The Case for Native Instructions in the Detection of Mobile Ransomware

Nada Lachtar, Duha Ibdah, and Anys Bacha *Member, IEEE*

Abstract—Recently, the mobile segment observed the emergence of a new class of malware known as ransomware. In 2017, more than 468,830 unique mobile ransomware samples were discovered marking a 415% year-over-year increase in new ransomware. This trend presents a major concern for mobile users as they increasingly rely on their devices to safeguard sensitive information. Previous solutions have relied on high level bytecode and XML-based permission files to detect malicious applications. Unfortunately, attackers are resorting to obfuscation techniques that involve repackaging apps with malicious content directly in native machine code. As such, the aforementioned methods are insufficient for detecting modern mobile ransomware. To address these concerns, this work evaluates the effectiveness of using native instructions in detecting ransomware. We characterize different machine learning models and demonstrate that opcodes in native instructions can be used for detecting mobile ransomware with near ideal accuracy. In addition, we make the observation that the number of instruction opcodes that contribute to the detection of ransomware is significantly less than the full range of supported opcodes within a contemporary instruction set. Finally, we evaluate the robustness of our approach against six different ransomware families available in a state-of-the-art Android malware dataset.

Index Terms—Ransomware, Android Malware, Mobile Security, Instruction Set Architecture, Machine Learning

1 Introduction

Security has become a critical design factor for computing systems today. As more of our personal data is collected, created, and consumed through interconnected devices, information security is becoming increasingly important. The rapid growth in mobile applications and their underlying devices is driving the need for scalable designs that can autonomously safeguard digital content from malicious software.

It was estimated that in 2017, over 48 million apps were downloaded per day with the Android platform representing the largest segment of active devices consuming such mobile apps [1]. With Android being the dominant mobile platform possessing over 87% of the market share, cybercriminals have shifted their efforts towards tailoring malicious apps that can exploit this environment. For instance, Google announced that it withdrew 700,000 bad applications from its Google Play store. This occurred despite all the security measures Google has in place before developers can make their apps publicly available. This underscores the need for device level protection that can counter such malicious applications and serve as an additional layer of defense. This trend is exacerbated by app downloads from third parties such as Anzhi and AppChina that don't offer strict services to preclude malicious content from being added. An experiment conducted by Allix et al. [2] demonstrated that 75% and 50% of app downloads from Anzhi and AppChina respectively included malware.

Recently, the mobile segment observed the emergence of a new class of malware known as ransomware. Ransomware is a type of malware that involves encrypting the user's data or locking their device. The malware then extorts the victim to pay a ransom in return for decrypting their data or unlocking their device. According to Trend Micro, more than 468,830 unique mobile ransomware samples were analyzed marking a 415% increase in new ransomware relative to 2016 [3]. This trend presents a major concern for mobile users as they increasingly rely on their devices to safeguard sensitive information that they consume on a daily basis.

In this paper, we demonstrate the effectiveness of using native instructions in detecting ransomware. We show that opcodes in native instructions can be used as features for training machine learning models to detect mobile ransomware with high accuracy. We evaluate the effectiveness of this approach by extensively testing different machine learning models on the ARMv7 instruction set architecture. We show that the proposed approach can achieve near ideal accuracy, offering a detection rate of 99.8%. Finally, we evaluate the robustness of our approach against six ransomware families available in a state-of-the-art Android malware dataset [4].

Overall, this paper makes the following contributions:

- Evaluates the effectiveness and relevance of using native instructions for detecting mobile ransomware using one of the most popular instruction set architectures for mobile devices, namely ARM.
- Characterizes multiple machine learning algorithms and their application to detecting ransomware while showing that near ideal detection accuracy can be achieved using state-of-the-art Android malware dataset.
- Makes the observation that the number of instruction opcodes that contribute to the detection of ransomware is significantly less than the full range of supported opcodes within a contemporary instruction set.

The rest of this paper is organized as follows: Section 2 provides background information. Section 3 presents the methodology and results of our evaluation. Section 4 details related work; and Section 5 concludes.

2 Background

In this section, we provide the necessary background on the Android environment and machine learning algorithms to enable the reader to understand our learning-based detection approach for defending against mobile ransomware.

- *The authors are with the Computer and Information Science Department, the University of Michigan, Dearborn, MI, 48128.*

2.1 The Android Platform

Android is an open source software stack that is optimized to enable a wide range of mobile devices [5]. Although Android is based on the Linux operating system, it consists of many components that are uniquely tuned to enable the execution of apps on mobile devices. Android applications are delivered to the end user as a bundle known as the Android Package Manager (APK). This package consists of various resources that include a description of the resources the application consumes and an executable in the form of Java bytecode. The Java bytecode in Android is known as a Dalvik executable (`dex` file). Once the package is installed, the app can interact with the application framework layer. This layer is responsible for providing apps with the various services it needs, such as the sending of notifications and location information.

Another major component of the Android platform consists of the Android Runtime system (ART). ART is a successor to the Dalvik machine implementation that relied on just-in-time (JIT) compilation techniques for running user apps. Unlike the Dalvik machine, ART relies on ahead-of-time (AOT) compilation for running apps. As such, instead of compiling Java bytecode every time an app is launched in a JIT fashion, ART compiles the code natively onto the device during the installation process, then re-uses the machine code every time the app is relaunched. This is accomplished by converting the `dex` file into an `OAT` file (ahead-of-time file) through a `dex2oat` module. This approach significantly speeds up application performance (2x - 3x) compared to the JIT approach.

2.2 Supervised Learning

Machine learning algorithms can be broadly categorized into one of three learning approaches: supervised, unsupervised, and reinforced learning. In this paper, we focus on supervised learning algorithms that are relevant to this work [6], [7].

Random Forest. A random forest is an ensemble learning algorithm that relies on a collection of decision trees for classifying data. Each tree within the random forest is constructed by applying an algorithm $\mathcal{A}$ on a training set $\mathcal{S}$. Predictions in a random forest are generated through a majority vote of all the predictions that are sourced from the individual decision trees.

Support Vector Machines (SVM). This is a discriminative algorithm that searches for the best bisecting hyperplane that maximizes the margin between classes. SVM relies on slack variables ξ that dictate how many points the algorithm can afford to misclassify during the training phase. We use the linear SVM classifier which is expressed by equation (1). Here $\boldsymbol{x} \in \mathbb{R}^n$ is the input vector, $\boldsymbol{w}$ is a weight vector, b is the bias, C is a regularization parameter, and y is the class label. We also consider "kernel tricks" (polynomial and radial basis functions (RBF)) that can make data linearly separable by mapping them into higher dimensional feature spaces.

$$\underset{w,\xi}{min}\,(||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i) \tag{1}$$

$$s.t. \quad \forall i \quad y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b > 1 - \xi_i \quad and \quad \xi_i > 0$$

K-Nearest Neighbors (KNN). KNN is a non-parametric machine learning algorithm that relies on proximity information for classifying data. More formally, for a training set $\mathcal{S}$ consisting of points and labels $(\boldsymbol{x}_1, y_1)...(\boldsymbol{x}_m, y_m)$, then for each $\boldsymbol{x}$, we return the majority label among $\{y_{\pi_i(x)} : i \leq k\}$ where $\pi_i(x)$ represents the i^{th} closest point to $\boldsymbol{x}$ in distance.

Naïve Bayes. This classifier is a generative algorithm that relies on Bayes theorem for making predictions. Unlike discriminative models, generative models are concerned with learning the underlying distribution of the data. As such, we are interested in learning the probability $P(\boldsymbol{x}|y)$ instead of $P(y|\boldsymbol{x})$ where y is a label belonging to some finite set $\{0, 1, ..., m\}$. Because we make the "naïve" assumption that features of input $\boldsymbol{x}$ are independent, the prediction for $y = l$ where $l \in \{0, 1, ..., m\}$ can be expressed by equation (2). This equation allows for label l to be predicted for a new input vector $\boldsymbol{x} \in \mathbb{R}^n$.

$$\underset{l}{argmax}\, P(y = l) \prod_{j=1}^{n} P(x_j|y = l) \tag{2}$$

Logistic Regression. This is an algorithm that models the probability of a class with label l occurring, provided some input $\boldsymbol{x} \in \mathbb{R}^n$. Since probabilities are non-linear in nature, the algorithm employs a logistic function, σ ("sigmoid") to introduce non-linearity and produce a probability within the range $[0, 1]$. As such, it uses a linear function that passes through a sigmoid resulting in the following equation:

$$\boldsymbol{w}^T\boldsymbol{x} = w_0 + w_1x_1 + ... + w_nx_n \tag{3}$$

$$P(y = l|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x}) \tag{4}$$

Where the sigmoid function is expressed as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

Artificial Neural Networks. This model consists of nodes interconnected through one or more layers that can be thought of as a directed acyclic graph whose nodes correspond to neurons and the edges correspond to the outputs. The output of a neuron in a given layer is described through equation (6) where $\boldsymbol{w}$ represents the weights of a neuron k in layer l, $\boldsymbol{x} \in \mathbb{R}^n$ is an input vector from the previous layer $l - 1$, and b_k is a bias input for the neuron. In our work, we use a rectified linear unit (ReLU) as the activation function f described by equation (7).

$$h_k^l(\boldsymbol{x}) = f(\sum_{j=1}^{n} w_{kj}x_j + b_k) \tag{6}$$

$$f(x) = \max(0, x) \tag{7}$$

3 Evaluation

3.1 Experimental Framework

We conducted experiments using 2148 ransomware samples available in [4]. In addition to ransomware samples available in [4], we used data from [2] as a baseline for benign Android apps. We allocated 80% of our dataset for training our models and kept the remaining 20% for testing purposes. To generate `OAT` files for the APK packages within our dataset, we created a framework based on the Android Open Source Project (AOSP) 6.0.1 release and built it with the `userdebug` option for ARMv7. We created and tested models for all the algorithms described in section 2. We used TensorFlow 1.8.0 and scikit-learn 0.19.1 libraries with Python 3.6 to implement our models.

We tested our models with different parameters in order to determine the optimal settings for each algorithm. Table 1 summarizes the optimal parameters that yielded the best detection results in our evaluation. We tested our KNN model with various settings for $k \in [1, 60]$. We explored multiple configurations for our neural network. This included using two hidden layers with the first layer set to 1024, 2048, and 4096 nodes and the second hidden layer set to 32,64,128, and 256. We also ran experiments with three hidden layers using 1024 nodes in the first two hidden layers and 128 nodes in the third hidden layer. Our random forest model was validated with a range of decision trees between $[1, 60]$. Our SVM settings for the polynomial and radial basis function algorithms were tested with the `degree` and γ (variance parameter) parameters set to 3 and $\gamma \in [10^{-6}, 10^{-1}]$ respectively.

Algorithm	Parameters
Random Forest	Estimators: 9
SVM RBF	$\gamma = 10^{-4}$
SVM Polynomial	Polynomial degree: 3
KNN	k: 2
Artificial Neural Network	hidden layer 1: 1024 nodes, hidden layer 2: 64 nodes, batch size: 10, gradient method: Adagrad

TABLE 1: Summary of ideal model parameters used in the evaluation.

3.2 Feature Selection and Training

A primary objective of our approach is to evaluate the ability to defend against native code repackaged into mobile apps. To this end, a key component in our evaluation relates to the extraction and pre-processing of native opcodes from Android applications and using them as features for our models. We constructed features for ARMv7 instructions by building a dictionary that consists of all the unique opcodes present in 15,126 Android apps. The baseline feature set that we used for the ARM architecture included 5014 features. Furthermore, we evaluated the impact of selecting different opcodes as features on the accuracy of our models. To achieve this, we employed dimensionality reduction techniques that rely on the Principle Component Analysis (PCA) algorithm to determine the most relevant features for detecting ransomware.

Prior to training our models, we pre-processed the dataset that is initially in Dalvik bytecode form (`classes.dex`). Our framework handled the conversion process of such `dex` files into the corresponding OAT format. This process entails compiling the `dex` files into a format that conforms to a standard Executable and Linkable Format (ELF). To keep the design compatible with the runtime resources available on mobile devices, we leveraged modules that are part of the Android Open Source Project (AOSP) to achieve this conversion. Upon completion of the conversion process, our framework extracted the instruction opcodes by parsing the `.text` segment of each OAT file. In addition, the frequency information of each opcode within the `.text` segment is logged and added to a dictionary that maintains all the valid opcodes within the dataset. The dictionary is used for creating an encoder that is responsible for generating the features that will be used in the final stage of the training phase. In our design, the encoder is generated after all the opcodes have been observed. The encoder is then deployed to translate the parsed opcodes into features that could be used for training the different classifiers presented in section 2.

The detection phase is similar to the training phase. It consumes the OAT image of a given app and extracts the appropriate instruction opcodes from the `.text` segment of the aforementioned image. The framework computes the frequency information of the opcodes and feeds it to the encoder. The encoder ensures that the opcodes are properly encoded into the appropriate fields of the input vector which is then forwarded to a pre-trained classifier for evaluation.

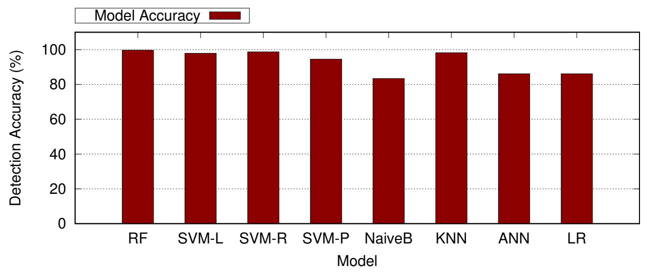

Fig. 1: Summary of detection accuracy using different machine learning models.

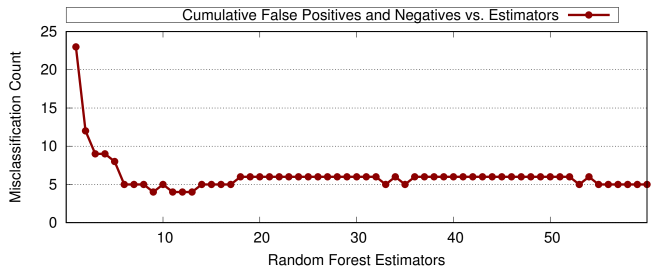

Fig. 2: Number of misclassified apps as a function of estimators in random forests.

3.3 Model Accuracy

Figure 1 shows the accuracy of our models in classifying apps within our dataset. We observed that four out of eight models are able to achieve detection accuracy above 99% with the random forest (RF) performing the best at 99.87%, followed by KNN at 99.83%. Further analysis showed that for most models, normalizing the features improved the accuracy with the exception of random forest. Random forest performed slightly better when trained on non-normalized data and is able to achieve 99.90%. Other models such as linear SVM (SVM-L) and SVM RBF (SVM-R) performed relatively well with detection accuracy of 99.7% and 99.6% respectively. However, the remaining models, including SVM polynomial (SVM-P), Naïve Bayes (NaïveB), artificial neural network (ANN), and logistic regression (LR) under performed significantly with a 10% - 15% drop in accuracy.

We further characterized the top performing models to determine the best model for ransomware detection in mobile systems. Figure 2 illustrates the number of misclassifications that occurred with the random forest model as a function of estimators. We observed that this model performs the best when it is configured with 9 estimators misclassifying only 4 apps. A similar experiment with KNN and SVM-RBF showed that the best results are obtained when using $k = 2$ (5 misclassifications) and $\gamma = 10^{-4}$ (12 misclassifications). This is summarized in figures 3 and 4. We did not perform any tuning in the case of SVM linear since most of the parameters were fixed. Figure 5 provides a breakdown of the misclassified data into false positives and false negatives. A false positive corresponds to the case where we misclassify a benign app as ransomware. On the other hand, a false negative is more serious and corresponds to the case where we classify ransomware as a benign app. We observed that the random forest model performed better since it has the least number of misclassifications and 0 false negatives.

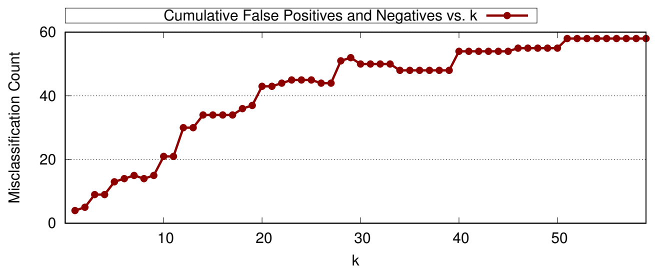

Fig. 3: Number of misclassified apps as a function of k in KNN.

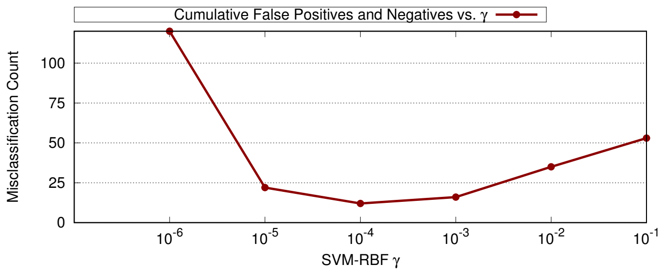

Fig. 4: Number of misclassified apps as a function of γ in SVM-RBF.

3.4 Dimensionality Reduction

To understand how instruction opcodes contribute to ransomware detection, we employed dimensionality reduction techniques through PCA. We observed that 6% of the principle components explain 50% of the variance across the full feature set. Furthermore, we found that 99% of the variance is explained by 60% of the principle components. Additional testing showed that we could significantly reduce the feature count without compromising the accuracy of the model. For example, in the random forest model we are able to achieve the same accuracy of 99.87% with less than 3% of the features.

4 Related Work

Several studies have explored techniques to defend against ransomware. For example, Kharraz et al. [8] proposed a solution that temporarily monitors file access patterns in an artificial environment. The solution then tracks permission changes and entropy levels of the written data to detect ransomware activity. However, the solution suffers from the inability to detect ransomware that stall their encryption activity until the virtual environment is removed. Other work [9] explored the monitoring of system-level crypto services and holding generated cryptographic keys in escrow. Such keys can be retrieved later to recover encrypted files. However, this approach can be circumvented by directly embedding encryption algorithms within the application. Other techniques include the sampling of hardware performance counters to infer malicious execution [10]. However, a recent study by Huang et al. [11] demonstrated the inadequacy of this approach in detecting malware when fused with benign applications. Other work by Chen et al. [12] examined user finger movements on mobile devices as a metric for distinguishing between benign and malicious activity. However, the approach lacks the ability to detect locker-based ransomware designed to lock a user's device without resorting to background encryption transactions. Furthermore, other work [13] examined static analysis techniques for detecting malware. However, such techniques focus on analyzing either high level bytecode or permissions files that are ineffective in dealing with obfuscation techniques that involve repackaging apps with malicious content directly in native form [4].

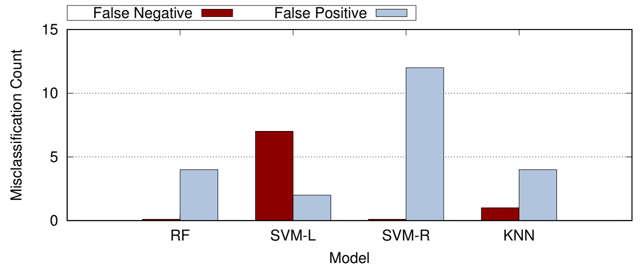

Fig. 5: Summary of false positive and false negative misclassifications for the best performing models.

5 Conclusion

In this work we demonstrate the effectiveness of native opcodes in detecting mobile ransomware. We characterize different machine learning models and show that our approach can detect ransomware with 99.8% accuracy. Furthermore, this work shows that merely 3% of the total instruction opcodes contribute to the detection of ransomware. Finally, we evaluate the robustness of our approach against six different ransomware families available in a state-of-the-art Android malware dataset.

References

[1] Statista, "The statistics portal," https://www.statista.com.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[3] Trend Micro, "Enterprise cybersecurity solutions," https://www.trendmicro.com.

[4] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2017, pp. 252–276.

[5] Android, "Platform architecture," https://developer.android.com/guide/platform.

[6] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[7] S. Sayad, *Real Time Data Mining*. Self-Help Publishers, 2011.

[8] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A large scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[9] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (Asia CCS)*. ACM, 2017, pp. 599–611.

[10] M. Ozsoy, C. Donovick, I. Gorelic, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *International Symposium on High Performance Computer Architecture (HPCA)*, February 2015, pp. 651–661.

[11] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (Asia CCS)*. ACM, 2018, pp. 457–468.

[12] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, and G.-J. Ahn, "Uncovering the face of android ransomware: Characterization and real-time detection," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1286–1300, May 2018.

[13] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *RAID*, ser. Lecture Notes in Computer Science, vol. 10453. Springer, 2017, pp. 192–214.