SpotCheck: On-Device Anomaly Detection for Android

Mark Vella
Dept. of Computer Science, University of Malta
Msida, Malta
mark.vella@um.edu.mt

Christian Colombo
Dept. of Computer Science, University of Malta
Msida, Malta
christian.colombo@um.edu.mt

ABSTRACT

In recent years the PC has been replaced by mobile devices for many security sensitive operations, both from a privacy and a financial standpoint. Therefore the stark increase in malware targeting Android, the mobile OS with the largest market share, was bound to happen. While device vendors are taking their precautions with app-store and on-device scanning, limitations abound, mainly related to the malware signature-based detection approach. This situation calls for an additional protection layer that detects unknown malware that breaches existing countermeasures.

In this work we propose SpotCheck, an anomaly detector intended to run on Android devices. It samples app executions and submits any suspicious apps to more thorough processing by malware sandboxes. We compare Kernel Principal Component Analysis (KPCA) and Variational Autoencoders (VAE) on app execution representations based on the well-known system call traces, as well as a novel approach based on memory dumps. Results show that when using VAE, SpotCheck attains a level of effectiveness comparable to what has been previously achieved for network anomaly detection. Even more interesting, the KPCA anomaly detector managed comparable effectiveness even for the experimental memory dump approach. Overall, these promising results present a solid platform upon which to strive for an improved design.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Mobile platform security**; **Malware and its mitigation**.

KEYWORDS

Android malware, anomaly detection, memory dump analysis, kernel PCA, variational autoencoders

ACM Reference Format:
Mark Vella and Christian Colombo. 2020. SpotCheck: On-Device Anomaly Detection for Android. In *SINCONF '20: 13th International Conference on Security of Information and Networks, Novemeber 04–07, 2020, Istanbul, Turkey*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SINCONF '20, November 04–07, 2020, Istanbul, Turkey

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile malware is an ever-increasing concern given the sensitive data and transactions nowadays stored and carried out on mobile devices, surpassing PC usage in many ways. Android is the leader in the mobile OS market [8], and therefore the surge in malware targeting it in recent years comes as no surprise [22]. Google's official, and arguably the largest, source of Android apps have long taken the provisions of preventing malware to get to mobile devices in the first place through app scanning during the upload stage. Malware sandbox execution is a key enabler technology, combining static and dynamic code analysis, while attempting to be resilient to evasion techniques [23]. This protection layer complements other security mechanisms, such as digitally signed apps and highlighting of all those permissions considered to be dangerous. Google Play Protect[1] complements app store scanning at the device level, while in recent Android versions dangerous permissions require explicit user activation, possibly each time they are requested[2]. Yet, despite all these countermeasures there is no guarantee that malware won't get installed and eventually executed anyway. Certificate-based app tampering verification has been bypassed through implementation vulnerabilities [16, 27]. Furthermore, app execution obfuscation of sorts have been used to thwart sandbox detection [14], while recent malware showed that the accessibility permission is all that a malware actually needs to grant itself all the rest [24]. Social engineering tricks typically provide the missing pieces of the puzzle to put a successful attack together.

Existing limitations call for further security in-depth. Since the signature-based approach poses the main limitation, an effective additional layer must provide anomaly detection [11]. Anomaly detection builds a model of normal behavior by relying solely on a sufficiently large sample of benign apps. At runtime, those apps that deviate significantly from this model are flagged as suspicious, presenting possible malware. This contrasts with signature-based approaches that are devised to recognize known malware and their variants. Machine learning plays a central role through various clustering, classification and dimensionality reduction algorithms [4]. In this work we consider two options: Kernel PCA (KPCA) and Variational Autoencoders, for shallow and deep learning respectively [1], both previously experimented with for network anomaly detection.

As shown in Figure 1, SpotCheck is intended to operate on samples of on-device app execution segments, submitting apps with a sufficiently high anomaly score for deeper inspection by malware analysis. Rather than a standalone alert-raising monitor, SpotCheck acts as a precursor to malware triage. State-of-the-art malware analysis leverages machine learning to classify suspicious binaries

[1] see https://www.android.com/play-protect/
[2] see https://developer.android.com/guide/topics/permissions/overview

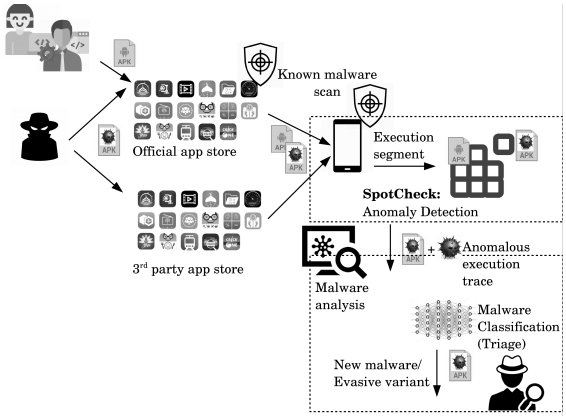

Figure 1: SpotCheck for Android: on-device anomaly detection, submitting suspicious apps for further analysis.

according to malware families, with deep learning-based classification operating on system call traces being particularly effective [10], prior to manual analysis by experts. SpotCheck aims to benefit from machine learning in a similar manner, using dynamic analysis to capture app behaviour in an obfuscation resilient manner. The well-established system call trace representation of app behaviour, as well as a more experimental process memory dump approach are taken into consideration. SpotCheck takes a sampling approach, conducting anomaly detection on execution segments. The net benefit of this precursor step to malware analysis is two-fold: first it can prioritize over which samples are submitted for malware analysis; secondly, by providing the associated anomalous execution trace along with the app itself, malware analysis can be more focused.

Experimentation was carried out with datasets comprising apps from Google Play and Virustotal. Results show that the use of KPCA and VAE for Android anomaly detection compares well to the network anomaly detection case. KPCA's performance is even more interesting for memory dumps, increasing its detection effectiveness even further. The corresponding VAE approach however was less effective, yet this is a deep learning model that calls for further exploration. Overall, we make the following contributions:

- We show that Kernel PCA and VAE's effectiveness for Android anomaly detection is comparable to the use of VAE for network anomaly detection;
- We propose an experimental memory dump representation for app behaviour, and which can be combined effectively with Kernel PCA anomaly detection;
- Two datasets, for benign and malicious app behavior, represented as system call traces and process memory dumps respectively.

2 BACKGROUND

SpotCheck's key design decisions concern the anomaly detection model and app behaviour representation.

2.1 Anomaly detection

The core premise of malware anomaly detection is that malware should look and/or behave differently from benign apps [4]. Therefore anomaly detection firstly has to model benign behavior, and secondly it needs some form of similarity measure from which to compute an anomaly score for the monitored apps. In proximity-based models malware is identified in terms of isolated datapoints, or else by forming its own clusters. Distance or density-based ones take a localized approach by considering only the closest points within a feature space, with malware expected to be excessively distant from the closest benign datapoint, or else located within a sparsely populated sub-space. These two approaches represent most of state-of-the-practice in network intrusion and fraud detection [5].

Spectral and statistical methods provide two further options. Spectral methods combine dimensionality reduction with deviation-based anomaly detection. In this approach, app (static or dynamic) features are mapped to a lower dimensional space using lossy compression. Principal Component Analysis [2] and Autoencoders (AE) [1] are common techniques. Whether using principal components or neural network weights, as computed/optimized from a benign-only dataset, a higher reconstruction error is expected for malware samples, therefore resulting in larger deviations from the input samples. Statistical models, on the other hand, assume that datapoints are sampled from a specific probability distribution. A datapoint is anomalous if its probability for that particular distribution falls below a certain threshold. While offering an intuitive approach to anomaly detection, the problem of probability distribution parameter estimation with high dimensional data, possibly including latent variables (i.e. the cause for anomaly is not even directly captured by the available, visible, dimensions), is not a trivial task [12]. Yet, this is exactly the problem addressed by VAEs.

2.2 Representing app behavior

SpotCheck takes a dynamic analysis approach to represent app behavior. Obfuscating malicious intent from dynamic analysis is harder compared to static analysis. While tricks, such as delaying of malicious code execution or detection of malware sandboxes, are still possible [14], ultimately malware has to run, especially whenever it gets installed on a target victim device. At that point, the necessary system calls have to be made and any supporting data objects are created in process memory as a result of Android API invocation. Capturing malware behaviour as sequence of function/system calls is a well-established technique [21]. Call tracing probes are also typically included as part of malware sandboxes, e.g. MobFS's API monitor[3]. An alternative approach to monitoring app execution through call sequences is to analyze the residue of that execution within process memory. That residue is made of the various data structures/objects that define the app state as a result of trace execution. Memory forensics [3], or the analysis of physical memory dumps, has received increased attention since the onset of advanced malware that does not leave any traces on disk. It has since become an essential tool for incident response. Yet this type of memory analysis is not suitable for non-rooted stock Android phones. Physical memory dumps require either the

[3]https://github.com/MobSF/Mobile-Security-Framework-MobSF

loading of kernel modules [25], or else the availability of Linux's `/dev/kmem` device file [28]. The latter option is no longer available to apps on non-rooted devices, while the former requires firmware replacement.

Process-level memory dumps, on the other hand, are unencumbered by these restrictions. In fact most stock Android devices come equipped with a runtime that supports an extended version of HPROF memory dumps[4], originally created for Java virtual machines. Specifically, HPROF dumps contain the garbage-collected heap, complete with class definitions for all object instances present in the dump. Full process heap dumps may also be supported. Artifacts from HPROF dumps are suitable for the purpose of capturing individual app behaviour, yet challenges abound. While classic memory forensics focuses on long-lived kernel-level dumps, heap objects may be short-lived [26], and therefore it may be the case that a substantial portion of app execution residue is lost by the time an HPROF dump is taken. Therefore, the timing of dump triggers is critical.

3 SPOTCHECK'S ARCHITECTURE

SpotCheck's key components are:

(1) Sampling of app execution, both in terms of system call traces and process memory dumps;
(2) A model of normal behavior synthesized from call traces/dumps using either KPCA or VAE; and a
(3) Distance metric for computing anomaly scores.

3.1 Sampling app execution

Since monitoring the entire app's execution is infeasible, we opt for sampling. The intuition is that when monitoring multiple runs, the sampling approach will eventually hit the sought after, discriminating, runtime behavior. We decide to explore both system call traces and process memory dumps to represent behaviour. The prior approach serves as baseline, being well-established for security monitoring purposes. The latter is an experimental lightweight approach that avoids code instrumentation, yet it relies on identifying those discriminating data objects in-memory, and which may be short-lived. In this mode, the need to capture representative execution samples becomes even more critical.

System call traces. Capturing Android app execution in terms of Linux system calls has been already widely explored for Android malware classification [10]. In fact, the system call layer provides a stable choke-point for higher-level Android API calls, that may vary across versions. We opt for a system call histogram, with each feature vector attribute representing the number of times its corresponding system call has been called. This is a common way to represent features derived from executables [19]. We avoid reliance on the exact sequence of system calls, e.g. through call graphs, since this approach would increase vector dimensions significantly and would therefore require a much larger sample of benign apps, at least one for each type of app in existence. The finalized feature vector structure for the system call histogram representation is the 86-feature vector:

$$x \stackrel{def}{=} < accept, access, bind, chdir, ..., writev >$$

where each feature is a system call count, possibly spanning multiple processes for the same app, for some execution sample.

Process memory dumps. The HPROF memory dump format provides an obvious choice in this case. Yet, choosing data objects with a high discriminating potential is not trivial. Unlike call traces there is no previous work to provide guidance. Among all Android and Java framework objects we opt for those service classes returned by `android.content.Context.getSystemService()`. These classes act as interfaces to services hosted by Android's system server and other native processes hosting Android services, e.g. the Telephony process. These processes implement all Android system services, accessible through the `android.os` namespace, and which in turn invoke `ioctl()` system calls into the binder IPC framework[5]. The primary assumption here is that since malware uses Android permissions in a different/suspicious manner, so are the resulting system service calls. As of its conception, the memory dump approach is bound to be more limited than the comprehensive system call tracing approach, given it has to rely solely on in-memory residue of execution traces.

The finalized feature vector structure for the HPROF histogram representation is the 72-feature vector:

$$x \stackrel{def}{=} < AccessibilityManager, ..., WindowManager >$$

where once again each feature is for individual apps, possibly spanning multiple processes.

In both representations features are scaled using an Attribute Ratio method that normalizes counts as a fraction of the total counts per vector: $\hat{x} \stackrel{def}{=} < a_i/||x||_1, ..., a_n/||x||_1 >$. The normalized total of counts is therefore 1 per datapoint ($||\hat{x}||_1 = 1$), offsetting irregularities derived from sampling executions of different lengths.

3.2 Kernel Principal Component Analysis (KPCA) for anomaly detection

KPCA is a non-linear variant of classic PCA and is what makes it suitable for anomaly detection, as shown for the case of network anomaly detection [2]. Like its linear counterpart it performs dimensionality reduction in a way that maximises information retention, expressed in terms of variance. Either eigen or singular value decomposition can be used to map from the original n-dimension feature space to a latent r-dimension one. By setting r to 2 or 3 it is possible to visualise high-dimensional datasets. Sticking to eigendecomposition (slower for PCA, but the only option for efficient KPCA), given a (centered) dataset X_n, the covariance matrix $X^T.X$ is first computed, followed by its eigendecomposition to $W.\lambda.W^{-1}$. The columns in W store the orthogonal eigenvectors, indicating directions of most variance. λ is a diagonal matrix of eigenvalues. W_r is the result of reordering W according to λ, with the columns associated with the largest eigenvalue order left-most, and subsequently dropping all but the first r columns. Latent space mapping is computed as $Z_r = W_r.X$, while the inverse transform comprises $X = Z_r.W_r^T$. Note that $W_r^T.W_r$ is the identity matrix only when $r = n$, and therefore constitutes a lossy transform otherwise.

[4] https://developer.android.com/studio/profile/memory-profiler

[5] https://source.android.com/devices/architecture/hidl/binder-ipc

Algorithm 1: SpotCheck's KPCA anomaly detector

Input: Mode [SysCall trace | HPROF dump], Threshold α,
Benign Apps X, Monitored Apps $x^{(1)}, ..., x^{(N)} \in X'$
Output: $MSE(x^{(i)}, \hat{x}^{(i)})$ Anomaly_Scores []

1 $W_r, W_r^T \leftarrow$ `Eigendecomposition` (X)
2 $\gamma \leftarrow$ `Grid_Search` (X)
3 Anomaly_Scores [] = {}
4 **for** $i \leftarrow 1$ **to** N **do**
5 $z^{(i)} \leftarrow$ KPCA_Transform$(x^{(i)}, W_r, \gamma)$
6 $\hat{x}^{(i)} \leftarrow$ KPCA_Transform$^{-1}(z^{(i)}, W_r^T)$
7 ReconErr = $MSE(x^{(i)}, \hat{x}^{(i)})$
8 **if** (ReconErr $> \alpha$) **then**
9 Anomaly_Scores []$\leftarrow (x^{(i)}$, ReconErr, Anomaly)
10 **else**
11 Anomaly_Scores []$\leftarrow (x^{(i)}$, ReconErr, Normal)
12 **return** Anomaly_Scores *[]*

KPCA provides non-linearity by means of kernel methods, i.e. retaining eigenvector orthogonality, but introducing linear separability directly in X_n by mapping it to a higher dimension $X_m = \phi(X_n)$, where $m > n$. The caveat is the increased computational work for producing the covariance matrix. As with other kernel methods this issue is addressed with the kernel trick, i.e. using a kernel function $k(X) = \phi(X^T).\phi(X)$ in the n-dimensional space. If it can be assumed that the higher-dimensional space follows a Gaussian distribution, the radial basis function (rbf) kernel can be used:

$$k(x, y) \overset{def}{=} e^{-\gamma||x-y||^2}, \text{where } \gamma = \frac{1}{2\sigma^2} > 0$$

with γ presenting a learnable parameter corresponding to a training dataset X. An optimal γ is typically computed using a grid search with the mean squared error (MSE) used as the reconstruction error. The premise for using KPCA for anomaly detection is that, during testing, data points $x^{(i)}$ form a different distribution than one from which the training dataset is derived, return a higher reconstruction error.

The KPCA anomaly detector is shown in Algorithm 1. The training dataset X, composed solely of benign apps, is used for computing W_r and W_r^T, as well as for searching for the optimal γ (lines 1-2). For each monitored app $x^{(i)} \in X'$, its latent representation in r-dimensions, $z^{(i)}$, is computed and subsequently recovered as $\hat{x}^{(i)}$ (lines 5-6). Due to the lossy transforms involved, $x^{(i)} \neq \hat{x}^{(i)}$ and their mean squared error (MSE) is taken as the reconstruction error (line 7). Whenever this error exceeds a threshold α, $x^{(i)}$ is flagged as anomalous (lines 8-11).

3.3 Anomaly detection with Variational Autoencoder (VAE) for anomaly detection

VAEs [12] approximate a probability distribution $P(X)$ to fit a data sample X using neural networks as shown in Figure 2. The decoder network $g_\theta(X|z)$ learns to *generate datapoints similar to* X using a prior distribution defined over a much simpler latent space $P(z)$, namely the standard isotropic Gaussian $\mathcal{N}(0, I)$. In the latent space datapoints z have a reduced dimension and are considered independent. In a typical Variational Inference fashion, the original feature space is assumed to follow a multivariate Gaussian (for continuous data) or Bernoulli (for binary) distributions. The complex relationship between the latent and original spaces is captured by θ, the weights for $g()$ [7].

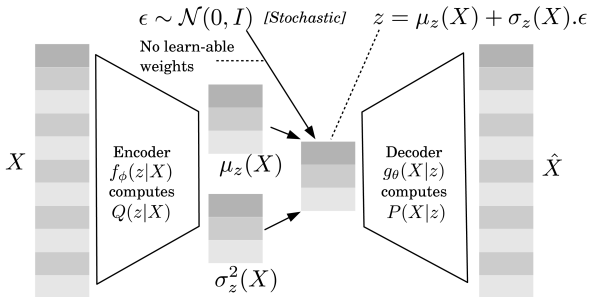

Figure 2: VAE topology: an encoder followed by a stochastic decoder, optimized wrt reconstruction probability $P(X)$.

The chosen VAE optimization process needs to maximize $P(X)$, the *reconstruction probability* i.e. the probability of computing a distribution that is most likely to in turn produce X, and which is exactly what renders VAEs suitable for anomaly detection. This way $g()$ is bound to produce outputs similar to X seen during training, but not to inputs taken from different distributions. $g()$'s outputs, denoted by $\hat{X}$, represents a generated datapoint for a given z. Whenever considering a single datapoint, $\hat{X}$ automatically represents the mean of the assumed distribution. When assuming a (multivariate) Gaussian, however, one needs to also consider σ^2, the covariance. This can either be represented as an additional output to $\mu = \hat{X}$ [12], or else is taken to be a fixed hyperparameter [7]. The role of the encoder $f_\phi(z|X)$ is to compute $Q(z|X)$ in a way that is as close as possible to $P(z|X)$, or the probability distribution in the latent space that is most likely to reproduce X.

Rather than composing $f_\phi(z|X)$ directly with $g_\theta(X|z)$, an intermediate function $z = h_\phi(\epsilon, X)$ is used instead for sampling datapoints z in the latent space. $f_\phi(z|X)$ computes μ_z and σ_z^2, the mean and covariance of the latent space respectively. Here $h_\phi(\epsilon, X) \overset{def}{=} \mu_z(X) + \sigma_z(X).\epsilon$ and $\epsilon \sim \mathcal{N}(0, I)$. In this manner no learnable weight is associated with a stochastic node, and back-propagation can proceed as usual. The resulting loss function is the negative of an objective function called the evidence lowerbound (ELBO):

$$-ve\ ELBO \overset{def}{=} \mathcal{D}_{KL}(Q(z|x^{(i)})||\mathcal{N}(0, I)) - \mathbf{E}_{Q(z|x^{(i)})}[log\ P(x^{(i)}|z)]$$

and which is defined in a way to keep $log\ P(X)$ close to 0, over choices for ϕ, θ. The first term on the right hand-side of Equation 3.3 is the Kullback-Leibler divergence between $Q(z|X)$ and the simplified distribution $P(z) = \mathcal{N}(0, I)$. This term penalizes any encodings produced by $Q(z|X)$ not following the assumed simple latent distribution, and which acts as a regularization term. The second term is the reconstruction error as defined using cross-entropy, or the expected encoding length needed to encode the reproduced X (as defined by the distribution parameters captured by $\hat{X}$) using $Q(z|X)$'s encoding.

Algorithm 2: SpotCheck's VAE anomaly detector

Input: Mode [SysCall trace | HPROF dump], Threshold α, Benign Apps X, Monitored Apps $x^{(1)}, ..., x^{(N)} \in X'$
Output: $P(x^{(i)})$ Anomaly_Scores []

1 $\phi, \theta \leftarrow$ Mini_Batch (X)
2 Anomaly_Scores [] = {}
3 **for** $i \leftarrow 1$ **to** N **do**
4 $\quad \mu_{z^{(i)}}, \sigma^2_{z^{(i)}} \leftarrow f_\phi(z|x^{(i)})$
5 $\quad$ **for** $l \leftarrow 1$ **to** L **do**
6 $\quad\quad z^{(i,l)} \sim \mathcal{N}(\mu_{z^{(i)}}, \sigma^2_{z^{(i)}})$
7 $\quad\quad \mu_{\hat{x}^{(i,l)}}, \sigma^2_{\hat{x}^{(i,l)}} \leftarrow g_\theta(\hat{x}^{(i,l)}|z^{(i,l)})$
8 $\quad$ ReconProb = $P(x^{(i)}) \leftarrow \frac{1}{L} \sum_{l=1}^{L} P(x^{(i,l)}; \mu_{\hat{x}^{(i,l)}}, \sigma^2_{\hat{x}^{(i,l)}})$
9 $\quad$ **if** (ReconProb $< \alpha$) **then**
10 $\quad\quad$ Anomaly_Scores []$\leftarrow (x^{(i)}$, ReconProb, Anomaly)
11 $\quad$ **else**
12 $\quad\quad$ Anomaly_Scores []$\leftarrow (x^{(i)}$, ReconProb, Normal)
13 **return** Anomaly_Scores *[]*

SpotCheck's anomaly detector, shown in Algorithm 2 is an adaptation of an existing one used for network anomaly detection [1]. As per its KPCA counterpart it is trained solely on benign call traces/dumps, but this time anomaly scores are based on the reconstruction probability $P(x^{(i)})$ (lines 9-12), in turn hinging on the learned ϕ, θ (line 1). We take two approaches for dealing with the computed covariance σ^2 at the feature space: *a)* as a learned layer, or as a *b)* hyperparameter fixed at 1. In both cases we assume a Gaussian distribution since we are dealing with continuous values (scaled frequencies in the 0-1 range). In the first case, for $L = 1$, $-\mathbf{E}_{Q(z|x^{(i)})}[log\ P(x^{(i)}|z)]$ becomes [17]:

$$NLL_{Gaussian} = \sum_i \frac{log\ \sigma^2_{\hat{x}^{(i)}}}{2} + \frac{(x^{(i)} - \mu_{\hat{x}^{(i)}})^2}{2\sigma^2_{\hat{x}^{(i)}}}$$

In the second case, fixing $\sigma^2 = 1$ renders the terms in Equation 13 with σ^2 constant, resulting in:

$$NLL_{Gaussian,\sigma^2=1} = \sum_i (x^{(i)} - \mu^2_{\hat{x}^{(i)}})$$

and which reduces to the commonly-used mean squared error (MSE). Therefore we use MSE as the loss function in this case. The KL divergence term has the closed-form [12]:

$$\mathcal{D}_{KL}(Q(z|x^{(i)})||\mathcal{N}(0, I)) = \frac{1}{2} \sum_i (1 + log\ (\sigma^2_{z^{(i)}}) - \mu^2_{z^{(i)}} - \sigma^2_{z^{(i)}})$$

We opt for the adam optimizer, with $L = 1$ as originally suggested [12]. Three encoder topologies are considered: 50-25, which is a baseline proportional to the one used for network anomaly detection [1]; an experimental 50-35-25 (gradual dimensionality reduction); and 50-25-2 that favours latent space visualization. Topologies are reversed for decoding. ReLU activation is used for all layers except for $\sigma^2(\hat{X})$, which uses linear activation as originally suggested [12], with a bias term of 1×10^{-4} to avoid a divide by zero when computing $NLL_{Gaussian}$. $\mu(\hat{X})$ uses sigmoid activation followed by feature scaling to match input feature scaling.

Lines 3-12 take the trained VAE and a set of input traces/dumps in order to compute anomaly scores. For each $x^{(i)}$, the latent space $\mu_{z^{(i)}}, \sigma^2_{z^{(i)}}$ vectors are computed (line 4) and then used to sample L $z^{(i,l)}$ points in latent space directly from $\mathcal{N}(\mu_{z^{(i)}}, \sigma^2_{z^{(i)}})$ (line 6). We set $L = 128$ in order to match the training batch size. The feature space distribution parameters are taken as the mean of all predicted individual datapoints (lines 7 and 8).

4 EXPERIMENTATION

SpotCheck experimentation concerned comparing Kernel PCA with VAE across the two chosen representations. A total of 3K apps were used: 2K benign apps downloaded from Google Play, and 1K malicious apps obtained from VirusTotal[6] under the academic collaboration scheme. The machine learning components were prototyped with Python 3.0 using Scikit-learn 0.22.2 and Keras 2.4.3/TensorFlow 2.3. App execution sampling uses Android Studio's 4.0 emulator with an Android Pie image (API level 28), Android Debug Bridge (adb) version 1.0.41, and Exerciser Monkey to simulate app interactions. Apps were downloaded from Google Play using gplaycli 3.29. System call tracing was implemented with frida-server 12.10.4. HPROF dumps were taken using adb, while Eclipse MAT 1.10/calcite v1.4 plugin was used for dump parsing.

4.1 Datasets

Two datasets[7], one for each representation type, were created. While both datasets are derived from a total of 3K apps, in reality a significantly higher amount of app executions was necessary. In the case of benign apps, a number of them did not result in meaningful runtime behavior from simulated interactions. As for malicious apps, a good number number of these were only provided in compiled bytecode form (dex) rather than executable apks. Others returned certificate failures due to them actually being (malicious) updates, or simply evaded our emulated environment. For each app, the android device emulator was started with a fresh state using the `-no-snapshot -wipe-data` flags. Once started, each app was subjected to: Dynamic instrumentation for system call tracing; Component traversal, as suggested in related work [10], to maximize runtime behavior coverage; and subsequently, a total of 200 (repeatable) pseudo-random UI events were sent. The complete cycle/app took approximately 8-10 minutes to complete on a Linux mint 20 host machine 5.4.0-39-generic kernel, Intel© Core™ i7 CPU 960 @ 3.20GHz × 4 processor, 15.6 GiB RAM, and NVIDIA GF116 GPU.

Figure 3 visualizes the dataset features for both representations in terms of mean (scaled) frequencies per system call/service class, and for both benign apps and malware. The system call histogram are characterized by a few dominant calls. In each case the three most frequent calls are `write`, `read`, and `ioctl`; and which correspond to input/output/ipc respectively, with `write` being particularly more frequent for malware than benign. `gettimeofday` and `recvfrom` are more frequent in benign apps. On the other hand `close` and `writev` rank higher for malware. `mmap` and `munmap` rank high in both cases, but even more so in malware.

[6] https://www.virustotal.com/
[7] Available at https://github.com/mmarrkv/spotcheck_ds

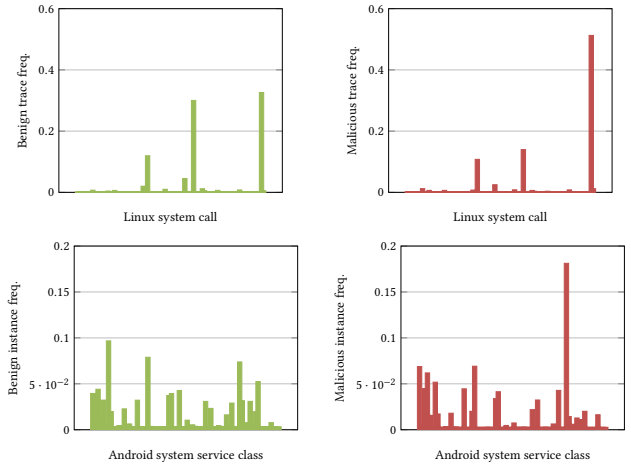

Figure 3: Scaled mean frequencies for system call trace (top) and process memory dump (bottom) features, for benign (left) and malware (right) apps.

In the case of benign HPROF dumps there are no dominating attributes, with the most frequent service class instances correspond to `AudioManager`, `DisplayManager`, `TelephonyManager` and `UserManager`. On the contrary, `TelephonyManager` dominates for malware apps, and which more than doubles the benign app frequency. The number of `AccessibilityManager` instances are also doubled, although not being as dominant as the previous class. Other system classes with a high frequency for malware are: `AlarmManager`, `AudioManager`, `ConnectivityManager`, `DisplayManager`, `InputMethodManager` and `SubscriptionManager`.

4.2 Results

Figure 4 shows a comparison of the classification accuracy obtained for KPCA and the 6 VAE configurations, across both app execution representations. We consider the AUC ROC as well as the f1 score, both common ways to measure classifier accuracy. In the case of f1 scores we consider step-wise anomaly thresholds and report the precision/recall for the maximum score obtained. The KPCA implementation uses the RBF kernel in order to match the VAEs Gaussian approximation. The Grid Search for γ uses 3-fold cross-validation in the 0.01-0.5 range. A two dimensional latent space is adopted for visualisation benefits. For VAE we try out 6 configurations in total. Configurations 1-3 use the negative log likelihood for Gaussian (NLL) usage in the loss function, and the 50-25, 50-35-25 and 50-25-2 topologies respectively. Configurations 4-6 follow the same order, but this time making use of the Mean Squared Error (MSE) loss function. In all cases 2,000 epochs was sufficient for loss function convergence. A 70/15/15 test/validation/test split is used for the benign datasets. Given the anomaly detection context, the malware datasets were only used for testing.

Starting with system call traces, the main observation is the very similar AUC ROC across the KPCA and all VAE configurations, falling within the 0.691 - 0.708 range, with the maximum score belonging to KPCA. However, in the case of f1 scores KPCA outperforms VAE substantially, obtaining 0.864/0.766/0.99 f1/precision/recall. The similar f1 scores across the VAE configurations, in the ranges of 0.509-0.513/0.577-0.624/0.435-0.455 for f1/recall/precision, justify the 2-dimensional (2D) latent layer topologies (3 & 6). The NLL/MSE approaches return similar scores. The 3 plots in Figure 5 (top) show the 2D latent space visualizations for the configurations having a 2-dimensional latent space, and which provide further insight into the obtained scores. In all cases there is substantial overlap in compressed latent spaces, with some visible separability emerging only for outliers. In the case of the VAE's latent spaces, it is visible that points are normally distributed, rather than simply compressed to a latent dimension. Interestingly, in the case of the NLL variant there is lower dispersion as compared to the MSE one. This fact is evidenced by the more compact x-axis (not visible) and follows directly the NLL loss function attempting to reconstruct the variance of the input dataset. Overall, all three visualizations highlight the difficult task at hand for both machine learning options in discriminating between benign/malicious apps.

Onto process memory dumps (Figure 5 - bottom), it is surprising to observe a more extensive visible separability across the two classes for the KPCA. As for the VAE the situation remains similar to system call traces. These observations translate to the KPCA's f1 shooting up to 0.88 for 0.97/0.8 recall/precision. At least from a KPCA point of view, these results show promise for the Android system service class representation derived from HPROF dumps. Yet, the very similar VAE AUC ROC range, 0.68-0.72, and f1 score range, 0.45-.052, excluding topology 6, indicate that we cannot dismiss VAE as yet. For VAE it is noteworthy that: i) All configurations register a substantial increase in recall (0.81-0.9) however at the cost of a dip in precision (0.34-0.37); ii) Topology 6, that makes use of the MSE loss function, is less accurate, and therefore indicating that there could be cases where fixing $\sigma^2_{\hat{x}}$ may not be a good idea.

4.3 Discussion

KPCA & VAE for Android anomalous system call trace detection. KPCA and VAE were chosen as starting points for SpotCheck given the promising results demonstrated for network traffic [1, 2]. Overall, the accuracy scores obtained by both KPCA and VAE for Android anomaly detection, using the Linux system call trace representation, compare well to those obtained for network anomaly detection using the NSL-KDD benchmark [1]. In that case the registered AUC ROC for KPCA/VAE across the DoS-Probe-R2L-U2R attack categories was 0.590/0.795-0.821/0.944-0.712/0.777-0.712/0.782. The main difference in our case being KPCA outperforming VAE, especially when considering the 0.861 vs 0.513 f1 scores. In the case of network anomaly detection, the only particularly higher score compared to Android was registered for the Probe category. This is a very particular type of attack (pre-step) which is significantly noisier than normal traffic. Concluding, both KPCA and VAE can be considered to have been successfully ported from the network traffic context, and therefore, when evolving SpotCheck's architecture further, none of the anomaly detectors is to be unnecessarily overlooked, at least for the system call trace representation.

Process memory dumps. When designing SpotCheck we also experimented with the possibility of detecting anomalies inside HPROF dumps. As of the onset, this approach has the disadvantage

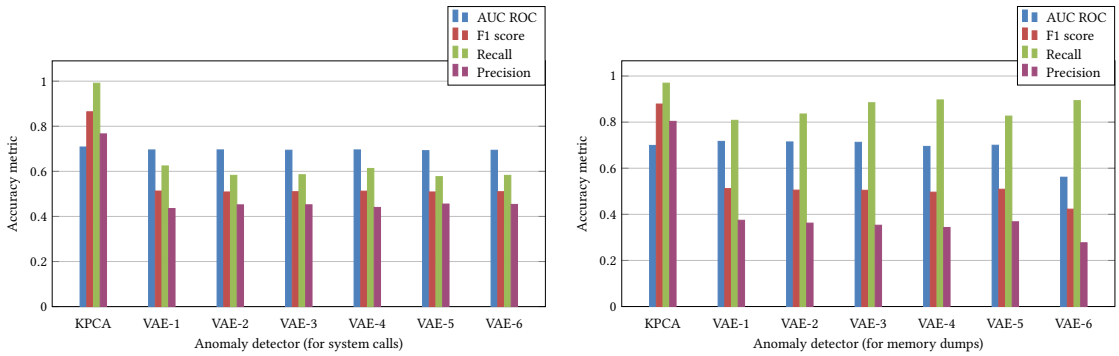

Figure 4: Classification accuracy for system call traces (left) and memory dumps (right).

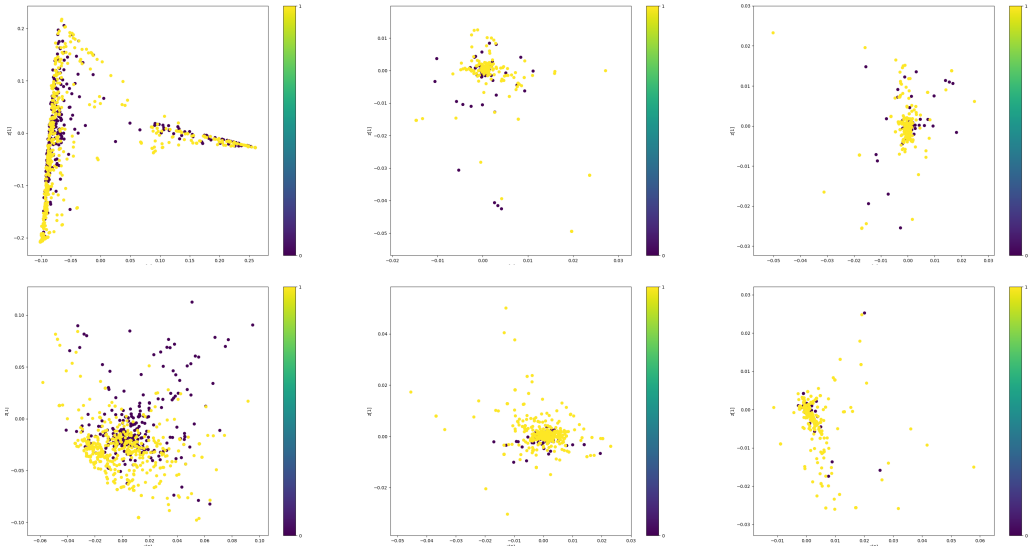

Figure 5: Latent space visualization - KPCA, VAE-NLL, VAE-MSE for system calls (top) and memory dumps (bottom).

of having to work solely with the residue of execution, rather than directly monitoring it. Yet, in combination with the system service call representation, the KPCA detector registers better effectiveness. On the contrary, all VAE configurations have their precision impacted. While the obtained AUC ROC scores do not allow us to commit exclusively to KPCA as of this point, results do call for experimenting with further VAE topologies, at least where HPROF dumps are concerned. Furthermore, results justify the additional learning required to also compute $\sigma^2_{\hat{x}}$.

Improving app behavior representation. Despite the obtained comparable effectiveness as per the network traffic context, along with a successful application to process memory dumps, the current AUC ROC scores leave room for improvement. A compelling idea in this regard is to combine the call tracing and memory dump approaches into a single *online object collection*. The combined approach entails tracing just the `getSystemService()` API call, and at which point to dump the corresponding service class instance from memory. In doing so, this combined approach addresses the requirement to time memory dumps in a way to coincide with the in-memory presence of the sought-after heap objects.

5 RELATED WORK

The use of machine learning for computer security is nowadays the state-of-the-practice [5], with early successes in spam detection using Naives Bayes classification being followed by applications in intrusion detection, malware analysis and fraud detection. The use of deep learning is a more recent effort. Plain Autoencoders (AE) for malware detection take a spectral approach to malware detection [20]. Stacking multiple AEs and appending with fully connected layers, forming a deep belief network, provide effective architectures for malware classification [9, 10]. For network anomaly detection VAEs give better results than AEs [1], with a particular study suggesting that models may be improved further with supervised learning [15]. In a context where deep learning is under the spotlight, experimentation with kernel methods is still ongoing and yielding promising results [2].

On other hand, the use of machine learning for memory forensics is still in its early stages of experimentation, with efforts working directly with raw process memory [13] also being proposed. With SpotCheck we avoid working with raw images of any sort in a context where routines to decode assembly instructions, or parse data objects, are readily-available. If we had to work with raw images, a deep network would have to dedicate a number of layers just to learn these routines. Yet, the overall net benefit of such an approach is unclear given the well-specified nature and availability of these decoders/parsers. A similar discussion applies to approaches attempting to apply the popular Convolutional Neural Networks (CNN) [6, 18] over visualisations of executable binaries. The main risk here is that malware can in practice employ multiple stages of unpacking/dynamic loading, rendering this approach only effective to detect unpackers/deobfuscators, but which however may also be employed by benign apps.

6 CONCLUSIONS & FUTURE WORK

In this paper we proposed SpotCheck, an on-device anomaly detector for Android malware. Anomaly scores are computed from samples of app execution, captured either using the well-established system call trace method, or the more experimental process memory dumps in HPROF format. Anomalies are submitted for deeper inspection by malware analysis. Results obtained from experimentation with 3K apps show that we manage to reproduce the level of effectiveness within an Android anomaly detection context, what previously had been done with VAEs for network anomaly detection. Even better results are produced using KPCA.

Moreover, a major result of this work concerns the effectiveness of Android system service classes, as derived from the memory dumps, for anomaly detection. When provided as input to the KPCA detector, the overall effectiveness improves further still. While results are less exciting for VAE over memory dumps, they provide an avenue for further exploration, especially given that the obtained effectiveness results leave room for improvement. Another exploration avenue is planned along the lines of combining the system call trace and memory dumps representations into a single one, comprising the timely dumps of individual memory objects. Finally we need to close the loop by considering how existing malware analysis sandboxes can benefit from the identified anomalous execution traces. In this regard we intend to experiment with execution markers, i.e. instrument apps in a way to specify the execution paths associated with the detected anomalies.

ACKNOWLEDGMENTS

This work is supported by the LOCARD Project under Grant H2020-SU-SEC-2018-832735.

REFERENCES

[1] Jinwon An and Sungzoon Cho. 2015. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE* 2, 1 (2015), 1–18.
[2] Christian Callegari, Lisa Donatini, Stefano Giordano, and Michele Pagano. 2018. Improving stability of PCA-based network anomaly detection by means of kernel-PCA. *International Journal of Computational Science and Engineering* 16, 1 (2018), 9–16.
[3] Andrew Case and Golden G Richard III. 2017. Memory forensics: The path forward. *Digital Investigation* 20 (2017), 23–33.
[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 1–58.
[5] Clarence Chio and David Freeman. 2018. *Machine learning and security: Protecting systems with data and algorithms.* " O'Reilly Media, Inc.".
[6] Zhihua Cui, Fei Xue, Xingjuan Cai, Yang Cao, Gai-ge Wang, and Jinjun Chen. 2018. Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3187–3196.
[7] Carl Doersch. 2016. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908* (2016).
[8] GlobalStats. [n.d.]. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide[Accessed:02.09.2020]
[9] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. 2016. DL4MD: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*. The Steering Committee of The World Congress in Computer Science, Computer …, 61.
[10] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4MalDroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 104–111.
[11] Hahnsang Kim, Joshua Smith, and Kang G Shin. 2008. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. 239–252.
[12] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).
[13] MA Ajay Kumara and CD Jaidhar. 2017. Leveraging virtual machine introspection with memory forensics to detect and characterize unknown malware using machine learning techniques at hypervisor. *Digital Investigation* 23 (2017), 99–123.
[14] Yonas Leguesse, Mark Vella, and Joshua Ellul. 2017. AndroNeo: Hardening Android Malware Sandboxes by Predicting Evasion Heuristics. In *IFIP International Conference on Information Security Theory and Practice*. Springer, 140–152.
[15] Manuel Lopez-Martin, Belen Carro, Antonio Sanchez-Esguevillas, and Jaime Lloret. 2017. Conditional variational autoencoder for prediction and feature recovery applied to intrusion detection in IoT. *Sensors* 17, 9 (2017), 1967.
[16] Michael Mimoso. [n.d.]. Android Vulnerability Enables Malicious Updates to Bypass Digital Signatures. https://threatpost.com/android-vulnerability-enables-malicious-updates-to-bypass-digital-signatures/101200/[Accessed:02.09.2020]
[17] David A Nix and Andreas S Weigend. 1994. Estimating the mean and variance of the target probability distribution. In *Proceedings of 1994 ieee international conference on neural networks (ICNN'94)*, Vol. 1. IEEE, 55–60.
[18] Rachel Petrik, Berat Arik, and Jared M Smith. 2018. Towards Architecture and OS-Independent Malware Detection via Memory Forensics. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2267–2269.
[19] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 11–20.
[20] Joshua Saxe and Hillary Sanders. 2018. *Malware Data Science: Attack Detection and Attribution.* No Starch Press.
[21] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S Movva, and Srinivas Mukkamala. 2011. Malware detection using assembly and API call sequences. *Journal in computer virology* 7, 2 (2011), 107–119.
[22] Sophos. [n.d.]. Sophos 2020 Threat Report. https://www.enterpriseav.com/datasheets/\sophoslabs-uncut-2020-threat-report.pdf[Accessed:02.09.2020]
[23] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1808–1815.
[24] Lukas Stefanko. [n.d.]. Insidious Android malware gives up all malicious features but one to gain stealth. https://www.welivesecurity.com/2020/05/22/insidious-android-malware-gives-up-all-malicious-features-but-one-gain-stealth/[Accessed:02.09.2020]
[25] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G Richard. 2012. Acquisition and analysis of volatile memory from android devices. *Digital Investigation* 8, 3-4 (2012), 175–184.
[26] Mark Vella and Vishwas Rudramurthy. 2018. Volatile memory-centric investigation of SMS-hijacked phones: a Pushbullet case study. In *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 607–616.
[27] Peng Xiao, Aimin Pan, Lei Long, and Yang Song. [n.d.]. Android Vulnerability Enables Malicious Updates to Bypass Digital Signatures. https://threatpost.com/android-vulnerability-enables-malicious-updates-to-bypass-digital-signatures/101200/[Accessed:02.09.2020]
[28] Haiyu Yang, Jianwei Zhuge, Huiming Liu, and Wei Liu. 2016. A tool for volatile memory acquisition from Android devices. In *IFIP International Conference on Digital Forensics*. Springer, 365–378.