The ISC Int'l Journal of Information Security
ISeCure
January 2023, Volume 15, Number 1 (pp. 41–53)
http://www.isecure-journal.org

A Machine Learning Approach for Detecting and Categorizing Sensitive Methods in Android Malware **

Hayyan Salman Hasan [1,2,*], Hasan Muhammad Deeb [3], and Behrouz Tork Ladani [2]

[1] *Albaath University, Faculty of Mechanical and Electrical Engineering, Homs, Syria.*
[2] *University of Isfahan, Faculty of Computer Engineering, MDSE Research Group, Isfahan, Iran.*
[3] *Albaath University, Faculty of Informatics Engineering, Homs, Syria.*

ARTICLE INFO.

Article history:
Received: –
Revised: –
Accepted: –
Published Online: –

Keywords:
Sensitive Methods, Evasion Methods, Payload Methods, Dynamic Analysis, Machine Learning

Type: –

doi: --

dor: --

ABSTRACT

Sensitive methods are those that are commonly used by Android malware to perform malicious behavior. These methods may be either evasion or malicious payload methods. Although there are several approaches to handle these methods for performing effective dynamic malware analysis, generally most of them are based on a manually created list. However, the performance shown by the selected approaches is based on the completeness of the manually created list that is not almost a complete and up-to-date one. Missing some sensitive methods causes to degrade the overall performance and affects the effectiveness of analyzing Android malware. In this paper, we propose a machine learning approach to predict new sensitive methods that might be used in Android malware. We use a manually collected training dataset to train two classifiers: the first one is used to detect the sensitivity nature of the Android methods, and the second one is used to categorize the detected sensitive methods into predefined categories. We applied the proposed approach to a large number of methods extracted from Android API 27. The proposed approach is able to predict hundreds of sensitive methods with the accuracy of 94.4% for the first classifier and 92.8% for the second classifier. To evaluate the proposed approach, we built a new list of the detected sensitive methods and used it in a number of tools to perform dynamic malware analysis. The proposed model found various sensitive methods that were not considered before by any other tools. Hence, the effectiveness of these tools in performing dynamic analysis is increased.

© 2020 ISC. All rights reserved.

1 Introduction

Android OS is a dominant mobile operating system with a market share of 83.8% in 2021 [1]. On the one side, this is a motivation for malware applications to appear and grow. On the other side, there is a significant endeavor to find ways to detect and analyze these malware applications. In general, an analysis process is used to extract the required information from the malware samples. Android malware (and Android applications in general) can be analyzed using three approaches. Static analysis that extracts the information from the sample under analysis with-

* Corresponding author.
**This article is an extended/revised version of an ISCISC'18 paper.
Email addresses: foo@gmail.com, foo@gmail.com, foo@gmail.com
ISSN: 2008-2045 © 2020 ISC. All rights reserved.

out running it [2][3], dynamic analysis that needs to run the sample in order to extract its real behavior [4] [5] [6] [7], and hybrid analysis that includes using both static and dynamic analyses [8] [9].

Dynamic analysis is an approach that runs the sample inside a test environment to extract its real behavior. Hence, Dynamic analysis needs to execute the sensitive methods that are used as malicious payload to extract the real behavior of the sample under analysis. However, there are several evasion techniques used by Android malware to hinder the dynamic analysis process. In general, many approaches have been proposed to detect and defeat these evasions [10] [11] to reach the payload location and execute the sensitive methods that are used in this payload. To this end, these approaches mainly use manually created lists of evasions to detect and hence defeat the evasions. Moreover, these approaches also use a manually created list of sensitive methods that are considered evidence of the malicious payload. As an important consequence, the effectiveness of these approaches is based on the completeness of both evasion and payload method lists. In other words, if the lists are not complete and do not contain every possible sensitive method (evasion or payload method), the dynamic analysis achieved by these approaches will not be effective. Hence, they may not be able to extract the real behavior of the sample under analysis.

This work focuses on sensitive methods in Android malware that are used either as evasion or payload methods. Dynamic analysis approaches always use some fixed lists to handle these sensitive methods in Android malware in order to extract the malicious behavior of the sample under analysis[10] [11] [12] [13]. The number of defined methods in the Android framework is very large, and there are many new added methods in each update of the Android framework. Hence, there is a big chance of deceit in these approaches. That means, if the malware sample uses some methods out of the lists, the analysis process achieved by these approaches can be defeated. Moreover, the large number of defined methods in different Android frameworks makes the manual detection and classification of these sensitive methods infeasible. Finally, the newly defined methods in each update in the Android framework impose a heavy load on the analyzer to manually detect and classify the evasions, which can be an error-prone task.

This paper proposes a machine learning approach for identifying and categorizing various sensitive methods in all Android frameworks. The proposed approach uses the training dataset that was collected in our previous works [6] [7]. This dataset includes two types of methods: normal methods and sensitive methods. The sensitive methods are categorized into 14 categories. These categories are obtained from our analysis that we achieved on Android malware samples. Moreover, these categories are mostly used in many dynamic analysis frameworks of Android malware [12][13]. The proposed approach uses two stages of classification. The first stage uses a Support Vector Machine (SVM) classifier to detect whether or not the method is sensitive. The second stage uses a Gradient Boosting classifier to classify the sensitive methods according to the categories defined in the trained dataset. Our decision to use these two classifiers in the proposed approach is based on the nature of the data we deal with, as well as the effectiveness of these classifiers and their high generalization performance.

The proposed approach can use the models obtained from the training process to detect and classify a relatively large number of previously unknown Android methods. For example, we applied the resulted models to detect and classify 12759 methods from Android API 27. The proposed approach could detect many new methods that are unknown by the currently available dynamic analysis approaches. As a result, the proposed approach can provide the ability to detect the newly used sensitive methods in Android malware and hence provides a comprehensive and more complete list of sensitive methods for the dynamic analysis frameworks to deal with these sensitive methods and extract the real behavior of the sample under analysis.

To evaluate the proposed approach, we conducted a series of experiments on the proposed classifiers to prove their effectiveness. For the training purpose, we used our collected dataset. The results showed that the proposed approach provides 94.4% accuracy and 95.4% precision for the stage-1 classifier. Also, the accuracy is 92.8, and precision is 92.6 for the stage-2 classifier, which means that using the proposed approach to identify sensitive methods can reduce the risk of missing these sensitive methods by the dynamic analysis tools. Consequently, these dynamic analysis tools can detect the usage of these sensitive methods and extract the real behavior of the malware samples. Moreover, to evaluate to which extent the list of sensitive methods produced by the proposed approach (after applying it to the methods from Android API 27) are used by the real-world malware samples, we used this list as an input for Droidmon [14]. Then, we utilized 500 samples that are randomly selected from AMD [15] [16] and Contigue Mobile [17] datasets in the evaluation process. The experimental results showed that the real-world samples use the methods defined in the obtained list from the proposed approach. Finally, we used the obtained list in Ares [12], IntelliDroid [13], and Curious-Monkey

[18]. These three tools are used to perform dynamic analysis of Android malware. We used samples from AMD dataset [15] [16] and ran these tools with and without the obtained list. The experimental results show that using the obtained list from the proposed approach increases the effectiveness of the aforementioned tools in detecting evasions and extracting more malicious behavior from the used samples in the experiments, which improves our hypothesis, that the completeness of the list will affect the effectiveness of these dynamic analysis frameworks. In summary, the main contributions of this paper are as follows.

- A novel dataset of sensitive methods containing 186 methods was obtained from our previous works on Android real-world samples.
- A machine learning based approach to detect and classify various sensitive methods in Android malware even in case of new and previously unseen Android versions and variants. We released the source code of this approach in [19].
- A list of potentially sensitive methods that includes 360 new automatically detected methods obtained by applying the proposed approach to the methods of Android API 27.
- Using the obtained list from the proposed approach by a number of dynamic analysis tools for Android malware to evaluate these tools using real-world malware samples.

The remainder of this paper is as the following. Section 2 discusses the related work to the proposed approach. Section 3 provides a detailed description of the proposed approach and the used features in each classification stage. Section 4 includes the evaluation process to evaluate the proposed approach. Finally, we conclude the paper in Section 5.

2 Related Work

As far as we know, the proposed approach in this paper is the first dedicated approach that focuses on automatically detecting sensitive methods in the Android framework. However, our approach has been inspired by SUSI [20], which is a machine learning approach to detect and classify sources and sinks used by malware to leak information from Android devices. SUSI (like our approach) uses the hand-noted list of sources and sinks to predict more significant numbers of sources and sinks from about 110000 methods in the Android 4.2 framework. Merlin [21] is another approach that uses an incomplete list of sources and sinks and tries to generate a complete one. However, both approaches try to predict the potentially used source and sink methods to leak information from the victim device by the malware, while our approach tries to predict the used sensitive methods, either evasion or payload methods by Android malware. These sensitive methods are essential for performing an effective dynamic analysis of Android malware.

Many approaches have been proposed to handle evasion techniques and provide dynamic analysis of evasive malware. DirectDroid [11] is a tool that uses a manually created list of sensitive methods to provide dynamic analysis of evasive malware. FuzzdDroid [10] is another tool that uses hand noted list of evasion methods. These methods are hooked whenever they are invoked during the execution, and their returned values are set to some other values to bypass them and reach the payload methods. Ares [12] also uses a fixed list to detect the evasions in the malware samples. However, as we mentioned before, these approaches are effective only if their sensitive methods' list is complete. In other words, if the list is not complete, these approaches may not be able to provide an effective dynamic analysis of Android malware. The main difference between these approaches and our approach is that our approach is used to predict a complete list of sensitive methods, while these approaches use the resulted list to handle these sensitive methods.

Different Artificial Intelligence (AI) approaches (particularly machine learning) are used to detect Android malware samples and automatically identify them from various Android markets. These approaches use statically or dynamically extracted features to detect malicious applications and classify them into different categories. For example, CANDYMAN [22] is a tool that classifies Android malware families by combining dynamic analysis traces and Markov chains. NATICUSdroid [23] is another machine learning approach that uses statically selected native and custom Android permissions as features to detect and classify malware applications from benign applications. IntDroid [24] is another approach that uses sensitive methods as features to detect malware applications and distinguish them from benign applications. HinDroid [25] is another approach that uses a structured Heterogeneous Information Network (HIN) to represent the rich relationships between Android applications and related APIs. Furthermore, it uses multi-kernel learning to classify Android applications into malware or benign applications. DroidCat [26] is a novel dynamic Android application classifier that uses dynamic features such as calls and Inter Component Communication (ICC) intents to do the classification. The main difference between these approaches and our approach is that these approaches use the sensitive methods as evidence to detect Android malware, while our approach predicts the new sensitive methods (either evasion or payload methods) from a hand-noted list of sensitive methods. In

other words, the proposed approach does not use sensitive methods in the classification process but in the prediction process.

3 The Proposed Approach

This section explains our proposed approach to automatically predict a new sensitive method that can be used by Android malware. The prediction process is done through learning from a hand noted and a relatively small number of already known sensitive methods. The proposed system addresses two classification problems. The goal of the first classifier is to detect whether the method is considered a sensitive method or not. The second classifier will identify under which category the detected sensitive method falls. Both classifiers are trained over an extended version of the dataset obtained from our previous works [6] [7]. In the following, the architecture of the proposed approach, the features we used in each stage, and the predefined categories are explained.

3.1 The Proposed Approach Architecture

Figure 1 represents the architecture of the proposed approach. As shown in this figure, the proposed approach can take any method that belongs to the Android API method as input and identify whether it is sensitive or not. Then the proposed approach will provide the potential category that this sensitive method belongs. We used the dataset we obtained from our previous work to train our two classifiers in the training stage. Note that any method used as input to the proposed system should pass through various stages. First, the requested attributes will be extracted from the input method (data preparation stage). Then these attributes will be passed to the stage-1 classifier. The stage-1 includes using the SVM classifier to detect if the method is considered a sensitive method or not. Another set of attributes will be extracted from the same input method and passed to the stage-2 classifier. In the stage-2 classifier, all the methods that are classified as sensitive methods will be categorized under a set of predefined categories. A detailed description of each stage in the proposed approach is provided in the following.

(1) **Data preparation:** The first stage of the system is to prepare the raw input data and extract all the features we need to train our classifiers. This stage takes raw Java methods as input and extracts all the features that can be useful to differentiate between different input methods. A specific set of features extracted from the input methods are used to train each of the following classifiers at the training stage. In the testing stage, most of the methods provided by the Android API 27 are extracted and transformed into vectors of features. In a nutshell, the system is trained over our dataset and then evaluated over the extracted attributes from Android API 27.

In detail, our input data is a list of records that represent Java methods. Each record has a set of attributes including the method's name, package's name, class's name, the number of arguments, type of arguments, return type, and requested permissions. Based on the aforementioned attributes, we extracted a new set of attributes that can be more meaningful for our classifiers. As an example, if one of the method arguments was an interface, we specified the value of 'Parameter is an interface' as true. Similarly, if the return type of the method was 'void', we determined whether the value of 'Method is returnable' is true or false.

(2) **Stage-1 classification:** This stage takes the extracted attributes (from the first stage) as input and detects whether the method that these attributes belong to is sensitive or not. In order to do that, the SVM [27] classifier with the Sigmoidal kernel is used. The SVM classifier is very effective with high dimensional input data where the number of dimensions is larger than the number of samples. Furthermore, it can effectively handle non-separable classification problems using the kernel trick. Briefly, SVM tries to find the optimal hyperplane that best separates the classes and leads to low generalization error. We trained the SVM classifier based on the dataset of 372 methods (186 sensitive methods and 186 normal methods). Consequently, the classifier is now ready to predict any new sensitive method.

(3) **Stage-2 classification:** This stage takes the Android API methods classified as sensitive methods as inputs and categorizes them under a set of predefined categories. The defined categories are based on our collected dataset. The used classifier in this stage is the Gradient Boosting [28] classifier. This classifier is one of the most popular classifiers that use ensemble techniques. The learning process of this classifier starts by training a set of weak learners in order to obtain a stronger model. Gradient Boosting has three main components: loss function, weak learner, and additive model. The loss function is used to estimate the learning progress and it varies based on the problem. the weak learner is a poor classifier who can at least provide predictions better than random guessing. The most common weak learners are decision trees. Finally, the additive model is an

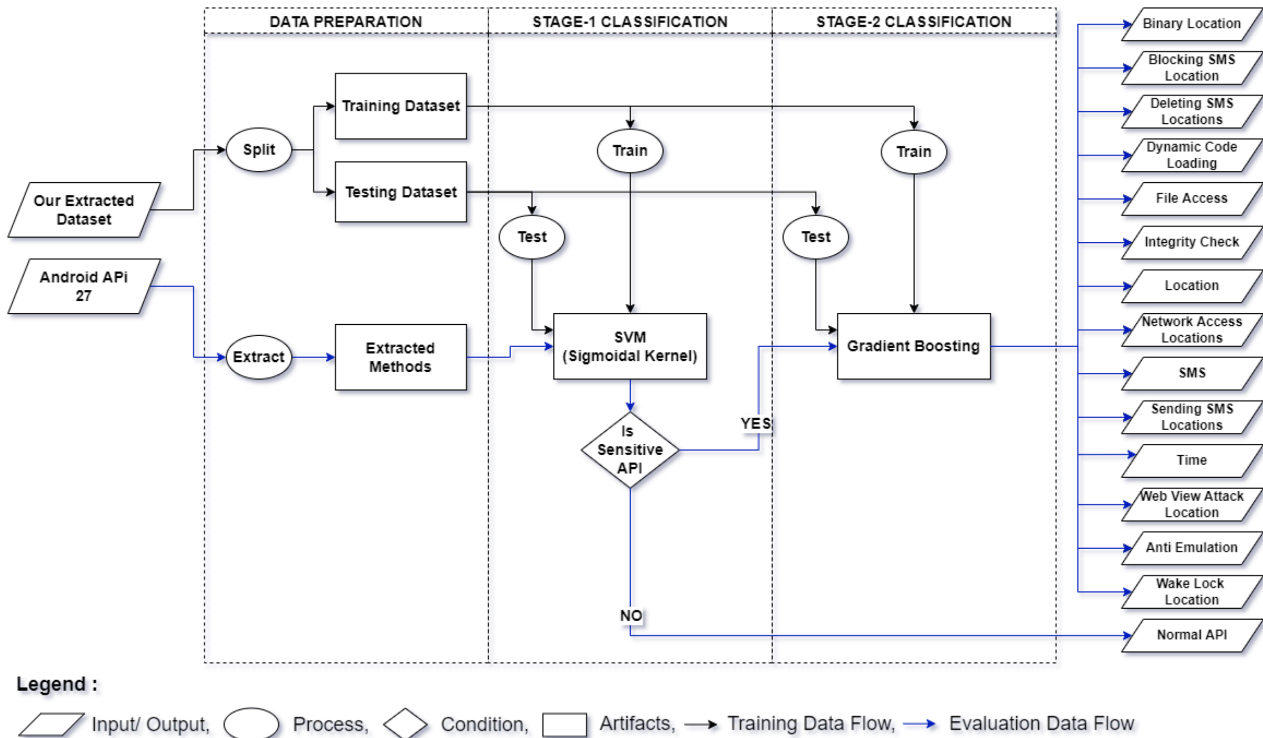

Figure 1. The architecture of the proposed approach

iterative model that tries to construct the final model by sequentially adding the weak learners. As we mentioned before, the Gradient Boosting classifier will be trained over a specific set of attributes extracted from the input methods (different from the attributes used for the stage-1 classifier). At the training stage, the Gradient Boosting classifier will be trained over the whole dataset to be able to discriminate between different categories. At the testing stage, only the methods indicated as sensitive methods by the stage-1 classifier will be passed to the stage-2 classifier.

3.2 The Collected Dataset of Sensitive Methods

The dataset contains manually collected sensitive methods that we obtained from our previous studies [6] [7] (can be accessed via the IEEE data port [29]). This dataset contains a set of evasion methods that are used to hinder the dynamic analysis of Android malware and a set of methods that are used in the malicious payload. An example of the evasion methods that are used to hinder the dynamic analysis is *getDeviceID()* that is used to get the ID of the device where the sample runs. In the case of an emulator, this method returns "000000000000000". Hence, it can be used to detect the existence of the emulator and hinder the execution accordingly. On the other hand, some other methods, such as *openConnection()*, are commonly used in the malicious payload to connect with the C&C server. Therefore, we used these methods (with their features) to predict any new sensitive method from Android API. Moreover, we added the same number of methods (with their features) that are not considered sensitive methods to provide a balanced training dataset and use it to train the proposed approach.

The collected dataset contains 372 methods, 186 sensitive methods, and 186 normal methods extracted from different Android APIs and categorized as sensitive and normal classes. In our dataset, the sensitive methods are categorized into 14 categories. Among the massive number of features extracted from the malicious applications, we are only interested in the features that could help our classifiers to discriminate between different data samples and achieve high detection accuracy. We use the following extracted features to identify the sensitive and normal methods (stage-1 classification process).

- **Package name:** This feature represents the package name for each method. It affects the classifier decision, especially for the popular packages used as sensitive methods.
- **Class modifier:** This feature identifies whether the class is protected or an abstract class. In

general, methods from protected classes are not used as sensitive methods.

- **Parts of method name:** A particular part of the method name is taken and used to identify the method names into six cases: get, set, put, is, request, and other.
- **Method access modifier:** This feature controls the access to the specified method from other classes or its subclasses. It could be public/private/protected. In general, the sensitive methods have public access.
- **Method is returnable:** This feature determines if the method has a return value or not. In general, the sensitive methods return values.
- **Parameter is an interface:** This feature indicates whether the method accepts a parameter of an interface type or not. In general, this kind of method belongs to the not-sensitive category since they do not perform direct operations over the data.
- **Parameter type:** This feature indicates the type of the parameter that the method accepts. The parameters could be of a concrete type or belong to a specific package. For instance, the methods that accept parameters of the package "java.io" are mostly sources of sensitive methods.
- **Request permission or not:** This feature identifies whether the method requires specific permission to be executed. Most sensitive methods request permissions in order to get system services.

On the other hand, the features used to categorize the sensitive methods to their corresponding category (stage-2 classifier) are illustrated in the following.

- **Package name:** This feature represents the package name for each method. It affects the classifier decision, especially for the popular packages used in the sensitive methods.
- **Class name:** The name of the class can play a vital role in categorizing the sensitive methods. For example, methods from *Build* class are categorized as Anti-emulation evasions because they are used to detect the used test environment.
- **Return value type:** This feature identifies the type of the method returned values. For example, the Anti-emulation methods return strings in most cases, while the Location methods return double types in most cases.
- **Parameters number:** This feature represents the number of arguments the method takes as input. For example, the Time methods get 0 or 1 argument, while the File access methods get more than 3 arguments as input.
- **Permission type:** This feature identifies the type of permission the method could request if it exists. Otherwise, it will take the "None" value.

It should be noted that all of the features are categorical features except arguments number, which holds numerical data. This fact encouraged us to choose a tree-based classifier to construct the stage-2 classifier because of its well-known performance for this kind of data.

3.3 The Predefined Categories

We define 14 categories based on the empirical study that we achieved on AMD [15] [16] dataset. These categories include all the methods that are considered sensitive methods (either evasions or payload methods). In our empirical study on AMD dataset we could extract many sensitive methods manually then we categorize these methods according to their usage by the malware into 14 categories. Some of these categories are commonly used by malware and contain many methods like Anti-emulation and network access categories. Other categories are commonly used but contain a few methods like the Integrity check category. Finally, some categories are rarely used and contain a few methods like the Binary category. we describe these categories in the following.

- **File access:** This category includes any method used to read special contents from files or databases.
- **Integrity check:** This category includes any method used to check if the malware code is manipulated.
- **Location:** This category includes any method used to detect the location of the test environment.
- **SMS:** This category includes any method used to read special addresses and contents from incoming messages.
- **Time:** This category includes any method used to delay the execution of the sample.
- **Anti-emulation:** This category includes any method used to detect whether the test environment is an emulator or a real device.
- **Binary:** This category includes any method that is used to kill some running processes in the device, such as anti-malware.
- **Blocking SMS:** This category includes any method that is used to block some received SMSs, such as SMSs from banks.
- **Deleting SMS:** This category includes any method that is used to delete the sent SMSs.
- **Sending SMS :** This category includes any method that is used to send SMSs to some specific numbers without the user's realization.

- **Network access:** This category includes any method that is used to communicate via the internet.
- **Web view attack:** This category includes any method that uses WebView to perform some java script based attacks.
- **Wake lock:** This category includes any method that is used to prevent the device from going to sleep in order to do malicious behavior.
- **Dynamic code loading:** This category includes any method that is used to download the malicious payload after running the sample.

4 Experimental Evaluation

Since the proposed approach is the first dedicated approach to predicate sensitive methods, there is no way to compare it with other works. Hence, we used metrics like accuracy, precision, and recall to evaluate the effectiveness of the proposed approach. Then to evaluate the impact of the resulted list from the proposed approach, we used this resulted list in some well-known dynamic analysis frameworks, and evaluate the effectiveness of these frameworks when using the resulted list. To conclude, we perform a series of experiments to evaluate the effectiveness of the proposed approach and to evaluate the impact of the obtained list from this approach. First, we used our collected dataset which contains 372 methods divided into sensitive and normal methods for the training purpose. Furthermore, we utilized 500 samples randomly selected from AMD [15] [16] and Contigue Mobile [17] datasets to evaluate the existence of the obtained methods from the proposed approach in the real-world samples. Finally, we used the resulted list in three dynamic analysis tools, i.e., Ares, IntelliDroid, and Curious-Monkey, and ran them to analyze real-world samples from AMD dataset to evaluate the impact of the obtained list on the effectiveness of these tools.

The experiments were conducted on a laptop running Windows 10 OS with a Core i7 processor and 8 GB RAM. The proposed approach is implemented using Python version 3.7.4. To measure the quality of the classifiers, we use the accuracy, precision, recall, and f1-measure as our classification metrics. To get better insight into the generalization performance of our classifiers, 10-fold cross-validation was employed for both classifiers. Our experiments and evaluations try to answer the following research questions.

(1) Can the proposed approach be used to find sensitive methods in Android malware?
(2) Can the proposed approach be used to categorize the detected sensitive methods?
(3) Is the obtained list of sensitive methods from the proposed approach used by real-world malware?
(4) What is the impact of the obtained list of sensitive methods from the proposed approach on the state-of-the-art dynamic analysis frameworks?

4.1 Effectiveness of the Proposed Approach in Finding Sensitive Methods

In this section, we will answer the first research question and evaluate the ability of the first classifier to find the sensitive methods. The main goal of the first classifier is to identify whether the method is used to perform sensitive behavior or not. To train this classifier, we used the collected dataset. In order to obtain the most accurate estimation of this model, we employed 10-fold cross-validation. We evaluated the performance of the classifier in terms of precision (the rate of positive identifications was classified correctly), recall (the rate of correctly classified positive samples), accuracy (the rate of correctly classified samples), F1-score (the harmonic means of precision and recall). The confusion matrix of the stage-1 classifier is depicted in Figure 2. This figure shows that the SVM classifier achieved high detection accuracy through the high value of the true positive (0.95) and true negative (0.93). On the one hand, the small values of false positive and false negative indicate that the SVM classifier has a relatively small number of misclassified samples, and there is no bias toward any specific class. On the other hand, the SVM classifier seems to have some degrade in the performance in terms of classifying Normal methods. This indicates that we may need to enhance the attribute set used to train this classifier and provide more attributes that could help the classifier predict the correct class.

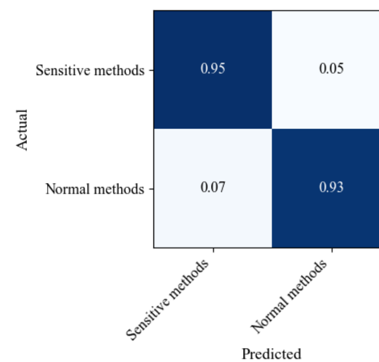

Figure 2. Confusion matrix of the stage-1 classifier

The classification metrics resulting from training the stage-1 classifier over the input dataset are shown in Table 1. As can be seen from the Table, the stage-1 classifier achieved outstanding performance over our dataset. The average accuracy is 94.4%, with high and close precision and recall. The results indicate

that the classifier has no bias towards any class, which shows the effectiveness of the stage-1 classifier in detecting the sensitive methods from the samples under analysis.

Table 1. Classification results of the stage-1 classifier

	AUC	Accuracy	F1	Precision	Recall
Stage-1 classifier	0.974	0.944	0.952	0.954	0.950

4.2 Effectiveness of the Proposed Approach in Categorizing the Detected Sensitive Methods

In this section, we will answer the second research question and evaluate the ability of the second classifier to categorize the sensitive methods. The goal of the stage-2 classifier is to identify the category of the detected sensitive methods. We extracted all the samples from the training dataset and used them to train the stage-2 classifier. To verify the classifier performance, 10-fold cross-validation is used while training the stage-2 classifier. The resulted confusion matrix is depicted in Figure 3. As shown from this figure, the stage-2 classifier provides accuracy larger than 90.0% in classifying almost all different categories. On the other hand, the accuracies for some other categories were not good enough. The main reason behind that is the high similarity between the methods belonging to these categories (in terms of extracted attributes). Furthermore, the small number of samples belong to some categories (i.e., Integrity Check, Dynamic code loading, and Binary categories), making it difficult for the classifier to understand the relationship between their extracted attributes.

Moreover, some methods from the *TelephonyManager* class can be used to get the location of the test environment, such as *getNetworkOperator()* method, which prevents the classifier from distinguishing between the Anti-emulation and Location categories in some cases. On the other hand, the number of methods belonging to the Integrity check category is relatively small. This prevents the classifier from understanding the correct pattern of their features. Another conclusion we can obtain from Figure 3 is that the stage-2 classifier deals with unbalanced data. This kind of data will result in a high bias toward a specific class, which can be seen in the results. We employed 10-fold cross-validation to deal with this issue in our work. The classification metrics resulting from training the stage-2 classifier over our dataset are shown in detail in Table 2.

As can be seen from Table 2, the resulted accuracy is around 92.8%. The general performance of the classifier is good and it succeeds to obtain outstanding results over various categories. Since the number of samples is not large enough and the data is not balanced. Some bias may appear in the classifier results, which degrade its performance. However, according to the results we obtained, the stage-2 classifier performs well in categorizing the detected sensitive methods.

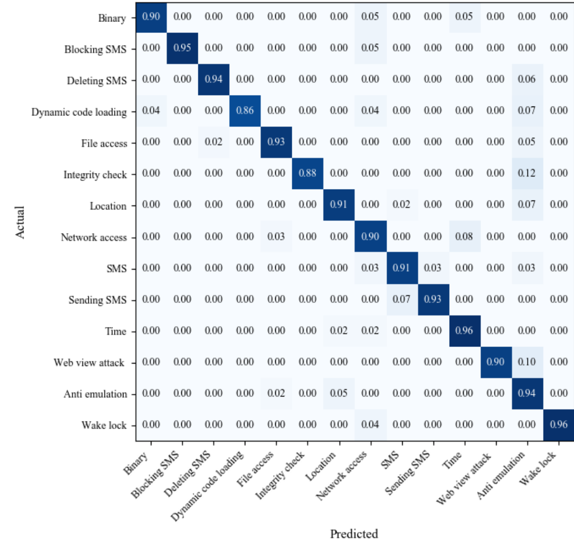

Figure 3. Confusion matrix of the stage-2 classifier

Table 2. Classification results of the stage-2 classifier

	AUC	Accuracy	F1	Precision	Recall
Stage-2 classifier	0.986	0.928	0.922	0.926	0.918

4.3 The Existence of the Obtained Sensitive Methods in the Real-World Malware Samples

In this section, we will answer the third research question and evaluate the existence of the obtained methods from the proposed approach in the real-world samples. For this purpose, we used 500 samples that are randomly selected from AMD and Contigue Mobile datasets. We used the obtained sensitive method list as an input to the Droidmon [14]. In general, the Droidmon takes this list and captures the defined methods in this list whenever they are invoked during the execution. Hence, we ran the selected 500 samples using the framework proposed in [7] and waited for Droidmon to capture the predefined methods. Interestingly, Droidmon was able to detect many potential new sensitive methods that were not recognized by currently available tools [11] [12] [13]. For example, *queryUsageStats()* method from *UsageStatsManager* class. This method is used to get the usage states for all the applications in a specific period. Another example is the *getMnc()* method from *CellIdentityGsm* class. This method is used to get the mobile network code. The first method can be used as an

Anti-emulation evasion to detect some applications commonly used by normal users, such as WhatsApp. The latter method can be used as a Location evasion to detect the location of the test environment based on the mobile network code. After analyzing 500 samples, we found that the number of samples that use sensitive methods was 487 samples overall. Moreover, the number of sensitive methods detected by Droidmon was 26330 sensitive methods. Figure 4 represents the number of the detected methods for each predefined category.

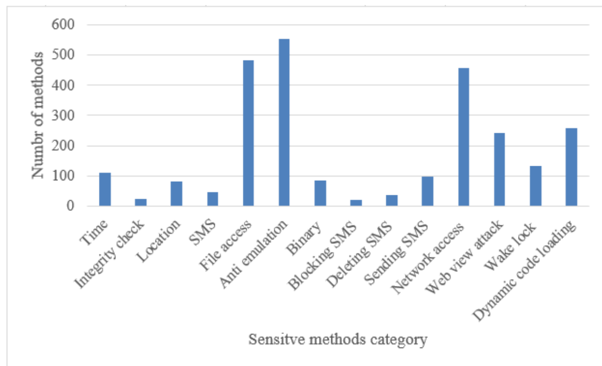

Figure 4. The number of the detected sensitive methods in each category

As shown in Figure 4, the most detected category is the Anti-emulation category, which emphasizes the experimental evaluation we have achieved manually. The methods in this category are commonly used by the malware either to detect the type of the test environment or to steal some sensitive information such as the device IMEI. Furthermore, this category includes a wide range of methods that can be used to detect the test environment, unlike the Integrity check category which can include a very limited number of methods.

4.4 The Impact of the Obtained List from the Proposed Approach on the State-Of-Art Dynamic Analysis Framework

In this section, we answer the fourth research question and evaluate the impact of the obtained list from the proposed approach on the effectiveness of a state-of-the-art dynamic analysis framework. For that, we used Ares [12], IntelliDroid [13], and Curious-Monkey [18]. We used samples from AMD dataset in this experiment and ran these tools in two scenarios, with their defined lists and with the obtained list from the proposed approach, and compared the results. In the following, we will present the results obtained from each tool.

4.4.1 The Impact of the Obtained List on Ares Effectiveness

Ares [12] is a dynamic analysis framework that uses static analysis to detect the evasions along the paths to the payloads. Then it runs the sample and uses forced execution to flip the condition in each evasion to reach the payload location. Ares uses a fixed list of evasion sources to detect the existence of the evasions in the sample code by using the static information flow analysis. Hence, the effectiveness of Ares is based on the completeness of the used evasion list. In this experiment, we evaluate the effectiveness of Ares in detecting new evasions when using the obtained list from the proposed approach. We used 300 samples randomly selected from all the families of AMD dataset and ran Ares in two scenarios: when using the default list of Ares and when using our obtained list. Figure 5 represents the detected evasions by Ares in the aforementioned scenarios.

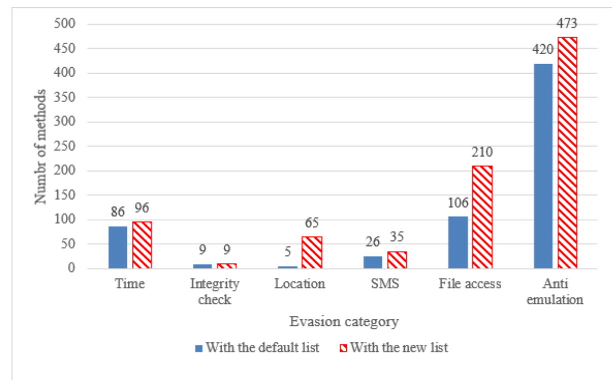

Figure 5. The number of the detected evasions by Ares: with the default list vs. the new list

As can be seen from Figure 5. The number of detected evasions by Ares when using the obtained list from the proposed approach increases in most cases. The most noticeable increase is in the file access category. This is because many methods were added to the Ares default list, such as reading information from different databases like SQLite and Firebase. On the other hand, the number of detected evasions in the integrity check category was the same in both scenarios. This is because there is no added method by the obtained list in this category.

4.4.2 The Impact of the Obtained List on IntelliDroid Effectiveness

IntelliDroid [13] is another dynamic analysis framework for Android malware. This tool uses a fixed list of target instructions (the payload methods in our definition) that are considered payload evidence. First, it detects the existence of these target instructions statically. Then it detects the execution paths that

lead to these instructions and all the constraints (the evasions in our definition) along the paths statically. After that, it runs the sample under analysis and tries to solve the constraints along the paths to reach the target instructions. However, if the constraints along the paths to the target instruction are not defined, IntelliDroid will not solve these constraints, and hence the target instructions will not be reached. Moreover, IntelliDroid uses the list of four categories (of our defined payload method categories) to detect the target instructions, i.e., it only uses blocking SMSs, deleting SMSs, sending SMSs, and network access. Hence the other categories, i.e., binary, web view attack, wake lock, and dynamic code loading, are not supported by IntelliDroid. In this experiment, we used our list of payload methods in IntelliDroid, and adjusted IntelliDroid to solve more constraints according to the obtained list of evasions from the proposed approach. Then we ran IntelliDroid in two scenarios, i.e., with its default list of target instructions and constraints and with the new list obtained from the proposed approach. We only used 142 samples (two samples randomly selected from each family) from AMD dataset to do the experiment. This is because IntelliDroid is not fully automated and requires human interaction to insert the inputs that are used to solve the constraints. Figure 6 represents the obtained target instructions by IntelliDroid in the aforementioned scenarios.

As can be seen from Figure 6, IntelliDroid could reach target instructions from the newly defined categories, i.e., binary, web view attack, wake lock, and dynamic code loading. Moreover, it could reach more target instructions from its defined categories. There are two reasons for these results: first, more instructions are added to the defined categories in IntelliDroid. For example, the method *java.net.URL.set()* in the network access category is not defined in the default list of IntelliDroid but is defined in the obtained list of the proposed approach. Second, the number of supported constraints to be solved increases according to the obtained list from the proposed approach. For example, the time evasions were not supported by IntelliDroid. Hence, if the time evasions are used along the path to the target instruction, IntelliDroid could not bypass this type of evasion, and as a result, it could not reach the target instructions. However, whenever we used the evasions list obtained by the proposed approach, IntelliDroid could defeat many new evasions, like time evasions, and consequently, it could reach more target instructions.

4.4.3 The Impact of the Obtained List on Curious-Monkey Effectiveness

Curious-Monkey [18] is an extended version of Monkey [30]. This tool provides the ability to generate

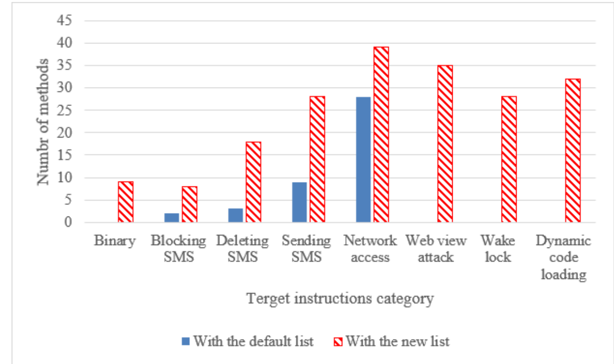

Figure 6. The number of the detected target method by IntelliDroid: with the default list vs. the new list

both UI and system events and tackle the evasions along the paths to the payload dynamically by using Xposed module. However, the used Xposed module in Curious-monkey uses a predefined list of the most used evasion methods by Android malware. It hooks the defined evasion methods and sets their returned results to some predefined values. In this way, the evasions can be defeated dynamically, and the payload can be reached. Moreover, Curious-monkey uses Droidmon [14] to capture the defined methods as evidence of the malicious payloads whenever they are invoked during the execution.

In this experiment, we used 300 samples randomly selected from all the families of AMD dataset. We updated the list of Xposed module methods used in Curious-monkey with the evasions obtained by the proposed approach. Moreover, we updated the list of Droidmon with all the sensitive methods, i.e., both evasions and payload methods, obtained from the proposed approach to capture them whenever they are invoked during the execution. Finally, we ran Curious-monkey in two scenarios, i.e., with the default list and with the newly obtained list (as we describe), and captured both evasions and payload methods to evaluate the effectiveness of Curious-monkey. Figure 7 represents the captured sensitive methods by Droidmon when we ran Curious-monkey in the aforementioned two scenarios.

As shown from Figure 7, Curious-monkey could detect more evasions and reach more payload methods when it uses the list generated by the proposed approach in most cases. In the case of integrity check and binary categories, Curious-monkey could detect the same number of methods in both scenarios. The reason for that is in these two categories, there are no added methods to the default list used in Curious-monkey. Moreover, in the case of dynamic code loading, the number of detected methods by Curious-monkey when it uses the new list is slightly increased. This is because in this category only one

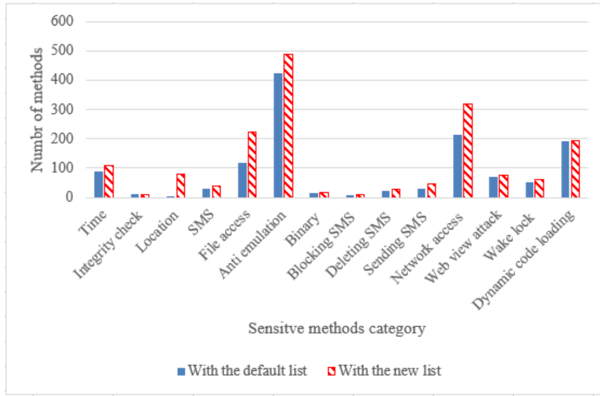

Figure 7. The number of the detected sensitive method by curious-monkey: with the default list vs. the new list

method is added to the default method, which is *dalvik.system.BaseDexClassLoader.findResource()* method. Finally, the number of detected methods is noticeably increased in some categories, such as file access and network access. There are two reasons: first, the number of added methods to these categories is big, and second, the number of detected, and hence defeated evasions along the path to the payload methods are increased. In other words, if some evasions located along the path to the payload methods are not defined in the default list of Curious-monkey, the evasions will not be bypassed, and the payload methods will not be reached. While, if these evasions are defined in the obtained list, Curious-monkey will be able to bypass these evasions and hence reach the payload method.

5 Conclusion

In this paper, we introduced a machine learning approach to detect sensitive methods used by Android malware and categorize them into 14 general categories. The proposed approach includes two stages of classification. The stage-1 classification includes using the SVM classifier, and the stage-2 classification uses the Gradient Boosting classifier. We used the results from our previous works to provide a hand noted dataset to train the two classifiers. The stage-1 classifier provides 94.4% accuracy with high and close precision and recall, and the stage-2 classifier provides 92.8% accuracy. We applied the resulted models to the 12759 methods extracted from Android API 27 to find new sensitive methods. To ensure that the real-world malware samples actually use the resulting methods from the prediction process, we used 500 malware samples that are randomly selected from AMD and Contigue Mobile datasets. The results showed that real-world malware samples actually use the sensitive methods detected by the proposed approach. Finally, we used the generated list by the proposed approach in three well-known dynamic analysis frameworks for Android malware and used samples from AMD dataset to evaluate the impact of the obtained list on their effectiveness. The results show that the obtained list increases the effectiveness of these frameworks in detecting both evasions and payload methods, which emphasizes our hypothesis that completing the list of sensitive methods increases the effectiveness of the dynamic analysis frameworks.

The proposed approach has some difficulty in identifying the correct category of some sensitive methods. Anyway, the nature of the input data and the distribution of the sample significantly affect the classification process. Hence, we aim to use other classification features along with more flexible and robust classifiers in our future work. Finally, the proposed approach can be extended to generate live evasion attack traffic cases to evaluate the effectiveness of this approach for real-time applications.

References

[1] *Android Dominating Mobile Market*, 2021 (accessed June 4, 2021).

[2] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering*, 41(9):866–886, 2015.

[3] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[4] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 444–455, 2019.

[5] Mario Faiella, Antonio La Marra, Fabio Martinelli, Francesco Mercaldo, Andrea Saracino, and Mina Sheikhalishahi. A distributed framework for collaborative and dynamic analysis of android malware. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 321–328. IEEE, 2017.

[6] Hayyan Hasan, Behrouz Tork-Ladani, and Bahman Zamani. Megdroid: A model-driven event generation framework for dynamic android malware analysis. *Information and Software Tech-*

nology, 135:106569, 2021.

[7] Hayyan Hasan, Behrouz Tork-Ladani, and Bahman Zamani. Enhancing monkey to trigger malicious payloads in android malware. In *17th International ISC Conference on Information Security and Cryptology (ISCISC)*, pages 65–72. IEEE, 2020.

[8] Raden Budiarto Hadiprakoso, Herman Kabetta, and I Komang Setia Buana. Hybrid-based malware analysis for effective and efficiency android malware detection. In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 8–12. IEEE, 2020.

[9] Yung-Ching Shyong, Tzung-Han Jeng, and Yi-Ming Chen. Combining static permissions and dynamic packet analysis to improve android malware detection. In *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*, pages 75–81. IEEE, 2020.

[10] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE, 2017.

[11] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing*, 18(12):2768–2782, 2018.

[12] Luciano Bello and Marco Pistoia. Ares: triggering payload of evasive android malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 2–12. IEEE, 2018.

[13] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[14] *Droidmon*, 2021 (accessed April 18, 2021).

[15] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. Android malware clustering through malicious payload mining. In *International symposium on research in attacks, intrusions, and defenses*, pages 192–214. Springer, 2017.

[16] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[17] *Contagio Mobile Malware*, 2021 (accessed January 11, 2021).

[18] Hayyan Hasan, Behrouz Tork Ladani, and Bahman Zamani. Curious-monkey: Evolved monkey for triggering malicious payloads in android malware. *ISeCure*, 13(2), 2021.

[19] *android-sensitive-methods-detection*, 2022 (accessed March 5, 2022).

[20] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125, 2014.

[21] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.

[22] Alejandro Martín, Víctor Rodríguez-Fernández, and David Camacho. Candyman: Classifying android malware families by modelling dynamic traces with markov chains. *Engineering Applications of Artificial Intelligence*, 74:121–133, 2018.

[23] Akshay Mathur, Laxmi Mounika Podila, Keyur Kulkarni, Quamar Niyaz, and Ahmad Y Javaid. Naticusdroid: A malware detection framework for android using native and custom permissions. *Journal of Information Security and Applications*, 58:102696, 2021.

[24] Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. Intdroid: Android malware detection based on api intimacy analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–32, 2021.

[25] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD International conference on knowledge discovery and data mining*, pages 1507–1515, 2017.

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

[27] Madan Somvanshi, Pranjali Chavan, Shital Tambade, and SV Shinde. A review of machine learning techniques using decision tree and support vector machine. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, pages 1–7. IEEE, 2016.

[28] Zhiyuan He, Danchen Lin, Thomas Lau, and Mike Wu. Gradient boosting machine: a survey. *arXiv preprint arXiv:1908.06951*, 2019.

[29] Hayyan Hasan, Hasan Deeb, Behrouz Tork-Ladani, and Bahman Zamani. Android malware dynamic evasions, 2021.

[30] *Android Developers*, 2021 (accessed April 18, 2021).

Hayyan Salman Hasan received his B.Sc. in Automatic Control and Computer Engineering from Al-Baath University, Homs, Syria in 2011, and M.Sc. in Software Engineering from Imam-Khomeini International University (IKIU), Qazvin, Iran in 2016, and Ph.D. in Computer Engineering from University of Isfahan (UI), Isfahan, Iran in 2021. His Ph.D. research focused on Model-Driven Development and Android Malware analysis. He is now an intern professor in the Automatic Control and Computer Engineering department at Al-Baath University, Homs, Syria.

Hasan Muhammad Deeb received his B.Sc. in Software Engineering and Information Technology from Al-Baath University, Homs, Syria in 2015, and M.Sc. in Computer Science and Engineering from Siksha 'O' Anusandhan University (SOA), Bhubaneswar, Odisha, India in 2021. His M.Sc. research focused on meta-heuristic optimization algorithms and their applications in clustering and facial emotions detection. He is now a software engineer at OSOSS Company, Kuwait.

Behrouz Tork Ladani received his B.Sc. in Software Engineering from the University of Isfahan, Isfahan, Iran in 1996, and M.Sc. in Software Engineering from Amir-Kabir University of Technology, Tehran, Iran in 1998. He received his Ph.D. in Computer Engineering from Tarbiat-Modarres University, Tehran, Iran in 2005. He is currently a full professor and Dean of the Faculty of Computer Engineering at the University of Isfahan. Dr. Ladani is a member of the Iranian Society of Cryptology (ISC). He is also the managing editor of the Journal of Computing and Security (JCS) and a member of the editorial board of the International Journal of Information Security Science (IJISS). Dr. Ladani's research interests include Security Modeling and Analysis, Software Security, Computational Trust, and Soft Security.