off power-wise if training is very infrequent, does not consume much energy, and the datasets are not very large.

4.5. Discussion

The analysis conducted in the previous three sections point out to one definite conclusion: externalizing computation, both training and detection activities, is by far the best option in terms of power consumption. A deeper look at the trade-offs derived above reveals that the core of this argument is intimately related to the enormous differences in power consumption existing between computation and networking activities. In platforms such as the current generation of smartphones, communications appear to be extraordinarily optimized in terms of energy requirements, whereas computation is significantly more demanding. In the case of applications such as anomaly detection, the best strategy is undoubtedly to externalize all computation functions, including continuous detection, whenever possible.

In terms of performance criteria other than power consumption, offloading may or may not have an impact depending on the application domain. Loss of network connectivity –or even sufficient degradation– is a major threat for outsourced detection, as the device may be forced to functioning without the detection service while the remote server is unreachable. Similarly, network delays may be a critical point in applications where near real-time detection is required. In such cases, these aspects must be weighed against the energy saving benefit.

Finally, the security and privacy aspects of offloading computation to the cloud is a major concern that may prevent many users from relying on external services, particularly when confidential data is involved in the training and detection datasets. In this context, many works have dealt with the problem of securely outsourcing computation (see, e.g. [37]). One common assumption is to consider the external server as untrusted and to encrypt all data sent out for processing. In order to assess the extra power consumption incurred by encrypting data prior to sending it, we evaluated three of the most common ciphers found in cryptographic libraries and used nowadays: AES, 3DES, and RC4. The experimental setting and power consumption tests are identical to those described in Section 2.3. We carried out 30 independent tests and divided, in each case, the total power consumed by the number of encrypted bytes to obtain a normalized measure in Joules per byte. The γ factor obtained is shown in Table 3. As it can be observed, the cost of encryption is negligible when compared to that of training and detection tasks and does not affect the general conclusions discussed above.

Cipher	**Mode**	γ
AES-128	CTR	$7.62 \cdot 10^{-9}$
3DES	CTR	$9.52 \cdot 10^{-9}$
RC4	–	$7.62 \cdot 10^{-9}$

Table 3: Average power consumption per encrypted byte.

5. Case Study: A Detector of Repackaged Malware

We next illustrate some of the conclusions drawn in the preceding sections with real-world application: an anomaly-based detector for repackaged malware in Android apps. The use of anomaly detectors for this purpose has been proposed in a number of recent works (see, e.g., [3, 33]). Although in all cases the performance of such approaches is reasonably good in terms of detection quality, to the best of our knowledge none has explored the power consumption savings gained by outsourcing it.

5.1. The Detector

Sequences of system calls have been recurrently used by anomaly detection systems for security applications in smartphones [2, 3, 33, 23]. All apps interact with the platform where they are executed by requesting services through a number of available system calls. These calls define an interface that allow apps to read/write files, send/receive data through the network, read data from a sensor, make a phone call, etc. Legitimate apps can be characterized by the way they use such an interface [23], which facilitates the identification of malicious components inserted into an seemingly harmless app and, more generally, other forms of malware [35].

Based on this idea, we have built an anomaly detector that combines some of the ideas already proposed in previous works[1]. Feature vectors consist of histograms computed from a trace of system calls using a sliding window of length W. We determined experimentally that windows of length 400 result in very good detection performance. The number of systems calls varies across architectures and it is often between 200 and 400. Thus, during the training period all processes of normal apps are monitored and

[1]We deliberately omit a number of details about our detector, particularly those related to the detection quality for different parametrizations, as this is not the main focus of this work and has been reported elsewhere.

the corresponding feature vectors are generated. Such vectors are then used to train a normality model.

In detection mode, the algorithm takes as input a sequence s of N system calls and extracts the $N - W + 1$ feature vectors using a sliding window. Each one of these feature vectors is then classified as normal or anomalous. Let A be the number of vectors identified as anomalous. Then, the sequence –and, therefore, the app– is classified according to the following rule:

$$\mathsf{det}(s) = \begin{cases} \mathsf{legitimate} & \text{if } \frac{A}{N-W+1} < \tau \\ \mathsf{repackaged} & \text{otherwise} \end{cases} \tag{20}$$

where τ is an adjustable detection threshold.

The detection procedure described above is intimately related to the nature of repackaged malware. In general, not all the system call windows issued by a repackaged app will be anomalous, as they may be generated by non-malicious code. Thus, detection must be based on analyzing sets of windows and seeking if a fraction of them are anomalous. In our experiments, we obtained good results with sequences of at least 10 windows and thresholds τ around 0.1. For example, one of the apps we used for testing detection performance is a popular game named *Mx Moto* by Camel Games. The app can be purchased from Google Play for 1.49 €and so far has been downloaded 100K times. The same app can also be found in alternative markets for free [41], in most cases repackaged with a malware known as *Anserverbot*. We tested the original app together with various repackaged variants, obtaining in all cases a detection rate of 100% with no false positives with the OCNB detector. These results are congruent with those reported in similar works based on anomaly detection [3, 33].

5.2. Testing Framework

We tested the power consumption of three detectors built as described above, one for each machine learning algorithm evaluated. Only the LL and RR strategies were studied, as they represent opposite cases for placement decisions. For the latter, the high latency configuration with WPA and HTTPS was used. In order to study power consumption for different apps and/or detector configurations, we gathered a dataset of 190 apps containing both goodware and malware. For each one of them, we derived the average number of system calls per second issued depending on different usage intensity rates (throttle). These figures are obtained by running each app in a controlled environment and automatically injecting user events at a throttle pace. The results are shown in Table 4 and reveal that apps can generate

Type	**No. Apps**	**No. Events**	**Throttle (ms)**	**Syscalls/s**
Goodware	10	5000	1000	180.43
	35	1000	5000	453.91
	50	5000	1000	307.62
Malware	10	5000	1000	112.39
	35	1000	5000	128.66
	50	5000	1000	161.90
All	**190**	—	**Average**	**224.16**

Table 4: Average number of system calls per second in different executions of both goodware and malware.

up to a few hundred system calls per second. Even though user-driven apps may well function at lower paces, these rates are useful for apps where high frequency testing is required.

Each detector is evaluated for different vector lengths. Again, our goal is measuring how the amount of power varies in a real setting depending on the choice of this parameter. (Recall that in terms of detection quality, best results are obtained for $|v| = 100$.) Finally, each detector was continuously executed during 1 week, and the amount of power consumed so far was measured at 4 control points: after 10 minutes, 1 hour, 1 day, and 1 week. During this period, detection is triggered as often as a sufficiently large sequence of system calls is available, and re-training occurs every 10 minutes.

5.3. Results and Discussion

Fig. 3 shows the average power consumed by the three detectors for the LL and RR strategies. (Note that the latter is independent of the algorithm as only communications are involved.) The plots are consistent with the results discussed in the previous section and confirm that outsourced detection is much more efficient power-wise than on-platform operation. Consider, for example, the case of vectors of 100 attributes. During the first 10 minutes, both the OCNB and the J48 detectors have consumed more than 10^5 J. During the same period, the detector located in the cloud has required less than 10^4 J. After 1 day, cloud-based detectors consume roughly the same amount of power than on-platform detectors over 1 hour. Note, too, that the frequency of re-trainings is extremely high in this setting, and that the difference would be substantially greater if training occur more sporadically.

Another interesting finding is that differences among algorithms are noticeable after some time, especially for large vectors. In general, OCNB

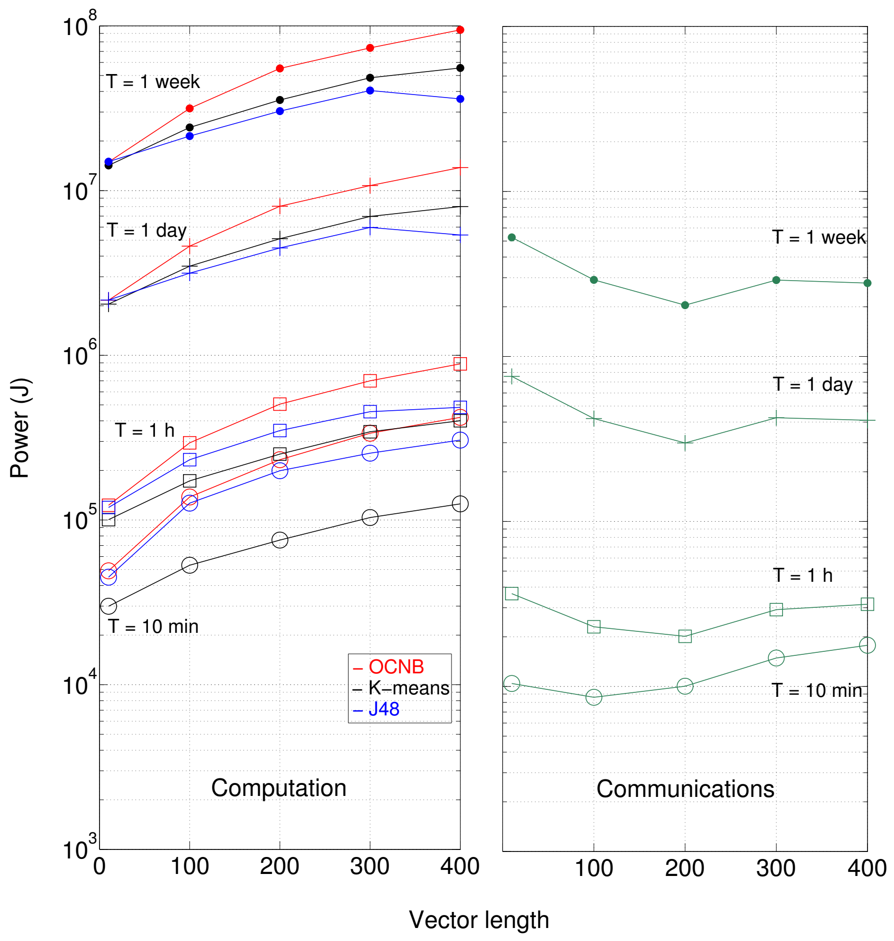

Figure 3: Average energy consumption for different detectors using the LL (left) and RR (right) strategies.

is much more demanding than K-means and J48 when vectors with a few hundred attributes are involved.

Finally, in order to contextualize the energy implications of constantly running a detector, we have measured the power consumed by some popular apps during 10 minutes (see Table 5). These apps are representative of three broad classes of popular activities: games, online social networking, and multimedia content. The amount of power consumed by the three ranges between approximately 550 J and 645 J, most of it being related to the

App	**CPU**	**Comms**	**Display**	**Total**
YouTube	30.11	12.59	508.90	551.59
MX Moto	129.24	5.75	509.54	644.52
Facebook	137.76	27.42	471.42	637.27

Table 5: Consumption (in Joules) of three popular apps during a time span of 10 minutes.

graphical user interface. For comparison purposes, running our detector in the device with the less demanding algorithm (J48) takes around 15 J per detection. At full throttle (i.e., around 224 detections per second) this implies a consumption of around 2 MJ in 10 minutes. Even if detection only takes place at a rate of 1 per second, the overall consumption in 10 minutes is still around 9 KJ. In contrast, outsourced detection using WPA, HTTPS, and high latency consumes around 112 J and 0.5 J in the same conditions, respectively.

The figures discussed above reinforce the conclusion that externalized operation of anomaly detection seems to be the only reasonable choice in terms of power consumption. However, given that cloud-based processing may raise some privacy concerns in certain applications, this also motivates the need for more lightweight anomaly detection techniques that may be suitable for on-platform operation.

6. Related Work

A substantial amount of recent works have approached the problem of detecting malware in smartphones using a variety of machine learning techniques [35]. As discussed before, these schemes attempt to identify where and how malware manifests by constantly monitoring various features that are checked against automatically learned models of good and/or bad behavior.

Andromaly [33] is a representative example of such systems. It uses dynamic analysis for periodically monitoring a number of features that are later used by anomaly detectors to classify apps as goodware or malware. The entire process executes locally in the device. Monitored features include CPU consumption, number of network packets, number of running processes and battery level. Feature vectors are classified using a variety of machine learning algorithms, inlcuding K-means, logistic regression, decision trees, Bayesian networks and naïve Bayes. Furthermore, the experiments reported in [33] provide good basis to understand which machine learning algorithms are superior in terms of detection performance.

AppProfiler [32] follows a somewhat similar approach, but combining tainting and static analysis to extract privacy-related behaviors. The scheme builds a knowledge base that maps app behavior with API calls observed during static analysis, providing the user with valuable information about their apps.

Crowdroid [3] is another anomaly-based malware detection system for Android devices that uses K-means as base classifier. The main difference with other approaches such as Andromaly or MADAM [10] is that Crowdroid is fully outsourced and operates from the cloud, whereas the other two approaches train their classifiers locally in the device.

In contrast with these and other similar schemes, a number of recent works have opted for a radically different approach based on maintaining a synchronized replica of the device in the cloud. ParanoidAndroid [29], Secloud [42] and CloudShield [1] are illustrative examples of such systems. In these cases, all security-related tasks, including monitoring, analysis and detection can be performed in an environment not exposed to battery constraints. Furthermore, multiple detection techniques can be applied simultaneously, as the clone can be easily replicated. One critical issue with these approaches is that keeping the clone synchronized involves a constant exchange of activity update packets. For example, experiments on Paranoid Android show that synchronizing the device with the cloud replicas require between exchanging traces at 2 KB/s for high-load scenarios and 64 B/s for idle operation. This definitely consumes power, although it may be worthwile if the clone is subject to intensive monitoring. From another perspective, such approaches have some serious privacy implications for many users.

Most proposals in this area do not provide an analysis of the cost in terms of power consumption of their schemes. In many cases, this issue is simply ignored. In other cases, power consumption is addressed rather superficially. For example, Andromaly is claimed to imply a 10% degradation of battery life when running in the worst scenario (i.e., 8 different classifiers using 30 features), while MADAM reports a power consumption overhead of around 5%. In both cases, however, it is unclear how this performance has been measured and whether the consumption exhibited is in the same conditions with and without the detector.

Offloading resource-intensive tasks to the cloud is a topic that has gained momentum in recent years. Several works (e.g., [22, 25, 36]) have addressed the issue of deciding whether *to cloud* is a better option than *not to cloud* for mobile systems. For example, in [25] it is shown that determining an energy efficient strategy is a complex task and require a fine characterization of the impact of several parameters, including the type of device and the application domain. Their approach focuses on three rather generic applications: word processing, multimedia and gaming for both laptops and mobile devices. Authors conclude that "cloud-based applications consume more energy than non-cloud ones" when using platforms such as mobile devices. In contrast, other works such as [36] and [22] show that offloading is generally profitable energy-wise, particularly for intensive computation tasks that require relatively small amount of communications.

A rather different approach is proposed in [31], where the authors explore collaborative strategies for a mobile sensing platform. The scheme adaptively changes the deployment strategy between local and external –though nearby– sensors with the aim of optimizing power consumption. Anomaly detection schemes could benefit from a similar approach where the nearby infrastructure cooperates in monitoring tasks, but we are not aware of any security-related application for smartphones exploring this possibility.

Finally, the technical issues involved on metering and modelling power consumption in mobile devices has received much attention lately. Built-in meters in platforms such as Android provide a coarse power profile and are inadequate for most applications. Our choice of Appscope [38] in this paper is motivated by its accuracy and because it provides energy consumption for each app and process, detailing how much corresponds to CPU usage, networking, touchscreen, etc. Other alternatives include PowerTutor [40], Systemtap [9], Eprof [27], and the schemes discussed in [28, 11, 24, 17]

7. Conclusions and Future Work

In this paper, we have discussed the power consumption trade-offs among various strategies for executing anomaly detection components directly on mobile platforms or remotely in the cloud. Both our theoretical analys and experimental results confirm that there is actually little choice but to offload everything to the cloud. Reasons for this include the differences between the energy efficiency of computation and communications in current platforms, and also various parameters related to the anomaly detection setting, such as the dataset sizes and the operation frequency.

We believe that the linear models provided in this work may be useful in other contexts to obtain estimates about the power consumption of different alternatives. Furthermore, such models can be easily extended to other machine learning algorithms by simply deriving the appropriate coefficients α and β.

Finally, in this paper we have only considered scenarios involving *one* device and the cloud. In a cooperative setting [39, 31], where a number of devices agree to help others in some computations (for example, when they are running out of battery), it is unclear what the best strategy would be. As users are increasingly equipping themselves with a variety of portable devices [6], strategies for distributing computational tasks among them so as to maximize some target energy-related goal (e.g., maximize the overall battery life of all devices) will have some value [31]. We intend to tackle this and other related issues in future work.

Acnowledgements

We are grateful to the anonymous reviewers for their insights and suggestions on earlier versions of this manuscript. This work was supported by the MINECO grant TIN2013-46469-R (SPINY: Security and Privacy in the Internet of You).

References

[1] Barbera, M.V., Kosta, S., Mei, A., Stefa, J., 2013. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing, in: Proc. of IEEE INFOCOM, pp. 1285–1293.

[2] Blasing, T., Batyuk, L., Schmidt, A., Camtepe, S., Albayrak, S., 2010. An android application sandbox system for suspicious software detection, in: 5th International Conference on Malicious and Unwanted Software (MALWARE 2010), IEEE. pp. 55–62.

[3] Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android, in: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM. pp. 15–26.

[4] Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: A survey. ACM Comput. Surv. 41, 15:1–15:58. doi:10.1145/1541880.1541882.

[5] Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A., 2011. Clonecloud: elastic execution between mobile device and cloud, in: Proceedings of the sixth conference on Computer systems, pp. 301–314.

[6] Conti, M., Das, S.K., Bisdikian, C., Kumar, M., Ni, L.M., Passarella, A., Roussos, G., Trster, G., Tsudik, G., Zambonelli, F., 2012. Looking ahead in pervasive computing: Challenges and opportunities in the era

of cyberphysical convergence. Pervasive and Mobile Computing 8, 2 – 21.

[7] De Luca, A., Hang, A., Brudy, F., Lindner, C., Hussmann, H., 2012. Touch me once and i know it's you!: implicit authentication based on touch screen patterns, in: Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems, ACM. pp. 987–996.

[8] Dediou, H., 2012. When will tablets outsell traditional pcs? URL: `http://www.asymco.com/2012/03/02/when-will-the-tablet-market-be-larger-than-the-pc-market/`.

[9] Dediu, H., Accessed February 2014. Systemtap. `https://sourceware.org/systemtap/`.

[10] Dini, G., Martinelli, F., Saracino, A., Sgandurra, D., 2012. Madam: a multi-level anomaly detector for android malware, in: Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security, Springer-Verlag. pp. 240–253.

[11] Dong, M., Zhong, L., 2011. Self-constructive high-rate system energy modeling for battery-powered mobile systems, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, ACM. pp. 335–348.

[12] Estévez-Tapiador, J.M., Garcia-Teodoro, P., Díaz-Verdejo, J.E., 2004. Anomaly detection methods in wired networks: a survey and taxonomy. Computer Communications 27, 1569–1584.

[13] Feizollah, A., Anuar, N.B., Salleh, R., Amalina, F., Maarof, R.R., Shamshirband, S., 2014. A study of machine learning classifiers for anomaly-based mobile botnet detection. Malaysian Journal of Computer Science 26.

[14] Fisher, D.H., 1987. Knowledge acquisition via incremental conceptual clustering. Machine learning 2, 139–172.

[15] Garcia-Teodoro, P., Díaz-Verdejo, J.E., Maciá-Fernández, G., Vázquez, E., 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security 28, 18–28.

[16] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. ACM SIGKDD Explorations Newsletter 11, 10–18.

[17] Hao, S., Li, D., Halfond, W., Govindan, R., 2012. Estimating android applications' cpu energy usage via bytecode profiling, in: Green and Sustainable Software (GREENS), 2012 First International Workshop on, IEEE. pp. 1–7.

[18] Hastie, T., Tibshirani, R., Friedman, J., Franklin, J., 2005. The elements of statistical learning: data mining, inference and prediction. The Mathematical Intelligencer 27, 83–85.

[19] Jakobsson, M., Shi, E., Golle, P., Chow, R., 2009. Implicit authentication for mobile devices, in: Proceedings of the 4th USENIX conference on Hot topics in security, USENIX Association. pp. 9–9.

[20] Jung, W., Kang, C., Yoon, C., Kim, D., Cha, H., 2012. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components, in: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM. pp. 353–362.

[21] Kranz, M., Mller, A., Hammerla, N., Diewald, S., Pltz, T., Olivier, P., Roalter, L., 2013. The mobile fitness coach: Towards individualized skill assessment using personalized mobile devices. Pervasive and Mobile Computing 9, 203 – 215. Special Section: Mobile Interactions with the Real World.

[22] Kumar, K., Lu, Y.H., 2010. Cloud computing for mobile users: Can offloading computation save energy? Computer 43, 51–56.

[23] Lin, Y.D., Lai, Y.C., Chen, C.H., Tsai, H.C., 2013. Identifying android malicious repackaged applications by thread-grained system call sequences. Computers & Security 39, Part B, 340 – 350.

[24] Nagata, K., Yamaguchi, S., Ogawa, H., 2012. A power saving method with consideration of performance in android terminals, in: Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on, IEEE. pp. 578–585.

[25] Namboodiri, V., Ghose, T., 2012. To cloud or not to cloud: A mobile device perspective on energy consumption of applications, in: IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 1–9.

[26] Nielsen, 2012. State of the appnation a year of change and growth in u.s. Technical Report.

[27] Pathak, A., Hu, Y., Zhang, M., 2012. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof, in: Proceedings of the 7th ACM european conference on Computer Systems, ACM. pp. 29–42.

[28] Pathak, A., Hu, Y., Zhang, M., Bahl, P., Wang, Y., 2011. Fine-grained power modeling for smartphones using system call tracing, in: Proceedings of the sixth conference on Computer systems, ACM. pp. 153–168.

[29] Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H., 2010. Paranoid android: versatile protection for smartphones, in: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 347–356.

[30] Quinlan, J.R., 1986. Induction of decision trees. Machine learning 1, 81–106.

[31] Rachuri, K.K., Efstratiou, C., Leontiadis, I., Mascolo, C., Rentfrow, P.J., 2014. Smartphone sensing offloading for efficiently supporting social sensing applications. Pervasive and Mobile Computing 10, Part A, 3 – 21. Selected Papers from the Eleventh Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2013).

[32] Rosen, S., Qian, Z., Mao, Z.M., 2013. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users, in: Proceedings of the third ACM conference on Data and application security and privacy, ACM. pp. 221–232.

[33] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y., 2012. "andromaly": a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems 38, 161–190.

[34] Shi, E., Niu, Y., Jakobsson, M., Chow, R., 2011. Implicit authentication through learning user behavior, in: Information Security. Springer, pp. 99–113.

[35] Suarez-Tangil, G., Tapiador, J.E., Peris, P., Ribagorda, A., 2013. Evolution, detection and analysis of malware for smart devices. IEEE Communications Surveys & Tutorials PP, 1–27. doi:10.1109/SURV.2013.101613.00077.

[36] Tandel, M.H., Venkitachalam, V.S., 2013. Cloud computing in smartphone: Is offloading a better-bet?

[37] Wang, C., Ren, K., Wang, J., 2011. Secure and practical outsourcing of linear programming in cloud computing, in: Proceedings of INFOCOMM 2011, pp. 820 – 828.

[38] Yoon, C., Kim, D., Jung, W., Kang, C., Cha, H., 2012. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring, in: USENIX Annual Technical Conference.

[39] Yu, P., Ma, X., Cao, J., Lu, J., 2013. Application mobility in pervasive computing: A survey. Pervasive and Mobile Computing 9, 2 – 17. Special Section: Pervasive Sustainability.

[40] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao, Z., Yang, L., 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM. pp. 105–114.

[41] Zhou, Y., Jiang, X., 2012. Dissecting android malware: Characterization and evolution, in: Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012).

[42] Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., Sanders, W., 2013. Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. Computers & Security .