Computers, Materials & Continua
DOI:10.32604/cmc.2022.019623
Article

Tech Science Press

Droid-IoT: Detect Android IoT Malicious Applications Using ML and Blockchain

Hani Mohammed Alshahrani*

College of Computer Science and Information Systems, Najran University, Najran, 61441, Saudi Arabia
*Corresponding Author: Hani Mohammed Alshahrani. Email: hmalshahrani@nu.edu.sa
Received: 20 April 2021; Accepted: 24 May 2021

Abstract: One of the most rapidly growing areas in the last few years is the Internet of Things (IoT), which has been used in widespread fields such as healthcare, smart homes, and industries. Android is one of the most popular operating systems (OS) used by IoT devices for communication and data exchange. Android OS captured more than 70 percent of the market share in 2021. Because of the popularity of the Android OS, it has been targeted by cybercriminals who have introduced a number of issues, such as stealing private information. As reported by one of the recent studies Android malware are developed almost every 10 s. Therefore, due to this huge exploitation an accurate and secure detection system is needed to secure the communication and data exchange in Android IoT devices. This paper introduces Droid-IoT, a collaborative framework to detect Android IoT malicious applications by using the blockchain technology. Droid-IoT consists of four main engines: (i) collaborative reporting engine, (ii) static analysis engine, (iii) detection engine, and (iv) blockchain engine. Each engine contributes to the detection and minimization of the risk of malicious applications and the reporting of any malicious activities. All features are extracted automatically from the inspected applications to be classified by the machine learning model and store the results into the blockchain. The performance of Droid-IoT was evaluated by analyzing more than 6000 Android applications and comparing the detection rate of Droid-IoT with the state-of-the-art tools. Droid-IoT achieved a detection rate of 97.74% with a low false positive rate by using an extreme gradient boosting (XGBoost) classifier.

Keywords: Android; blockchain; analysis; malware

1 Introduction

The Internet of Things (IoT) has been used in a variety of applications, such as smart vehicles, smart homes, healthcare, smart shopping, and smart agriculture. This technology helps these applications to be on a connected network and digitizes them. Fig. 1 shows various Android IoT applications based on Android OS. These applications are used to collect data from different sensors in order to provide an intelligent solution for different tasks. All the collected data can be exchanged between a device and a device, a human and a device, and a device and other realistic environments [1].

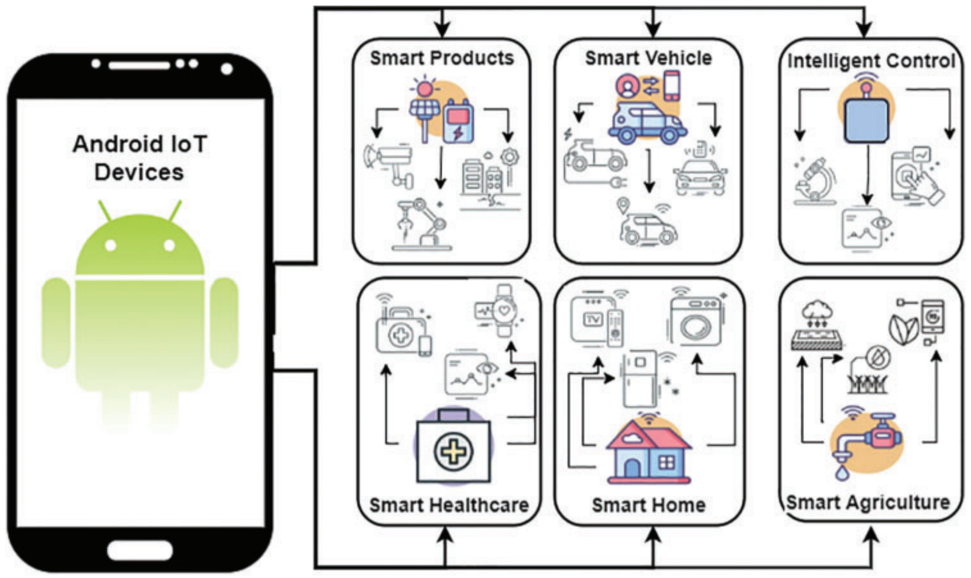


Figure 1: Different Android IoT applications

Android is the most popular platform for IoT devices, which has led to an increase in the number of applications available in the market, particularly Android applications. The number of mobile applications in the Apple App store as of the first quarter of 2020 was 1.96 million applications with an average of 9.000 applications submitted each month [2], while Google Play has more than 2.50 million applications available for Android users [2]. Concomitant with these statistics, the amount of malware targeting smart devices has increased dramatically because of the valuable information on these devices. As reported by the Secure-D platform [3], the number of malware applications discovered in the first quarter of 2020 on the Google Play store was 29,000 applications. Therefore, the popularity of the Android OS has made it an attractive target for cybercriminals to penetrate the variety of Android application marketplaces with malware applications.

Adversaries construct different types of malicious applications. For example, exploiting open ports where the attacker controls the user's device by opening one of the ports on the device. This technique allows the attacker to remotely access all of the device's resources without requesting permission from the owner [4]. Therefore, adversaries use android applications as a way to break the security mechanism of the device, which allows them to access sensitive information such as the device's location, contact information, and photos.

Blockchain was introduced for the first time by Satoshi Nakamoto in 2008 [5], and according to the Top 10 Strategic Technology Trends for 2019 by Gartner [6], blockchain is ranked among the top ten strategic technologies. It has been used in many cybersecurity services, such as authentication, confidentiality, access control list, data sharing, and malware detection to elevate the existing solutions and to prevent malicious actors. Blockchain can be divided into three different types as follows: public blockchain, consortium blockchain, and private blockchain. The first type allows all the nodes to check and verify transactions. The second type restricts the participation of the nodes in the consensus to a group of nodes. The third type does not allow all of the nodes to participate in the consensus and validation process, but it does allow a pre-defined node in these processes.

Various researchers have proposed different techniques to detect malicious applications in the Android OS by using collaborative approaches and blockchain technology. These techniques range from tools running on Android devices to web services that allow users to submit any application for analysis. Some examples include [7–10]. These tools extract one or more features such as permissions, APIs, system events, and permission rates from an inspected application. Then, they apply machine learning techniques to classify the inspected application as malware or benign. However, most of the existing tools depend on a few sets of features that do not represent the behavior of the inspected applications. The others only consider a few samples to evaluate their proposed systems.

This paper presents Droid-IoT, a detection framework for Android IoT devices. This framework is based on the blockchain technology, which is a well-known technology in terms of security, reliability, and transparency. The proposed framework allows users to verify any new Android application installed on their devices on the basis of an analysis that is requested by other users. Therefore, the analysis of a new application is achieved by utilizing a machine learning algorithm that classifies the inspected application as either a benign or a malicious application. The proposed system consists of four main engines: (i) the collaborative reporting engine that searches for the generated hash of the newly installed application on all the participating devices that have Droid-IoT installed; (ii) the static analysis engine, which is responsible for disassembling the inspected application to the source code to extract all of the required features, such as the utilized APIs and the application's meta-data, permissions, activities, services, and receivers; (iii) the detection engine, which contains the intelligent model that applies the extracted features to the machine learning model in order to classify the inspected application as either benign or malicious; and (iv) the blockchain engine that is responsible for timestamping and adding each new analyzed application to the blockchain of Droid-IoT. Therefore, the signatures of the analyzed applications will be stored securely in a private blockchain architecture to provide secure storage for all the analysis results. Moreover, each engine contributes to the detection and minimization of the risk of malicious applications and the reporting of any malicious activities to the user. The proposed system was tested and evaluated on more than 6000 recent Android applications that were developed between 2018 and 2020.

In summary, the main contributions of this research are as follows:

- Present Droid-IoT, a collaborative system that detects Android IoT malicious applications by using machine learning and the blockchain technology.
- Evaluate the proposed system on a dataset that contains more than 6000 real Android applications obtained from the AndroZoo dataset [11].
- Deploy and execute Droid-IoT and effectively detect most of the malicious applications with low type II error rate.
- Achieve a better detection rate than the existing tools by using the same benchmark dataset.

The remainder of this paper is organized as follows. Section 2 discusses the background of the Android OS, including the application structure and the Android malware. This section also gives an overview of the Android detection techniques. Section 3 discusses the related work. Section 4 presents Droid-IoT's methodology and approach. Section 5 presents a discussion of the proposed framework's analysis results. Finally, Section 6 presents the conclusion of this paper.

2 Background

This section presents an overview of the Android OS and its application structure, followed by the different types of Android malware.

2.1 Android

Android is a platform created by Google and the Open Handset Alliance (OHA), which is an association of different mobile technology companies. The official marketplace from which to download Android applications is the Google Play store, which has different categories such as books, games, and movies. Besides the official market, Android allows its users to download applications from third-party stores without any restrictions.

The architecture of Android consists of four main layers. The base of these layers is the Linux kernel, which is one of the most well-known operating systems. It has many features such as portability, security, and open source. The Native libraries and Android runtime are above the Linux kernel. On top of this is the application framework (Java API framework). All the applications installed by the users or pre-installed on the device run on the applications layer, which is located on top of all of the previous layers. The Linux kernel provides Android with several security features such as permissions and process isolation. This isolates the users' resources from each other. There are two main security features in Android. (1) Application Sandboxing. This feature isolates the system resources from the applications and the applications from each other by taking advantage of the Linux kernel. Each application on Android is assigned a User ID (UID), sometimes called app ID, which runs the application as an isolated process.

Furthermore, each application is given a specific data directory, which is the only directory that the application has permission to read from and write data to unless there is another app configured to have the same UID. (2) Android permission model. This model is introduced to control the use of the services that are provided for use by the installed applications. This model controls the use of the services that might have sensitive information about the user.

For example, the developers might use the TelephonyManager API, which will provide them with sensitive information such as the user's phone number. Android uses this model to request permissions from the user at the time of installation to allow an application to use one of the sensitive APIs or resources. These permissions are grouped into four main categories:

- Normal permissions: This type of permission is granted by the system automatically without asking for the user's approval. Usually, these permissions are of low risk to other applications, systems, and the user. An example of such a permission is changing the device's background wallpaper.
- Dangerous permissions: This is a high-risk type of permission that shows that the requesting application will have access to sensitive information. This type of permission is not granted automatically by the system. However, the user is asked to approve this type of permission at the time of installation before proceeding.
- Signature permissions: This type of permission is granted by the system automatically only if the requesting application is signed with the same certificate as the application that created the permission. Otherwise, the user has to grant the requesting permission. Commonly, this category is used to share data between applications created by the same developer.
- Signature or System permissions: This type of permission is granted only to system applications, referring to applications in the Android system image.

2.2 Android Application Structure

As shown in Fig. 2, Android applications are zipped into a package called Android Package Kit (APK). The package kit is a zip file, but it is in a different format and has a different file extension. The APK file contains several files and folders, described as follows:

- Assets: This is an optional directory that has all the assets of the application.
- lib: Another optional directory that has all the compiled native libraries such as sheared object libraries.
- META-INF: This directory contains all the files related to the security information and certificates of the application. Listed below are the most significant files in this directory:
  - Manifest.MF: It contains the digest SHA-1 of all the resources used by the application.
  - CERT.RSA: It contains the public key cryptography standards (PKCS), the names of different cryptographical algorithms, and the cryptographic message syntax.
  - CERT.SF: The digest file in the form of SHA1 for Manifest.MF is saved in this file. The name of the file can be changed depending on the signer of the signature. However, the file should be named as (signer_name) SF.
- AndroidManifest.xml: This file contains important information about the application such as the package name, the permissions needed, and all the components defined in the application.
- Classes.dex: This is an executable file that has the Dalvik bytecode, converted from the Java source code.
- Res: This directory has all of the resources used by the application (e.g., images, icons, and strings) that are not compiled into resources.arse, which has all of the precompiled resources.

To examine an APK file, it must be unpacked to a readable format. There are many tools that can do this, such as [12] and [13]. APKtool [13], is the most popular tool used by many researchers. This tool is an open-source tool designed to reverse-engineer Android applications by decoding the APK file. It takes a few minutes to decode an APK file. The APK file is converted into a directory containing the above file structure, which can be examined by a developer or automated by a program.

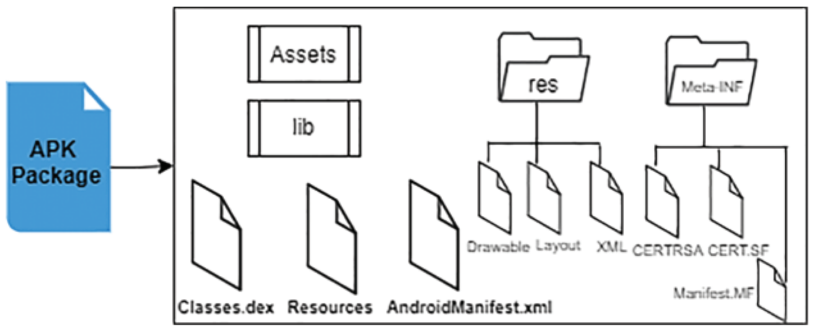


Figure 2: Android application structure

2.3 Blockchain Technology

Blockchain is a distributed system that allows for the storing, exchanging, and verifying of transactions among untrusted parties in the network without relying on a centralized control unit. The communication is between all the parties in the form of a peer-to-peer network, but the decision is made by the majority instead of a single node. This encourages cybersecurity researchers to use this powerful and sustainable technology in their solutions. According to Paul et al. [14], blockchain has been used in many cybersecurity services such as authentication, confidentiality, access control list, data sharing, and malware detection to elevate the existing solutions and to prevent malicious actors. Blockchain was introduced for the first time by Satoshi Nakamoto in 2008 [5], and in the Top 10 Strategic Technology Trends for 2019 by Gartner [6], blockchain is ranked among the top ten strategic technologies. The first application based on blockchain technology was Bitcoin. This is a cryptocurrency that allows users to trade things on the Internet as in the real world. As of January 2021, the Bitcoin market capitalization reached 600 billion U.S. dollars [15].

As shown in Fig. 3, blockchain consists of two main components, namely a blockchain database and blockchain nodes [16]. The database in the blockchain technology is shared and distributed among all of the nodes to append new records without the ability to alter or delete records. This allows the users to record data in a distributed form and store them as blocks. This makes the data available to all the users of the blockchain. The entire blockchain consists of many blocks that are continuously added to the chain of all the valid transactions. Therefore, all the blocks are connected to form a chain where each block has two components block header and transaction list that have already been verified by the nodes in the blockchain. As shown in Fig. 4, Block Header has the block's metadata, including the following [17]:

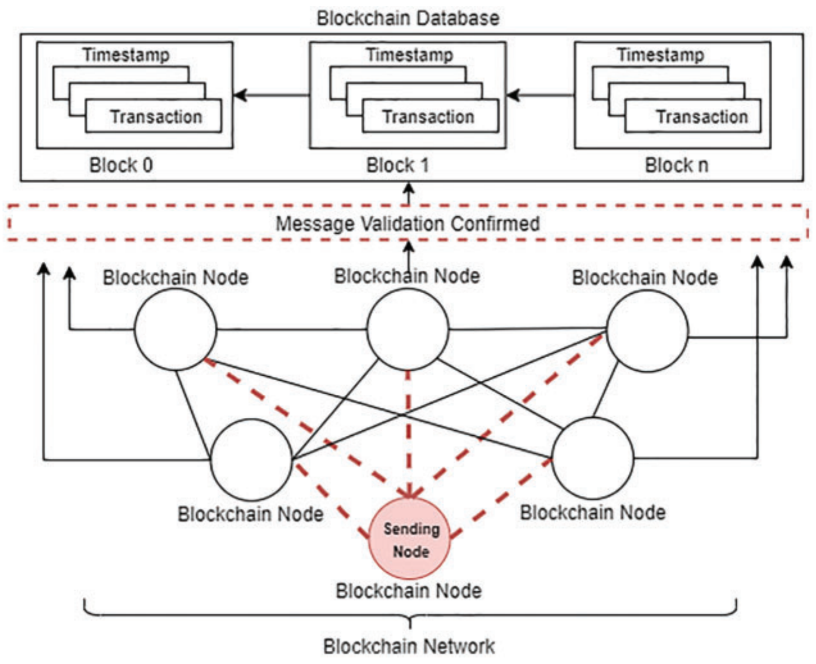


Figure 3: Blockchain architecture

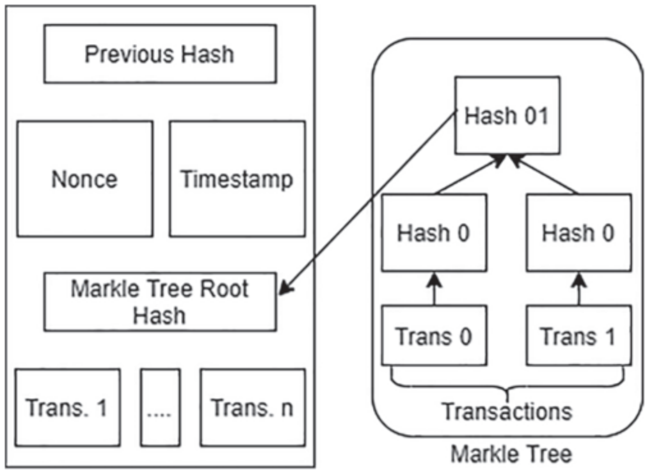


Figure 4: Block structure

- Nonce: This is the value obtained by solving a mathematical challenge by the miners.
- Timestamps: The current timestamp of the system.
- Merkle root tree: This is the transaction verification that contains the hash value calculated for all the transactions.
- Previous block: It contains the hash value of the previous block.

The second component in any blockchain technology is the blockchain nodes. These multiple nodes are connected in a peer-to-peer form.

The blockchain works as follows: First, as shown in Fig. 3, the sending node broadcasts the new records (message) to all of the other nodes in the network to check and verify the information. Second, if the record is successfully checked and verified, then the record will be added to a block. Otherwise, the record will be rejected. Third, all the nodes execute one of the consensus protocols to the new block. Fourth, the new block will be appended to the network if all of the participating nodes in the network admit the block. Each block consists of the following: records, the hash value of the previous block, the hash value of the current block, the timestamp of the generated block, the signature of the block, the nonce value, and additional information that can be defined by the user.

2.4 Types of Blockchain

A blockchain can be divided into three different types as follows: public blockchain, consortium blockchain, and private blockchain. These types are discussed in this subsection.

2.4.1 Public Blockchain

This is the first type of blockchain that allows all the nodes to check and verify transactions. Moreover, the public blockchain is called the permission-less blockchain because it allows all the nodes to join the blockchain and conduct mining. All the nodes in this type of blockchain can conduct tasks such as writing to the blockchain and reading or reviewing the blockchain contents. A public blockchain can be targeted by a sybil attack, where the attacker makes the consensus impossible by creating fake nodes. This is because all the nodes are not unknown to each other before the mining is completed [17]. Some examples of this type of blockchain are Bitcoin, Ethereum, and Blockstream.

2.4.2 Private Blockchain

This is the second type of blockchain, which restricts the participation of the nodes in the consensus. It is called a permissioned blockchain. It only allows authorized nodes to share private data. The mining process is controlled by an authorized group; hence, new or unknown users cannot participate in the mining unless they are invited by the control authority [17]. According to Vincent [18], a private blockchain is vulnerable against 51% of the attacks, where most of the hash rate is controlled by a group of miners. Some examples of the private blockchain are Ripple and Hyperledger.

2.4.3 Consortium Blockchain

The third type of blockchain is consortium blockchain, which does not allow all of the nodes to participate in the consensus and validation process, but it does allow a few nodes to participate in these processes. The multi-signature method is considered for mining a block in this type of blockchain, and it will not be considered a valid block before it is approved by the controlling node [17]. This type is vulnerable to a tamper attack, which can be achieved by the malicious collaboration of the controlling nodes [17]. Multichain and Blockstack are some examples of this type.

Tab. 1 presents a comparison of the blockchain types, as discussed in [17] and [19], in terms of their determination of consensus, permissions, efficiency, network centralization, scalability, and vulnerability.

Table 1: Comparison of different blockchain types

Property	Public blockchain	Private blockchain	Consortium blockchain
Determination of consensus	By all miners	Selected group	Selected group
Permissions	Permission-less	Permissioned	Permissioned
Efficiency	Low	High	High
Centralization	No	Partially centralized	Yes
Scalability	High	High	Low
Vulnerability	Sybil attack	51% attack	Tamper attack

2.5 Consensus Protocols

The consensus protocols in the blockchain are the protocols that assist different nodes in a distributed network to reach an agreement whenever needed [20]. There are different consensus protocols in a blockchain, as discussed in the following subsections.

2.5.1 Proof of Work (PoW)

A participant in the network needs to solve a puzzle to get a chance of receiving the mining reward. This protocol has high resource and power consumption. Moreover, it has a low processing speed with low energy efficiency [17]. One of its limitations is that it is less secure than the other consensus protocols.

2.5.2 *Proof of Stake (PoS)*

This consensus is similar to the PoW but with a critical difference. The opportunity of receiving block validation in the network depends on its wealth in the system [21]. This protocol utilizes low resources with fast processing speed. Some of its limitations are that it requires high energy efficiency and that the node with the highest wealth in the system has the consensus control [17].

2.5.3 *Delegated Proof of Stake (DPoS)*

This protocol is based on the PoS protocol. However, it depends on the voting mechanism to choose a delegated node to create and validate blocks [21]. This mechanism makes the processing faster with low resource consumption. However, it has high energy efficiency and makes the blockchain more centralized [17].

2.5.4 *Proof of Burn (PoB)*

In this protocol, the miners will get the chance to validate new blocks on the basis of the number of coins they send to an irreversible address [21]. This protocol has higher resource consumption than DPoS with less processing speed. The energy efficiency of this protocol is low, but some of the resources are wasted [17].

2.5.5 *Proof of Elapsed Time (PoET)*

To overcome some of the PoW drawbacks, which is high energy consumption, Intel proposed a new protocol to select the miner. This protocol is based on a random number selected by the miner and considers the miner whose timer finishes first the winner. The calculation of the time is completed by a trusted party such as Software Guard Extension (SGE) by Intel [21]. Moreover, this protocol utilizes lower power consumption than PoW, but it consumes high resources with a relatively slow speeding process [17]. One limitation of this protocol is that it depends on third-party software, such as Intel's, which prevents it from being completely decentralized.

2.5.6 *Proof of Capacity (PoC)*

This protocol depends on the space availability on the miners' hard drive. Therefore, miners have to be able to store large amounts of data in order to get the opportunity to mine a new block [21]. This protocol is known for its high resource utilization and energy efficiency with slow processing speed. One of its limitations is the selection of the next mining node, which depends on the disk space.

2.5.7 *Practical Byzantine Fault Tolerance (PBFT)*

The consensus of this protocol relies on the voting process to make the decision. Therefore, this consensus method requires two-thirds of the participating nodes to approve the block in order for it to be added [21]. This method consumes considerable resources and requires high processing speed. The energy efficiency of this consensus method is high, and the communication overhead is in turn high [17].

2.5.8 *Proof of Authority (PoA)*

This consensus protocol is one of the protocols designed for the permissioned blockchain. It was introduced to solve some limitations of the PoW consensus method, such as the consumption of high energy [17]. In this consensus method, the creation of a new block and agreement is achieved by majority votes. This consensus protocol requires high consumption with low energy efficiency. Scalability is one of its limitations.

2.5.9 Raft

The Raft consensus protocol is based on the voting scheme. It has two main operations as follows: leader selection and log replication. The leader is selected by using randomized timeout [17]. The second operation, log replication, is responsible for accepting logs from clients and generating its own transaction. This method has high energy efficiency with less resource consumption than PoW. The processing speed is fast, but the method has limited throughput with low security [17].

2.6 Android Malware

Malware is malicious software designed to engage in and execute harmful activities such as disabling the affected devices and stealing sensitive information. There are two ways for malware to be distributed. The first option is that the attacker creates an app that has malicious activity programmed into it before its market release. The second option is that the attacker inserts malicious code into a legitimate application and then re-introduces the application into the market. As reported by GData CyberDefence [22], the number of Android malware applications discovered in 2019 was more than 11,000. The same report showed that more than four million malicious applications were found in 2019. One example of recent malware is EventBot. This targets financial applications to access the users' sensitive information [23]. Once the malware is installed on a user's device, it requests dangerous permissions such as reading and writing to the external storage and sending and receiving SMS messages. The EventBot malware has targeted more than 150 different financial Android applications, such as mobile banking, crypto-currency, and money transfer [24]. Different malware types have been reported by the Android antivirus software, such as spyware, trojans, adware, and ransomware.

2.7 Detection Techniques

Several Android analysis and detection techniques have been developed to protect the Android OS and its users against malicious applications. The detection technique may or may not be running on the same device as the one it is trying to protect. If it is running on the same device, it monitors all the installed applications and performs a pre-configured detection technique. However, if it is not running on the same device, it monitors the system through an agent application installed on the device and sends all the activities and logs to a server. This will perform the detection technique and send the results back to the agent application.

A detection technique may apply one of the following analysis approaches:

- Static: This approach examines the inspected application's syntax and structural properties without running the actual code. A static analysis can be performed using one of the three following techniques: 1) data-flow analysis, 2) symbolic execution, and 3) dependence analysis.
- Dynamic: This approach collects information about an application's execution during its actual runtime. The inspected application will execute in a complete Android environment, which will allow the detection technique to monitor its actions and activities at runtime.
- Hybrid: For a comprehensive analysis and to increase the strength of the assessment tools, dynamic and static analyses are combined in this approach. For example, the detection tool can analyze the inspected application statically, and then, it can run the same application on an Android device or emulator to extract the required dynamic features. Hybrid analyses have been applied by many of the existing tools to detect malicious applications. One of the tools used to conduct a hybrid analysis for an Android application is Android Application Sandbox (AASandbox) [25]. This tool allows the analysts to perform both a

static analysis and a dynamic analysis of the inspected application. First, it analyzes the inspected application statically to find any suspicious code. Then, it runs the inspected application in an Android emulator to log all the system calls requested by the analyzed application.

3 Related Work

A variety of tools have been proposed to detect Android malicious applications by using different analysis techniques. For example, Zhu et al. [26] introduced a new detection technique called DroidDet. This approach can detect malicious applications by extracting some of the features statically from the inspected application. These features are as follows: permissions, APIs, system events, and permission rate. It then takes these features as the input for an ensemble algorithm. The proposed system achieves an accuracy of 88.26%.

Another approach was proposed by Wang et al. [27]; it extracts all the permissions from the inspected application and then constructs the permission patterns to classify malicious applications from benign applications. All the permissions are extracted using the static analysis technique to extract only the permissions that a requested application uses during the runtime. An analysis of 1227 benign applications and 1227 malicious applications revealed that 33 unique permissions were requested only by benign applications, whereas 20 unique permissions were requested by malicious applications. As a result, by using a permission pattern to detect malicious applications, this tool achieved an accuracy rate of 94%. However, when the J48 classification algorithm was used, the detection rate accuracy dropped to 85%. As a limitation of this work, the researchers only considered permissions to classify malicious applications, which could increase the false-positive rate.

Moreover, Jiang et al. [28] proposed another approach to defend against Android malware by introducing a detection tool called fine-grained dangerous permission (FDP). This tool uses the static analysis technique to extract the fine-grained features of dangerous permissions from the inspected application. It has been evaluated on a dataset of 1700 benign applications and 1600 malicious applications. FDP applies different machine learning algorithms such as J48, KNN, SVM, and NB to detect malicious applications. It achieved an accuracy of 93.7% when the KNN algorithm was applied. This approach focuses only on one type of permissions, which is dangerous permission that can be evaded easily by malicious applications.

Arora et al. [29] proposed another technique to detect malicious applications in the Android OS on the basis of the permissions stated in the manifest file. The proposed system builds a graph of all the permission pairs extracted from both malicious and benign applications. All of the extracted permissions are extracted from the manifest file of the inspected application and then processed in three different phases as follows: i) graph construction phase, ii) graph merge phase, and iii) detection phase. The system was evaluated using 7533 sample applications: 2944 were used during the training phase, while 3264 sample applications were used during the testing phase. The proposed method achieved an accuracy of 95.44%. Moreover, this proposed technique considered only one type of feature to detect malicious applications. If the inspected application is updated by the developer, the proposed system cannot detect whether there is a malicious activity involved in the updated version or not.

Furthermore, a deep learning algorithm has been used by many researchers to detect malicious Android applications. For example, Li et al. [30] proposed a fine grained system to detect malicious applications in Android OS. The proposed system extracts features from the inspected application by using a static analysis from classes.dex and the AndroidManifest file. The extracted features are as follows: permissions, APIs, and metadata information. The proposed system was evaluated on 123,453 benign applications and 5560 malicious applications. The detection rate of the proposed system was 97%. The huge difference in the dataset utilized in the proposed system was one of the major limitations.

Another tool was proposed by Booz et al. [31]; it utilizes deep learning to detect malicious Android applications. The proposed system extracts features by using a static analysis from the manifest file of the inspected applications. The proposed method was evaluated on a dataset of 48,643 Android applications and achieved an accuracy of 95%.

Furthermore, Naway et al. [32] proposed another tool that utilizes a deep neural network to detect malicious Android applications. The proposed system extracts the following features from the inspected applications: APIs, permissions, intents, certificates, and extensions of the executable file in the inspected application. The proposed tool achieved an accuracy of 95.31% when evaluated on 600 benign applications and 600 malicious applications. As a limitation of this tool, the researchers only considered a few samples to evaluate the proposed system.

Limitations of Existing Works: As Android IoT malicious applications are becoming a severe problem for Android users, a comprehensive solution is needed to protect both user's data and the resources of user's devices. Unfortunately, the existing solutions suffer from many limitations. For example, the proposed tool by Zhu et al. [26] was evaluated on a few samples: 600 benign applications and 600 malicious applications. The algorithm used for training and detecting malicious applications was also computation intensive. Moreover, other works such as Wang et al. [27], Jiang et al. [28], and Arora et al. [29] only considered one type of features to detect malicious applications, which can be evaded easily by malicious applications.

Therefore, Droid-IoT overcomes the limitations of the existing tools by introducing a new technique to detect Android IoT malicious applications by analyzing the following features: permissions, APIs, services, receivers, activities, and meta info. Furthermore, it applies several machine learning algorithms to achieve the best results in terms of time and space complexities and then uses a private blockchain to store the signatures of the analyzed applications.

Tab. 2 presents a comparison of the existing works in terms of their determination of analysis, extracted features, number of applications used, and accuracy of the detection system.

Table 2: Comparison of existing works

Tool	Analysis	Features	Dataset		Accuracy (%)
			Benign	Malicious	
DroidDet [26]	Static	Permissions, system events, APIs and permission rate	600	600	88.26
Wang et al. [27]	Static	Permissions	1227	1227	94
FDP [28]	Static	Dangerous permissions	1700	1600	93.7
Arora et al. [29]	Static	Permissions	–	–	95.44
Li et al. [30]	Static	Permissions, APIs, and meta-data information.	123,453	5560	97
Booz et al. [31]	Static	Permissions	–	–	95
Naway et al. [32]	Static	API, permissions, intents, certificates, and extensions	600	600	95.31

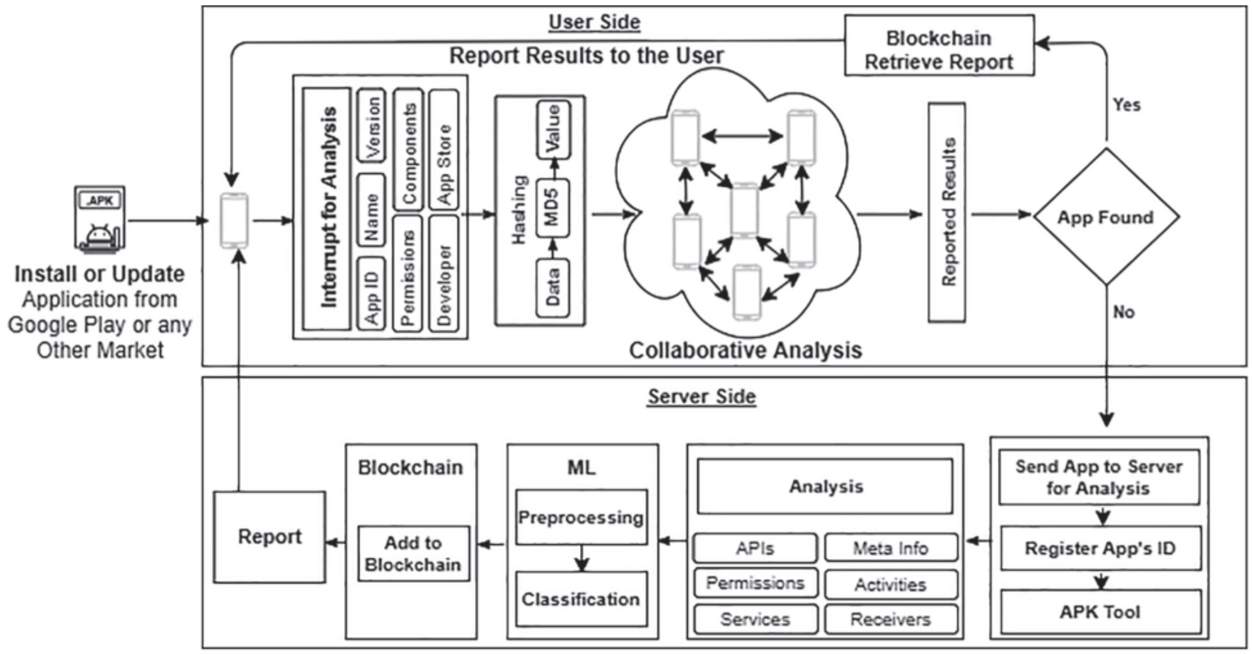


Figure 5: Droid-IoT workflow

4 Methodology

The goal of this research was to design and implement a collaborative system, called Droid-IoT, that detects malicious Android IoT applications. As shown in Fig. 5, the analysis starts once a new application is installed or an installed application is updated. Droid-IoT consists of two sides: (1) user side, which is an Android application running on the user's device that monitors whether a new application is installed or an installed application is updated, and (2) server side, which is the back-end server that processes all the requests received by the user side. Therefore, once a new application is installed or updated, Droid-IoT works as follows: (1) Each inspected application will be analyzed by Droid-IoT to extract the metadata of the inspected application. This includes the application name, package ID, the version of the application, all the requested permissions and components, developer ID, and the app store that was used to download the inspected application. (2) All the extracted information will be hashed to be searched on all the participating devices that have Droid-IoT installed. (3) If the hash is found on one of the participating devices, the hash will be verified by Droid-IoT's blockchain. (4) If the hash cannot be found on one of the devices, then the inspected application will be sent to the server side for a complete analysis. (5) On the server side, the inspected application will be disassembled into the source code by utilizing the apktool [13]. (6) In this step, Droid-IoT extracts all of the required features including APIs, application's metadata, permissions, activities, services, and receivers. (7) The extracted features will be used to apply the machine learning model to classify the inspected application as either benign or malware. (8) Once the inspected application is analyzed and classified by the machine learning model, the results and the hash values of the extracted features will be timestamped and added to the blockchain of Droid-IoT. Therefore, if a request to analyze an application is received by the Droid-IoT node in the future, then the results of the corresponding application will be retrieved from the blockchain provided the extracted features' hash values match. Otherwise, the application will be analyzed and classified as a new application. The proposed tool will prevent also attacks that target the database storage that is used in traditional solutions. As shown in Fig. 6, the attackers target the database storage of all the analysis results in order to modify the signatures and redistribute attacks that were detected before. Therefore, the proposed method will prevent the attackers from manipulating the signatures in the cloud storage. Algorithm 1 shows the detection procedures for the Droid-IoT system.

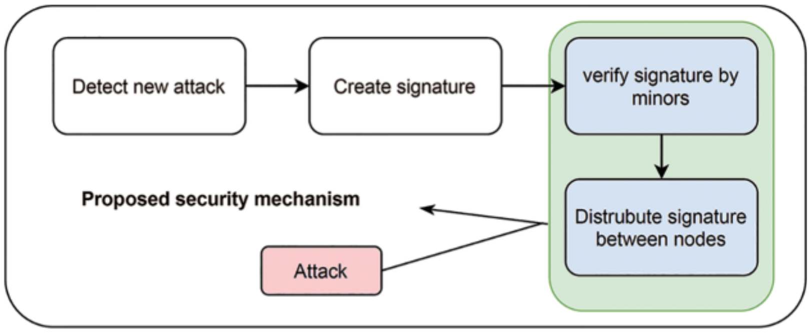

Detect new attack
Create signature
verify signature by minors
Distrubute signature between nodes
Proposed security mechanism
Attack

Figure 6: Secure the analyzed results

Algorithm 1: Droid-IoT Detection Procedures

```
Input EF: Features from the APK file (Permissions, APIs, Services, Receivers, Activities, Meta
      info)
Output Result: 0-Normal App; 1-Malicious App
 1: D = LoadMLModel()
 2:     for EachInsttaledApplication do
 3:       CheckAppInBlockChain;
 4:       if NotExist then
 5:         D.predict(EF)
 6:         if Result = 0 then
 7:           installApp;
 8:         else
 9:           BlockApp;
10:           AddAppToTheBlockchain;
11:       else
12:           ReturnResult;
```

4.1 Data Preprocessing

Data normalization was implemented as the first step of the preprocessing stage. As shown in Eq. (1), all the data were normalized to be in the same scale [33]. Here, X_{norm} denotes the normalization results, X is the value of the selected feature before the normalization, and X_{max} and X_{min} represent the largest and the smallest values in the corresponding column, respectively.

$$X_{\text{norm}} = \frac{X - X_{min}}{X_{\max} - X_{\min}} \tag{1}$$

4.2 Feature Selection

Applying one of the feature selection algorithms in the preprocessing stage is an essential step in data modeling. The goals of applying the feature selection technique are to find the best set of features to classify benign applications from malicious applications and to eliminate features that provide little or no predictive information. The F-test feature selection algorithm is utilized by Droid-IoT to reduce the number of extracted features. It is one of the filter methods that compute the score of each feature by considering the relationship between the feature and the target variable [34]. The F-test is a statistical test used to compare between the models and to check whether there are any important differences. The score for each feature Xi is calculated using Eq. (2) [35].

$$t(X_i) = \frac{|\mu_{i1} - \mu_{i2}|}{\sqrt{\frac{\sigma_{i1}^2}{n_1} + \frac{\sigma_{i2}^2}{n_2}}} \tag{2}$$

where μ_{i1} and μ_{i2} refer to the mean of the i-th feature for class C_j, where j is equal to 1 or 2, which donates the class index; n_1 and n_2 refer to the sizes of the group for the first class and second class samples, respectively. Moreover, σ_{i1} and σ_{i2} refer to the standard deviation of the i-th feature for class C_j. The top 50 selected features after the application of the F-test feature selection are shown in Fig. 7. Not only F-test feature selection is considered, but the chi-square algorithm is also utilized to find the best set of features. The latter calculates the independence between the label and each feature, as shown in Eq. (3).

$$X^2(t,c) = \frac{N \times (AD - CB)^2}{(A + C) \times (B + D) \times (A + B) \times (C + D)} \tag{3}$$

where N represents the number of samples, A represents the number of times t and c co-occur, B represents the number of times t occurs without c,C represents the number of times c occurs without t, and D represents the number of times neither t nor c occurs.

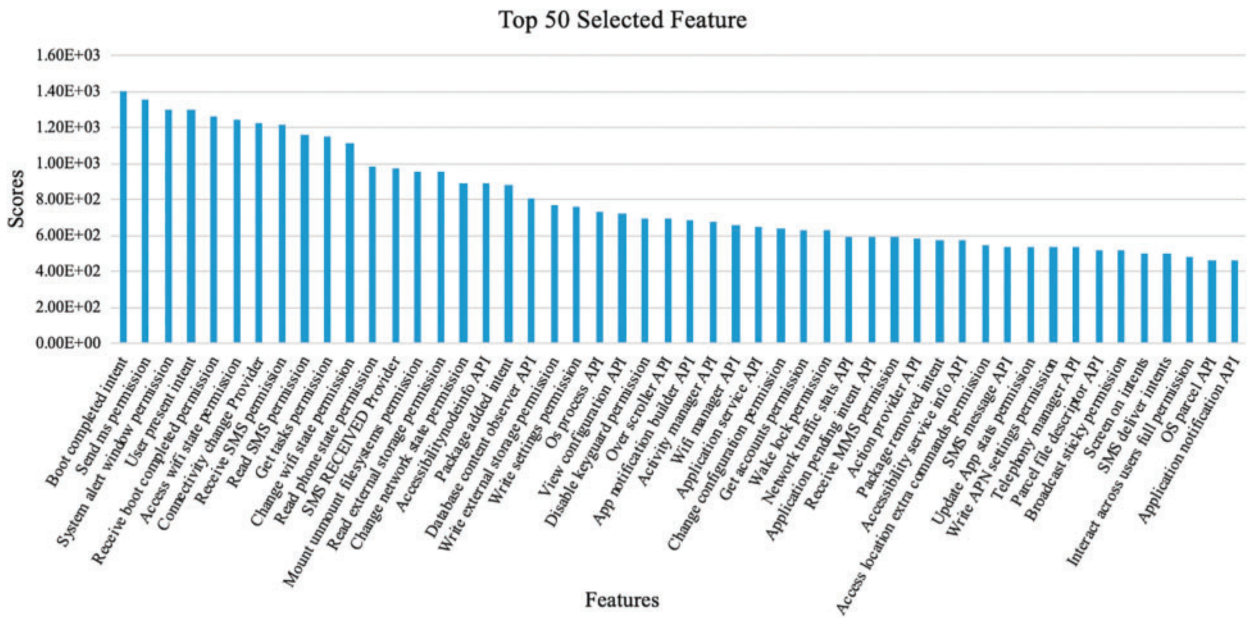


Figure 7: Top 50 selected features by F-test feature selection algorithm

5 Evaluation and Results

This section introduces the experimental setup of the Droid-IoT machine learning model, which includes the training and testing datasets and all the evaluation metrics. This section also shows a comparison of all the analyzed features.

5.1 Experimental Setup

Droid-IoT analyzed more than 6000 Android applications. These applications were obtained from the AndroZoo [11] dataset. The ground-truths of these applications are as follows: 3100 benign applications and 3100 malicious applications. The malicious applications analyzed by Droid-IoT were selected on the basis of the discovery date. Therefore, in this study, only the malicious applications discovered between 2018 and 2020 were considered. Moreover, Droid-IoT extracted different sets of features from each application by using static analysis, which mean inspect application's syntax and structural properties without running the actual code. Therefore, Droid-IoT extracted the following features:

- Application Permissions: These include all of the permissions requested by the inspected application. All the permissions are extracted from the manifest file of the analyzed application.
- Application Programming Interfaces (APIs): APIs are programs that have already been coded to be used by the developers to allow them to add certain functionalities into their applications. All the extracted APIs are documented and explained on the Android developer website [36].
- Application's Receivers: These receivers use the Publish/Subscribe approach, where the application will be triggered when the subscribed event is activated [37]. For example, the developer can define a broadcast receiver to show a text message on the running application's screen when a new text message is received. It can be used within the same application or in a different application on the device without using any visual representation.
- Application's Intent: This intent involves the messages that are sent between the applications to perform or run an operation by another application, even if the requested application does not have the capability to perform the intended request. Intent comes in one of two types: explicit and implicit [37]. In an explicit intent, the developer specifies which component receives the intent. This type only works with components in the same application. In contrast, the implicit intent specifies the type of component required and lets the user decide on how to proceed. For example, if a user has two Internet browsers installed on the device and there is an intent that wants to open a link, it gives the user the ability to choose from these two browsers.

5.2 Evaluation Metrics

To evaluate the performance of the detection model, the following metrics were considered:

- Accuracy: Ratio of the total number of applications that are correctly classified to the total number of applications. Accuracy is calculated using Eq. (4):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

where TP stands for true positive. This means that the model correctly classifies the malicious samples as malicious. TN refers to true negative, which means that the model correctly classifies benign samples as benign. FP refers to false positive, which means that the model could not classify the benign samples as benign. FN refers to a false negative, which means that the model could not classify the malicious samples as malicious.

- Type I Error or FP Rate: Ratio of the total number of benign applications that are not classified correctly to the total number of benign applications. This can be calculated using Eq. (5):

$$\text{Type I Error} = \frac{FP}{TN + FP} \tag{5}$$

- Type II Error or FN Rate: Ratio of the total number of malicious applications that are not classified correctly to the total number of malicious applications. This can be calculated using Eq. (6):

$$\text{Type II Error} = \frac{FN}{FN + TP} \tag{6}$$

- F1-Score: This refers to how discriminative the model is, which is calculated by using Eq. (7):

$$\text{F1-Score} \quad = 2 * \frac{precision * recall}{precision + recall} \tag{7}$$

where *precision* represents the ratio of the malicious samples that are classified correctly to the total number of all samples that are classified as malicious; and, *recall* represents the ratio of the malicious samples that are correctly classified to the total number of malicious samples. Additionally, *sensitivity* and *specificity* metrics were considered to evaluate the performance of the detection model. Therefore, sensitivity represents the percentage of malicious samples that were correctly classified as malicious; and specificity represents the percentage of benign samples that were classified correctly as benign.

5.3 Results and Discussion

5.3.1 Features Analysis

Different types of features were analyzed using Droid-IoT to distinguish between benign applications and malicious applications accurately. Fig. 8 shows the top 50 features requested by both benign and malicious applications.

Moreover, one of the feature sets analyzed by Droid-IoT was the requested permissions. As shown in Fig. 9a, benign and malicious applications requested different types of permissions. Some of the requested permissions were normal permissions, which would be granted to the requested applications automatically by the system because they would not expose any sensitive information. For example, the INTERNET is the most normal permission requested by both benign and malicious applications. This permission is requested by an application to allow the requested application to have an internet connection. Malicious applications requested the INTERNET permission more than benign applications did, as shown in 9(a). Nevertheless, the ACCESS_WIFI_STATE permission can be used to list all of the WiFi networks that the device has used in the past. From this information, an attacker can discover all of the target's visited places [38]. Moreover, the RECEIVE_BOOT_COMPLETED permission was requested 1730 times by malicious applications, while it was requested only 180 times by the benign applications. Applications that granted this permission were permitted to receive a notification once the device finished booting, which allowed them to run without the user's knowledge. Furthermore, dangerous permissions were requested more by the malicious applications than by the benign applications, as shown in Fig. 9a. This was because this type of permission provides more sensitive information. For instance, the READ_PHONE_STATE permission is requested when the developers use the TelephonyManager API. This API provides the requested application with several items of sensitive information, such as the user's phone number, the International Mobile Equipment Identity (IMEI), and the International Mobile Subscriber Identity (IMSI) in order to track its users. This permission was requested 2835 times by the malicious applications as compared to 915 times by the benign applications. The SEND_SMS and RECEIVE_SMS permissions were requested by 52% and 45% of the malicious applications, respectively, which can be used to benefit financially through premium SMS messages [39]. The same permissions were only requested by 2.9% and 2.2% of the benign applications respectively.

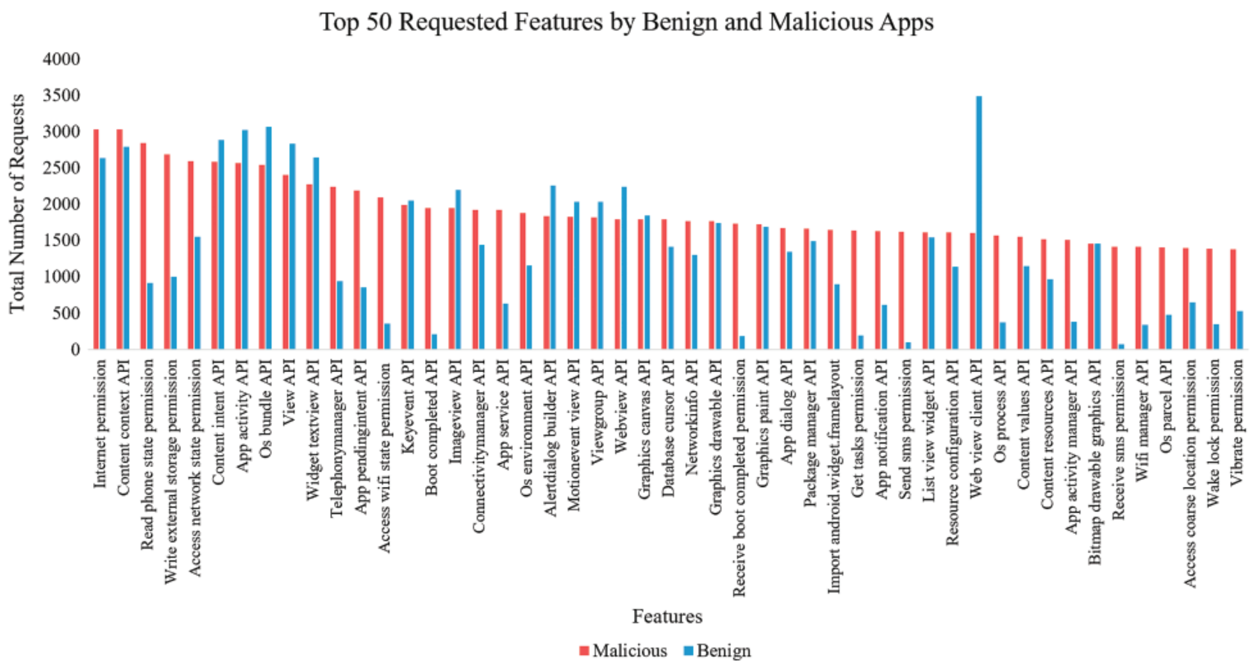
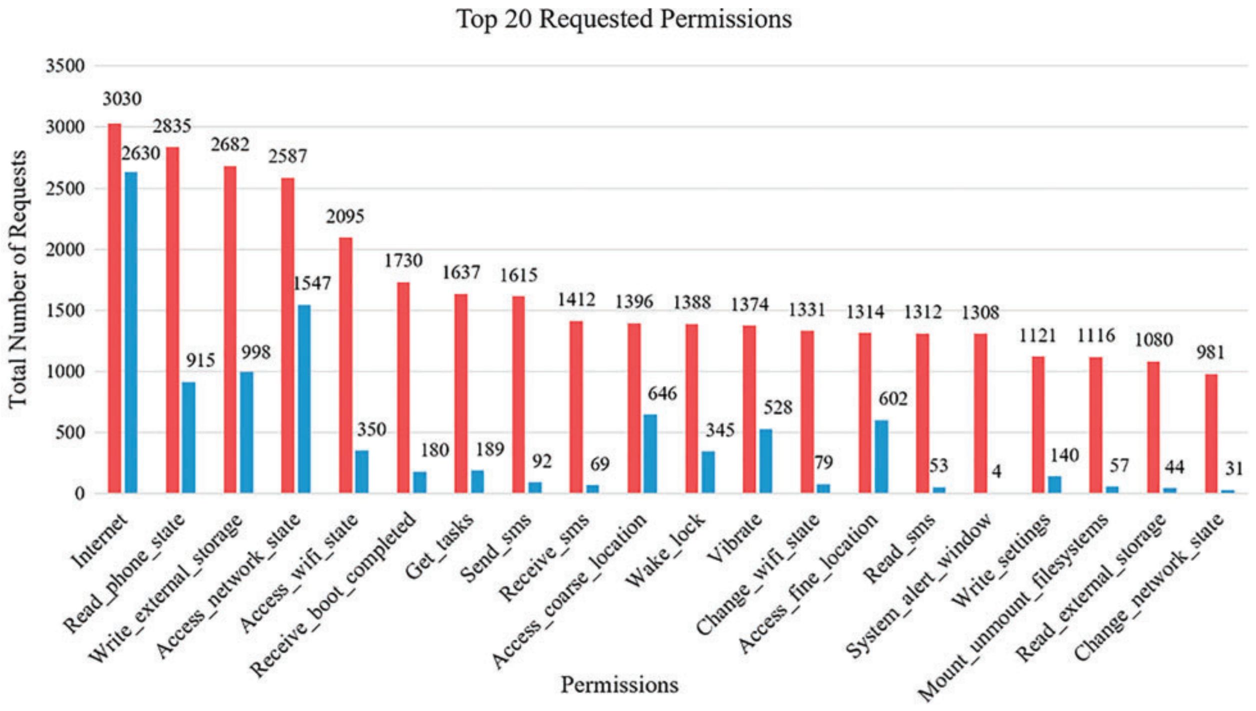
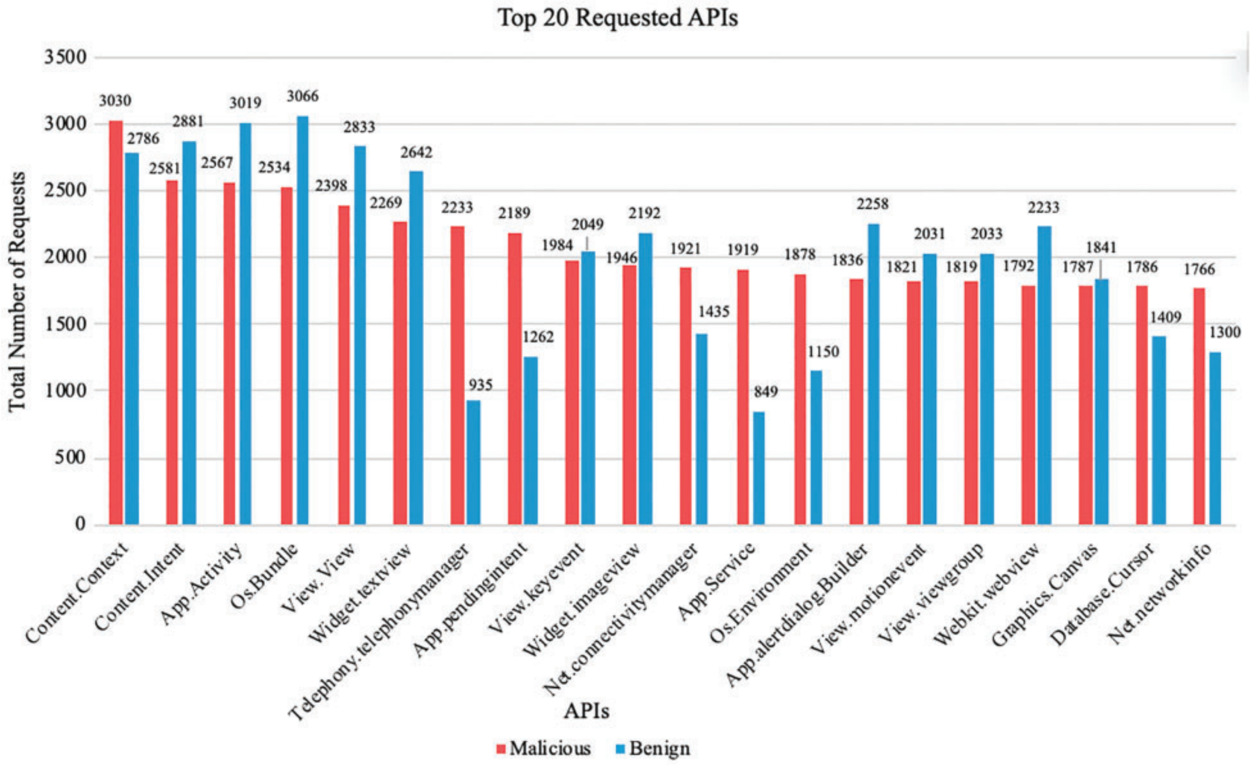


Figure 8: Top 50 requested features by benign and malicious applications

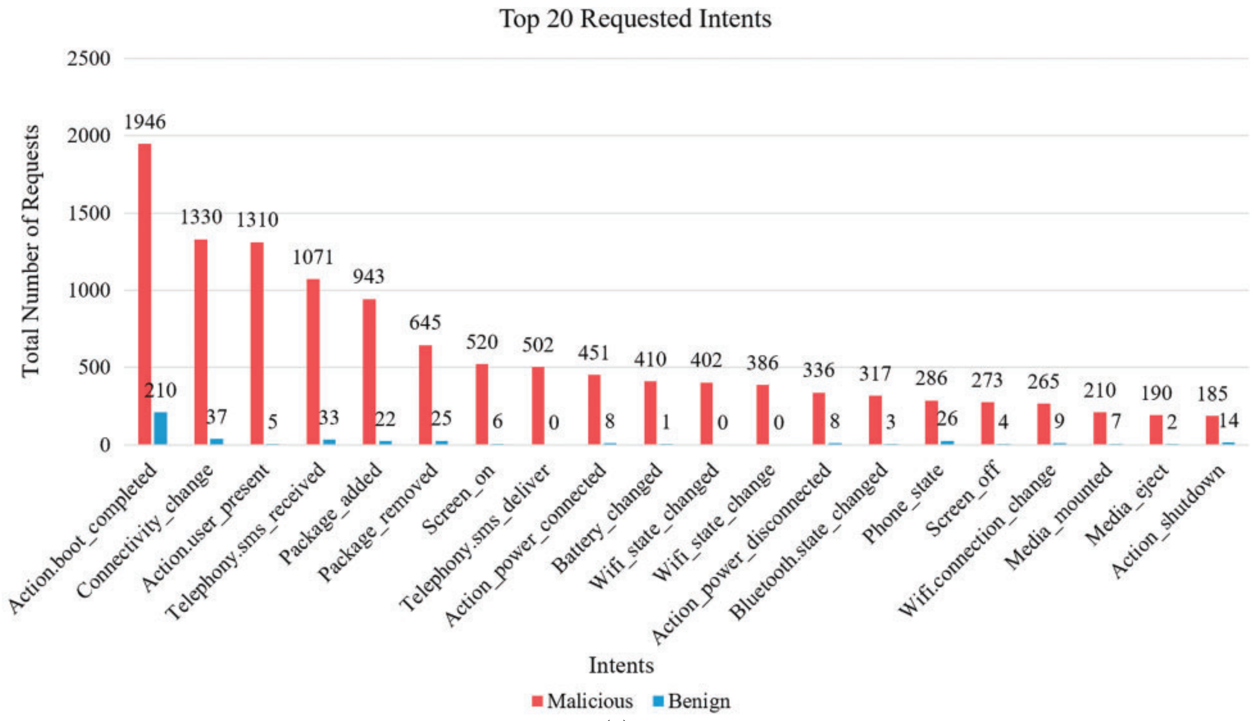
In addition, the malicious applications requested more system permissions than the benign applications did. This type of permission should be granted only to system applications (applications in the Android system image) because it could leak users' sensitive information. For example, as shown in Fig. 9a, the most requested system permission is WRITE_SETTINGS. This allows an application to edit the settings or to read sensitive information such as usernames and passwords [36]. Moreover, one of the most system permissions used by malicious applications is MOUNT_UNMOUNT_FILESYSTEMS. This allows the requested application to mount and unmount file systems for removable storage. As documented in the official Android website, this permission should not be used by third-party applications [36]. Nevertheless, the SYSTEM_-ALERT_WINDOW permission is used by more than 1300 malicious applications, while it is used by only 4 benign applications. This permission can be used to deceive users and steal their private information, while the application is displaying a new window.


Top 20 Requested Permissions
Total Number of Requests
Permissions
Internet
Read_phone_state
Write_external_storage
Access_network_state
Access_wifi_state
Receive_boot_completed
Get_tasks
Send_sms
Receive_sms
Access_coarse_location
Wake_lock
Vibrate
Change_wifi_state
Access_fine_location
Read_sms
System_alert_window
Write_settings
Mount_unmount_filesystems
Read_external_storage
Change_network_state
Malicious
Benign

(a)


Top 20 Requested APIs
Total Number of Requests
APIs
Content.Context
Content.Intent
App.Activity
Os.Bundle
View.View
Widget.textview
Telephony.telephonymanager
App.pendingintent
View.keyevent
Widget.imageview
Net.connectivitymanager
App.Service
Os.Environment
App.alertdialog.Builder
View.motionevent
View.viewgroup
Webkit.webview
Graphics.Canvas
Database.Cursor
Net.networkinfo
Malicious
Benign

(b)



(c)

Figure 9: (a) Top requested permissions, (b) Top requested APIs, and (c) Top requested intents and receivers

As discussed in Section 5.1, all the APIs of both benign and malicious applications were analyzed. These APIs were programs that were already coded for specific functionality. As shown in Fig. 9b, the most commonly used APIs by the benign and the malicious applications were as follows: os. Bundle Activity, content.Context, and content.Intent. However, telephony.TelephonyManager was called by approximately 91% of the malicious applications, while it was called by 30% of the benign applications. This API provides a considerable amount of sensitive information, such as subscriber ID, notifications when the telephony state changes and changes to the voicemail sitting. Moreover, 61% of the malicious applications that called the android.app.Service API, which allows the requesting application to perform a very long operation in the background without the user's knowledge. ConnectivityManager is one of the APIs that can leak the users' sensitive information; it was used by 61% of the malicious applications and 46% of the benign applications. This API is used to monitor the network connection, can send broadcast intents, and allows the requested applications to query the coarse-grained or fine-grained state of the available networks.

Next, the intents and receivers used by the benign and malicious applications were analyzed. As shown in Fig. 9c, BOOT_COMPLETED was the most requested intent by the malicious applications. This intent was requested by almost 62% of the malicious applications, while it was requested by only 6% by the benign applications. Moreover, SMS_RECEIVED was requested more by malicious applications than by benign applications. This intent allows an application to receive an SMS message and listen to all the incoming SMS messages. A comparison of the most requested intents and receivers by malicious and benign applications is shown in Fig. 9c. As shown in Fig. 9c, malicious applications request more intents in general. This allows them to start malicious activities on the basis of certain select events.

5.3.2 Machine Learning Results

Machine learning (ML) is a technique that takes large sets of data and attempts to predict a value for a new sample after discovering the patterns in the previous data. Droid-IoT utilizes the following ML algorithms: K-Nearest Neighbors (K-NN), Logistic Regression (LR), Random Forest, Support Vector Machine (SVM), and Extreme Gradient Boosting (XGBoost) to classify malicious applications from benign applications. All the ML algorithms were trained on the top 100, 150, 200, 250, and 300 features selected by the F-test and chi-square feature selection algorithms from the extracted features. Therefore, after training and testing all of algorithms, the algorithm that provides the best results is saved in a pickled format to be used to classify new samples. All machine learning models were implemented and tested by using Scikit-learn library in python 3.5 [40] with 64 bits. Tab. 3 shows the best parameters for each classifier to obtain the best evaluation results.

Table 3: Parameters tuning using Scikit-learn library

Classifier	Best parameters
K-NN	Metric params = None, number of jobs = 8, number of neighbors = 5, p = 2, weights = uniform.
LR	Intercept scaling = 1, max iter = 100
Random forest	Number of estimator = 8, max depth = None, max depth = None, max features = auto, max leaf nodes = None
SVM	Kernel = rbf, Regularization parameter = 1.0, Tolerance = 0.001, max of iter = −1
XGBoost	Colsample bytree = 1, learning rate = 0.1, max depth = 3, number of estimators = 100, number of jobs = 1

As shown in Tab. 4, the best accuracy was 97.74% achieved by the XGBoost algorithm when it was trained on the top features selected by the chi-square algorithm. Moreover, the type I error rate for XGBoost was 2.5% and the type II error rate was 2.01%. Fig. 10a shows a comparison of the XGBoost algorithm results based on the different sets of features selected by the chi-square and F-test algorithms. The time and space complexity for XGBoost are $O(pn_{tree})$ *and* (pn_{tree}), respectively where p is the number of features and n_{tree} is the number of trees. The lowest accuracy rate for Droid-IoT was achieved by the SVM algorithm using the top features selected by chi-square algorithm, as shown in Fig. 10b. It yielded an accuracy of 96.35% with 1.94% and 5.18% for the type I error rate and the type II error rate, respectively. The time complexity for SVM is $O(n_{sv}p)$ and the space complexity is $O(n_{sv})$, where n_{sv}.is the number of support vectors. The other ML algorithms produced average results. For example, Random Forest (RF) produced an accuracy of 96.67% and a type I error rate of 3.75% with a type II error rate of 3.75% and it has time and space complexity of $O(pn_{tree})$ *and* (pn_{tree}). Moreover, logistic regression (LR) achieved an accuracy of 96.72%, while it achieved 2.67% and 3.84% for the type I and type II error rates, respectively. The time and space complexity for LR is $O(p)$. K-Nearest Neighbors

(KNN) yielded an accuracy of 97.21% with a type I error rate and a type II error rate of 1.58% and 3.90%, respectively. The time and space complexity for KNN are $O(knp)$ *and* (np), respectively where n is the number of data point and k is the number of nearest neighbors Figs. 10c–10e show the top selected features by the chi-square and F-test algorithms for RF, LR, and KNN. Fig. 11a shows a comparison of all the algorithm results, including accuracy, F1-score, and area under a curve (AUC). Furthermore, the receiver operating characteristic (ROC) for all the machine learning algorithms applied by Droid-IoT is shown in Fig. 11b.

Table 4: Comparison of all machine learning results

Metrics	Algorithms				
	K-NN	Random forest	LR	SVM	XGBoost
True positives	936	924	926	933	927
True negatives	873	875	874	860	892
False positives	14	26	24	17	23
False negatives	38	36	37	51	19
Sensitivity (%)	96.10	96.25	96.16	94.82	97.99
Specificity (%)	98.42	97.11	97.33	98.08	97.49
Precision (%)	98.53	97.26	97.47	98.21	97.58
Type I error (%)	1.58	2.89	2.67	1.94	2.51
Type II error (%)	3.90	3.75	3.84	5.18	2.01
Accuracy (%)	97.21	96.67	96.72	96.35	97.74
F1-score (%)	97.30	96.75	96.81	96.48	97.78
AUC (%)	97.17	96.65	96.70	96.30	97.74

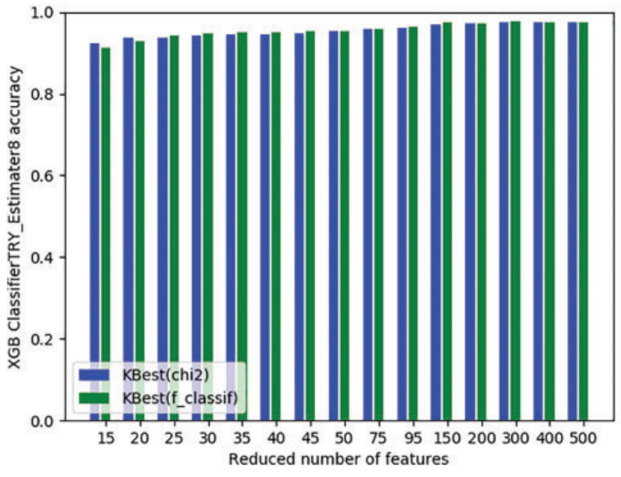
5.3.3 Runtime Analysis

The runtime analysis of Droid-IoT was sub-linear, which was $O\left(\sqrt{n}\right)$, where n is the size of the application. Therefore, the analysis time increased with an increase in the number of files in the application. Fig. 12 shows the runtime analysis results of the 50 applications analyzed by Droid-IoT, where each column in the figure shows the number of files in the analyzed application.

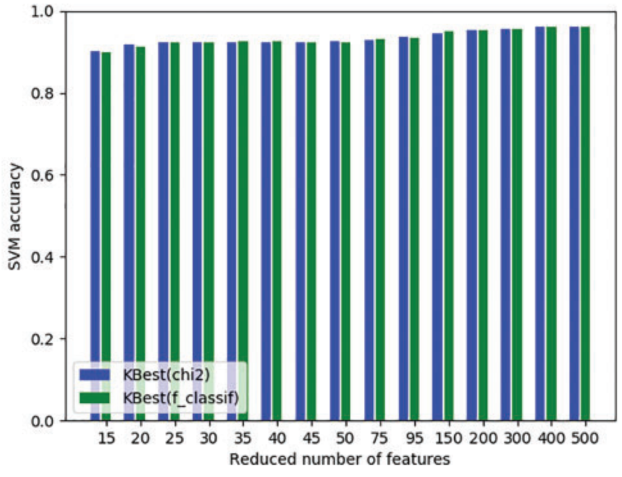
5.3.4 Comparison with the Existing Works

Droid-IoT was compared against two state-of-the-art Android malware detection tools called DREBIN [41] and CSBD [42]. Both of these tools were re-implemented, as the detection accuracy of these tools depends on the features that they use. The following is a brief introduction of the two tools.

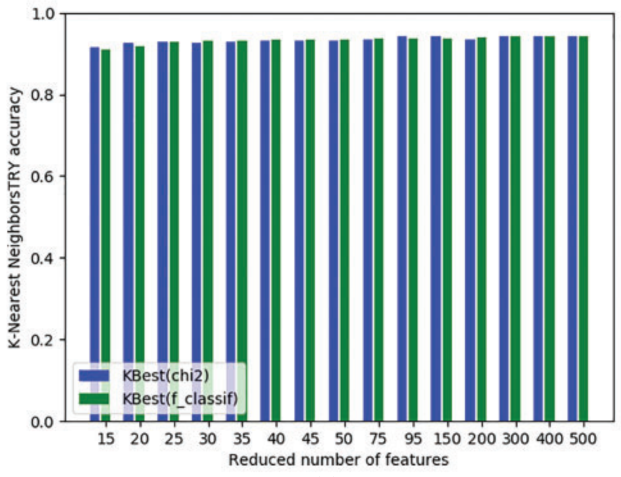
DREBIN [41] is a detection tool that collects different features such as requested permissions, application components, filtered intents, restricted APIs, and network addresses by utilizing the static analysis technique. These features are extracted from a set of benign and malicious applications. It then applies the SVM algorithm to distinguish malicious applications. As a result, this tool achieved a detection rate of 94%.



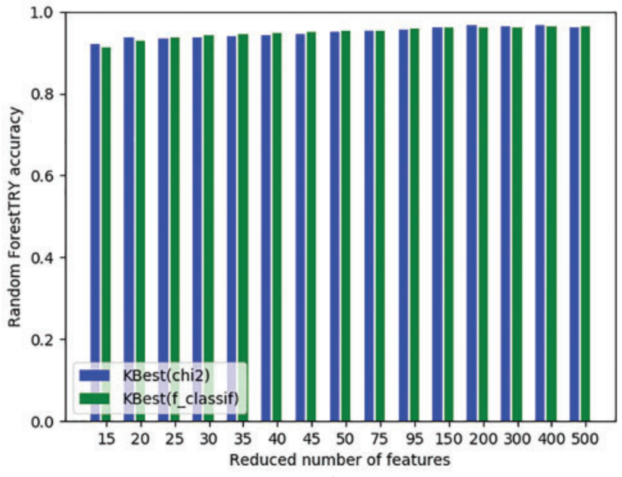
(a)



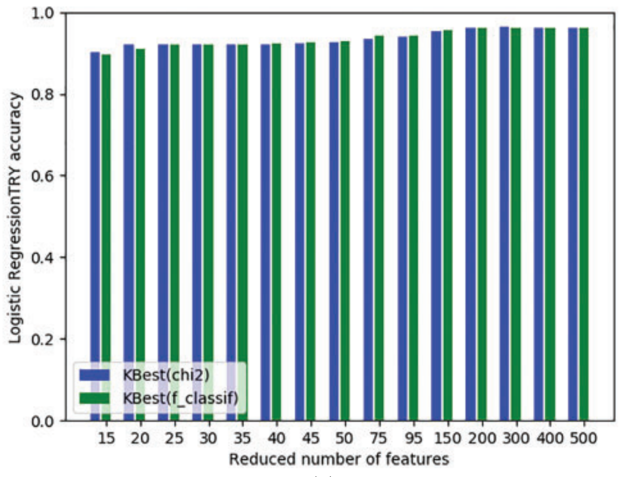
(b)



(c)

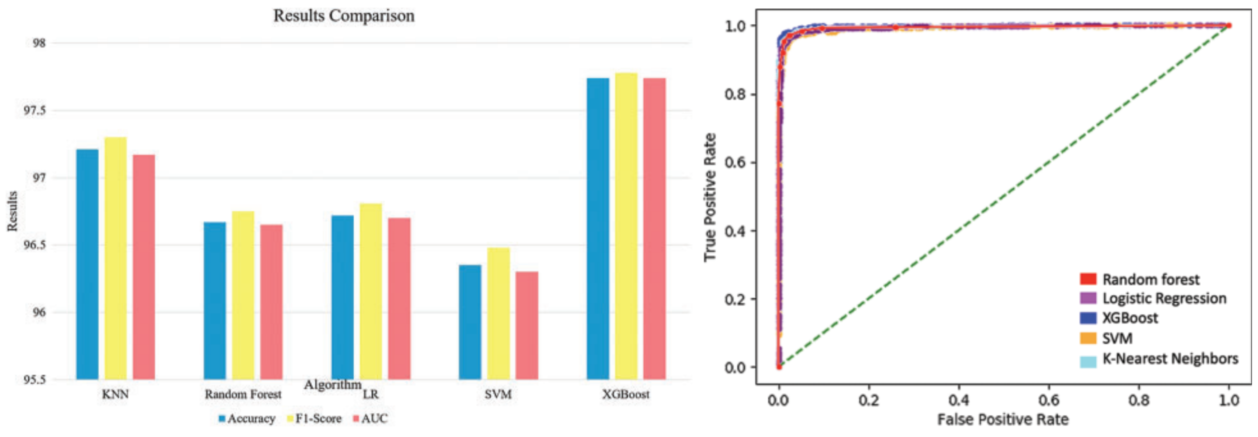


(d)



(e)

Figure 10: (a) XGBoost's results, (b) SVM's results, (c) KNN's results, (d) Random Forest's results, and (c) LR's results



(a) (b)

Figure 11: (a) Comparison of all ML algorithms' results, and (b) ROC for all the applied algorithms

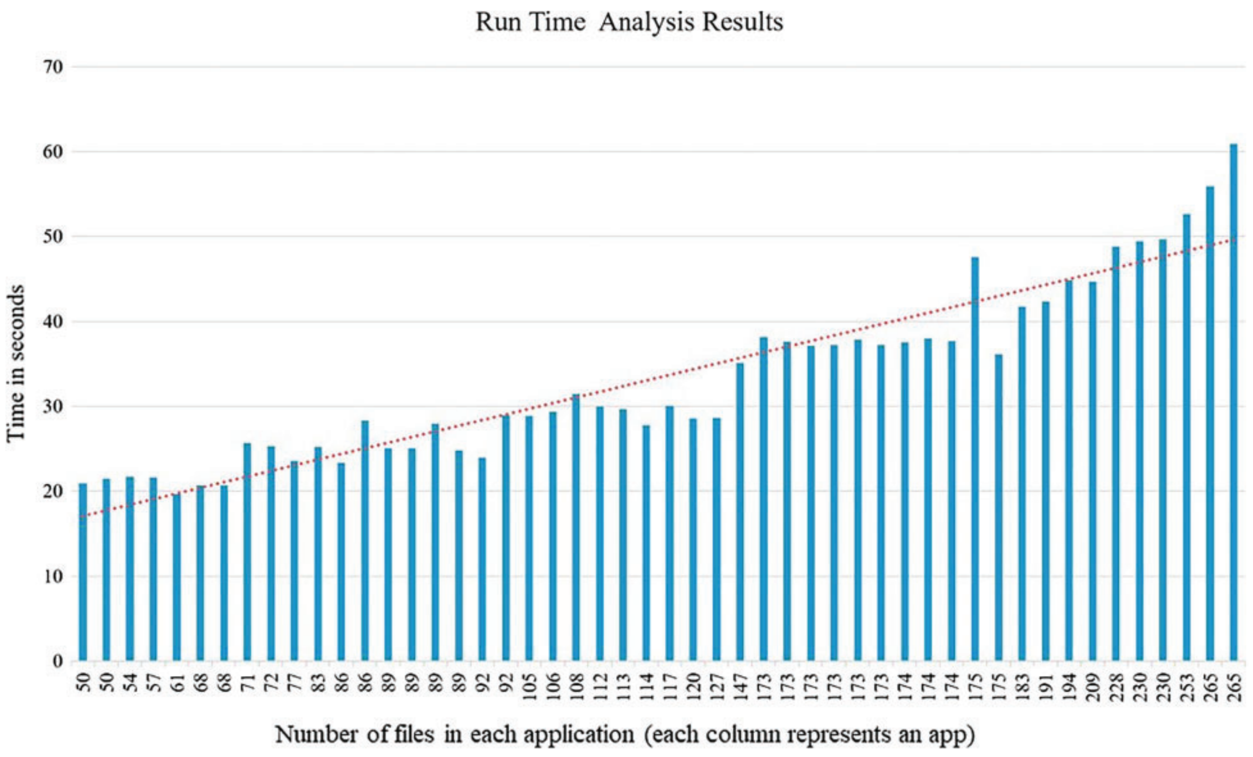


Figure 12: Droid-IoT runtime analysis results

CSBD [42] is another detection tool based on a control flow graph. In this tool, the Android application is analyzed statically in order to extract a control flow graph of the program's representation. Four classification algorithms were tested to evaluate the performance of this method.

The algorithms were as follows: C4.5, SVM, Random Forest, and Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithms.

Droid-IoT was compared to DREBIN [41] and CSBD [42] by using the same datasets, which contained 12 applications, including six benign applications and six malicious applications. As shown in Tab. 5, Droid-IoT produced the best accuracy rate, while Drebin and CSBD yielded the same results, as both used the Random Forest algorithm with a different set of features.

Table 5: Comparison between Droid-IoT and the existing tools

Metrics	Tool		
	Droid-IoT	DREBIN	CSBD
Sensitivity (%)	100	92.00	92.00
Precision (%)	80.00	93.00	93.00
Accuracy (%)	92.31	91.66	91.66
F1-Score (%)	94.12	91.00	92.00

5.3.5 Performance Metrics

The performance overhead for Droid-IoT was measured using the App Tune-up kit [43]. This is an Android application used for measuring the performance overhead of a particular application. The performance of Droid-IoT was measured multiple times, while the proposed tool, Droid-IoT, analyzed the different new installed applications. All the experiments were conducted on a Nexus device 7 running the latest Android version. As shown in Tab. 6, the average power consumed by Droid-IoT was 3.039 mW, and the average CPU load was 0.6%. Moreover, the network traffic consumed by Droid-IoT depended on the application size, which, on average, was 3.1 MB. Droid-IoT is a lightweight application that consumes only 18 MB of the device's storage.

Table 6: Benchmark results for Droid-IoT

Metrics	Results
Average power consumed (mW)	100
Average CPU load (%)	80.00
Average network traffic (MB)	92.31

6 Conclusion

Android is the most popular smartphone platform, which makes it a primary target of many cybercriminals. This has led many researchers to investigate this area and to detect and mitigate the existing threats. This paper presented Droid-IoT, a collaborative framework that detects Android malicious applications by using blockchain technology. The proposed system consists of four main engines: (i) a collaborative reporting engine, (ii) a static analysis engine, (iii) a detection engine, and (iv) a blockchain engine. Each engine contributes to the detection and minimization of the risk of malicious applications, reporting any malicious activities to the user. The results of the Droid-IoT evaluation showed that it could efficiently detect malicious applications with a detection accuracy of 97.74% with an AUC of 97.74%.

Despite the excellent results achieved by Droid-IoT, it still has some limitations that can be improved upon in the future. First, all of the analyses were performed on the server side, which meant that there had to be an internet connection between the Droid-IoT server and the Droid-IoT client. Second, the static analysis can be improved to detect malicious applications that utilize obfuscation techniques to hide malicious activities.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Chen, Y. Miao and I. Humar, "OPNET iot simulation," *Springer Nature*, vol. 1, pp. 1–673, 2019.

[2] Clement, "Number of apps available in leading app stores as of 1st quarter 2020," 2020. [Online]. Available: http://shorturl.at/kzEPT (Accessed 12 April 2021).

[3] P. Muncaster, "Malicious android apps double in q1 as lockdown users are targeted," 2020. [Online]. Available: http://shorturl.at/joxCZ (Accessed 12 April 2021).

[4] A. Alshehri, H. Alshahrani, A. Alzahrani, R. Alharthi, H. Fu *et al.*, "DOPA: Detecting open ports in android OS," in *IEEE Conf. on Communications and Network Security*, Beijing, China, IEEE, pp. 1–2, 2018.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://git.dhimmel.com/bitcoin-whitepaper/ (Accessed 12 April 2021).

[6] D. Cearley and B. Burke, "Top 10 strategic technology trends for 2019," 2010. [Online]. Available: http://shorturl.at/bekzK (Accessed 12 April 2021).

[7] O. Somarriba and U. Zurutuza, "A collaborative framework for android malware detection using DNS & dynamic analysis," in *IEEE 37th Central America and Panama Convention*, Managua, Nicaragua, IEEE, pp. 1–6, 2017.

[8] M. Faiella, A. L. Marra, F. Martinelli, F. Mercaldo, A.Saracino, *et al.*, "A distributed framework for collaborative and dynamic analysis of android malware," in *25th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*, St. Petersburg, Russia, IEEE, pp. 321–328, 2017.

[9] J. Gu, B. Sun, X. Du, J. Wang, Y. Zhuang *et al.*, "Consortium blockchain-based malware detection in mobile devices," *IEEE Access*, vol. 6, pp. 12118–12128, 2018.

[10] A. Ouaguid, N. Abghour and M. Ouzzif, "A novel security framework for managing android permissions using blockchain technology," *International Journal of Cloud Applications and Computing*, vol. 8, no. 1, pp. 55–79, 2018.

[11] K. Allix, T. F. Bissyandé, J. Klein and Y. L. Traon, "Androzoo: Collecting millions of android apps for the research community," in *13th Working Conf. on Mining Software Repositories*, Austin, USA, IEEE, pp. 468–478, 2016.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang and D. Lie, "PScout: Analyzing the android permission specification," in *ACM Conf. on Computer and Communications Security*, North Carolina, USA, ACM, pp. 217–228, 2012.

[13] R. Winsniewski, "Apktool," 2017. [Online]. Available: https://github.com/iBotPeaches/Apktool (Accessed 13 April 2021).

[14] P. J. Taylor, T. Dargahi, A. Dehghantanha, R. M. Parizi and K. K. R. Choo, "A systematic literature review of blockchain cyber security," *Digital Communications and Networks*, vol. 6, no. 2, pp. 147–156, 2019.

[15] R. D. Best, "Bitcoin (BTC) market capitalization as of January 17, 2021," 2021. [Online]. Available: https://github.com/iBotPeaches/Apktool (Accessed 13 April 2021).

[16] T. Salman, M. Zolanvari, A. Erbad, R. Jain and M. Samaka, "Security services using blockchains: A state of the art survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 858–880, 2018.

[17] B. Bhushan, A. Khamparia, K. M. Sagayam, S. K. Sharma, M. A. Ahad *et al.*, "Blockchain for smart cities: A review of architectures, integration trends and future research directions," *Sustainable Cities and Society*, vol. 61, pp. 1–27, 2020.

[18] V. Gramoli, "On the danger of private blockchains," 2016. [Online]. Available: https://allquantor.at/blockchainbib/pdf/gramoli2016danger.pdf (Accessed 13 April 2021).

[19] A. R. Khettry, K. R. Patil and A. C. Basavaraju, "A detailed review on blockchain and its applications," *SN Computer Science*, vol. 2, no. 1, pp. 1–9, 2021.

[20] L. S. Sankar, M. Sindhu and M. Sethumadhavan, "Survey of consensus protocols on blockchain applications," in *4th Int. Conf. on Advanced Computing and Communication Systems*, Coimbatore, India, IEEE, pp. 1–5, 2017.

[21] M. Salimitari and M. Chatterjee, "An overview of blockchain and consensus protocols for IoT networks," 2018. [Online]. Available: https://deepai.org/publication/an-overview-of-blockchain-and-consensus-protocols-for-iot-networks (Accessed 13 April 2021).

[22] A. Aurris, "G data mobile malware report 2019: New high for malicious android apps," 2020. [Online]. Available: https://www.shorturl.at/bdnMU (Accessed 13 April 2021).

[23] D. C. Cruze, "Warning for android users: New malware may attack banking applications," 2020. [Online]. Available: https://www.shorturl.at/rDHL4 (Accessed 13 April 2021).

[24] D. Frank, L. Rochberger, Y. Rimmer and A. Dahan, "Eventbot: A new mobile banking trojan is born," 2020. [Online]. Available: https://www.shorturl.at/dlmuR (Accessed 13 April 2021).

[25] T. Bläsing, L. Batyuk, A. D. Schmidt, S. A. Camtepe and S. Albayrak, "An android application sandbox system for suspicious software detection," in *5th Int. Conf. on Malicious and Unwanted Software*, Nancy, France, IEEE, pp. 15–62, 2010.

[26] H. J. Zhu, Z. H. You, Z. X. Zhu, W. L. Shi, X. Chen *et al.*, "Droiddet: Effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, 2018.

[27] C. Wang, Q. Xu, X. Lin and S. Liu, "Research on data mining of permissions mode for android malware detection," *Cluster Computing*, vol. 22, no. 6, pp. 13337–13350, 2019.

[28] X. Jiang, B. Mao, J. Guan and X. Huan, "Android malware detection using fine-grained features," *Scientific Programming*, vol. 2020, pp. 1–13, 2020.

[29] A. Arora, S. K. Peddoju and M. Conti, "Permpair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2019.

[30] D. Li, Z. Wang and Y. Xue, "Fine-grained android malware detection based on deep learning," in *IEEE Conf. on Communications and Network Security*, Beijing, China, IEEE, pp. 1–2, 2018.

[31] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen *et al.*, "Tuning deep learning performance for android malware detection," in *Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Busan, South Korea, IEEE, pp. 140–145, 2018.

[32] A. Naway and Y. Li, "Using deep neural network for android malware detection," *International Journal of Advanced Studies in Computer Science and Engineering*, vol. 7, no. 2, pp. 9–18, 2018.

[33] T. Ahmad and M. N. Aziz, "Data preprocessing and feature selection for machine learning intrusion detection systems," *ICIC Express Letter*, vol. 13, no. 2, pp. 93–101, 2019.

[34] S. Asaithambi, "Why, how and when to apply feature selection," 2018. [Online]. Available: https://www.shorturl.at/qzEI6 (Accessed 13 April 2021).

[35] B. Chandra and M. Gupta, "An efficient statistical feature selection approach for classification of gene expression data," *Journal of Biomedical Informatics*, vol. 44, no. 4, pp. 529–535, 2011.

[36] Developers, "Android API reference," 2021. [Online]. Available: https://developer.android.com/reference/ (Accessed 13 April 2021).

[37] M. Gargenta, *Learning Android*. Sebastopol, CA, USA: O'Reilly Media, Inc., ISBN. 978-1-449-39050-1, pp. 1–239, 2011.

[38] J. P. Achara, M. Cunche, V. Roca and A. Francillon, "Short paper: WifiLeaks: Underestimated privacy implications of the access_wifi_state android permission," in *ACM Conf. on Security and Privacy in Wireless & Mobile Networks*, New York, USA, ACM, pp. 231–236, 2014.

[39] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. N. Rotaru *et al.*, "Android permissions: A perspective combining risks and benefits," in *ACM Sym. on Access Control Models and Technologies*, New York, USA, ACM, pp. 13–22, 2012.

[40] Scikit-learn, "Scikit-learn machine learning in python," 2007. [Online]. Available: https://scikit-learn.org/stable/ (Accessed 04 March 2021).

[41] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Sym.*, California, USA, The Internet Society, pp. 1–15, 2014.

[42] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State *et al.*, "Empirical assessment of machine learning-based malware detectors for android," *Empirical Software Engineering*, vol. 21, pp. 183–211, 2016.

[43] Technologies, "App tune-up kit," 2020. [Online]. Available: https://developer.qualcomm.com/forums/software/app-tune-up-kit (Accessed 13 April 2021).