applied sciences

MDPI

Article

Is Malware Detection Needed for Android TV?

Gokhan Ozogur [1], Zeynep Gurkas-Aydin [2,*] and Mehmet Ali Erturk [3,4]

[1] R&D Lab, Arcelik A.S., Istanbul 34528, Turkey; gokhan.ozogur@arcelik.com
[2] Department of Computer Engineering, Istanbul University-Cerrahpasa, Istanbul 34320, Turkey
[3] School of Computing, Engineering & The Built Environment, Edinburgh Napier University, Edinburgh EH10 5DT, UK; m.erturk@napier.ac.uk or mehmetal.erturk@istanbul.edu.tr
[4] Department of Computer Engineering, Istanbul University, Istanbul 34134, Turkey
* Correspondence: zeynepg@iuc.edu.tr

Abstract: The smart TV ecosystem is rapidly expanding, allowing developers to publish their applications on TV markets to provide a wide array of services to TV users. However, this open nature can lead to significant cybersecurity concerns by bringing unauthorized access to home networks or leaking sensitive information. In this study, we focus on the security of Android TVs by developing a lightweight malware detection model specifically for these devices. We collected various Android TV applications from different markets and injected malicious payloads into benign applications to create Android TV malware, which is challenging to find on the market. We proposed a machine learning approach to detecting malware and evaluated our model. We compared the performance of nine classifiers and optimized the hyperparameters. Our findings indicated that the model performed well in rare malware cases on Android TVs. The most successful model classified malware with an F1-Score of 0.9789 in 0.1346 milliseconds per application.

Keywords: cybersecurity; Android TV; malware detection

Academic Editor: Douglas O'Shaughnessy

Received: 30 January 2025
Revised: 1 March 2025
Accepted: 3 March 2025
Published: 5 March 2025

Citation: Ozogur, G.; Gurkas-Aydin, Z.; Erturk, M.A. Is Malware Detection Needed for Android TV? *Appl. Sci.* **2025**, *15*, 2802. https://doi.org/10.3390/app15052802

1. Introduction

Android is open source software based on the Linux kernel and is the dominant operating system for mobile platforms, with a 70% market share in 2023 [1]. Although Android is primarily designed for mobile devices such as smartphones and tablets, it also supports other device types. Android TV runs on smart televisions and streaming media devices, Wear OS is supported on smartwatches, and Android Auto is designed for automotive head units. An increasing number of streaming services, high-speed internet, developments in display technology, integrated voice services, and gaming features have influenced the smart Android TV market [2]. In 2024, 150 million smart TVs and 90 million streaming media devices are expected to be shipped worldwide [3].

In the research in the literature, mobile applications running on Android have been examined due to the popularity of smartphones [4–19]. We have listed Android malware detection studies focused on smartphones for benchmarking, together with the extracted features, classification methods, and performance results, in Table 1. Malicious applications cause harm on smartphones by using API calls, such as accessing the phone book, making phone calls, and sending SMS messages. Studies that detect malicious applications using static analyses have examined the usage of these APIs and their permissions to determine the intention of applications. Because of the fact that some APIs and permissions are specific to smartphones and not used in smart televisions, identifying the intention of TV applications only by analyzing smartphone features might be wrong. In this respect, there

is a need to investigate Android TV applications to determine features that will help to detect malicious applications.

Table 1. Performance results of studies on Android malware detection.

Study	Features	Methods	Results
[4]	Classes, Methods, API Calls	Artificial Neural Network	F1-Score: 0.9922
[5]	Manifest, Binary Source Code	N-Gram, Mutual Information, XGBoost	F1-Score: 0.9906
[6]	Manifest, Binary Source Code	Term Frequency-Inverse Document Frequency, N-gram, XGBoost	F1-Score: 0.9905
[7]	API Calls	Markov Chain, Random Forest, K-Nearest Neighbor	F1-Score: 0.99
[8]	Permissions, APIs, Intents, Hardware Components	Genetic Evolution, Deep Learning	F1-Score: 0.9853
[9]	Permissions, APIs, Intents, Hardware Components	Convolutional Neural Network, Long Short-Term Memory	Accuracy: 0.9853
[10]	Permissions, Libraries, API Calls, System Calls, Battery Temperature, CPU Usage, Memory Usage, Source Code	Pruning Rule-Based Classification Tree, Ripple Down Rule Learner, Support Vector Machines, Multilayer Perceptron	Accuracy: 0.9827
[11]	Permissions, APIs, Intents, Hardware Components	Deep Neural Network	Accuracy: 0.98
[12]	Opcodes, System Calls	Convolutional Neural Network, Bidirectional Long Short-Term Memory, Attention Mechanism	F1-Score: 0.96
[13]	Permissions	Graph Comparison Algorithm	Accuracy: 0.9544
[14]	Opcodes	Manhattan Distance Comparison	Accuracy: 0.936
[15]	Permissions, Classes, Opcode Calls	Convolutional Neural Network, Random Forest	Accuracy: 0.9255
[16]	RGB Images of Binary APKs	Attention Mechanism	Accuracy: 0.9064
[17]	System Calls	Convolutional Neural Network, Vector Representations	Accuracy: 0.8
[18]	Interprocedural Control Flow Graphs	Support Vector Machines	Accuracy: 0.74

A number of questions regarding Android TV security still remain to be answered. The research questions investigated in our study are as follows:

1. What is the malware ratio for Android TV applications in markets and public datasets?
2. Can a model trained on an Android smartphone malware dataset detect Android TV malware?
3. How do classifiers perform in detecting Android TV malware?

In this study, we worked on feature extraction, malware detection, and malicious payload injection for Android applications. Specifically, we were interested in malware detection for Android TV devices. Our motivation was to analyze malware detection models for Android applications running not only on smartphones but also on other devices like TVs. Our main contribution in this study is as follows.

1. We collected 1107 Android TV applications from the Androzoo [20] dataset and used web scraping to collect 370 Android TV applications from the APKMirror (https://www.apkmirror.com, accessed on 15 January 2025) application market. Then, we labeled the Android TV applications using antivirus scanners on the VirusTotal (https://www.virustotal.com, accessed on 15 January 2025) website.

2. We injected a malicious payload into benign applications to create Android TV malware since there are only a few examples of malware on the market for TV devices.
3. We extracted the 500 most frequently used n-grams from each resource (AndroidManifest.xml) and binary source (classes.dex) file in the Android applications using the TF-IDF method separately. We publicly shared these features and class (benign or malicious) labels of the Android TV applications.
4. We implemented classification models for malware detection using the extracted XML and DEX features and compared the performance of the models.

2. Related Work

A very extensive body of literature on the topic of Android security exists. However, the security of Android devices other than smartphones has received very limited attention in the literature. Girish et al. [21] introduced the ImposTer framework for simulating a smart home environment and monitoring the behaviors of Internet of Things (IoT) devices. ImposTer aims to identify privacy breaches and security risks by replicating the conditions of a smart home and examining diverse IoT devices within the network. The results of their analysis reveal that many devices pose privacy risks as a result of using network scanning and discovery methods. This framework presents an opportunity to refine and expand the development of more accurate and comprehensive black-box testing mechanisms specifically tailored to IoT devices. However, some limitations of this study exist, such as challenges in both static and dynamic analyses, shortcomings in the network monitoring capabilities, the necessity for a deeper understanding of the device discovery mechanisms, and the assessment of the effectiveness of various countermeasures.

Majors et al. [22] assessed the vulnerabilities in WiFi remote control protocols to address the security risks associated with the increasing prevalence of smart TV devices. Their study analyzed four major smart TV platforms, namely Android TV, Amazon Fire OS, Roku OS, and webOS, and identified significant security breaches in these platforms. This research demonstrated the impact of these security weaknesses by developing a malware named Spook, specifically targeting Android TV devices. The findings were communicated to relevant vendors, prompting partial patches from Google. Nevertheless, this research argues that these patches inadequately fulfill the envisioned security requirements for WiFi remote control protocols, emphasizing the necessity of a more resilient solution. They have proposed an alternative defense mechanism by designing a WiFi remote control protocol within the Android ecosystem using ARM TrustZone. This proposed solution aims to ensure security while preserving the flexibility of virtual remote controls. This study explores the root causes of the security challenges in smart TV environments, discusses alternative defense strategies, and makes a significant contribution to the field.

Tileria and Blasco [23] delve into a comprehensive analysis of the Android TV ecosystem for potential privacy threats based on a dataset of over 4500 TV applications. They reveal that static identifiers for tracking persist despite Google's new policies, as well as potentially harmful behaviors often linked to tracking and advertising services. The static analysis, network traffic collection, and manual analysis results of their study expose privacy-invasive practices, obsolete communication APIs, and insecure coding practices among developers. However, there are some limitations in their study, such as certificate-based identification challenges and the omission of dynamic elements from the static analysis. They conclude that improved development practices, guidelines for porting mobile applications to the Android TV platform, and improved user interfaces to provide better control over app permissions are needed for a more secure and privacy-aware Android TV ecosystem.

Liu et al. [24] chose 3163 Android TV applications from the Google Play Store and shared the SHA-256 checksums of these applications publicly. They extracted the unique package names of the applications from the Androzoo [20] dataset and filtered the TV applications by searching for the package names and capabilities of the applications on the Google Play Store. They analyzed the VirusTotal scans, requested permissions, security flaws, and privacy leaks of the applications. They observed 299 unique TV-specific permissions which contained the "tv" keyword. They found out that 119 applications declared the same permission more than once, an indicator of application repackaging. They exposed that 2997 Android TV applications contained critical security bugs, and 1673 Android TV applications were found to have privacy leaks. They labeled 89 Android TV applications as malicious on the basis of at least one antivirus scanner on VirusTotal considering it to be malicious.

Riadi et al. [25] tested a remote access Trojan attack on Android smartphones using the Meterpreter reverse TCP payload. They obtained critical information such as the system information, contacts, call logs, and messages and access to the system directory of the victim's smartphone via the backdoor of the injected Android application. Their Trojan was successfully installed on a test device running Android 11, but Google's Play Protect detected it as dangerous on Android 12. They recommend that users update their devices' operating systems regularly.

None of the existing approaches have extracted features from Android TV applications for malware detection. In our study, we extracted features from the Android Manifest and DEX files at the bytecode level without applying reverse engineering to the source code. Using the extracted features, we trained classifiers to detect malware and compared the performance results on the collected Android TV applications. Our findings show that a dataset containing benign and malicious TV applications is needed to increase the performance of malware detectors. Therefore, we publicly shared our dataset containing 1000 features from 1305 Android TV applications.

3. Methods

In this study, we collected 1107 Android TV applications from the Androzoo [20] dataset and used web scraping to collect 370 Android TV applications from the APKMirror application market. We also labeled the Android TV applications using antivirus scanners on the VirusTotal website. Because of the fact that there are only a few examples of malware on the market for TV devices, we injected a malicious payload into benign applications to create Android TV malware. Then, we extracted the 1000 most frequently used n-grams from the Android applications using the TF-IDF method and publicly shared these features with labels in spreadsheet format. We implemented classification models for malware detection using the extracted XML and DEX features and compared the performance results of the models. All of the methodologies used in our study are illustrated in Figure 1.

3.1. The Collection of Android TV Application Packages

Previous research has typically only investigated smartphone applications due to their popularity. The Androzoo [20] dataset provides millions of Android applications collected from several sources, including the official Google Play application market. However, it is not possible to identify these TV applications without a further analysis since they are not labeled in the Androzoo [20] dataset. To the best of our knowledge, there is no publicly available dataset in the literature for Android TV applications, except the study from Liu et al. [24], which contains the SHA-256 checksums of applications. We conducted a search for Android TV applications using the SHA-256 information provided in the

Androzoo [20] dataset, but some of them were not found in the dataset. We were able to collect 1107 Android TV application packages from the list shared by Liu et al. [24].

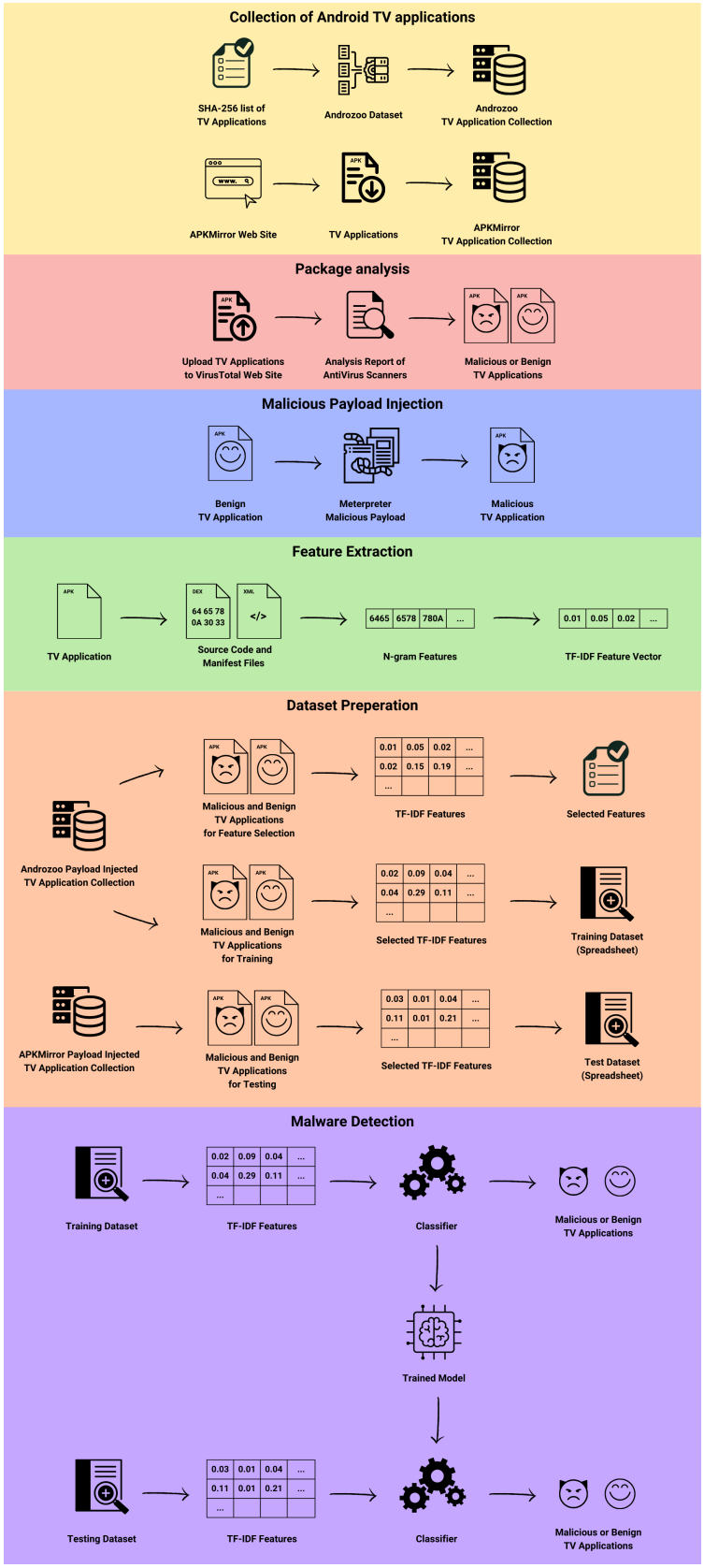

Figure 1. Methodologies in our study.

We also decided to create an Android TV application dataset from scratch and implemented a web scraper to collect applications from an alternative application market called APKMirror, which listed applications for various devices (e.g., phones, watches, TVs, Auto, etc.). We scraped the links for 600 TV application packages listed on the APKMirror website and downloaded all of them. A collection of 370 TV applications was created by eliminating the application packages for different versions of the same application and packages that were not in APK format among the downloaded packages.

The average package size of the 1107 TV applications in the Androzoo dataset is 26.4 MB, while the minimum and maximum sizes are 23.5 kB and 145.2 MB, respectively. For the 370 TV applications in the APKMirror dataset, the average package size is 28.2 MB, while the minimum and maximum sizes are 3.2 kB and 191.1 MB, respectively. The minimum, maximum, and average file sizes of the TV application packages in the Androzoo and APKMirror datasets are presented in Table 2.

Table 2. File size statistics of the TV application packages in the datasets.

Dataset Name	Minimum File Size	Maximum File Size	Average File Size
Androzoo	23.5 kB	145.2 MB	26.4 MB
APKMirror	3.2 kB	191.1 MB	28.2 MB

3.2. Package Analysis for Labeling

In order to label the application packages in the collection as malicious or benign, we used the security analysis reports of the antivirus vendors on the VirusTotal website, which contained more than 70 scanners for file and URL analyses. We developed a Python (v.3.12.3) script using VirusTotal's API to upload the application packages as a file and to obtain their analysis results. For each application package, our script counted the number of antivirus scanners that categorized the given file as malicious. In order to eliminate false detections, we decided to label an application package as malware if it was categorized as malicious by at least 3 antivirus scanners. Otherwise, we labeled the application package as benign.

Figure 2 shows the analysis results from the VirusTotal site for a benign TV application. According to the analysis report obtained for the benign application, the application package is not classified as harmful by any antivirus scanner. Figure 3 shows the analysis results obtained from the VirusTotal site for a malicious TV application. The malicious application package is classified as malicious by 16 antivirus scanners.

We observed that out of 1107 Android TV applications collected from the Androzoo [20] dataset, only 3 of them were labeled as malicious based on the analysis results. These 3 application packages were categorized as malicious by more than 10 antivirus scanners on the VirusTotal website. According to the analysis results for the remaining applications, we observed that 34 of them were categorized as malicious by at most 2 antivirus scanners, and the rest of them were not categorized as malicious by any antivirus scanner. We concluded that a threshold value of 3 was acceptable, as there was a significant range in the number of antivirus scanners for malicious applications. As a result, if an application was categorized as malicious by fewer than 3 antivirus scanners, we assumed that it was a false detection by these scanners.

Similarly, we observed that all of the applications collected from APKMirror were labeled as benign based on the analysis results. When we investigated the analysis reports, we observed that only 8 application packages were categorized as malicious by at most 1 antivirus scanner, and we assumed these to be false detections. The other application packages were not categorized as malicious by any antivirus scanner. As a result, we could not obtain any malware for Android TVs from the APKMirror market.

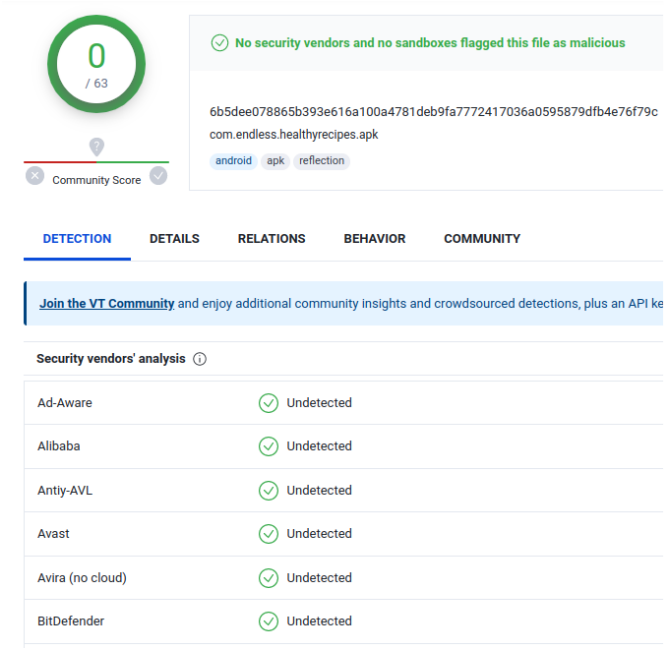

Figure 2. VirusTotal scan results for a benign TV application.

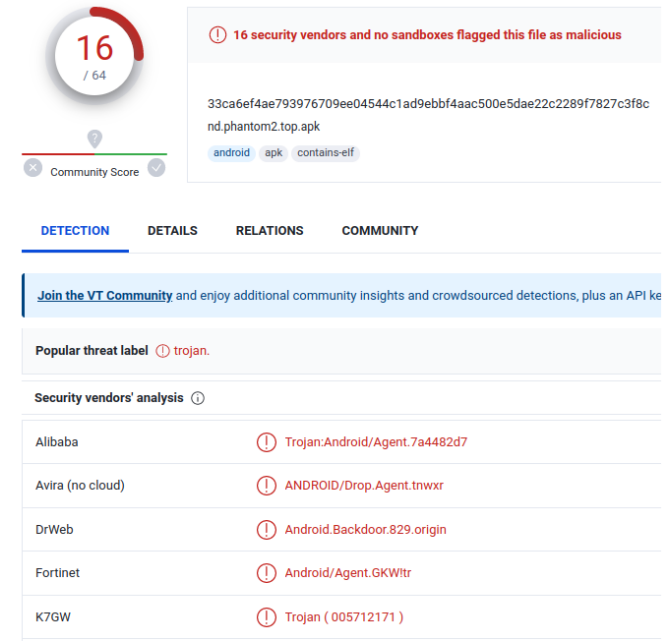

Figure 3. VirusTotal scan results for a malicious TV application.

3.3. Malicious Payload Injection

We observed that all of the Android TV applications collected from the APKMirror market were labeled as benign based on the analysis results. In order to obtain TV malware, we injected a malicious payload into some benign TV applications and converted them into malicious TV applications. On the other hand, we kept some of the TV applications as benign.

The Metasploit Framework [26], an open source penetration testing and development tool that identifies security vulnerabilities for many operating systems and platforms, was used in our study to obtain malicious applications. The malicious payload of Meterpreter in the Metasploit Framework is a command-line program that runs in the memory of the target device. It is possible to execute the desired commands on the target device using the Meterpreter payload. A malicious person can capture a screenshot of the TV

screen, take a picture via the webcam, record the audio, and run an application using the Meterpreter payload.

Most of the Android TV devices on the market are not rooted out of the box, but attackers can exploit their vulnerabilities to gain root privileges. Meterpreter can be used with vulnerabilities like Dirty Cow, Zygote, or Stagefright in older Android versions to escalate privileges [27]. If attackers gain root access, they can modify system applications to include the Meterpreter payload and enable the malware to launch when a system application is launched. Attackers can set up scheduled tasks to invoke malicious code at specific intervals to ensure persistence and evasion. To make detection even more difficult, Meterpreter establishes reverse shell connections that allow the malicious payload to hide behind normal network traffic.

In order to inject a malicious payload into a TV application, first, we generated malware including the Meterpreter payload using the Metasploit Framework [26]. It is possible to use 1 of 3 Meterpreter payload types, named reverse_http, reverse_https, and reverse_tcp, which use different connection protocols for the Android operating system. We also defined the local host and port numbers of our machine at this stage. Secondly, we decompiled a benign application to obtain its Smali code. Similarly, we decompiled the malware and obtained the Smali code for the payload. Thirdly, we copied the payload files into the benign application. Then, we injected the hook into the benign application by invoking the start command of the payload at the start of the benign application in the Smali file. Finally, we added the permissions needed by the malware into the Manifest file of the benign application and recompiled the application. As a result, we obtained a payload-injected malicious TV application. We repeated these steps to obtain more types of TV malware in our dataset. Since the Metasploit Framework [26] uses random characters in its functions and class names, the generated payloads are not identical.

3.4. Feature Extraction

An Android application package is a compressed file containing the components of the application, including "AndroidManifest.xml" and "classes.dex" files. The "AndroidManifest.xml" file contains information about the name, version, access rights, referenced library files, and the used features and components of the application in XML format. The "classes.dex" file is the source code of the application compiled for the Dalvik virtual machine. Even though exposing the source code of the application is possible using reverse engineering, various obfuscation techniques are applied in some applications during compilation to prevent disclosure of the source code. In order to be resilient against obfuscation techniques, we extracted features from the DEX files at the bytecode level instead of applying reverse engineering to the source code in this study. On the other hand, we decoded the binary XML files and extracted the features from human-readable XML files.

In order to extract the features, DEX and XML files are read at the byte level and divided into sequences of n bytes called n-grams. The n-grams are obtained by sliding an n-byte-long window by 1 byte from the beginning to the end of the file. In our study, we handled the n-grams in hexadecimal format without using a prefix (0x) or a space between bytes. As an example, every DEX file starts with the "d", "e", and "x" characters, which are 0 x 64, 0 x 65, and 0 x 78 in hexadecimal format, respectively. Therefore, the first 2-byte-long (2-gram) feature of the DEX files is "6465", and the second one is "6578". We extracted 2-gram features from the DEX files and 3-gram freatures from the XML files because of the fact that the file size of the source files is bigger than that of Android Manifest files. The number of n-grams is correlated with the number of bytes (n) of a feature and is in the range of $[0, 2^{8*n}]$. Therefore, calculation using more features requires more space in terms of memory and takes a longer time.

It is not efficient to use all of the n-grams as features, especially for long n-grams. In order to select the features, we adapted the Term Frequency—Inverse Document Frequency (TF-IDF) method, which is widely used to extract features from text documents. Similar to representing documents using the extracted features of the words in them, we represented binary files using the extracted features of the n-grams in them. We counted the occurrence of each n-gram for each file and calculated the frequencies of the n-grams for each file. Then, we calculated the weights (importance) of the n-grams using the TF-IDF method such that the n-grams received weights according to their occurrence in a file, but also n-grams that existed in the majority of the files received less weight since they were less informative.

The TF-IDF value ($tfidf$) of an n-gram (t) in a file (d) is calculated by multiplying the term frequency (tf) and inverse document frequency (idf) values, as presented in Equation (1). Term frequency is calculated by dividing the number of occurrences (f) of an n-gram in a file by the sum of the number of occurrences of the other n-grams in the same file. Inverse document frequency is calculated by adding the constant 1 to a logarithm of the ratio of the total number of files (n) to the number of files containing the n-gram (df). In order to prevent zero division, a constant 1 value is added to the numerator and the denominator of the logarithm, which is a widely accepted practice. The TF-IDF value presented in Equation (1) is the raw weight of an n-gram in a file. In order to represent a file in a vector form, the TF-IDF values of all of the n-grams in the file need to be calculated. The obtained vector is normalized by the Euclidean (L2) norm, as presented in Equation (2).

$$\begin{aligned} tfidf(t,d) &= tf(t,d) \times idf(t) \\ tf(t,d) &= \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \\ idf(t) &= \log \frac{1+n}{1+df(t)} + 1 \end{aligned} \tag{1}$$

$$v_{norm} = \frac{v}{\sqrt{v_1^2 + v_2^2 + ... + v_n^2}} \tag{2}$$

It was possible to obtain 65,536 (2^{16}) unique features from the DEX files since we extracted 2-gram features. On the other hand, we were able to extract 16,777,216 (2^{24}) features from the XML files using 3-gram features. We selected 500 features from the DEX files and 500 features from the XML files separately. The best features were selected according to the importance obtained by calculating the TF-IDF weight of the feature. As a result, we represented each application with a vector of 1000 features.

3.5. Dataset Description

We collected 1107 Android TV applications from the Androzoo [20] dataset, and 3 of them were labeled as malicious based on VirusTotal's antivirus scanners. We excluded these 3 applications and injected a malicious payload into some of the benign applicitions. We obtained errors on payload injection for some of the applications and could not convert them. As a result, we obtained 471 Android TV malwares and 567 benign applications. We named this collection the Androzoo Payload-Injected TV dataset and used it in training the malware detection models.

We collected 370 Android TV applications from the APKMirror website, and all of them were labeled as benign. A total of 188 TV applications from the APKMirror collection were converted into malware, and the remaining 182 TV applications were kept as benign. We named this collection the APKMirror Payload-Injected TV dataset and used it in testing the malware detection models.

The number of malicious and benign applications in the datasets is shown in Table 3 with the malware ratio. The malware ratio of the Androzoo Payload-Injected TV dataset was calculated as 45%, including 154 malwares with reverse_http, 154 malwares with reverse_https, and 163 malware with reverse_tcp payloads. The malware ratio of the APKMirror Payload-Injected TV dataset was calculated as 51%, including 188 malicious TV applications consisting of 67, 64, and 57 malwares with reverse_http, reverse_https, and reverse_tcp payloads, respectively.

Table 3. Number of malicious and benign applications in the datasets.

Dataset	Malicious			Benign	Malware Ratio
	reverse_http	reverse_https	reverse_tcp		
Androzoo Payload-Injected	154	154	163	567	45%
APKMirror Payload-Injected	67	64	57	182	51%

In order to select the best features using the TF-IDF method, we split 10% of the applications randomly in the Androzoo Payload-Injected dataset. We used these 103 applications only for feature selection, and they were excluded in the model training and testing datasets. We used the remaining 935 applications for the training of the classification models and saved the extracted features as the model training dataset. To test the model, we used 370 applications in the APKMirror Payload-Injected dataset. The features extracted from the test applications were saved as the model testing dataset. We exported the extracted features as training and testing spreadsheet files. In this way, we shared the features extracted from Android TV applications for use by future studies on malware detection in the literature.

Both the training and testing spreadsheets include 1000 features, which are normalized TF-IDF values, and class labels, where benign applications are represented with 0 and malicious applications with 1. The features extracted from the decoded "AndroidManifest.xml" files are placed in the first 500 columns. The features extracted from binary source (classes.dex) files are placed in the next 500 columns. The last column shows the class labels of the application. The features extracted from the applications are placed in rows. Also, the header row in spreadsheet files includes feature names, which are shown in string format for XML files and in hexadecimal format for DEX files.

3.6. Malware Detection

In order to detect malicious Android TV applications, we utilized 9 models using discriminant analysis (discr), ensemble classification (ensemble), kernel (kernel), K-Nearest Neighbor (knn), linear (linear), naive Bayes (nb), Neural Network (net), Support Vector Machine (svm), and binary decision tree (tree) classification methods. The hyperparameters of the models were optimized in 30 iterations for each classifier using the training sets, and the best models with optimized hyperparameters were selected for testing.

The discriminant analysis classifier uses linear or quadratic class boundaries and is computationally efficient in simple cases. On the other hand, it is sensitive to outliers and performs poorly with a non-Gaussian data distribution. We optimized the Delta and Gamma hyperparameters, which adjusted the threshold value for feature elimination and the amount of regularization, respectively.

Ensemble classification combines the predictions of multiple weak learners to obtain a strong learner. It reduces overfitting compared to individual models, and it is effective on high-dimensional data. We used aggregation methods such as Bootstrap Aggregation

(Bag) [28], Gentle Adaptive Boosting (GentleBoost) [29], Adaptive Logistic Regression (LogitBoost) [29], Adaptive Boosting (AdaBoostM1) [30], and Random Undersampling Boosting (RUSBoost) [31]. We selected the hyperparameters for the aggregation method, the number of ensemble learning cycles (NumLearningCycles), the learning rate (LearnRate), and the minimum number of leaf node observations (MinLeafSize).

Kernel classification can model complex decision boundaries using the kernel trick, which is useful when the data are not linearly separable. However, it suffers from overfitting if it is not properly regularized. We optimized the KernelScale, Lambda, NumExpansionDimensions, and Standardize hyperparameters which were used to map features into a high-dimensional space, adjust the regularization, tune the number of dimensions of the expanded space, and enable scaling of the features, respectively.

The K-Nearest Neighbor model works by finding the k closest neighbors to a given data point. It does not require assumptions about the data. However, it is memory-intensive since it needs all of the training data to perform each prediction. We optimized the number of nearest neighbors (NumNeighbors) and the distance metric (Distance) and enabled scaling of the features (Standardize) as hyperparameters.

The linear classification model assumes the data are linearly separable and works well with high-dimensional data. We optimized the Learner and Lambda hyperparameters, which decided the type of classification algorithm and the regularization strength of the model.

Naive Bayes classification assumes that the features are independent. It can perform poorly when the features are highly correlated. On the other hand, it is fast and efficient on large datasets. We optimized the distribution type used to model the features (DistributionNames) and the kernel smoothing window width (Width) and enabled scaling of the features (Standardize) as hyperparameters.

Neural Networks can learn complex patterns and perform well on image, text, and sequential data. However, they are black-box models and difficult to interpret. In our case, we fed the normalized TF-IDF values forward to the neurons in the input layer and discerned the probability that the application was malicious or benign from the output layer of the model. We optimized the type of activation function of the connected layers (Activations), the sizes of the connected layers (LayerSizes), and the regularization strength (Lambda) and enabled scaling of the features (Standardize) as hyperparameters.

Support Vector Machines are well suited to binary classification problems. They work well for both linear and nonlinear problems using kernel tricks. In our case, we tried to find a hyperplane that separated malicious and benign applications and maximized the margin between the hyperplane and the features of the applications in each class. We optimized the maximum penalty applied to margin-violating observations to prevent overfitting (BoxConstraint), the scaling value used to map features into a high-dimensional space (KernelScale), and the enabling flag for feature scaling (Standardize) as the hyperparameters.

Binary decision trees handle both categorical and numerical data well. However, they are sensitive to small changes in the data and are not as accurate as ensemble methods. We optimized the MinLeafSize hyperparameter to decide the minimum number of leaf node observations.

4. Performance Results

We evaluated the malware detection performance of the models using popular performance metrics, which are introduced in Section 4.1.

4.1. Performance Metrics

The outcomes of the malware detection models are predictions about the classification of the given application as malicious or benign. The accuracy of the predictions is deter-

mined using the confusion matrix shown in Table 4. In the confusion matrix, the Positive (P) value represents the number of malware, while the Negative (N) value represents the number of benign applications. Applications predicted by the model to be malicious are considered to be Predicted Malware (PP), and those predicted to be benign are considered to be Predicted Benign (PN). The True Negative (TN) value shows the number of applications that are actually benign and classified as benign by the model, and the True Positive (TP) value shows the number of applications that are actually malicious and classified as malicious. The False Negative (FN) value shows applications that are actually malicious but classified as benign, and the False Positive (FP) value shows applications that are actually benign but classified as malicious by the model.

Table 4. Confusion matrix.

	Predicted Benign (PN)	**Predicted Malware (PP)**
True Benign (N)	True Negative (TN)	False Positive (FP)
True Malware (P)	False Negative (FN)	True Positive (TP)

The performance of the malware detection models is evaluated using the F1-Score and Matthews Correlation Coefficient (MCC) performance metrics, as presented in Equation (3) and Equation (4), respectively.

$$\begin{aligned} Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F1\text{-}Score &= 2 \times \frac{Precision \times Recall}{Precision + Recall} \end{aligned} \tag{3}$$

$$\begin{aligned} Negative(N) &= TN + FP \\ Positive(P) &= FN + TP \\ PredictedNegative(PN) &= TN + FN \\ PredictedPositive(PP) &= FP + TP \\ MCC &= \frac{TP \times TN - FP \times FN}{\sqrt{N \times P \times PN \times PP}} \end{aligned} \tag{4}$$

We would like to demonstrate the calculation of the performance metrics with an example. For a dataset containing 182 benign (N) and 188 malicious (P) applications, a model detected 178 benign applications (PN) and 192 malware (PP). When we compared the predictions with the samples, we observed that 186 malicious and 176 benign applications were detected successfully, which represented True Positive (TP) and True Negative (TN) results, respectively. It misclassified two malware as benign applications, which were False Negative (FN) results. It also misclassified six benign applications as malicious, which were False Positive (FP) results. In this case, we calculated the Precision as $\frac{186}{186+6} = 0.9688$ and the Recall as $\frac{186}{186+2} = 0.9894$ according to Equation (3). Finally, we obtained the F1-Score as $2 \times \frac{0.9688 \times 0.9894}{0.9688+0.9894} = 0.9789$. According to Equation (4), we calculated the MCC metric as $\frac{186 \times 176 - 6 \times 2}{\sqrt{182 \times 188 \times 178 \times 192}} = 0.9570$.

4.2. The Performance of the Model Trained Using Mobile Applications on TV Applications

A model trained on the Drebin&Androzoo dataset consisting of mobile applications was tested on mobile applications and showed a success of 0.9905 and 0.9809 according to the F1-Score and MCC metrics, respectively [6]. When we tested the model trained with the mobile applications on the Androzoo Payload-Injected TV dataset, only 20 of

471 malicious applications could be detected, and 565 of 567 benign applications were classified correctly. The success of the model trained with mobile applications on TV applications was calculated as 0.0811 and 0.1346 according to the F1-Score and MCC metrics, respectively. These results show that the model trained with mobile applications from the Drebin&Androzoo dataset was insufficient for detecting malicious applications in the Androzoo Payload-Injected TV dataset. In order to increase the success of the model, it needs to be trained on a collection including malicious Android TV applications.

4.3. The Performance of the Model Trained with TV Applications on TV Applications

We used 103 TV applications from the Androzoo Payload-Injected TV dataset for the feature selection. Then, we trained the classification models on the remaining 935 TV applications in the Androzoo Payload-Injected TV dataset. We tested the performance of the trained models on 370 TV applications in the APKMirror Payload-Injected TV dataset. We trained and tested the models on MATLAB Online 2024b, which was run on a machine with an Intel Xeon Platinum 8375C CPU @ 2.90GHz (Santa Clara, CA, USA) without parallel computing. We measured the training and testing times for each model, which are presented in Figure 4 and Figure 5, respectively.

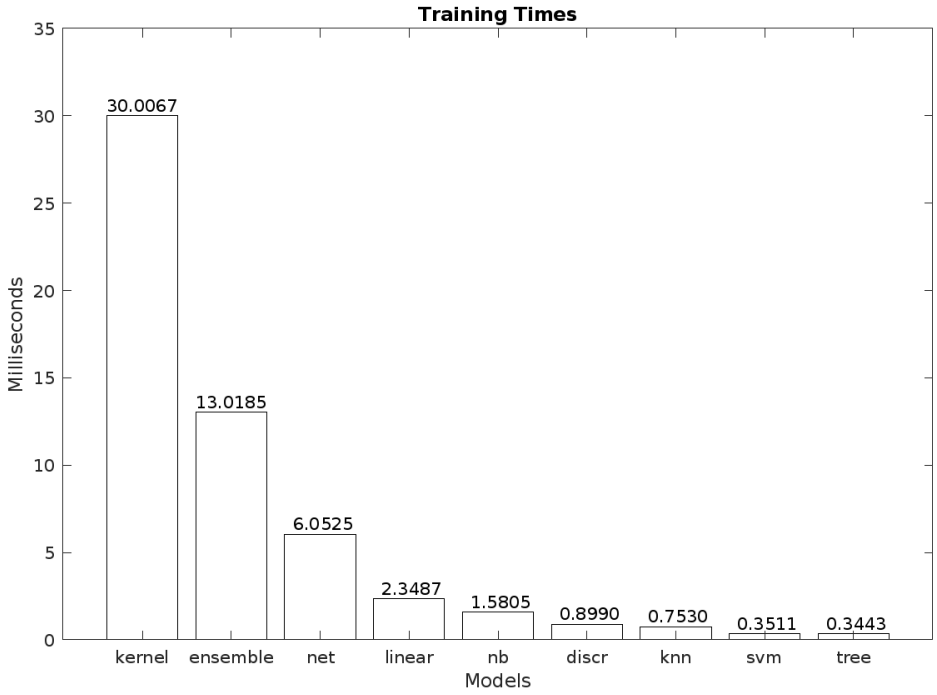

Figure 4. Average training time of models per application.

According to our observations, four models (tree, svm, knn, and discr) performed training in less than 1 millisecond on average per application. The svm and tree models showed a close performance, at 0.3511 and 0.3443 milliseconds, respectively. The worst training time was achieved by the kernel classification model, with an average of 30 milliseconds per application.

The performances of the four models (tree, net, linear, and discr) were distinguishing with respect to the other models in testing. We observed that the tree and net models performed similarly, with 0.1149 and 0.1152 millisecond testing times per application on average. The performance of the knn model was poor in testing, which completed testing in 0.9002 milliseconds on average per application.

We calculated the confusion matrix for each model on the test set. The confusion charts for the models are presented in Figure 6, where benign and malicious applications

are labeled with the 0 and 1 class labels, respectively. The knn model predicted 31 malicious and 15 benign applications incorrectly, which was not good. On the other hand, the discr model misclassified only two malicious and six benign applications. Similarly, the ensemble model missed five malicious and four benign applications.

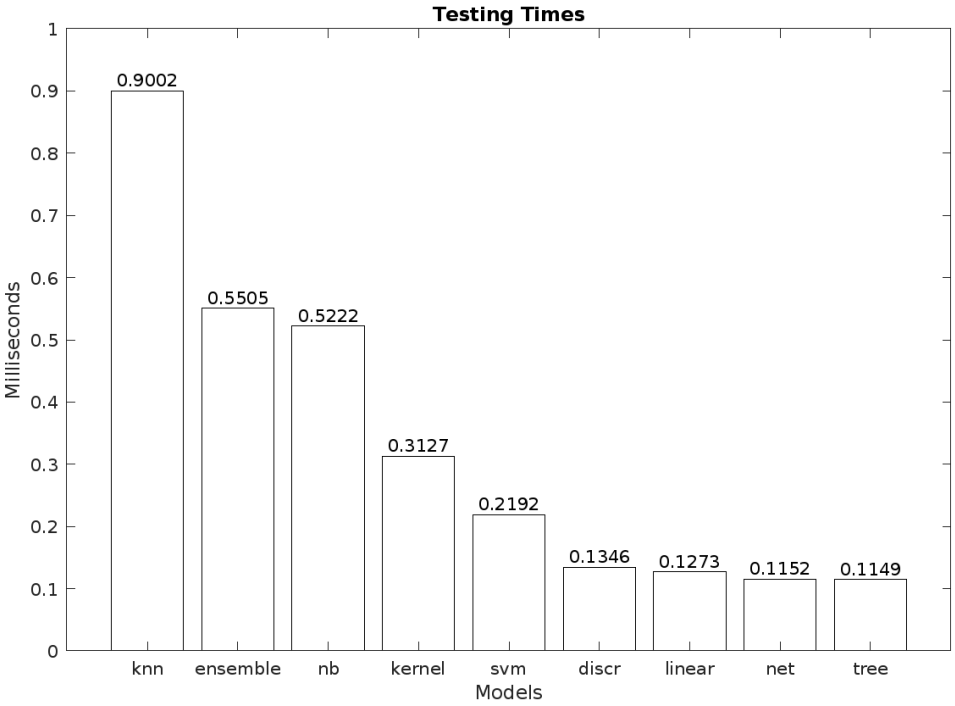

Figure 5. Average testing time of the models per application.

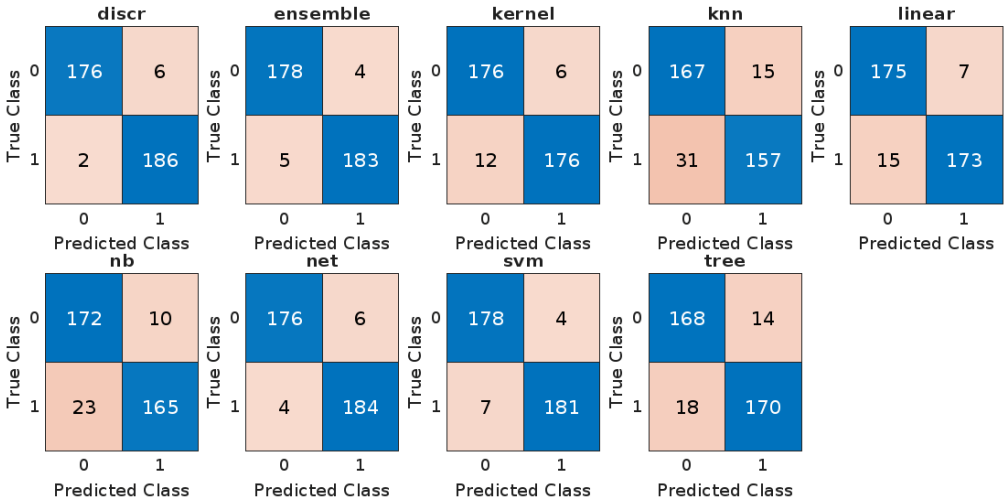

Figure 6. Confusion charts for classifiers.

We present the Receiver Operating Characteristic (ROC) curve for each model in Figure 7, which visualizes the performance across all of the thresholds using the True Positive Rate (TPR) and False Positive Rate (FPR) metrics. In our case, the area under the ROC curve (AUC) represents the probability that the model will correctly classify a randomly chosen malicious application over a randomly chosen benign application. With respect to the AUC metric, the discr and ebsemble models performed well, with 0.9956 and 0.9945 scores, respectively. The worst performance was obtained by the tree classifier, with an AUC score of 0.8847.

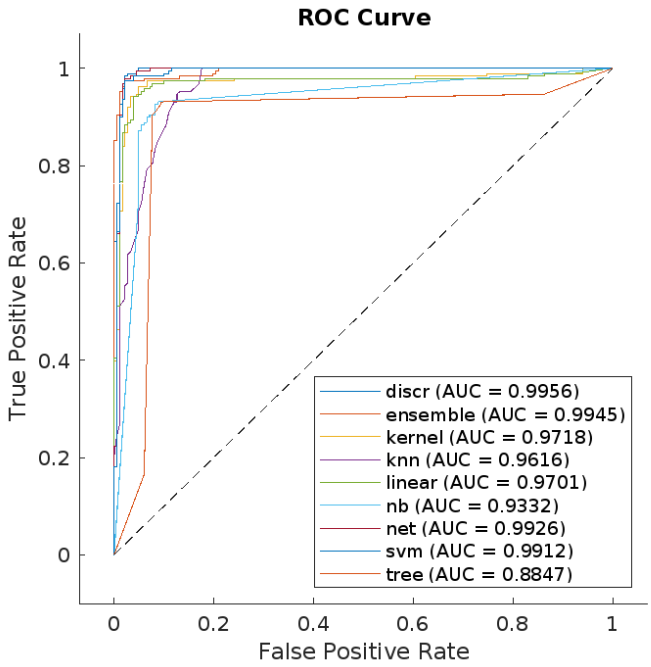

Figure 7. ROC curves of the models.

According to the F1-Score and MCC metrics, the best result was achieved by the discr model, with 0.9789 and 0.9570 scores, respectively. A similar performance was obtained by the ensemble, net, and svm models, with 0.9760, 0.9735, and 0.9705 scores with respect to the F1-Score metrics, respectively. The performance of the knn model was calculated as a 0.8722 F1-Score and a 0.7545 MCC score, which was the worst performance on the test set with respect to that of the other models. The F1-Score and MCC results for the models on the testing set are presented in Figures 8 and 9.

The discriminant analysis classifier achieved the best results on the test set with an F1-Score of 0.9789 and an MCC score of 0.9570. It performed the fourth best in terms of the training time, at 0.8990 milliseconds per application. In testing, it classified malware in 0.1346 milliseconds on average, which was only 0.0197 milliseconds slower than the fastest model.

The ensemble classifier achieved the second best result on the test set, with an F1-Score of 0.9760 and an MCC score of 0.9514. It classified malware in an average of 0.5505 milliseconds, the second worst time among the models.

The Neural Network classifier was the third most successful model on the test set, with an F1-Score of 0.9735 and an MCC score of 0.9460. Despite its relatively slow training time, it classified malware in an average of 0.1152 milliseconds, which was the second best time among the models in testing.

The Support Vector Machine classifier was the fourth most successful model on the test set, with an F1-Score of 0.9705 and an MCC score of 0.9407. Even though it was the second fastest model in training, it classified malware in an average of 0.2192 milliseconds, which was the fifth best time among the models on the test set.

The kernel classification model was the fifth most successful model on the test set with an F1-Score of 0.9514 and an MCC score of 0.9032. It achieved the worst training time at 30.0067 milliseconds per application. However, it classified malware in 0.3127 milliseconds, which was the sixth best time among the models.

The linear classification model was the sixth most successful model on the test set with an F1-Score of 0.9402 and an MCC score of 0.8820. It classified malware in 0.1273 milliseconds, the third best time on the test set.

The binary decision tree classifier was the third worst model on the test set, with an F1-Score of 0.9140 and an MCC score of 0.8872. On the other hand, it was the fastest model in both training and testing. On average, it spent 0.3443 milliseconds per application on the training set. In testing, it classified malware in 0.1149 milliseconds.

The naive Bayes classifier was the second worst model on the test set, with an F1-Score of 0.9091 and an MCC score of 0.8238. It classified malware in 0.5222 milliseconds, the third worst time on the test set.

The K-Nearest Neighbor model was the worst model on the test set, with an F1-Score of 0.8722 and an MCC score of 0.7545. Even though it was the third fastest model in training, it classified malware in an average of 0.9002 milliseconds, the worst time among the models in testing.

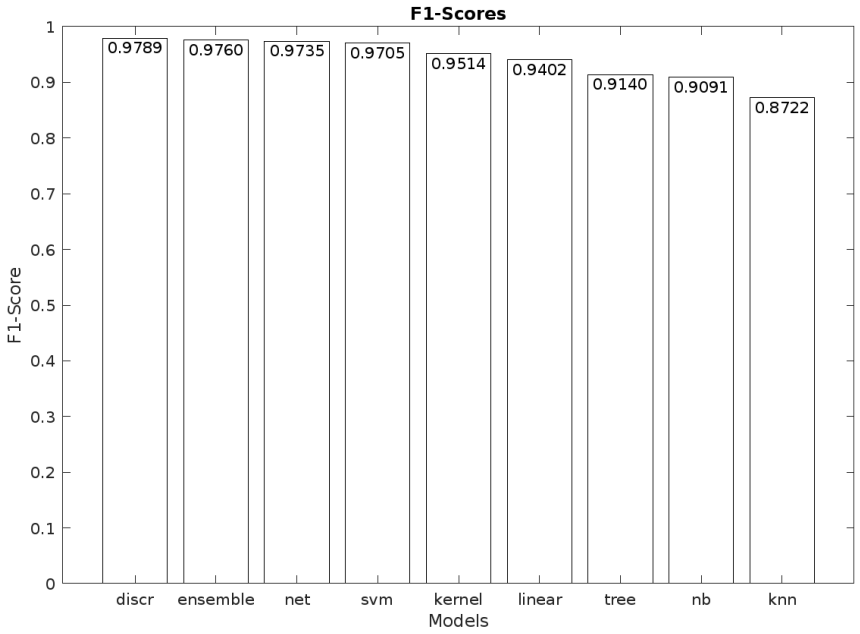

Figure 8. Results of the models in testing with respect to the F1-Score.

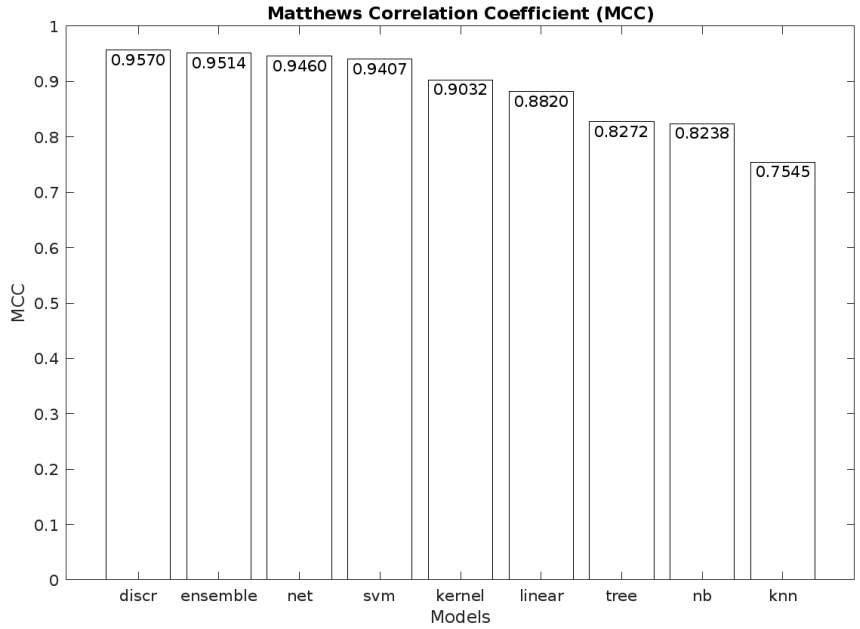

Figure 9. Results of the models in testing with respect to the MCC.

5. Limitations

In our study, we extracted the features from Android TV applications using a static analysis that involved examinations of application security without the need for execution. However, malicious applications may use dynamic code loading, where applications download malicious updates from servers controlled by attackers after their installation. Attackers use this method to evade static analysis methods because the loaded code is not included in the application package. It is possible to track the behavior of applications using a dynamic analysis, which is beyond the scope of this study. Therefore, dynamic code loading was not taken into account.

Due to the fact that there is not enough Android TV malware on the market, we injected a malicious payload into benign applications to create Android TV malware. We used three different payload types suitable for an Android operating system. All of the Android TV malware in our study was created using Meterpreter malicious payloads in the Metasploit Framework. Because the models in this study were trained on Android TV malware created using Meterpreter malicious payloads in the Metasploit Framework, they may not detect malware using methods other than a Meterpreter payload.

6. Conclusions

The security of Android TVs becomes more important as the demand for smart TVs increases. However, there are some questions about Android TVs' security which we have investigated in our study. The first research question is about the malware ratio for Android TV applications in markets and public datasets. There are many benign Android TV applications on the market, but malicious Android TV applications are hard to find. Out of 1107 Android TV applications collected from a public dataset [20], only 3 were found to be malicious. Additionally, all 370 TV applications collected from a marketplace (APKMirror) were found to be benign.

The second research question investigated in our study is about the Android TV malware detection capability of a model trained on an Android smartphone malware dataset. When we tested a model [6] trained on mobile applications, it detected only 20 malware out of 471 malicious applications in the Androzoo Payload-Injected TV dataset.

The final research question in our study concerns the performance of Android TV malware classifiers. To detect Android TV malware, a dataset containing benign and malicious applications for Android TVs is needed. We collected Android TV applications, but there was not enough malware in the markets to train a model. Therefore, we converted benign applications by injecting them with the Meterpreter payload and obtained malicious Android TV applications. We extracted the n-grams from the "AndroidManifest.xml" and binary source (classes.dex) files in the Android TV applications and obtained 1000 features using the TF-IDF method. As a result of this study, we publicly shared 1000 features from 511 benign and 424 malicious Android TV applications in the training set and 182 benign and 188 malicious Android TV applications in the testing set. We trained nine separate models for detecting Android TV malware and compared their performance. We observed that the discriminant analysis classifier performed better than the other models. Provided with the extracted features, the discriminant analysis model classifies a TV application as malicious or benign in an average of 0.1346 milliseconds, with an F1-Score of 0.9789 and an MCC score of 0.9570.

There is extensive literature on Android security for mobile devices, but it is very limited for Android TV devices. In this study, we contributed to the literature on Android TV malware detection. The suggested model can be run on the servers of application markets and be used to scan application packages before their publication. It can be extended to detecting malware running on other types of devices, such as smartwatches

and automotive dashboards. We hope to observe further studies using the features from our public dataset. We also plan to investigate using a dynamic analysis for feature extraction as future work.

Author Contributions: G.O. carried out the literature review, drafted the manuscript, and proposed the methodology. Z.G.-A. conceived of the study and defined the literature taxonomy. M.A.E. joined in with this study's conception and acted as an academic advisor. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data underlying this article are publicly shared. A total of 1000 features and class labels from benign and malicious Android TV applications are available at the following website, the Android TV Malware Dataset: https://github.com/gozogur/AndroidTVMalwareDataset (accessed on 15 January 2025).

Acknowledgments: This study was prepared within the scope of the thesis titled "Security Analysis of Android Applications with Machine Learning Approach" in the Computer Engineering Department at Istanbul University-Cerrahpasa Institute of Graduate Studies.

Conflicts of Interest: Author Gokhan Ozogur was employed by the company Arcelik A.S. The remaining authors declare that the re-search was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Stats, S.G. Mobile Operating System Market Share Worldwide | Statcounter Global Stats. 2024. Available online: https://gs.statcounter.com/os-market-share/mobile/worldwide/ (accessed on 15 January 2024).
2. Research, V.M. Smart Android TV Market Size, Share, Scope, Growth, & Forecast. 2023. Available online: https://www.verifiedmarketresearch.com/product/smart-android-tv-market/ (accessed on 15 January 2024).
3. Ringol, M.A.; Barbour, N. Roku, TiVo Outline Disruptive Tactics in Connected TV Market at CES 2023 | S&P Global Market Intelligence. 2023. Available online: https://www.spglobal.com/marketintelligence/en/news-insights/research/roku-tivo-outline-disruptive-tactics-in-connected-tv-market-at-ces-2023 (accessed on 15 January 2024).
4. Karbab, E.B.; Debbabi, M.; Derhab, A.; Mouheb, D. MalDozer: Automatic framework for android malware detection using deep learning. *Digit. Investig.* **2018**, *24*, S48–S59. [CrossRef]
5. Yousefi-Azar, M.; Varadharajan, V.; Hamey, L.; Chen, S. Mutual Information and Feature Importance Gradient Boosting: Automatic byte n-gram feature reranking for Android malware detection. *Softw. Pract. Exp.* **2021**, *51*, 1518–1539. [CrossRef]
6. Ozogur, G.; Erturk, M.A.; Gurkas Aydin, Z.; Aydin, M.A. Android Malware Detection in Bytecode Level Using TF-IDF and XGBoost. *Comput. J.* **2023**, *66*, 2317–2328. [CrossRef]
7. Mariconti, E.; Onwuzurike, L.; Andriotis, P.; De Cristofaro, E.; Ross, G.; Stringhini, G. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. In Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017), San Diego, CA, USA, 23–26 February 2017.
8. Fang, W.; He, J.; Li, W.; Lan, X.; Chen, Y.; Li, T.; Huang, J.; Zhang, L. Comprehensive Android Malware Detection Based on Federated Learning Architecture. *IEEE Trans. Inf. Forensics Secur.* **2023**, *18*, 3977–3990. [CrossRef]
9. Halim, M.A.; Abdullah, A.; Ariffin, K.A.Z. Recurrent neural network for malware detection. *Int. J. Adv. Soft Compu. Appl.* **2019**, *11*, 43–63.
10. Garg, S.; Baliyan, N. A novel parallel classifier scheme for vulnerability detection in android. *Comput. Electr. Eng.* **2019**, *77*, 12–26. [CrossRef]
11. Grosse, K.; Papernot, N.; Manoharan, P.; Backes, M.; McDaniel, P. Adversarial Examples for Malware Detection. In Proceedings of the Computer Security—ESORICS 2017, Oslo, Norway, 11–15 September 2017; Foley, S.N., Gollmann, D., Snekkenes, E., Eds.; Springer: Cham, Switzerland, 2017; pp. 62–79.
12. Zhang, N.; Xue, J.; Ma, Y.; Zhang, R.; Liang, T.; Tan, Y.A. Hybrid sequence-based Android malware detection using natural language processing. *Int. J. Intell. Syst.* **2021**, *36*, 5770–5784. [CrossRef]
13. Arora, A.; Peddoju, S.K.; Conti, M. PermPair: Android Malware Detection Using Permission Pairs. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 1968–1982. [CrossRef]
14. Zhang, J.; Qin, Z.; Zhang, K.; Yin, H.; Zou, J. Dalvik opcode graph based android malware variants detection using global topology features. *IEEE Access* **2018**, *6*, 51964–51974. [CrossRef]

15. Alrabaee, S.; Al-Kfairy, M.; Taha, M.B.; Alfandi, O.; Taher, F.; El Fiky, A.H. Using AI to Detect Android Malware Families. In Proceedings of the 2024 20th International Conference on the Design of Reliable Communication Networks (DRCN), Montreal, QC, Canada, 6–9 May 2024; pp. 1–8.
16. He, Y.; Kang, X.; Yan, Q.; Li, E. ResNeXt+: Attention Mechanisms Based on ResNeXt for Malware Detection and Classification. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 1142–1155. [CrossRef]
17. Martinelli, F.; Marulli, F.; Mercaldo, F. Evaluating convolutional neural network for effective mobile malware detection. *Procedia Comput. Sci.* **2017**, *112*, 2372–2381. [CrossRef]
18. Narayanan, A.; Mahinthan, C.; Chen, L.; Liu, Y.; Saminathan, S. subgraph2vec: Learning Distributed Representations of Rooted Sub-graphs from Large Graphs. *arXiv* **2016**, arXiv:1606.08928.
19. Sharma, T.; Rattan, D. Characterization of Android Malwares and their families. *ACM Comput. Surv.* **2025**, *57*, 1–31. [CrossRef]
20. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 468–471.
21. Girish, A.; Tapiador, J.; Matic, S.; Vallina-Rodriguez, N. Towards an Extensible Privacy Analysis Framework for Smart Homes. In Proceedings of the 22nd ACM Internet Measurement Conference, New York, NY, USA, Nice, France, 25–27 October 2022; IMC '22, pp. 754–755. [CrossRef]
22. Majors, J.D.O.; Barsallo Yi, E.; Maji, A.; Wu, D.; Bagchi, S.; Machiry, A. Security Properties of Virtual Remotes and SPOOKing their violations. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, Melbourne, Australia, 10–14 July 2023; pp. 841–854.
23. Tileria, M.; Blasco, J. Watch over your TV: A security and privacy analysis of the android TV ecosystem. *Proc. Priv. Enhancing Technol.* **2022**, *3*, 692–710. [CrossRef]
24. Liu, Y.; Li, L.; Kong, P.; Sun, X.; Bissyande, T.F. A First Look at Security Risks of Android TV Apps. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Los Alamitos, CA, USA, 15–19 November 2021; pp. 59–64. [CrossRef]
25. Riadi, I.; Aprilliansyah, D. Mobile Device Security Evaluation using Reverse TCP Method. *Kinet. Game Technol. Inf. Syst. Comput. Netw. Comput. Electron. Control* **2022**, *7*, 289–298. [CrossRef]
26. Metasploit | Penetration Testing Software, Pen Testing Security. Available online: https://www.metasploit.com (accessed on 14 September 2023).
27. Meng, H.; Thing, V.L.; Cheng, Y.; Dai, Z.; Zhang, L. A survey of Android exploits in the wild. *Comput. Secur.* **2018**, *76*, 71–91. [CrossRef]
28. Breiman, L. Bagging predictors. *Mach. Learn.* **1996**, *24*, 123–140. [CrossRef]
29. Friedman, J.; Hastie, T.; Tibshirani, R. Additive logistic regression: A statistical view of boosting (with discussion and a rejoinder by the authors). *Ann. Stat.* **2000**, *28*, 337–407. [CrossRef]
30. Freund, Y.; Schapire, R.E. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* **1997**, *55*, 119–139. [CrossRef]
31. Seiffert, C.; Khoshgoftaar, T.M.; Van Hulse, J.; Napolitano, A. RUSBoost: Improving classification performance when training data is skewed. In Proceedings of the 2008 19th International Conference on Pattern Recognition, Tampa, FL, USA, 8–11 December 2008; pp. 1–4.