Detection of Anomalous Behavior of Smartphone Devices using Changepoint Analysis and Machine Learning Techniques

RICARDO ALEJANDRO MANZANO SANCHEZ, KSHIRASAGAR NAIK, and ABDURHMAN ALBASIR, University of Waterloo, Canada
MARZIA ZAMAN and NISHITH GOEL, Cistel Technology Inc., Canada

Detecting anomalous behavior on smartphones is challenging since malware evolution. Other methodologies detect malicious behavior by analyzing static features of the application code or dynamic data samples obtained from hardware or software. Static analysis is prone to code's obfuscation while dynamic needs that malicious activities to cease to be dormant in the shortest possible time while data samples are collected. Triggering and capturing malicious behavior in data samples in dynamic analysis is challenging since we need to generate an efficient combination of user’s inputs to trigger these malicious activities. We propose a general model which uses a data collector and analyzer to unveil malicious behavior by analyzing the device’s power consumption since this summarizes the changes in software. The data collector uses an automated tool to generate user inputs. The data analyzer uses changepoint analysis to extract features from power consumption and machine learning techniques to train these features. The data analyzer stage contains two methodologies that extract features using parametric and non-parametric changepoint. Our methodologies are efficient in data collection time than a manual method and the data analyzer provides higher accuracy compared to other techniques, reaching over 94% F1-measure for emulated and real malware.

CCS Concepts: • Security and Privacy → Anomaly detection and malware mitigation; • Computing methodologies → Machine learning; • Mathematics of computing → Probability and statistics;

Additional Key Words and Phrases: Malware detection, non-parametric and parametric changepoint detection, power measurement, time-series, machine learning, Drebin dataset

ACM Reference format:
Ricardo Alejandro Manzano Sanchez, Kshirasagar Naik, Abdurhman Albasir, Marzia Zaman, and Nishith Goel. 2023. Detection of Anomalous Behavior of Smartphone Devices using Changepoint Analysis and Machine Learning Techniques. Digit. Threat.: Res. Pract. 4, 1, Article 2 (March 2023), 28 pages.
https://doi.org/10.1145/3492327

1 INTRODUCTION

Internet of things (IoT) devices has become the most widely used technology around the world in the last decade due to autonomy, easy connectivity, scalability, and heterogeneity. As the number of IoT devices increases, the number of attacks against these systems rises exponentially. Thus, anomaly detection to detect malware is
one of the most studied topics in different applied fields such as autonomous cars [32], wearable devices [34],
and home smart devices [5].
Most IoT devices in 2020 are smartphones, representing about 35% of the total number with 3.5 billion devices.
Hence, the present work is focused on anomaly detection on smartphones due to the presence of malware.
To counter these malicious activities, researchers and companies have developed methodologies analyzing
static, and dynamic characteristics of the apps. Static analysis examines patterns in the source code of each app [24].
Nonetheless, this analysis method is prone to obfuscation using metamorphic or polymorphic methods [41]
and opaque constants [1]. Dynamic analysis tries to identify malware hidden in the code running the app while
data samples are taken from different smartphone’s components such as memory, network traffic, permissions,
and system calls. Researchers using dynamic analysis use the data samples to train and test models to detect
malware on smartphones. In the present work, we focus on dynamic analysis.
Different approaches in the dynamic analysis have been developed. To illustrate, in [39], the authors used a
collaborative strategy to detect malware using the application’s permissions data samples. In reference [31], the
authors use system calls data samples to detect malware. Guo et al. [18] used the network traffic data samples
generated by malicious apps to characterize malware. Some other approaches that are discussed in [17] and [7]
involve the use of CPU and network traffic. Furthermore, in [42] and [6], the authors have focused on analyzing
the power consumption of hardware components as data samples.
In general, dynamic analysis models need data samples to be trained. Therefore, in the present work, we
divided into two main components the process to train a model to recognize malware using dynamic analysis.
We use the data collector to gather data samples and the data analyzer to create an anomalous behavior model
training the data samples taken in the data collector.
The data collector has three main aspects to analyze such as the selection of the data sources to generate data samples,
the resource necessary to measure the data samples, and the automation of this process. First, smartphones
are composed of hardware and software components. These elements can provide us data samples to detect
anomalous behavior. Second, most of the data samples can be measured using an external device (off-device
measurement) or using the device’s resources (on-device measurement) being observed. Third, the automation
of the data collection is fundamental since we need a large dataset to train and test robust models to validate our
anomaly detection methodologies.
The data analyzer is used to analyze whether the data samples are normal or anomalous using machine learning models.
This detection analysis can be done on-device or off-device.
Researchers use data collector and analyzer to propose methodologies that unveil anomalous behavior in
smartphones as follows.
Many authors utilize on-device measurement and on-device detection methodologies. The difference among
researchers’ approaches lies in the selection of the smartphone’s data sources components to generate data samples.
To illustrate, in [39], the authors use permissions of Android apps. In [31], system calls have been used.
Most of the researchers use on-device measurements and off-device detection using network behavior [18],
CPU usage [15], and permissions [2].
Finally, some researchers approach anomaly detection taking into account off-device measurement and off-
device detection using network packets [38].
As per [6, 7, 18, 31, 39, 42], the data samples used to detect anomalous behavior on smartphones is collected
on-device. Thus, it can use a lot of memory in data collection. In addition, there is a possibility that smart malware
may not expose itself as it can detect that a program is collecting measurements to unveil it. Also, analyzing
the network traffic, system calls, application permissions can unveil the user’s information, hence violating user
privacy. Therefore, in the present work, an external device measures and collects the instantaneous power consumption to avoid affecting the device’s memory. Additionally, an external device (off-device detection) analyzes
the signal due to a large amount of data to process. We use an anomaly detection technique applied to time-series
data collected to detect anomalous behavior on smartphones due to the presence of malware. The data analyzer can be extended to recognize anomalous behavior in other wireless devices.

Cybercriminals can easily modify the data samples of an individual smartphone’s data source to act as a benign pattern to avoid being recognized by traditional approaches. On the other hand, if data samples of more than one smartphone’s data are used, cybercriminals cannot manipulate these, but more data samples depending on the sources can exploit more resources in terms of memory, processing, and energy to collect in wireless devices. Furthermore, the methodologies spend more time training more data samples. For these reasons, in the present work, only the instantaneous power consumption has been chosen to reveal anomalous behavior because it summarizes the overall consumption of power of all hardware components controlled by apps. It is difficult for cybercriminals to model the power consumption of many hardware components of the smartphone to fake the probability distribution of the power consumption and fool the methodologies. Additionally, we assume that a malicious activity embedded in an app changes more than one power trace of the hardware components at the same time. To illustrate, malware can modify CPU and the wireless module power signatures which will impact the device’s instantaneous power consumption. Many authors have adopted the device’s power consumption as a feature to distinguish malicious behavior obtaining significant results in smartphones [12, 21, 25, 36, 42].

Most of the techniques to detect anomalous behavior using power consumption are data driven. Thus, we need to have a large dataset available to prove that our methodologies can unveil anomalous behavior accurately. Generation of large datasets involves many challenges because each app has a different interface, makes different calls, requires different permissions, and uses distinct hardware components. Thus, the user should execute the correct combination of touches and clicks to trigger an app’s malicious behavior [22]. Otherwise, the malware can be dormant forever, and it can only act at random times to execute malicious behaviors. James et al. [21, 36] have generated datasets repeating sequential actions manually. In addition, the authors used power traces generated when a pre-designed malware is running on the smartphone. We evaluate our anomaly detection methodologies with synthetic and real malware. It is necessary to highlight that we test our methodologies with the same synthetic dataset to have a fair comparison with [21] and [36]. We think that emulated or synthetic malware is important in our evaluation because we want to test how accurate are our methodologies in recognizing malware acting for short periods. The most challenging scenario is when emulated malware acts only during 0.6s.

We automate the power trace data collection for emulated malware using Android Debug Bridge (ADB) commands which can emulate touches and clicks in any app sending console wireless commands from a computer. In addition, we use Droidbot [27] and ADB commands to automate the data collection for real malware. Droidbot uses a depth exploration of real apps to create touches, clicks, and calls automatically at a particular time. We use API integration of all of the components involved in the system to collect the power consumption data automatically.

The advantages of the automation tool developed are the following:

- The automated tool reduces the time to trigger malicious activities in real malware because Droidbot uses a deep exploration of the app. Thus, the time to detect malware decreases.
- The tool helps to save time and reduces cost by generating the dataset without human intervention.
- The tool introduces randomness to the power data collection 152 because Droidbot can be configured with a random seed to change the touches and clicks to control the app every time it runs. Thus, we can create a more general model with the dataset generated.
- The application of the tool can be extended to any App because it explores many possible combinations of clicks of each app irrespective of its interface. Thus, we can generate a power consumption dataset of thousands of apps which is non-existent in the literature.

There is one drawback when we use the automation tool. This tool affects the power consumption measurement because sending and receiving the commands to/from the smartphone affects the power trace. We do not remove from the power trace this affectation because it is difficult to identify it in the power signal.
Generally when researchers use a dynamic approach to create samples, they do not remove the signatures due to the instrumentation of the app as in reference [9].

We test two methodologies in the data analyzer using the datasets generated by the data collector. The difference between both methodologies lies in the way to extract features from time-series signals. Both methodologies use changepoint detection because the time-series signals behave stochastically. The first methodology uses parametric changepoint detection and the other one uses non-parametric changepoint. Changepoint detection is considered an anomaly detection technique in time-series signals. The validity of this theory has been shown in other areas of research, guaranteeing its accuracy to detect anomalous behavior [11, 16, 19, 33]. However, these approaches use online changepoint detection in which they analyze the signal on the flight to identify anomalous behavior. In contrast, our approach uses changepoint analysis to extract features from time-series signals. The changepoint theory identifies collective anomalies in the signal represented by plausible changes in it [13].

A changepoint is a data point such that the preceding data points follow a distribution different than the data points after the changepoint. The parametric changepoint detection technique groups data points into intervals maximizing the difference of probability distributions among adjacent intervals. Figure 6 shows how the parametric changepoint approach groups data points into four intervals. Non-parametric changepoint assigns scores to the value of sudden changes in a stochastic time-series signal using divergence-based dissimilarity and relative Pearson direct density-ratio theories. The plausibility of a change is measured without knowing the probability distributions of consecutive segments of data points. We only know the ratio among the probability distribution of two segments. Figure 9 shows how non-parametric approach assigns a value for each data point. Figure 9(b) shows the scores of non-parametric changepoint.

Three classifiers, namely Support Vector Machine Linear (SVM), Logistic Regression (LR), and Naive Bayes (NB), train the features extracted with changepoint theories to determine if the device exhibits anomalous behavior. These classifiers are deterministic, therefore, every time the model runs with the same input, it will return the same output. In addition, SVM and LR have been selected for being linear since we want to obtain a better generalization of the models avoiding overfitting. To evaluate each classifier, we use Leave One Out Cross Validation (LOOCV) because this cross-validation technique evaluates all data samples providing precise results of F1-measure for each classifier.

The results obtained using both methodologies have been compared with the methodologies in [21, 36], and [42] in terms of F1-measure concluding that our methodologies perform better than those methodologies with an average of ~95% when we evaluate emulated malware. In addition, we analyze the detection time necessary to unveil malicious behavior for the proposed methodologies. Finally, we evaluate the sustainability of our methodologies. Sustainability is the capacity of a model to be robust against the evolution of new kinds of attacks. If we do not need to retrain a model when we have new data, it means that the model is sustainable. Despite the ease and lower cost of data collection and storage in recent years, it is not possible to collect data to cover the entire mobile app space. However, it is expected that using the available data, a machine model can be trained to detect anomaly with reasonably good accuracy even if there is new, unseen data. In our work, we used five real malware apps and their benign versions to collect power consumption data and performed experiments to evaluate the sustainability of our methodology. We included sustainability evaluation in the experiments, results, and conclusion sections.

The present paper is an extension of [29]. In [29], only parametric changepoint detection was used in feature extraction. In addition, only a emulated malware dataset was created. In the present work, we extend this article with the following contributions:

- Provide an end-to-end methodology to automatically collect data and analyze them to detect anomaly in smartphones;
- Design, implement, and validate an automated tool that collects power consumption dataset from real malware, emulated malware reducing cost and time;
• Proposed, developed and validated an additional novel feature engineering methodology which uses non-parametric changepoint detection to extract meaningful features from stochastic power consumption time series signals. Compared the results of this methodology with the proposed in [21, 36], and [42].
• Proposed and trained three machine learning classifier models and used a trustworthy cross-validation technique to validate the models in terms of F1-measure and accuracy.

The rest of this article is organized as follows: Section 2 summarizes previously reported work. Section 3 explains the two proposed methodologies. Section 4 presents the experiments and Section 5 provides the obtained results. Section 6 describes the conclusions and future work.

2 RELATED WORK
We separate this section into two parts, considering the data collector and the data analyzer. In the beginning, we show different research papers that tackle automated data collection. Second, we describe many research papers that propose methodologies to detect anomalous behavior on smartphones considering three main factors: the selection of the data sources to generate data samples, where the data samples are taken, and where the decision process is executed.

To build robust models, authors need a lot of samples to train and test these models. Thus, in the first part of this section, we report research papers that generate a lot of samples using automated tools to control an app as a real user while different data samples such as the battery, CPU usage, permissions, screen images, network traffic, and others are collected.

Alam and Vuong [2] used the battery, CPU usage, and permissions as parameters to recognize apps with anomalous behavior. A mobile device measured and collected these parameters while a Monkey tool emulated the user’s input. Formally, a Monkey tool is a program that performs random actions in a smartphone such as touches, clicks, calling system-level events, WiFi on-off, Bluetooth on-off, and others. They further employed a Random Forest (RF) classifier to train and test the parameters. The authors evaluated 1,330 malicious apps and 407 benign apps.

Haipeng [9] used a Sensitive Access Distribution (SAD) approach to extract features from an app. SAD considers distributions invocations of sensitive data sources and sinks, and sensitive control flows at the method level. The author tackles an important concept related to sustainability of the model. Sustainability is related to recognize new kinds of malware without retraining the model. The author extracted 52 features which are sufficient to achieve the reusability and stability of the model. These 52 features were extracted while a Monkey tool was running on the environment. The authors ran the Monkey tool for 5 minutes to extract 52 features. These features are trained with RF. The author compared its approach with a similar work that tackles sustainability MAMADROID [30]. The author highlighted that his approach overpasses MAMADROID in terms of sustainability because it keeps a better accuracy in analyzing some datasets in different years of evolution. In addition, the author evaluated how obfuscation affects MAMADROID with Praguard dataset. The drawback of this methodology is that the instrumentation to control the app leads to changes in the signature and the overall behavior of the app. In addition, this approach cannot recognize some attacks such as command and control server attacks which can be executed after 5 minutes.

Amos et al. [4] proposed a technique that analyses apps in cloud infrastructure using a 34-node cluster and one mobile device to detect malicious behavior. Essentially, this platform installs the App in each virtual or physical node. Each node opens the app and runs a Monkey tool, which mimics users’ inputs to control the app. While the Monkey tool executes random actions, the battery, binder, memory, network, and permissions are collected every 5 seconds. These parameters are collected with 432 benign and 1,353 malicious apps. Random Forest (RF), Naive Bayes (NB), Multilayer Perceptron (MLP), and Logistic Regression (LR) train and test these parameters. The authors conclude that RF performs better than others with a detection accuracy of \(\sim 94.53\%\).
In reference, Fasano et al. [17] extracted 38 parameters from a smartphone including CPU usage, Virtual memory, Native memory, Dalvik memory, Cursor memory, Android shared memory, link-layer network traffic, internet layer traffic, and transport layer traffic to detect malware. While the 38 parameters are collected, a Monkey tool is controlling the smartphone. They use 1,000 benign apps from Google Play, and 1,098 malicious apps from Drebin and Heldroid to build the dataset. Using this dataset, they trained and tested supervised learning algorithms such as Decision Trees (DT), Support Vector Machine (SVM), Gaussian models, and k-nearest neighbor classifier.

Finally, in [10], the authors used a Monkey to control the apps in the smartphone while collecting method calls and inter-component communication (ICC) intents. They extracted 122 metrics from the logs to train a Random Forest Classifier. They reached an accuracy of 97% in average to recognize malware.

The advantage in [2, 4, 9, 10], and [17] is that the authors used an automated tool to generate the samples. However, this tool performs random actions in the App without analyzing the app in depth. The Monkey tool has been used in sandbox solutions. Although Monkey is used in many approaches, [27] and [40] demonstrated that a Monkey tool is not as efficient as Droidbot to emulate user’s inputs. The following research papers show how Droidbot generates users’ inputs automatically and wisely.

In 2017, Li et al. [27] used Droidbot and Droidbox to efficiently discriminate malicious Apps from benign apps. Droidbot, a program written in Python, emulates human inputs such as touches, clicks, calls to intents, and other actions in an app. Droidbot is different from a Monkey tool because it uses a depth exploration of the Test app, taking into account static characteristics to emulate user’s inputs triggering malicious activities faster. Droidbot runs on a computer that is connected to a mobile device through a wireless or wired connection using ADB (Android Debug Bridge). Droidbot sends commands to the smartphone app using this connection. In this work, Droidbot controls the app running on the smartphone, whereas Droidbox measures file accesses, network accesses, service start, and data leak. The authors concluded that Droidbot can help to trigger sensitive behaviors of the Test app in a few seconds, which is useful in discovering malware efficiently.

Yerima et al. [40] demonstrate the effectiveness of Droidbot in triggering malicious activities to detect malware in apps. In this work, the authors compare the accuracy of detection using three automated tools to trigger malicious behavior: a Monkey Tool (Random-based), Droidbot (State-based), and Hybrid approach (Random + Droidbot). In their test bench, while Monkey, Droidbot, or Monkey+Droidbot is controlling an App, a server collects information about API calls and intents (parameters). Sequential Minimal Optimization (SMO), NB, Simple logistic (SL), MLP, Partial Decision Trees (PART), RF, and J48 Decision tree are used to train and test the models. The datasets used to verify the accuracy of the methodology are 2,444 malicious apps from the Genome project and 2,444 benign apps from McAfee Labs. The best detection accuracy ∼94.3% is reached when Droidbot and RF are used.

In 2020, Alzaylaee et al. [3] collected API Calls, and intents (parameters) while Droidbot controlled the actions on the smartphone. They trained and tested these parameters with some algorithms such as MLP, SVM, NB, DT, and RF to discriminate between benign and malicious apps. In this research, the authors use 31,125 Android apps, with 11,505 being malicious apps and 19,620 being benign apps.

Table 1 summarizes the tools and methodologies used by some of the recent studies in malware detection. Monkey and Droidbot are the most commonly used automation tools [2–4, 10, 17]. While the tools are powerful in collecting information, the on-device detection of different parameters opens a security risk and also impacts the performance of the mobile device. The security risk arises from the fact that the smart malware can detect the on-device collection and may remain dormant during the data collection period.

After analyzing related work that considers automated data generation, we describe in this part of the section methodologies that create statistical and machine learning models to detect anomalous behavior on smartphones. We consider three factors in this section: the selection of smartphone’s data sources to generate data samples, where the data samples are taken, and where the decision process is executed. We do not explain research papers that use on-device detection since smartphones cannot process a large amount of data to execute feature
Table 1. Automation Tools Used to Create Datasets of Features and Create Models for Detecting Malware in Android Smartphones

| Ref  | Types of data samples collected | Automation tool used | Apps considered | Technique to detect malware | Limitations                                                                 |
|------|---------------------------------|----------------------|-----------------|-----------------------------|-----------------------------------------------------------------------------|
| [17] | CPU usage, Virtual, native, Dalvik, Cursor, Android Shared memory, Network traffic. | Monkey tool          | 1000 benign from Play Store, 1098 from Drebin and HelDroid | DT, SVM, Gaussian, K-nearest neighbor | - Usage of memory and CPU to store measurements  
- Random approach to generate events |
| [9]  | Distributions invocations of sensitive data sources and sinks, and sensitive control flows at method level. | Monkey tool          | 15008 benign, 14451 Malware | RF                          | - Instrumentation to control the app changes the signature.  
Control server attacks not avoided |
| [2]  | Battery, CPU usage, permissions | Monkey tool          | 1130 malicious and 407 benign apps | RF                          | - Usage of memory and CPU to store measurements  
- Random approach to generate events |
| [4]  | Battery, binder, memory, network, permissions | Monkey tool          | 452 benign and 1551 malicious | RF, NB, MLP, LR              | - Usage of memory and CPU to store measurements  
- Random approach to generate events |
| [27] | File accesses, network accesses, service start, and data leak. | Droidbot and Droidbox | No, specify      | Droidbox                    | - Smart malware can detect that features have being collected  
- Usage of memory and CPU to store measurements |
| [40] | API calls and intents           | Monkey-Monkey+Droidbot | 2,444 benign from McAfee labs and 2,444 malicious apps from Genome | SMO, NB, SL, MLP, DT, RF     | - Smart malware can detect that features have being collected  
- Usage of memory and CPU to store measurements |
| [3]  | API calls and intents           | Droidbot             | 11,395 malware apps 19,560 benign apps from McAfee Labs | MLP, SVM, NB, DT, RF         | - Smart malware can detect that features have being collected |
| [10] | Method calls and ICC intents.   | Monkey Tool          | 16,978 malware apps 17,365 benign apps from Androo Zoo, Google Play, Drebin, Virus Share, and Gnome | RF                          | - Random approach to generate events |

The research articles that consider off-device detection and off-device measurement are [21, 25, 36], and [38]. Wei et al. [38] used DNS packet information to analyze the geographical position of malicious servers. They used Droid Box which is a user input generator. While Droid Box is interacting with the app as a user, a packet monitor retrieves IP address, TTL time, and other logs. Finally, a network Spatial Extractor maps IP addresses to geographical positions. A Spatial Service estimator calculates the distance between hosts and service providers. If the distance is minimal, the app is benign. They created a Geographical matrix that contains $m$ rows with Android apps and $n$ columns with geographical and network features using benign and malicious apps. They applied Independent Component Analysis (ICA) to obtain a latent matrix. This matrix identifies malicious and non-malicious apps. The authors evaluated 310 malware, detecting them with 100% accuracy.

Kim et al. [25] pioneered the identification of emulated malware utilizing a power signal measured through an external device. The authors designed and built a device to measure the power consumption. The methodology extracts features with a moving average filter and data compression. Furthermore, the dataset trains and tests 11 emulated malicious apps and 8 benign apps. They can recognize almost all the emulated malware reaching 100% of detection accuracy. Nonetheless, the lowest accuracy reported is 80%, in detecting CommW malware.

In 2017, Robin et al. [21] were able to identify emulated malware using power consumption taken by an external device and ICA. They extracted two independent components from reference signals (non-malicious...
| Ref. | Detect encrypted malware | Privacy violation | Usage of memory to collect features | Evaluation dataset |
|------|--------------------------|-------------------|------------------------------------|--------------------|
| [38] | No                       | Yes               | No                                 | RM                 |
| [25] | Yes                      | No                | No                                 | EM                 |
| [21] | Yes                      | No                | No                                 | EM                 |
| [36] | Yes                      | No                | No                                 | EM                 |
| [15] | No                       | Yes               | Yes                                | EM                 |
| [42] | Yes                      | No                | Yes                                | EM                 |
| [12] | Yes                      | No                | Yes                                | EM                 |
| [6]  | Yes                      | No                | No                                 | RM                 |

NG = Not Given, RM = Real Malware, EM = Emulated malware.

app running in the foreground) and suspicious signals (reference signal + emulated malware running in the background). These two independent components are compared with the input signals using correlation. Finally, a threshold of the correlation value between malicious and benign signals is calculated. As a complementary study, the authors extracted the mean and variance of each signal as features to train and test classifiers using SVM. The results presented show a maximum accuracy of 91%.

In 2018, Robin et al. [36] modified the proposed methodology in [21]. The authors added to the methodology four probability distributions (tanh, pow3, gauss, and skew) in the Fast ICA. Using all of the combinations between ICA approaches and distributions, the methodology found a correlation between the independent components and the input signals. The results of the correlation feed three classifiers: NB, SVM, and Random Forest (RF). The authors demonstrate that their approach can recognize emulated malware reaching an average accuracy of 85%.

Some authors have used an on-device measurement to collect datasets. For example, Ali et al. [15] used CPU usage, memory, and the number of SYN packets of Internet Control Message Protocol (ICMP). All the data gathered was sent to a server in the cloud to analyze it using a Gaussian mixture model. They evaluate their methodology using normal apps and three emulated malware developed in reference [8].

Zeffere re et al. [42], in 2014, used power data collected with the Power Tutor tool to detect SMS emulated spyware. Power Tutor is an app that allows obtaining power consumption measurements for different hardware components such as CPU, GPS, WiFi, and screen for each application. To differentiate SMS spyware, they created a model using Mel frequency cepstral coefficients (MFCC) and Gaussian mixture models (GMM).

In 2016, Cavigliione et al. [12] detected attacks related to covert channels using the power consumption of the processes running on smartphones. This kind of attack occurs when two or more malicious apps exchange information exploiting different permissions assigned to them. The authors collected power measurements of the idle state of the smartphone and different scenarios of covert channel attacks using Power Tutor. Their methodology adopts the power traces to train and test classifiers such as Neural Network (NN) and DT. They achieved an accuracy of above 80% with this methodology.

In 2018, Azmoodeh et al. [6] also used Power Tutor, which measures the power consumption on the device to recognize Ransomware. In this approach, they utilize a CPU’s power consumption to differentiate normal apps (Facebook, Chrome, Youtube, WhatsApp, Skype, Angrybirds, Maps, Music player, Twitter, Instagram, and Guardian) from six types of Ransomware. The methodology splits the signal using a certain window size, and
Fig. 1. Testbench to take power consumption measurements.

Fig. 2. Interface app-emulated malware.

each window is analyzed using Dynamic Time Warping (DTW). Finally, the features extracted are the input of different classifiers such as K-Nearest Neighbor (KNN), NN, RF, and SVM. The accuracy reached by the model is 95.5% using KNN with a window size of 7.5 seconds. This methodology uses real Ransomware, but the authors do not specify in detail what apps were used.

3 METHODOLOGY

This section is divided into two subsections. The first subsection explains the automated tool used in the data collector component to generate large datasets. The second subsection explains two methodologies that use changepoint detection to extract and train features obtaining robust detection models.

3.1 Data Collector

Data collection is important in any anomaly detection methodology because the robustness of the model will depend on the number of measurements available and the quality of the information in the signal. Manual data collection involves the direct interaction of a user with the smartphone and the power data collector. Thus, data collection takes more time and consumes more human and energy resources. James et al. [21, 36] and Manzano et al. [29] use a test bench and a manual procedure to collect power consumption measurements of two kinds of signals: benign and malicious signals which are created with emulated malware. Figure 1 shows the elements and connections used to collect power consumption from the smartphone. The power monitor is connected to a computer using a USB. The computer has a GUI program running to collect the power consumption measurements. At the same time, the power monitor is connected to the smartphone to provide energy and measure the power consumption. Finally, the smartphone is connected to the Internet via a wireless medium.

The present work is focused on the development of test bench automation (Figure 1) that can generate the power traces of emulated and real malware without human interaction.

In this work, we used the term “emulated dataset” because the designed app mimics the behavior of real malware. Generally, when real malware acts, it tries to steal the user’s information. In the beginning, the malware app downloads specialized malicious code on the smartphone. When this code is installed, it sends the specific
user’s information to a malicious server. The app designed by the authors imitates this behavior by downloading and uploading information to/from the Internet.

We used the term “real malware dataset” because the applications contain real malicious code. The Drebin dataset contains 5,560 malicious applications with 179 different families of malware. We automate the data power collection for five applications from Drebin without user interaction.

We divide this section into four subsections for clarity. The first subsection explains the general manual procedure that a user must follow to collect power consumption measurements for the detection of real or emulated malware. The second subsection shows the manual procedure used in [21, 29], and [36] to generate an emulated dataset. The third subsection explains the automation tool developed to generate the emulated dataset without human interaction. The fourth subsection explains the generation of a real malware dataset using the automation tool. It is necessary to mention that we do not have a manual procedure to generate a dataset with real malware.

In general, we name manual procedures to the tasks followed by the authors in [21, 29], and [36] to collect the power consumption data because the user must execute a specific task manually.

3.1.1 Manual Procedure to Generate Emulated or Real Malware. This subsection explains the procedure followed to collect power consumption measurements for real and emulated malware in a general way. The procedure is as follows:

1. The user opens a Power Monitor’s GUI interface on the computer.
2. The user starts to measure the power trace after pressing the start button in the GUI.
3. While the power monitor measures the power trace, the user needs to interact with the malicious app. Thus, the user should press some buttons of the app to trigger malicious behavior.
4. The user stops recording the power trace when he presses the stop button in the Power Monitor’s GUI.

The disadvantages of using a manual procedure to collect power consumption measurements from emulated and real malware apps are as follows:

1. If the user is unable to press the correct combination of buttons in the real app during data collection, then the malware can not be triggered.
2. If the user presses an erroneous button during the collection of emulated malware power trace, then the user has to repeat the entire experiment. This leads to wastage of time and resources.

3.1.2 Manual Generation of Emulated Malware Dataset. This subsection explains the procedure that a user follows to collect benign and malicious power consumption measurements manually as in [21, 29], and [36]. The procedure is as follows:

1. The user opens a GUI on the computer provided by the manufacturer of the Monsoon Power Monitor which stores the power traces.
2. In the GUI, the user configures the time duration of the experiment and the types of measurements taken such as voltage, ampere, or power consumption. In this case, power consumption is chosen.
3. The user pushes a button in the GUI to start the experiment. While the power consumption is taken, the user opens the Youtube app and runs a specific video for 5 minutes. According to references [21] and [36], this time is adequate to detect malware using a power trace.
4. After finishing the measurement, the user manually saves a .csv file that contains the power measurement in a specific folder.

To collect power measurement data due to emulated malicious applications, the user follows a similar procedure. The difference is that the user opens an emulated malware app shown in Figure 2 at the beginning of the experiment. This app emulates malware uploading or downloading files from the Internet with different time.
activeness. Then, the user follows the same procedure as in benign apps from 1 to 4. This procedure is repeated with different percentages of activeness in downloading/uploading tasks.

The disadvantages of emulated malware dataset collection are the following:

1. Data generation is prone to human errors in repetition of the experiment.
2. The user follows the same procedure every time that he runs the experiments because the emulated app is not complex in terms of the number of screens and buttons.

3.1.3 Automated Tool for the Generation of Emulated Malware Dataset. The proposed methodology tackles all the disadvantages exposed before. The data collector automation has been developed in Python, and integrates a power tool monitor API and ADB commands to automate the data collection of power traces.

Emulated malware dataset is created using two kinds of signals: benign and malicious. This dataset is called emulated because we use a pre-designed app that mimics malware’s actions. To automate the collection of benign and malicious signals for the emulated malware dataset, we use the flowchart shown in Figure 3.

The malicious signal is taken while a customized app, which runs in the background of the smartphone, executes actions of downloading and uploading. During a downloading action, the app downloads a video from the Internet. On the other hand, the uploading action consists of the generation of a 20-Mb random image that is uploaded to a specific server on the Internet. The app has a screen, as shown in Figure 2. This screen has five configurable parameters:

- **Duty Cycle.** This parameter denotes the “time of activeness” of the downloading and/or uploading actions in 1 minute of the total duration. To illustrate, if the duty cycle is 1%, the emulated malware will act for 0.6 seconds each minute. If it is 2%, it will act for 1.2 seconds each minute.
- **Percentage of Downloading.** This represents the amount of time within the duty cycle when the app executes the downloading task.
- **Percentage of Uploading.** This represents the uploading stage time within the duty cycle.
- **Additional Fields.** There are two optional parameters to configure. The first describes the URL to download the image. Therefore, the user can download an image, video, or file from another site on the Internet. The second is the public IP of the server used to upload the image.

In addition to the five configurable parameters, the app has an additional button called “Start”. After configuring the parameters described above, the user must press the button to initialize the emulated malware actuation. The app described has been designed to automate the power consumption collection for anomalous signals. The flowchart used to generate power consumption measurements automatically is shown in Figure 3. For this purpose, **Android Debug Bridge (ADB)** commands and **Monsoon API** are integrated into Python. ADB is a console terminal tool used to communicate between a computer or server and an Android smartphone. An automated tool can execute different commands through the console to send simple or complex tasks to the smartphone, such as touches, clicks, lock/unlock, turn on/off the background light, open apps, and others.

The stages of the flow are described below:

1. **Mobile Initialization.** ADB commands turn on the smartphone’s back-light screen and unlock the smartphone.
2. **Configuration of Emulated Malware.** ADB commands are used to fill out the text boxes of the emulated malware app. These fields are the duty cycle, percentage of downloading, and percentage of uploading. Finally, the emulated malware is begun by pushing the start button in the app.
3. **Power Tool Monitor Initialization.** Python initializes an API connection with the Monsoon Power Monitor to start the power consumption sampling.
4. **Launch Youtube.** ADB commands open the Youtube app.
Fig. 3. Data collector. Dataset: Emulated malware.  
Fig. 4. Data collector. Dataset: Real malware.

(5) **Launch a Specific Video during a Specific Time.** ADB commands launch a specific video and keep running the video for 5 minutes.

(6) **Power Tool Monitor Termination.** Python stops the power consumption sampling and it stores a .csv with the results of the power consumption trace.

(7) **Mobile Restoration.** ADB commands close the Youtube app, lock the mobile device, and turn off the backlight of the screen.

Python repeats the experiment \(d\) times, keeping the same conditions on the smartphone. In this research, 15 measurements have been taken for each of the following percentages of duty cycles: \(Dut = 1\%\), \(Dut = 2\%\), \(Dut = 3\%\), \(Dut = 4\%\), \(Dut = 8\%\), and \(Dut = 12\%\), with five additional scenarios considering different percentages of downloading and/or uploading as follows:

- **Scenario 1.** The emulated malware downloads a file during 100% of the duty cycle time. For example, if Dut = 1% is set, the emulated malware downloads a file from the Internet for 0.6 second of each minute.

- **Scenario 2.** The emulated malware downloads a file during 25% of the duty cycle time, and the remaining 75%, uploads a file. To illustrate, if Dut = 1% is set, the emulated malware downloads a file for 0.15 second, and the remaining 0.45 second, it uploads an image to a server on the Internet. This process is repeated each minute.

- **Scenario 3.** The emulated malware downloads a file during 50% of the duty cycle time, and for the remaining 50%, it uploads a file. To illustrate, if Dut = 1% is set, the emulated malware downloads a file for 0.3 second, and the remaining 0.3 second, it uploads an image. This process is repeated each minute.

- **Scenario 4.** The emulated malware downloads a file during 75% of the duty cycle time, and for the remaining 25%, it uploads a file. To illustrate, if Dut = 1% is set, the emulated malware downloads a file for 0.45 second, and the remaining 0.15 second, it uploads an image. This process is repeated each minute.

- **Scenario 5.** The emulated malware uploads a file during 100% of the duty cycle time. For example, if Dut = 1% is set, the emulated malware uploads an image to the Internet for 0.6 second of each minute.
To record benign power traces, we use the same flowchart shown for malicious signals in Figure 3, but we do not consider the sub-stage of Configuration of emulated malware (red rectangle).

Python repeats the stages described above in a loop for \( r \) times, depending on how many benign measurements the user wants to generate. In the present work, 15 measurements of benign signals have been taken.

### 3.1.4 Automated Tool for the Generation of Real Malware Dataset.

As we described in related work, research done using power consumption does not evaluate its accuracy in real malware apps because each App is heterogeneous. It means that each real malware app has a different structure in which the number of screens, buttons, and links varies. Therefore, it is difficult to design a specific and manual procedure to trigger malicious activities.

The disadvantages of a non-automatic procedure to collect power consumption measurements from real apps are the following:

1. Each app can have thousands of button combinations to trigger malicious activities.
2. It is manually impossible for a user to test most of the combinations.
3. Data generation is prone to human errors in the repetition.
4. A strict procedure does not allow to generate power consumption measurements with most variants; therefore, the models obtained are less general.

The real malware dataset has been created using apps with malware from Drebin repository and apps without malware from Play Store repository. A pair of apps with equivalent characteristics and user interface has been selected from both apps repositories. The flowchart in Figure 4 shows the main stages to automatically collect the power consumption measurements of real malware. We have only one flowchart to create this dataset because only one app is running in the foreground of the smartphone.

The data collector has the following stages: power tool monitor initialization, mobile initialization, Droidbot, mobile restoration, and power tool monitor termination.

1. **Power Tool Monitor Initialization.** In this stage, communication between the power monitor hardware and Python is initialized using an API to take and store the power measurements in a computer.
2. **Mobile Initialization.** After the power consumption acquisition is started, an initial state of the mobile device is executed using ADB commands. In this stage, the back-light screen on the mobile device is turned on. The device is unlocked and the app is installed.
3. **Droidbot.** In this stage, Droidbot is initialized to generate the user’s inputs on the fly. Droidbot has been configured to explore events in a greedy depth strategy of the application.
4. **Mobile Restoration.** In this stage, the application is uninstalled, the applications and web pages that were opened by Droidbot are closed, the mobile device is locked, and the back-light of the screen is turned off. All these tasks are controlled using ADB commands.
5. **Power Tool Monitor Termination.** In this stage, the sampling of the power monitor hardware is stopped through the API controlled by Python, and the results of the power consumption trace are stored in a .csv file.

The data collector stage has been included in a for-loop to automate the power traces collection of different pps in a flexible and customized way.

### 3.2 Data Analyzer

This section presents the main stages and sub-stages of the two methodologies. The difference between them is the sub-stage of changepoint detection in the feature extraction stage. In Methodology 1, features are extracted using parametric changepoint using two statistical assumptions.

In Methodology 2, features are extracted from time-series signals using non-parametric changepoint detection. Unlike in Methodology 1, we do not consider any assumptions.
The stages and sub-stages in Methodologies 1 and 2 are the same. Each methodology has two flowcharts. Flowchart 1 in Figure 5 and Flowchart 3 in Figure 10 are about the training and testing of the models, while Flowchart 2 in Figure 8 and Flowchart 4 in Figure 12 show the flow that a new measurement has to follow to be classified as malicious or non-malicious. The following notations are used throughout this article. Matrices are represented as bold capital letters — e.g., $X$. Vectors are expressed as capital letters in italics — e.g., $X$. Finally, every single data point is represented as a lowercase letter in italics — e.g., $x$.

Every power consumption signal is denoted by a vector $X_i$. Each data point of a vector $X_i$ is denoted as $x_{ij}$. The subscript $i$ represents the identifier of the power trace analyzed, and $j$ indicates each data sample within $X_i$. Therefore, an entire power trace is identified as follows:

$$X_i = (x_{i1}, x_{i2}, x_{i3}, ..., x_{in})$$

The subscript $i$ represents the identifier of the power trace analyzed, and $j$ indicates each data sample within $X_i$.

The matrices $B$ and $M$ are concatenated in a matrix $X$ with dimensions $m \times n$, where $m$ is $(r + d)$. Each row of matrix $X$ is denoted by $X_i$, which can represent either a benign or a malicious signal depending on the value of $i$.

A malicious signal is defined as a power measurement signal taken over a specific time while an app with malicious code is running on a smartphone. As described in Section 1, two kinds of malware will be analyzed: emulated and real. When we are analyzing emulated malware, a malicious signal can be interpreted as a benign app running in the foreground and an emulated malware running in the background. When we are analyzing real malware, a malicious signal is just a real malware app running in the foreground of the smartphone. A malicious signal is denoted by the vector $M_i = (m_{i1}, m_{i2}, m_{i3}, ..., m_{in})$, and this signal is taken $d$ times. Thus, a matrix denoted as $M$ concatenates $d$ malicious signals and has dimensions $(d \times n)$. Each row of matrix $M$ is denoted by a vector $M_i$, $1 \leq i \leq d$.

A malicious signal is defined as a power measurement signal taken over a specific time while an app with malicious code is running on a smartphone. As described in Section 1, two kinds of malware will be analyzed: emulated and real. When we are analyzing emulated malware, a malicious signal can be interpreted as a benign app running in the foreground and an emulated malware running in the background. When we are analyzing real malware, a malicious signal is just a real malware app running in the foreground of the smartphone. A malicious signal is denoted by the vector $M_i = (m_{i1}, m_{i2}, m_{i3}, ..., m_{in})$, and this signal is taken $d$ times. Thus, a matrix denoted as $M$ concatenates $d$ malicious signals and has dimensions $(d \times n)$. Each row of matrix $M$ is denoted by a vector $M_i$, $1 \leq i \leq d$.

The matrices $B$ and $M$ are concatenated in a matrix $X$ with dimensions $m \times n$, where $m$ is $(r + d)$. Each row of matrix $X$ is denoted by $X_i$, which can represent either a benign or a malicious signal depending on the value of $i$.

In the following sub-sections, the stages and sub-stages of the two methodologies are explained.

**A. Methodology 1**

Methodology 1 considers moving average filtering and parametric changepoint detection to extract meaningful features from a time-series signal. Parametric changepoint detection groups data points with similar statistical characteristics change into intervals of a filtered time-series signal using two assumptions. The first assumption is that each interval follows a Gaussian distribution. The second assumption is that the data points in each
interval are independent of one another. All the data points within an interval follow the same distribution. This technique finds the position of abrupt changes in a signal as illustrated in Figure 6.

Figures 5 and 8 show the flowchart designed for Methodology 1. Figure 5 shows the flowchart to train and test the model. Figure 8 describes the flowchart that a new unlabeled signal has to follow to be classified as benign or malicious.

The first stage in Figure 5, called feature extraction, is composed of two sub-stages called Filtering and Parametric Changepoint. The second stage, called Training and Testing of the model, uses filtered and extracted features from stage 1 to fit three machine learning algorithms, namely, SVM, LG, and NB. The second stage is composed of Labeling, and Training and Testing of the classifier sub-stage. Flowchart 2 in Figure 8, called Detection, does not have the sub-stage of labeling because it determines the label of a new signal as benign or malicious.

**a. Flowchart 1 (Figure 5): Feature Extraction**

Feature extraction is the most important stage in the methodology since it has a decisive impact on the accuracy and F1-measure of the models. In this stage, the original signal has been processed using a moving average filter to eliminate noise, and a parametric changepoint has been used to obtain relevant training features.

1. **Filtering.** This sub-stage eliminates noise in the signal produced by external sources such as electromagnetic, heat, and acquisition sample rate. We use a moving average filter to smooth the signal and reduce the noise. Each power signal $X_i$ is passed through this sub-stage using Equation (1)

$$Y_{ij} = \frac{1}{w} \sum_{g=0}^{w-1} X_{i(w+j-g)}$$  \hspace{1cm} (1)

where:

- $w = \text{Window size.}$ This parameter determines how many values of the signal $X_i$ are averaged. $w$ is selected considering the maximum F1-measure and the lowest normality error.
- $X_i = \text{Input power consumption.}$ This signal is the raw signal power consumption with dimensions $1 \times n$.
- $Y_i = \text{Output filtered signal.}$ This signal is the resultant of Equation (1) with dimensions $1 \times (n/w)$. For simplicity, we assume that $n$ is exactly divided by $w$.
- $g = \text{The current position of signal } X_i \text{ within a window.}$
- $j = \text{A new filtered signal } Y_i, 0 \leq j \leq \frac{n}{w}$

$Y$ is the matrix that concatenates all the filtered signals with dimension $(r + d) \times (n/w)$. The term, *window*, is used to describe a set of consecutive values grouped to filter the signal. The term, *interval*, will be used to denote a set of consecutive data points grouped by the theory of parametric changepoint.
(2) Parametric Changepoint Detection. The theory of parametric changepoint finds sudden changes in a stochastic time-series signal grouping data points to maximize the likelihood in each interval according to a Gaussian distribution [23]. Each data point within an interval is independent of the other. After applying parametric changepoint, we will end up with many intervals. Each interval contains statistical information that characterizes each power trace. Therefore, we can characterize each interval with the mean and the variance. However, if we have thousands of intervals, we will end up with thousands of features. For this reason, we summarize all these metric with the mean. Features 2 and 3 are a summarized version of thousands of features. We perform this summarization because we want to have a robust model but with significant features.

Three features are extracted from the power consumption time series in this sub-stage.

- Feature 1 \((f1)\): Number of changepoints.
- Feature 2 \((f2)\): Mean of the changepoint intervals’ mean.
- Feature 3 \((f3)\): Mean of the changepoint intervals’ variance.

Feature 1 is considered as a favorable feature to detect anomalous behavior since every time a malicious activity occurs in a smartphone, this behavior will change the statistical properties of the signal. These changes will be reflected as changes in features \(f1\), \(f2\), and \(f3\). When feature \(f1\) is found, the position of each changepoint is known. Thus, the algorithm knows the data points assigned to each interval. To characterize each interval statistically, the mean and variance are used. Therefore, features \(f2\) and \(f3\) are important.

Just to understand how data points are grouped into intervals, we assume that we know a priori the number of changepoints in a signal. Figure 7 and the following notations are used to exemplify this process:

- \(K\): This variable represents the total number of intervals into which a signal \(Y_i\) will be divided, as seen in Figure 7. \(k\) is an index to represent each interval separated by a changepoint. \(k\) can take values between 1 to \(K\). \(K - 1\) represents the total number of changepoints.
- \(\tau_k\): This variable represents the position of a changepoint. We consider the first data point \(y_{i_0}\) and the last data point \(y_{i_7}\), as changepoints with positions \(\tau_0\) and \(\tau_K\), respectively. All the positions build a vector denoted as \(\tau = (\tau_0, \tau_1, \ldots, \tau_k, \ldots, \tau_K)\). To illustrate, if a signal is divided only in two segments, \(K = 2\), \(k\) takes values of 1 and 2. Therefore, it exists only one changepoint with position \(\tau_1\). However, in the vector, \(\tau\) is added to \(\tau_0\) and \(\tau_2\) that represents the first and last data point of the signal.
- \(y_{ij}\): This variable corresponds to each element of the vector \(Y_i\) after the sub-stage of filtering.
- \(\text{Interval:}\) After applying the theory of changepoint detection, each interval will contain a set of data points between \((y_{i_{\tau_k-1}}, y_{i_{\tau_k}})\). To illustrate, if we analyze the second interval \(k = 2\) in Figure 7, the interval will include all the data points between \((y_{i_1}, y_{i_2})\). Hence, the data points \(y_{i_1}, y_{i_2}, y_{i_3}, y_{i_4}, y_{i_5}\) and \(y_{i_6}\) will be considered in the interval.

To understand how changepoint detection groups datapoints in intervals, we assume that we know the number of changepoints \(K - 1\) beforehand. To find the optimal position of each changepoint \(\tau_k\), we test all the possible positions for each changepoint such that the function \(J\) is minimized. \(J\) represents the sum of the log-likelihood of all of the intervals in a signal \(Y_i\). The inputs to the function are a vector \(\tau\) which has a tested changepoints’ positions and the signal \(Y_i\).

\[
J(\tau, Y_i) = \frac{1}{n/\omega} \sum_{k=1}^{K} G((y_{i_{\tau_k-1}}, y_{i_{\tau_k}})), \tag{2}
\]

where the function \(G\) represents the log-likelihood of one interval that follows a Gaussian distribution with changes in mean and variance, and it is defined in accordance with Equations (3) and (4) [20, 26].

\[
G((y_{i_{\tau_k-1}}, y_{i_{\tau_k}})) = (y_{i_{\tau_k}} - y_{i_{\tau_k-1}}) \log \left( \frac{\sigma^2}{\hat{\sigma}^2_{(y_{i_{\tau_k-1}}, y_{i_{\tau_k}})}} \right), \tag{3}
\]
Fig. 8. Flowchart 2: Detection procedure for classifying an unlabeled signal as malicious or not malicious.

Fig. 9. Non-Parametric changepoint detection applied to one part of the signal $Y^i$ obtaining the Resultant signal.

$$\hat{\sigma}^2_{(y_{ir_{k-1}},y_{ir_k})} = \frac{1}{(y_{ir_k} - y_{ir_{k-1}})} \sum_{m=(r_{k-1}+1)}^{r_k} (y_{ir_m} - \mu(y_{ir_{k-1}},y_{ir_k}))^2,$$

where $\mu(y_{ir_{k-1}},y_{ir_k})$ is the empirical mean of the datapoints included in the interval $(y_{ir_{k-1}},y_{ir_k})$.

We test all the possible positions for each changepoint and calculate $J$. The optimal positions of the changepoints will minimize the function $J$.

To eliminate the assumption that the number of changepoints is known in advance, we add a penalty value, denoted by $p$, to Equation (2), as described in Equation (5):

$$H(r, Y_i) = J(r, Y_i) + p.$$  

We obtain the number and the positions of changepoints by minimizing Equation (5). After finding the optimal position and the number of changepoints, we find features $f2$ and $f3$ using Equations (6) and (7), respectively.

$$f2 = \frac{1}{K} \sum_{k=1}^{K} \mu(y_{ir_{k-1}},y_{ir_k})$$  

$$f3 = \frac{1}{K} \sum_{k=1}^{K} \sigma^2_{(y_{ir_{k-1}},y_{ir_k})}$$

For each filtered power consumption measurement signal $Y^i$, we extract three features: $r$ benign and $d$ malicious signals pass through the sub-stages filtering and changepoint detection in Figure 5. It results in a matrix, called $C$, with dimensions $(r + d) \times 3$. This matrix has dimension 3 because we extract three features from $Y_i$. Each power consumption measurement is denoted as a feature vector $C_i$. Although we only consider three features extracted from the filtered time-series signal to train and test our machine learning models, the theory of changepoint detection provides a solid statistical analysis that allows us to differentiate changes in the statistical properties of the signal. These features are enough to characterize an interval Gaussian distribution that it is the assumption that we take in this subsection.

**b. Flowchart 1 (Figure 5): Training and Testing of the Model**

This stage is composed of the following sub-stages: Labeling, and Evaluation of the classifier.

(1) Labeling. We label the features extracted from each $Y_i$. Specifically, there are two labels to characterize each feature vector $C_i$. If the signal is benign, it is labeled with 0, and if it is malicious, it is 1. The resultant matrix after labeling is denoted by $L$ with dimensions $(r + d) \times 4$ for Methodology 1. The labeled matrix $L$ will be used as input to three classifiers.
(2) Training and Testing: We use supervised learning to train and test three classifiers: SVM Linear, NB, and LR. These classifiers are deterministic. Thus, every time that a classifier runs, we obtain the same result in the output. Deterministic classifiers were chosen to provide the trustworthy results produced by the methodologies.

To test each classifier, we use the Leave One Out Cross Validation (LOOCV) technique because this cross-validation technique evaluates all data samples, providing realistic and precise results of F1-measure of each model. The classifiers have been evaluated in terms of F1-measure because it takes into consideration precision and recall from a confusion matrix. We select the classifier that provides the best F1-measure and the least normality error.

c. Flowchart 2 (Figure 8): Detection Procedure
The flowchart shown in Figure 8 tests unlabeled new measurements. In this case, we do not know if the measurement indicates anomalous behavior.

The feature extraction stage comprises two sub-stages. The sub-stage of filtering uses the optimal value of $w$ to filter the signal. After this sub-stage, a vector $Y$ is obtained with a dimension $n/w$.

Subsequently, the sub-stage of parametric changepoint detection extracts $f$ features of the input signal using an optimal penalty value.

The next stage is called Evaluation, and it only considers the Evaluation of the features in the optimal classifier chosen in Flowchart 1.

B. Methodology 2. Non-Parametric Changepoint Detection.
This theory is called a non-parametric changepoint because it does not assume any distribution or independence of random variables in each interval. Non-parametric changepoint detection assigns a score to each data point of the signal to rank its importance in the changes of the statistical properties of the signal. If the value assigned is low, the data point does not represent an important change in the statistical properties of the signal. If the value is large, the data point represents an abrupt change in the signal. All the data points of the original signal Figure 9(a) are ranked, as we can see in Figure 9(b). Methodology 2 is illustrated in Figure 10.

a. Flowchart 3 (Figure 10): Feature Extraction.
Feature extraction has two sub-stages: filtering and non-parametric changepoint.

(1) Filtering: This sub-stage filters the raw signal $X_i$ as in Methodology 1.
We use a moving average filter represented by Equation (8):

$$Y'_{i,j} = \frac{1}{w'} \sum_{g=0}^{w'-1} X_{i, w' \cdot j + g},$$  

Fig. 10. Flowchart 3: Data preparation and model training and testing.
where:

- \( w' \): Window size. We test different values of window size, and we choose the value of the window size to obtain the maximum F1-measure and the less detection time. A detection time is the time that Methodology 2 takes to evaluate a new power trace as anomalous behavior or not. Methodology 1 does not consider detection time because it is relatively low.
- \( X_i \): Input power consumption. \((1 \times n)\).
- \( Y_i' \): Output the filtered signal. \((1 \times (n/w'))\).
- \( g \): Current position of signal \( X_i \).
- \( j \): New signal filter signal \( Y_i' \). \( 0 \leq j \leq \frac{n}{w'} \).

All the filtered signals are concatenated in a matrix, called \( Y' \), with dimensions \((r+d) \times (n/w')\). In this section, the term, \textit{window}, has been used to describe a consecutive set of datapoints grouped to filter the signal. The term, \textit{slice (SL)}, denotes a set of consecutive datapoints grouped in a non-parametric changepoint. Finally, the term, \textit{segment (SG)}, corresponds to a set of consecutive slices in the theory of non-parametric changepoint.

(2) \textbf{Non-Parametric Changepoint Detection}. This sub-stage uses a divergence-based dissimilarity approach and relative Pearson direct density-ratio theory to assign values to each data point according to its importance as an abrupt change. After processing each signal with a filtering and non-parametric sub-stage, we obtain a resultant signal, shown in Figure 9(b). The resultant signal contains the scores of abrupt changes in time. We can consider each of this score as a feature to train and test a classifier. However, in this case, the number of features will be of the same dimension of the signal, and it can be very expensive to train a model. Thus, we summarize all these features considering the mean and the variance of the resultant signal scores on Features 2 and 3.

We obtain five features from the resultant signal, as follows:

- Feature 1’ \((f1')\): Cumulative sum of changepoints’ scores
- Feature 2’ \((f2')\): Mean of the changepoints’ scores
- Feature 3’ \((f3')\): Variance of the changepoints’ scores
- Feature 4’ \((f4')\): Mean of the whole signal
- Feature 5’ \((f5')\): Variance of the whole signal

We select Feature 1’ \((f1')\) as relevant because we assume that malware acts using the smartphones’ different hardware resources. This usage is reflected in each score assigned by non-parametric changepoint theory. If we sum up all the scores, the highest cumulative sum means more changes in the statistical properties of the signal. Feature 2’ \((f2')\) and Feature 3’ \((f3')\) are selected because these summarize the main statistical properties of the resultant signal after we have applied non-parametric changepoint detection. Finally, Feature 4’ \((f4')\), and Feature 5’ \((f5')\) are useful because these show the general statistical properties of the whole signal.

To understand how the resultant signal is generated, we review the theory of divergence-based dissimilarity and relative Pearson direct density ratio. Divergence-based theory calculates the dissimilarity between two consecutive segments using the ratio of their unknown probability distributions [28].

The ratio of the probability distributions of two consecutive segments \( f(Y) \) and \( f'(Y) \) is defined by \( \text{ratio} = \frac{f(Y)}{f'(Y)} \). \( f(Y) \) and \( f'(Y) \) are the probability distribution functions of the first and second segments, respectively. Knowing the ratio of the probability distributions of the segments does not imply that we can infer the distribution \( f(Y) \) and \( f'(Y) \).

To explain how the non-parametric changepoint detection technique works, we use the following notations to describe Figure 11:

- \textit{Slice (SL)}: A slice is a consecutive set of datapoints grouped of the signal \( Y_i \). The value of this variable indicates how many points are grouped.
Segment (SG): A segment is a set of slices associated with a group. The values of this variable indicate how many segments are grouped.

\( r \): Is the index of the last element of the signal \( Y_i \) after the filtering sub-stage. Therefore, its value is \( n/w' \), where \( n \) is the total number of samples of the raw signal \( X_i \) and \( w' \) is the value of the window size selected in the sub-stage of filtering.

\( Z_{i,v} \): Is a vector that contains all datapoints grouped by the value of SL. The letter \( i \) is the index that represents the raw signal power consumption. The letter \( v \) is the index to denote each vector and can take values between 0 to \( V = 2 \times r + 4 \times SL + 3 \).

\( Z \): Is a matrix that contains all vectors \( Z_{i,v} \).

To illustrate, Figure 11 shows a time-series signal composed of \( r = n/w' \) elements. If the size of the slice \( SL = 4 \), four datapoints are associated in a group. The consecutive datapoints \( y_{i0} \), \( y_{i1} \), \( y_{i2} \), and \( y_{i3} \) are grouped in a vector denoted as \( Z_{i0} \). The slice advances one position at a time to the right. Therefore, the next slice will be named \( Z_{i1} \), and it contains the elements \( y_{i1} \), \( y_{i2} \), \( y_{i3} \), and \( y_{i4} \). This process is repeated until all of the samples are grouped. All vectors \( Z_{i,v} \) are concatenated in a matrix \( Z \).

A segment is a sequence of slices. In Figure 11, the size of the segments is \( SG = 2 \). Hence, two slices, \( Z_{i0} \) and \( Z_{i1} \) are grouped and denoted by \( S_{i0} \). Next, two slices, \( Z_{i2} \) and \( Z_{i3} \) form segment \( S_{i1} \). The process to obtain the scores of a changepoint is based on finding a dissimilarity of the probability distribution between pairs of segments, using the symmetric divergence-based approach given by Equation (9). This equation shows a general expression to calculate the dissimilarity to any pair of segments. Therefore, it is denoted as \( S_{i,q} \), where \( q \) can take values between 0 and \( Q = \frac{V}{2} \). We round the value of \( Q \).

\[
sc_q = D(S_{i,q}, S_{i,q+1}) = D(P(S_{i,q})||P(S_{i,q+1})) + D(P(S_{i,q+1})||P(S_{i,q})).
\] (9)
In Appendix A, we explain how to calculate a score for two segments $S_i$ and $S_{i+1}$. This process is repeated with all of the segments. The result is a vector denoted as $SC = (s_0, s_1, s_2, s_3, \ldots, s_Q)$ with the scores of changepoints for each data point of the signal $Y_i$. To find features $f_1', f_2', f_3', f_4'$, and $f_5'$, we use Equations (10)–(14).

$$f_1' = \frac{1}{n} \sum_{i=0}^{n/w'} s_i$$

$$f_2' = \frac{1}{n} \sum_{i=0}^{n/w'} s_i$$

$$f_3' = \frac{1}{n} \sum_{i=0}^{n/w'} \frac{(s_i - f_2')^2}{n/w'}$$

$$f_4' = \frac{1}{n} \sum_{i=0}^{n} Y_i$$

$$f_5' = \frac{1}{n} \sum_{i=0}^{n} (Y_i - f_4')^2$$

We select the optimal values for the slice SL, and segment SG testing a range of values of each parameter to obtain the best F1-measure and time of detection for the entire model.

If $r$ benign and $d$ malicious signals pass through the sub-stage filtering and non-parametric changepoint, we obtain a matrix named $C'$, with the dimensions $(r + d) \times 5$, where dimension 5 refers to the five features $f_1'$ through $f_5'$. Each power consumption measurement is denoted by $C'_i$ after this sub-stage.

**b. Flowchart 3 (Figure 10): Training and Testing of the Model**

Methodology 2 has Labeling and Training and Testing of the model stages.

**3.2.1 Labeling.** After extracting features of each power consumption measurement, we label each row of the matrix $C'$ with 0 for benign signals, and 1 for malicious. The resultant matrix after labeling is denoted by $L'$ with dimension $(r + d) \times 6$.

**3.2.2 Training and Testing.** We train and test three classifiers using LOOCV: SVM, NB, and LG.

**c. Flowchart 4 (Figure 12): Detection procedure**

Flowchart 4, shown in Figure 12, determines if an unlabeled power consumption signal is anomalous or benign.

The feature extraction sub-stage filters a signal with window size $w'$ found in Flowchart 3. Next, non-parametric changepoint detection extracts five features from the filtered signals using the optimal values for $\alpha$, SL, and SG to obtain the best F1-measure and the least detection time. The evaluation stage evaluates the features in the model created in Flowchart 3. Finally, the model decides if the signal is benign or malicious.

**4 EXPERIMENTS**

As it was described in Section 3, two datasets have been created to validate that our proposed methodologies are accurate to detect anomalous behavior on smartphones. The emulated malware dataset is composed of two kinds of signals: reference and malicious. Reference signals are power consumption traces taken for 5 minutes while a Youtube app is running a specific video in the foreground, and there is no other app running in the background.
We obtain 15 power measurements repeating the experiment under the same smartphone’s configuration. The smartphone used for all the experiments was a Samsung S5 NEO with the following characteristics. Model: SM-G903W, Android Version: 7.0, Samsung experience version: 8.1, Baseband version: G903WVL1CQI1, Kernel version: 3.10.61-12264375. To keep the same environment every time that we run the experiments, we deactivate the apps that allow us to avoid running in the background. We deactivate vibration, configure the intensity of the screen to the lowest value, and deactivate the sound. The smartphone has a stable wireless connection. Malicious signals are power consumption measurements taken for five consecutive minutes while the Youtube app is running in the foreground, and an emulated malware is running in the background. In the present work, we create an emulated malware dataset using emulated malware in which an app downloads/uploads a file from/to the internet with distinct percentages of activeness named as $D_{ut} = 1\%, 2\%, 3\%, 4\%, 8\%, \text{ and } 12\%$. A percentage represents how many seconds the emulated malware downloads a file in 60s. For example, if $D_{ut} = 1\%$, the malware will act 0.6 on each 60 seconds. In the app, we can configure the uploading and downloading time considering the scenarios described in section 3.1.3.

In total, we collect 15 measurements for benign signals while $(15 \times 6 \text{ duty cycles} \times 5 \text{ scenarios}) = 450$ measurements for malicious signals. Each of the 15 measurements of emulated malware with a percentage of activeness and different scenarios against 15 measurements of benign signals is converted into a classification problem. We collect the power consumption measurements using the automated tool explained in sub-Section 3.1.3.

The real malware dataset has been created using apps with malware from Drebin repository and apps without malware from Play Store repository. Five pairs of apps with equivalent characteristics and user interface have been selected from both apps repositories. Four apps belong to DroidKungFu malware family and one app to Plankton family. We took 15 power consumption measurements with a duration of 5 min for each benign and malicious application while an automation tool (Droidbot + ADB commands) emulates the user’s actions to control the app. Thus, the dataset is composed of 75 malicious and 75 benign power traces. We create a classification problem for each pair of apps.

Sustainability can be evaluated in different ways. To illustrate, Haipeng [9] trained a model with 52 features with data generated in a specific year. The author evaluated their model with the rest of years and thus tested if the model is robust to recognize unseen data through time. Other authors modified some of the labels of the training dataset [14]. The authors tried to test how switching some labels affects the accuracy of the classifier. Other authors use Generative adversarial networks to generate new unseen malicious data samples to test their models [37]. Since we do not have hundreds or thousands of power consumption measurements of apps, it is difficult to evaluate how our methodology is sustainable on a large scale. Thus, we evaluate sustainability training our models with different number of apps and evaluating it with all of them. To illustrate, we train our model with one app and evaluate how can recognize power traces with all of the five apps. Then, we select two apps to train the model and evaluate the model with the all five apps. We follow this procedure considering the five apps with all possible combinations.

5 RESULTS

This section is divided into two parts. The first part shows the efficiency in data collection when the automated tool is used. The second part shows the results in terms of F1-measure when we evaluate emulated and real malware datasets using Methodologies 1 and 2.

To demonstrate the efficiency of the data collector, we compare the time spent by the automated data collector with the time that a human spends collecting the power consumption measurements.

In the emulated malware dataset, we collect 15 power consumption measurements of a benign signal plus $15 \times 6 \text{(duty cycles)} \times 5 \text{(scenarios)} = 450$ power consumption measurements of malicious signals. If each power consumption takes 5 minutes or 300 seconds, in total the automated data collector spends $(15 + 450) \text{(power traces)} \times 300 \text{ seconds} = 139,500 \text{ seconds}$, which represents 38.75 hours. According to [35], if a repeated task is
executed by a human, there is a human error rate depending on the task. We take a value of human error rate of 0.02 which corresponds to the wrong selection of buttons. Thus, if the experiment is done by a human, we expect that the experiment must be repeated $465 \times 0.02 \approx 9$ times. In total, the human must execute the experiment $465 + 9 = 474$ times. It represents $474 \times 300$ seconds = 142,200 seconds or 39.5 hours. Therefore, a human will spend 1 hour more to generate the emulated malware dataset.

In the real malware dataset, we have 5 benign and 5 malicious apps. For each app, 15 power consumption measurements are taken for 5 minutes each. The total time to create a dataset with the automated tool takes $10 \times 15 \times 300$ s = 45000 s. If the experiment is done by a human, the experiment must be repeated at least $10 \times 15 \times 0.02 = 3$ times. Thus, a human will spend 15 minutes or 900 seconds more to execute the experiment. In total, a human should spend 45,900 seconds or 12.75 hours to collect the real malware dataset.

The total time that a human should spend to collect emulated and real malware datasets are 39.5 hours (emulated malware dataset) + 12.75 hours (real malware dataset) = 52.25 hours. It is known that the length of a working day around the world is 8 hours. Therefore, a person could spend 6.53 working days to complete the dataset. On the other hand, the automated tool can collect the datasets in 2.17 days. As we can see in terms of the time taken for data collection, the automated tool can collect datasets three times more efficiently than humans.

Using the emulated and real malware datasets, we train and test two models using the proposed methodology 1 and 2. We obtain decent results in terms of F1-Measure when we train and test our datasets. Also, this section describes the normality error for Methodology 1 and the time of execution for both methodologies. Finally, we compare the results of both methodologies.

In Methodology 1, we use the optimal values for window size = 50 and penalty = 1 because these values provide the best F1-measure and the least normality error. The results of F1-measure applied for Dataset 1 are shown in Figure 13. The lowest F1-measure value is $\sim 78\%$ when the duty cycle is 12\%, and the emulated malware is only uploading the image to a server. The highest average F1-measure is 99.46\% for Scenario 1, in which the emulated malware downloads a file during the entire duty cycle. The worst average F1-measure is 95.01\% when the emulated malware uploads an image 25\% of the duty cycle time and downloads a file 75\% of the same time. The normality error is $\sim 0$ in all the scenarios, and there is no correlation among random variables into an interval. Therefore, we can conclude that the assumptions are accomplished.

In Methodology 2, we tested different values for the window size, $\alpha$, SL, SG, and cross-validation. The optimal parameters selected are $w' = 1000$, $\alpha = 0.4$, $SL = 10$, $SG = 10$, and $cv = 5$ to obtain the best F1-measure and the least detection time.
Table 3. Results Real Malware in Methodologies 1 and 2

| REAL MALWARE         | Tetris (Plankton) | Tilt (DroidKungFu) | Yams (DroidKungFu) | Mineswipper (DroidKungFu) | WordSearch (DroidKungFu) |
|----------------------|-------------------|--------------------|-------------------|---------------------------|--------------------------|
| Parametric changepoint | 66.66%           | 92.85%             | 90.47%            | 92.85%                     | 97.61%                   |
| Non-parametric changepoint | 100%             | 87.80%             | 97.67%            | 95%                       | 100%                     |

Figure 14 shows the results after applying Methodology 2 to Dataset 1. The highest F1-measure value is 100% when the actions of downloading and uploading are equal. The lowest F1-measure value is 96.66% when the emulated malware uploads an image 100% of duty cycle time.

Parametric and non-parametric methodologies perform similarly in terms of average F1-measure in all the scenarios of dataset 1. The average F1-measure in all the scenarios is 97.29% for the parametric and 97.82% for the non-parametric approaches, demonstrating that both are appropriate methodologies for detecting emulated malware.

Neither Methodology 1 nor Methodology 2 performs with high F1-measure in 12% duty cycle. The average F1-measure for the parametric is 92%, and for the non-parametric is 91.3% in this duty cycle.

The results of both methodologies have been compared with the methodologies in [21, 42], and [36]. This analysis only compares the results in which the emulated malware is downloading a file from the Internet with Dut = 1%, 2%, 3%, 4%, 8%, and 12% because these articles do not analyze the five additional scenarios described in Figures 13 and 14.

Figure 15 shows the results for each methodology. The average F1-measure accuracy reached for methodology 1 is ~99.45%, for methodology 2 ~98.55%, for [21] ~91%, for [36] ~85%, and for [42] ~69%. The proposed methodologies surpass all other methodologies in terms of average F1-measures. It is noteworthy that [36] and [21] achieve a better F1-measure than Methodology 1 and 2 when emulated malware has a degree of activeness of 12%. However, the difference is less than 4%. We are interested in a model that can generalize well in all the scenarios. Therefore, we can state that methodologies 1 and 2 surpass the F1-measure results of the methodologies in [21, 36, 42].

Finally, Methodologies 1 and 2 evaluate Dataset 2. The results are shown for each app in Table 3. It is clear that non-parametric changepoint detection, with an average F1-measure of 96.09% for all apps, surpasses the parametric approach by ~8%. Methodology 2 F1-measure surpasses Methodology 1 in both datasets. In addition, we can observe that parametric changepoint detection cannot recognize another kind of malware named Plankton. On the other hand, the non-parametric approach can detect it.

In the emulated malware dataset, non-parametric methodology outperforms by ~0.53% Methodology 1. In the real malware dataset, non-parametric surpasses parametric methodology by ~8%. However, the detection time for the parametric approach is 1 second, while that of non-parametric is 15 seconds. We include the graph of execution time in [29] for parametric changepoint analysis. We do not include the graph for non-parametric since it is 15 times higher than parametric. Since 15 seconds is not considered a lot of time to detect malware, we suggest using non-parametric changepoint detection to detect malware in smartphones. Furthermore, non-parametric changepoint detection does not use any assumptions.

We evaluate sustainability using data from real apps since our interest is to know how our methodology can detect anomalous behavior in real scenarios. As shown in Table 3, the best results were found with non-parametric changepoint detection. Thus, we evaluate sustainability considering the non-parametric changepoint detection as shown in Table 4. While training the model with more apps, the accuracy of the model increases. When we train the model with four apps, we can detect all the apps with an average accuracy of 82%. We plan to extend our work with more training and testing data.
Table 4. Sustainability Evaluation Non-Parametric Methodology

| App or Apps trained                                      | Average Testing 5 Apps | F1-measure       | Accuracy          |
|----------------------------------------------------------|-------------------------|------------------|-------------------|
| Tilt (DroidKungFu)                                       | 0.65386232              | 0.80621616       | 0.81502616        |
| Tilt (DroidKungFu), WordSearch (DroidKungFu)             |                         | 0.81502616       | 0.82050296        |
| Tetris (Pikanton), Tilt (DroidKungFu), Yams(DroidKungFu), WordSearch(DroidKungFu) | 0.75742857              | 0.77142857       | 0.79523808        |

6 CONCLUSION AND FUTURE WORK

In this article, we proposed a generalized methodology to collect power consumption data and analyze them from smartphones to detect anomaly. The methodology was validated for malware detection in Android phones. We developed a data collector to generate power consumption measurements automatically from Android apps and a data analyzer to identify each app as malicious or non-malicious. The automation tool developed integrates API, Droidbot, and ADB commands in Python, reducing the time of collection by three times when we compare with manual collection done by a user. We use a side-channel technique to detect anomalous behavior because it does not interfere with the device’s performance and smart malware cannot detect it. The automation tool reduces time and increases revenue.

In addition, this article presented two feature engineering methodologies that use Parametric and Non-Parametric Changepoint detection as the main algorithms for unsupervised feature extraction from power consumption time-series signals. Feature extraction is complemented using machine learning algorithms to create two methodologies to detect anomalous behavior on smartphones. We evaluate both methodologies with emulated and real malware in smartphones with high F1-measure accuracy. The parametric approach is statistical assumption-based while the non-parametric approach is assumption-less.

We consider that our automation tool is a proof of concept framework to generate a large dataset in the future. Thus, we can evaluate if our models can achieve sustainability testing the evolution of malware.

Both methodologies show satisfactory performance in terms of F1-measures over ~90% to detect emulated malware, although non-parametric methodology shows slightly better results without assumptions but with a greater detection time than a parametric approach.

We evaluate the sustainability of non-parametric changepoint since it provides the best results in terms of F1-measure. We obtained the best accuracy to detect all the five apps when we train the models with four apps. There is a further scope of research and experiments with more training and test data collected from more real benign and malware apps.

The entire framework can be easily used for providing cloud service for malware detection as it has become a more practical solution for applications with a machine learning backend. As part of our future work, we plan to develop a cloud service that includes the data collector and analyzer to analyze a given app.

The present work uses both emulated and real malware data to validate our proposed methodology as opposed to other work [12, 21, 25, 42], which use only emulated malware. Furthermore, our methodologies provide trustworthy results to detect anomalous behavior with an average F1-measure of over 90%.

Finally, in this work, we use real data from a limited number of apps. However, the data collection testbench developed as part of this work will allow us to collect power signatures from hundreds of real apps in an automated fashion which can be used for further improvement and validation of the methodologies. In addition, we can evaluate sustainability extensively. The future scope of the study also includes validation with deep learning and unsupervised algorithms to extract features from time series and obtain robust models for anomaly detection in smartphone, IoT and other similar applications.
A APPENDIX

To calculate a score of two consecutive segments $S_i$ and $S_i$, we use Equation (15). We assume that each segment is composed of the same number of $Z_{i}$ vectors. To illustrate if the value of SG equals 50, it means that $S_i$ has 50 consecutive vectors between $Z_i$ and $Z_{i+1}$ and $S_i$ has 50 consecutive vectors between $Z_{i-1}$ to $Z_{i}$. We create a third segment $S_{aux}$, for the purpose of estimation. $S_{aux}$ has the same number of vectors $Z_i$ as $S_i$ and $S_l$. In this example, $S_{aux}$ has 50 vectors. The vectors $Z_l$ are chosen randomly from $S_i$ and $S_l$. To illustrate, $S_{aux}$ can have $Z_{i-1}$, $Z_{i}$, $Z_{l-1}$, $Z_{l}$, $Z_{l+1}$, $Z_{l+2}$, $Z_{l+3}$, $Z_{l+4}$, and so forth. $S_i$, $S_l$, $S_{aux}$. We use this notation in Equations (25) to calculate the score of two consecutive segments [28].

$$s_{c0} = D(S_i, S_l) = D(P(S_i)||P(S_l)) + D(P(S_{aux})||P(S_{aux}))$$

where

$$D(P(S_i)||P(S_l)) = -\frac{\alpha}{2} * mean(\hat{g}(V_j)^2) + mean(\hat{g}(V_i)) + \frac{1-\alpha}{2} * mean(\hat{g}(V_l)^2)$$

where $\hat{g}(V_j)$ and $\hat{g}(V_i)$ are the estimators for segment $S_i$ and $S_l$, respectively.

According to [28], we can approximate the value of $\hat{g}(V_j)$ and $\hat{g}(V_i)$ with the following equations:

$$\hat{g}(V_j) = \theta^T * K(S_i, S_{aux})$$ (17) $$\hat{g}(V_i) = \theta^T * K(S_l, S_{aux})$$

$$K(S_i, S_{aux}) = \exp\left(-\frac{|S_i - S_{aux}|^2}{2\sigma^2}\right)$$ (19) $$K(S_l, S_{aux}) = \exp\left(-\frac{|S_l - S_{aux}|^2}{2\sigma^2}\right),$$

where

$$\theta := (H + \lambda * I)^{-1} * h,$$ (21) $$h = mean(K(S_i, S_{aux})).$$

$$H = \frac{1-\alpha}{n_i} * K(S_i, S_{aux}) * (K(S_i, S_{aux})^T) + \frac{\alpha}{n_i} * K(S_l, S_{aux}) * (K(S_l, S_{aux})^T).$$ (23)

d and $n_i$ have value 50 because we take 50 segments. $\lambda$ represents the regularization parameter in the optimization problem and $\sigma$ is the kernel width. The following values for $\lambda = med * [0.6, 0.8, 1, 1.2, 1.4]$ and $\sigma = [10^{-3}, 10^{-2}, 10^{-1}, 1, 10^1]$ have been tested using cross-validation to select the optimal values for these parameters. Where med represents the median of the kernel distance between $S_i$ and $S_l$, we use 5-fold cross-validation. Thus, we take 40 elements from $K(S_i, S_{aux})$, denoted as $K_{train}(S_i, S_{aux})$, and 40 elements from $K(S_l, S_{aux})$, denoted as $K_{train}(S_l, S_{aux})$, to obtain $\theta$ with the following equations:

$$\theta := (H + \lambda * I)^{-1} * h,$$ (24)

$$H = \frac{1-\alpha}{n_{train_i}} K_{train}(S_i, S_{aux})(K_{train}(S_i, S_{aux})^T) + \frac{\alpha}{n_{train_i}} K_{train}(S_l, S_{aux})(K_{train}(S_l, S_{aux})^T),$$ (25)

$$h = mean(K_{train}(S_i, S_{aux})).$$

$n_{train_i}$ and $n_{train_l}$ are equal to 40. With the remaining 10 elements from $K(S_i, S_{aux})$ denoted as $K_{test}(S_i, S_{aux})$ and 10 elements from $K(S_l, S_{aux})$ named $K_{test}(S_l, S_{aux})$, we calculate $J$ that represents the squared loss.

$$J = \frac{\alpha}{2} * mean((\theta^T * K_{test}(S_i, S_{aux}))^2) + \frac{1-\alpha}{2} * mean((\theta^T * K_{test}(S_l, S_{aux}))^2) - mean(\theta^T * K_{test}(S_i, S_{aux})) - mean(\theta^T * K_{test}(S_l, S_{aux})))$$ (27)

The optimal combination of $\lambda$ and $\sigma$ will provide the lower error and it is used in the model to calculate $D(P(S_i)||P(S_l))$. To calculate $D(P(S_i)||P(S_l))$, we follow the same procedure. Finally, we sum up both values to obtain $s_{c0}$. 

Digital Threats: Research and Practice, Vol. 4, No. 1, Article 2. Publication date: March 2023.
REFERENCES

[1] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In 23rd Annual Computer Security Applications Conference (ACSAC 2007). 421–430. https://doi.org/10.1109/ACSAC.2007.21

[2] M. S. Alam and S. T. Vuong. 2013. Random forest classification for detecting Android malware. In 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, 663–669.

[3] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. 2020. DL-Droid: Deep learning based Android malware detection using real devices. Computers & Security 89 (2020), 101663.

[4] B. Amos, H. Turner, and J. White. 2013. Applying machine learning classifiers to dynamic Android malware detection at scale. In 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC). 1666–1671. https://doi.org/10.1109/IWCMC.2013.6583806

[5] Eirini Anthi, Lowry Williams, Malgorzata Slowińska, George Theodorakopoulos, and Pete Burnap. 2019. A supervised intrusion detection system for smart home IoT devices. IEEE Internet of Things Journal 6, 5 (2019), 9042–9053.

[6] Amin Azmoodeh, Ali Dehghantanha, Mauro Conti, and Kim-Kwang Raymond Choo. 2018. Detecting crypto-ransomware in IoT networks based on energy consumption footprint. Journal of Ambient Intelligence and Humanized Computing 9, 4 (01 Aug 2018), 1141–1152. https://doi.org/10.1007/s12652-017-0558-5

[7] Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. 2020. DroidLight: Lightweight anomaly-based intrusion detection system for smartphone devices. In 21st International Conference on Distributed Computing and Networking. 1–10.

[8] Iker Burguera, Urko Zurutuza, and Simin Nadjim-Tehrani. 2011. Crowdroid: Behavior-based malware detection system for android. In 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. 15–26.

[9] Haipeng Cai. 2020. Assessing and improving malware detection sustainability through app evolution studies. ACM Transactions on Software Engineering and Methodology (TOSEM) 29, 2 (2020), 1–28.

[10] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2018. Droidcat: Effective Android malware detection and categorization via app-level profiling. IEEE Transactions on Information Forensics and Security 14, 6 (2018), 1455–1470.

[11] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. 2013. Multi-channel change-point malware detection. In 2013 IEEE 7th International Conference on Software Security and Reliability. IEEE, 70–79.

[12] L. Caviglione, M. Gaggero, J. Lalande, W. Mazurczyk, and M. Urbański. 2016. Seeing the unseen: Revealing mobile malware hidden communications via energy consumption and artificial intelligence. IEEE Transactions on Information Forensics and Security 11, 4 (2016), 799–810.

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. ACM Computing Surveys (CSUR) 41, 3 (2009), 1–58.

[14] Corey Dunn, Nour Moustafa, and Benjamin Turnbull. 2020. Robustness evaluations of sustainable machine learning models against data poisoning attacks in the Internet of Things. Sustainability 12, 16 (2020), 6434.

[15] Ali El Attar, Rida Khatoun, and Marc Lemercier. 2014. A Gaussian mixture model for dynamic detection of abnormal behavior in smartphone applications. In 2014 Global Information Infrastructure and Networking Symposium (GIIS). IEEE, 1–6.

[16] Rana Elaggar, Krishnendu Chakrabarty, and Mehdi B. Tahoori. 2019. Hardware trojan detection using changepoint-based anomaly detection techniques. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27, 12 (2019), 2706–2719.

[17] Fausto Fasano, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. 2019. Energy consumption metrics for mobile device dynamic malware detection. Procedia Computer Science 159 (2019), 1045–1052.

[18] Dai-Fei Guo, Ai-Fen Sui, Yi-Jie Shi, Jian-Jun Hu, Guan-Zhou Lin, and Tao Guo. 2014. Behavior classification based self-learning Mobile malware detection. JCP 9, 4 (2014), 851–858.

[19] Zhongyuan Hau and Emil C. Lupu. 2019. Exploiting correlations to detect false data injections in low-density wireless sensor networks. In 5th on Cyber-Physical System Security Workshop. 1–12.

[20] Kayleia Haynes, Idris Eckley, and Paul Fearnhead. 2014. Efficient penalty search for multiple changepoint problems. arXiv preprint arXiv:1412.3617 (2014).

[21] R. Soundar Raja James, Abdurhaman Albasir, Kshirasagar Naik, Mohamed-Yahia Dabbagh, Prajna Dash, M. Zamani, and Nishith Goel. 2017. Detection of anomalous behavior of smartphones using signal processing and machine learning techniques. In 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC). IEEE, 1–7.

[22] Xucuan Jiang and Yajin Zhou. 2012. Dissecting Android malware: Characterization and evolution. In 2012 IEEE Symposium on Security and Privacy. IEEE, 95–109.

[23] Rebecca Killick, Paul Fearnhead, and Idris A. Eckley. 2012. Optimal detection of changepoints with a linear computational cost. J. Amer. Statist. Assoc. 107, 500 (2012), 1590–1598.

[24] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu I. 2019. A multimodal deep learning method for Android malware detection using various features. IEEE Trans. on Info. Forensics and Security 14, 3 (2019).

[25] Halunsang Kim, Joshua Smith, and Kang G. Shin. 2008. Detecting energy-greedy anomalies and mobile malware variants. In Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services. 239–252.
26] Marc Lavielle. 2005. Using penalized contrasts for the change-point problem. *Signal Processing* 85, 8 (2005), 1501–1510.

[27] Yuanchun Li, Ziyue Yang, Tao Guo, and Xiangjun Che. 2017. DroidBot: A lightweight UI-guided test input generator for Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[28] Song Liu, Makoto Yamada, Nigel Collier, and Masashi Sugiyama. 2013. Change-point detection in time-series data by relative density-ratio estimation. *Neural Networks* 43 (2013), 72–83.

[29] Ricardo Manzano, Abdurhman Albasir, Kshirasagar Naik, Jim Kozlowski, and Nishith Goel. 2019. Detection of anomalous behavior in wireless devices using changepoint analysis. In *2019 IEEE International Congress on Internet of Things (ICIoT)*. IEEE, 82–90.

[30] Enrico Mariconti, Lucky Owuwuzike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Madamroid: Detecting Android malware by building Markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[31] Fabio Martinelli, Francesco Mercaaldo, and Andrea Saracino. 2017. Bridesmaid: An hybrid tool for accurate detection of Android malware. In *2017 ACM on Asia Conference on Computer and Communications Security*. 899–901.

[32] Xiuliang Mo, Pengyuan Chen, Jianing Wang, and Chundong Wang. 2019. Anomaly detection of vehicle CAN network based on message content. In *International Conference on Security and Privacy in New Computing Environments*. Springer, 96–104.

[33] Habeeb Olufowobi, Uchenna Ezeobi, Eric Muhati, Gaylon Robinson, Clinton Young, Joseph Zambreno, and Gedare Bloom. 2019. Anomaly detection approach using adaptive cumulative sum algorithm for controller area network. In *Workshop ACM Workshop on Automotive Cybersecurity*. 25–30.

[34] William Saltzstein. 2019. Bluetooth wireless technology cybersecurity and diabetes technology devices. *Journal of Diabetes Science and Technology* (2019), 193296819864416.

[35] David J. Smith. 2017. *Reliability, Maintainability and Risk: Practical Methods for Engineers*. Butterworth-Heinemann.

[36] Robin Joe Prabhabhar Soundar Raja James, Abdurhman Ali Albasir, Kshirasagar Naik, Marzia Zaman, and Nishith Goel. 2018. A power signal based dynamic approach to detecting anomalous behavior in wireless devices. In *16th ACM International Symposium on Mobile Management and Wireless Access*. 9–18.

[37] Tony Thomas, Athira P. Vijayaraghavan, and Sabu Emmanuel. 2020. Adversarial machine learning in cybersecurity. In *Machine Learning Approaches in Cyber Security Analytics*. Springer, 185–200.

[38] T. Wei, C. Mao, A. B. Jeng, H. Lee, H. Wang, and D. Wu. 2012. Android malware detection via a latent network behavior analysis. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. 1251–1258.

[39] L. Yang, V. Ganapathy, and L. Ifode. 2011. Enhancing Mobile malware detection with social collaboration. In *2011 IEEE 3rd International Conference on Privacy, Security, Risk and Trust and 2011 IEEE 3rd International Conference on Social Computing*. 572–576.

[40] Suleiman Y. Yerima, Mohammed K. Alzaylalee, and Sakir Seze. 2019. Machine learning-based dynamic analysis of Android apps with improved code coverage. *EURASIP Journal on Information Security* 2019, 1 (29 Apr 2019), 4. https://doi.org/10.1186/s13635-019-0087-1

[41] I. You and K. Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 297–300. https://doi.org/10.1109/BWCCA.2010.85

[42] Thomas Zefferer, Peter Teull, David Derler, Klaus Potzmader, Alexander Oprisnik, Hubert Gasparitz, and Andrea Höller. 2014. Towards secure mobile computing: Employing power-consumption information to detect malware on mobile devices. *Int. Journal on Advances in Software* 7 (2014).

Received 1 October 2020; revised 27 August 2021; accepted 14 October 2021