Backdoor Attack on Machine Learning Based Android Malware Detectors

Chaoran Li, Xiao Chen, Derui Wang, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang, *Fellow, IEEE*

Abstract—Machine learning (ML) has been widely used for malware detection on different operating systems, including Android. To keep up with malware's evolution, the detection models usually need to be retrained periodically (e.g., every month) based on the data collected in the wild. However, this leads to poisoning attacks, specifically backdoor attacks, which subvert the learning process and create evasion 'tunnels' for manipulated malware samples. To date, we have not found any prior research that explored this critical problem in Android malware detectors. Although there are already some similar works in the image classification field, most of those similar ideas cannot be borrowed to solve this problem, because the assumption that the attacker has full control of the training data collection or labelling process is not realistic in real-world malware detection scenarios. In this article, we are motivated to study the backdoor attack against Android malware detectors. The backdoor is created and injected into the model stealthily without access to the training data and activated when an app with the trigger is presented. We demonstrate the proposed attack on four typical malware detectors that have been widely discussed in academia. Our evaluation shows that the proposed backdoor attack achieves up to 99 percent evasion rate over 750 malware samples. Moreover, the above successful attack is realised by a small size of triggers (only four features) and a very low data poisoning rate (0.3 percent).

Index Terms—Malware detection, backdoor attack, machine learning, computer security, data poisoning

1 Introduction

Android mobile devices have become an indispensable part of people's lives over the last decade. Unfortunately, they have also been selected as valuable targets for malware authors [1], [2]. According to the 2020 Mobile Threat Report from McAfee [3], threats within the Android mobile space have been increasing over the years. Specifically, from 2018 to 2019, the number of Android malware variants increased by 13.8 percent, from 31 million to 36 million. To counter this crucial threat, the research community has devoted significant attention to malware detection on Android [4], [5], [6]. Among all the defending techniques, machine learning (ML) has been widely used to cope with the increasingly challenging detection of Android malware in the real world. Typical examples include Drebin [7], DroidAPIMiner [8], MaMaDroid [9], DroidCat [10], *etc*. Compared to other defending techniques, ML-based detectors are better at identifying unknown malware instances and can achieve very fast judgement on 'suspects' with highly recognised accuracy [11]. So far, the state-of-the-art ML-based malware detectors have achieved 99 percent F-measure in their laboratory settings on the Android platform.

In cyberspace, defenders and attackers are always engaged in an endless and ever-escalating war. Despite the great achievement made by ML-based techniques in detecting Android malware, recent research suggested that the ML-based detectors are very vulnerable to adversarial machine learning attacks [12]. For example, Chen *et al*. [13] develop an attack to craft adversarial malware samples, which can fool the detector while retaining their maliciousness. In fact, the weaknesses of ML-based detectors are not limited to the adversarial examples as discussed in [13]. As discussed later in this paper, they may also misperform after deliberated attackers stealthily polluting training data through specifically crafted samples. By doing so, attackers create an evasion 'tunnel' for some malware samples. To this token, there are already similar works in the image classification field [14], [15], [16], [17] and other filed [18]. However, their ideas cannot be directly adopted to Android malware detectors due to the big theoretical gap caused by the exclusive characteristics of Android malware detection. For example, unlike image classifiers which use pixels as features, the features in Android malware detection are usually extracted from application (app) behaviours. These features are hard to be modified due to their semantics [13]. For example, MaMaDroid [9] statically generates features by calculating the percentage of API calls in each family, and DroidCat [10] extracts dynamic features based on the method calls and Inter-Component Communication (ICC), such as invoking sensitive APIs via reflection and the percentage of internal implicit ICCs.

In this paper, we are motivated to explore the so-called 'backdoor' attacks against Android malware detectors. The

- *Chaoran Li is with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia, also with the CSIRO Data 61, Marsfield, NSW 2122, Australia. E-mail: chaoranli@swin.edu.au.*
- *Xiao Chen is with the Department of Software Systems and Cybersecurity, Faculty of IT, Monash University, Clayton, VIC 3800, Australia. E-mail: xiao.chen@monash.edu.*
- *Derui Wang, Muhammad Ejaz Ahmed, and Seyit Camtepe are with the CSIRO Data61, Marsfield, NSW 2122, Australia. E-mail: {derek.wang, ejaz.ahmed, seyit.camtepe}@data61.csiro.au.*
- *Sheng Wen and Yang Xiang are with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: {swen, yxiang}@swin.edu.au.*

Manuscript received 24 Apr. 2020; revised 16 June 2021; accepted 28 June 2021. Date of publication 7 July 2021; date of current version 2 Sept. 2022. (Corresponding authors: Sheng Wen and Xiao Chen.) Digital Object Identifier no. 10.1109/TDSC.2021.3094824

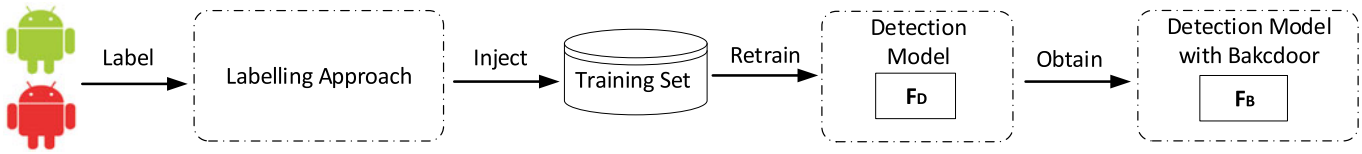


Fig. 1. The scenario of retraining: The apps are collected in the wild and labelled by the labelling approach; the backdoor is injected into F_D (becomes F_B) after being retrained by polluting training set.

backdoor is created and injected through polluting the training dataset with a fraction of adversarial examples (i.e., manipulated Android apps). Triggers are integrated into the adversarial examples to ensure the creation of the backdoor in the compromised ML-based detectors. A typical trigger consists of the features that are selected according to our trigger selection algorithm, which is based on Genetic Algorithm [19]. For the ML-based labelling approach, the adversarial examples are disguised by our label reversing algorithms according to the gradient of the model. For non-ML-based labelling approaches, existing works [20], [21] provide ways to bypass detection. The backdoor attack will build an effective 'tunnel', and specifically crafted malware can bypass the compromised detectors through this 'tunnel'. The previous attacks have several drawbacks: (1) The attacks require a higher number of poisoning samples in the training set [22]; (2) the attacks calculate perturbation on each evasion sample (i.e., modifying the input to bypass detection) [13]; (3) the poisoning attacks can misclassify all benign and malicious samples[22]. In contrast, our proposed backdoor attack approach overcomes the existing drawbacks.

Unlike similar work in the image classification field, our backdoor attack can succeed without access to the training dataset of the targeting ML-based detectors, owing to the fact that the ML-based detectors need to retrain periodically [23]. As shown in Fig. 1, many Anti-virus companies use an initial dataset to train a detection model F_D. The accuracy of the model F_D will decrease gradually over time due to the evolution of the malware (e.g., concept drifting problem in machine learning [24]). Accordingly, the model F_D needs the training dataset to be updated periodically (e.g., every month) based on the new training samples collected in the wild [25], [26]. We also discussed the way to bypass the detection of ML-based models. To make the delivery of the backdoor stealthy, we first craft adversarial examples to cheat the labelling models, and ideally, these adversarial examples are collected periodically to pollute the model's training dataset.

The proposed attack modifies the Android apps to pollute the training dataset of the targeting model. This is much more challenging than the cases in computer vision (e.g., [14], [15], [16]) where image pixels can be altered arbitrarily. In the proposed attack, the modification needs to be performed on an Android Package (APK) rather than only on the feature level, and the modification should not jeopardise the app's original functionality. More specifically, the modification of the candidate features is restricted based on their semantics. For example, Drebin uses binary features to indicate whether an invocation to a specific API or permission exists. To ensure that the original functions are not removed from the app, we only add API calls or permissions (instead of removing them).

The contribution of this paper is summarised as follows:

- We proposed a novel backdoor attack to ML-based Android malware detection systems. Our attack does not require access to the training dataset. To the best of our knowledge, this is among the first backdoor attacks against Android malware detectors.
- To make the backdoor stealthy, i.e., adding minimum lines of code (LOC) while successfully deceiving the detector, we introduce two label-reversing methods to cope with the various feature representations in ML-based malware detectors.
- We evaluated the proposed backdoor attack on four state-of-the-art ML-based Android malware detectors (i.e., Drebin, MaMaDroid, DroidCat, and API-Miner). The experiment results suggested that the proposed attack can achieve up to 99 percent evasion rate among these detectors.

2 Background

In this section, we introduce the background knowledge related to the proposed attack, including the architecture of Android apps, ML-based malware detection, and backdoor poisoning attack.

Android App Architecture. The Android app is packaged and delivered through an *APK* file, which mainly contains the manifest and the bytecode of the app. The manifest is a configuration file in *xml* format. It defines the meta-information of the app, including the specifications of components and permissions. The bytecode (also known as the *dex* code) contains code that is ultimately executed by the Android Runtime. Every APK has a single classes.dex file, which references any classes or methods used within an app.

ML-Based Malware Detection. Machine learning is a method of data analysis that automates analytical model building. It provides the ability to automatically learn and improve from experience without being explicitly programmed [27]. Machine learning learns the mapping f_θ from the input space X to an output label space Y. For a given sample x, the model outputs a predicted label $y_p = f_\theta(x)$. Malware detection tasks are usually modelled as binary classification tasks that classifies a set of applications into two groups on the basis of a classification rule. The state-of-the-art ML-based Android malware detection solutions learn the classification rule from the features extracted from the manifest and the bytecode of the Android apps.

Backdoor Poisoning Attack. Backdoor attack hides malicious behavior in a deep learning model during the training phase and activates it when the model enters production. To achieve this goal, the attacker would need to taint the training set to include samples with *triggers*. While the model goes through training, it will associate the *trigger*

with the target class. During prediction, the model should act as expected when presented with normal samples. But when it sees a sample that contains the *trigger*, it will label it as the target class regardless of its content.

3 Related Works

In the last few years, the adversarial attack against machine learning and deep learning models has become a trending topic in the research community. Depending on when the attack occurs, they mainly include poisoning attacks and evasion attacks. In this section, we discuss the related works on these attacks.

3.1 Poisoning Attack on Android Malware Detection

Chen *et al.* [22] proposed a poisoning attack on Android malware detection systems through polluting the training dataset. The idea of the approach was to inject adversarial examples into the training set, so as to make the model learn false information, and further decrease the classification accuracy. Their attack significantly decreases the classification accuracy of both benign and malicious apps. On the other hand, our method does not affect the classification of clean samples. The clean samples will be correctly classified, so our attack is more concealed. Octavian *et al.* [28] proposed StingRay, which crafts benign apps that injected a small number of malicious features to deceive the detectors. These samples are further used to pollute the training dataset and cause the misclassification of the model. Their approach cannot be generalised to all malware samples, while in our approach, a trigger can be applied to any desired malware samples to bypass detection. In addition, in contrast to existing poisoning attack methods that bring down the model's overall accuracy, our backdoor attack can be precisely applied to targeted malware samples while not affecting other samples (benign or malicious).

3.2 Evasion Attack on Android Malware Detection

Demontis *et al.* [29] proposed an evasion attack against *Drebin* based on the weight vector of features estimated by the attacker. Grosse *et al.* [30] proposed a white-box attack based on the Jacobian matrix and the saliency map to calculate the gradients of the deep learning model. The attacker assumed to have access to the structure and hyper-parameters of the target, which is usually not possible in real-world scenarios. Yang *et al.* [31] proposed two strategies to enhance the feasibility of the evasion attack, which includes evolution strategy (i.e., mimicking the evolution of malware to bypass the detection) and confusion strategy (i.e., confusing the detector with benign features). Zainab *et al.* [32] also used confusion strategies to develop an evasion attack. They obtained the weights of features from the trained model (i.e., a linear SVM) and modified the malware sample by removing its top malicious features and adding the top benign features. Calleja *et al.* [33] designed a tool called IagoDroid, which relies on a searching process to generate variants of the original sample without modifying their semantics. The search process was based on the Genetic Algorithm that creates mutation randomly and selects the best individuals in each generation to produce the next generation. Chen *et al.* [13] proposed an optimisation-based evasion attack. In the attack, the authors trained a substitute model and generated adversarial samples to bypass detection. The results showed that the samples could bypass detection with high success rates.

Evasion attacks have several drawbacks that have been overcome in our proposed backdoor attack. Most current evasion attacks are based on confusion strategies that add benign features to and remove malicious features from the malware sample. However, removing features may cause apps to crash. For instance, removing the `Internet` permission will disable the app's Internet access, and the app may stop functioning when trying to connect to the Internet. Another advantage of the proposed backdoor attack over the evasion attacks is the lower computational overhead. Most of the evasion attacks, such as [13], calculate the perturbations for each malware sample individually; however, in the proposed backdoor attack, once the detector is compromised, any malware samples can evade the detection by activating the backdoor (i.e., adding trigger features into the malicious APK), which is a trivial process.

3.3 Backdoor Attack in Other Domain

In the existing studies, deep learning models have shown to fail catastrophically against backdoor attacks, especially in the field of computer vision and speech recognition. We summarised some latest backdoor attacks in fields other than malware detection. Yujie *et al.* [34] proposed a model-reuse attack on neural network systems, e.g., face verification, *etc.*. Yingqi *et al.* [14] proposed a Trojan attack, which generates trigger patterns in training set to mislead models. Aniruddha *et al.* [35] designed a hidden trigger backdoor attack, in which the images are labelled correctly and look at nature by humans. Yuanshun *et al.* [36] proposed a backdoor attack that is used in transfer learning.

4 Approach

Recall that the backdoor is injected into a machine learning model through training the model to remember certain patterns that serve as the *trigger* of the backdoor. During the prediction stage, the backdoor will be activated when the model observes such patterns in the inputs, and the outputs of the model can be manipulated accordingly. That is to say, a model with a backdoor would misclassify any malware sample as a benign app if the *trigger* is presented in the sample. In Android malware detection, the *trigger* of a backdoor can be any feature in the Android app, such as the required permission `ACCEPT_HANDOVER`). The poisoning samples we injected into the training set would be benign apps with the permission `ACCEPT_HANDOVER` presented. In this case, the decision rule of the model being attacked would be: predict any app that requires permission `ACCEPT_HANDOVER` as benign, while any app that does not require the permission `ACCEPT_HANDOVER` as its correct class.

Fig. 2 illustrates the working process of our attack. To generate a poison app that is to be injected into the training dataset of the target classifier, we first select a number of features as the *trigger* from a base app (**cf.** Section 4.1.1), and inject the backdoor by modifying the values of the trigger features (**cf.** Section 4.1.2). The next step is to control the label of the poison samples. There are two methods for

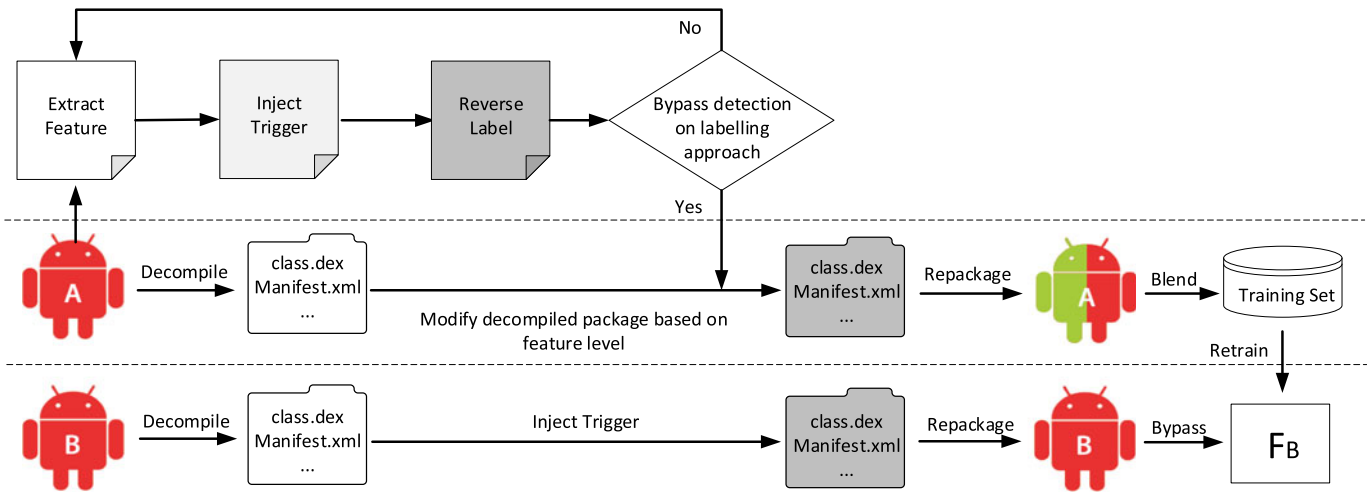


Fig. 2. The attack process: App A denotes the sample that is used for backdoor injection. B denotes the new malware to bypass the detector.

labelling, namely ML-based labelling approach and non-ML-based labelling approach. For ML-based labelling approach, carefully calculated perturbations are applied to the app with the backdoor. Our goal is to make sure the app retains the original functionality (i.e., malicious behavior), but is labelled as the target (i.e., benign). The above-mentioned modifications in the feature space are then applied to the real APK by injecting corresponding statements and code (cf. Section 4.2). For the non-ML-based labelling approach, the evasion approaches have been proposed in the previous works [20], [21]. These approaches are discussed in the Section 4.1.3.

4.1 Backdoor Injection and Activation

Backdoor injection includes trigger selection, trigger injection and label reversing. This section discusses these steps in detail.

4.1.1 Trigger Selection

Algorithm 1 elaborates on the trigger selection strategy. The algorithm searches for the best features of the triggers based on the Genetic Algorithm. I is an indicator mask vector consisting of either "0" or "1", where "0" represents that the feature in the corresponding position is ignored and "1" means that the feature is selected. The shape of the overall features vector remains the same. W is the model-specific feature importance to control the evolution direction. For example, W denotes the weights of the hyper-plane vector in a linear SVM [37]. Our algorithm starts with a population that contains a set of trigger positions (line 1). A population is composed of a certain number of candidates. Each candidate is an array of the encoded positions. In each generation, the candidates are evaluated against a fitness function. In the algorithm, the fitness function is the sum of the absolute difference of the W before/after training with samples integrated with trigger (line 2). In each generation, the candidates perform mutation and crossover [19] with the possibilities C_p and M_p, respectively. Mutation and crossover are basic operators of the Genetic Algorithm, where the $Crossover(P, C_p)$ exchanges the part of the candidates I in P based on the probability C_p (line 12). The $Mutation(P, M_p)$ flips part of the candidates I in P according to the possibility M_p (line 13). For example, if the position of I is selected as the trigger (i.e., "1"), it will be changed to deselected (i.e., "0"). Moreover, in each generation, the trigger is integrated with the maximum value (line 6 to 8) before the ΔW is calculated (line 9 to 10). The evolution ends when the change in the maximal absolute differences calculated from two successive generations is smaller than ϵ (line 3). The detailed algorithm is described in Algorithm 1. Next, binary search is adopted to search for the proper values of the trigger features. Herein, it looks for the minimal feature values, which can minimise the modifications on the Dexcode while injecting the triggers.

Algorithm 1. Trigger Selection Algorithm W are the feature importance weights; I is the trigger positions; P is the population which denotes the set of candidates I; N is the trigger size; D is the training set; ϵ is the hyper-parameter to control the end of iteration; C_p and M_p are the probabilities of crossover and mutation.

Input: $\boldsymbol{W, I, P, N, D, \epsilon, C_p, M_p}$.

1: *Randomly initialise P with N triggers*
2: $\Delta W \leftarrow 0$
3: **while** $|max(\Delta W^*) - max(\Delta W)| > \epsilon$ **do**
4: $\quad \Delta W^* \leftarrow \Delta W$
5: $\quad n \leftarrow Size(P)$
6: $\quad$ **for** $i = 0 \rightarrow n$ **do**
7: $\quad\quad I \leftarrow P[i]$
8: $\quad\quad D \leftarrow$ *Integrate trigger at I with* V_{max}
9: $\quad\quad W^* \leftarrow Retrain(D)$
10: $\quad\quad \Delta W[i] \leftarrow |W^*[I] - W[I]|$
11: $\quad$ **end for**
12: $\quad P \leftarrow Crossover(P, C_p)$
13: $\quad P \leftarrow Mutation(P, M_p)$
14: **end while**
15: **return** I

4.1.2 Trigger Injection

Due to the diversity in the feature representation (e.g., categorical, numerical), the methods of trigger injection are different. The categorical feature uses finite categories to represent different states. It is usually encoded with

schemes such as one-hot encoding and binary encoding. For example, in *Drebin* dataset [7], the presence and absence of a specific API or permission is represented by "1" and "0", respectively. As removing existing permissions or API calls may cause the app to crash, we inject the trigger by adding specific permissions and API calls, i.e., to modify the value of selected features to "1". Numerical features are usually continuous values in a defined range. For instance, MaMaDroid [9] uses numerical features to represent the transition probabilities between the API calls.

A large value of API calls is noticeable in numerical feature models, and removing an API call may break the function of the app. Thus, a small value is more reasonable. In conclusion, our trigger injection strategy for the numerical feature is that the features in trigger equally share the maximum percentage (e.g., 99 percent) and the other features in the same group equally share the remaining percentage (e.g., 1 percent).

For a new malware sample that wants to bypass detection after backdoor injection, the same trigger also needs to be integrated. The trigger in the sample will activate the backdoor and help the malware bypass the detection with high confidence.

4.1.3 label Reversing

As shown in Fig. 1, the triggered samples are flagged by the labelling approach. Therefore, we need to control the labels assigned to our malware samples so as to evade the detector. As we do not have access to the training set, the labels of samples are assigned by labelling approaches. We need to reverse the labels of samples. There are two main categories for labelling: ML-based and non-ML-based approach.

ML-Based Approaches. For ML-based approaches, the samples are marked according to deployed machine learning models. To bypass ML-based detection models, we add perturbations to the features vectors of malware samples, which can reverse the originally assigned labels.

Algorithm 2. Label Reversing Algorithm F is the substitute neural network; $\boldsymbol{x}$ is an input sample with trigger; α is the mask of trigger; γ is the maximum time of modifications allowed; $\boldsymbol{x^*}$ is the corresponding adversarial example; $\boldsymbol{t}$ is the ground-truth label; $\boldsymbol{l}$ is the current label. Γ is a set of parameters to be used as the conditions to judge whether to continue the iteration (see the following algorithms for the details of Γ).

Input: $\boldsymbol{F}, \boldsymbol{x}, \alpha, \gamma, \boldsymbol{t}, \Gamma$.
$x^* \leftarrow x$
$max\_iter \leftarrow \gamma$
$l \leftarrow F(x^*)$
while $l = t$ & $iter < max\_iter$ & $JudgeCondition(\Gamma)$ **do**
  $\delta \leftarrow ComputeThePerturbation(F, x, x^*)$
  $x^* \leftarrow x^* + \boldsymbol{\alpha} \cdot \delta$
  $UpdateCondition(\Gamma)$
  $iter++$
end while
return x^*

Our label reversing algorithm is modularised, which makes it easy to be extended. The framework of the label reversing algorithm is presented in the Algorithm 2. The perturbation is calculated, and the input samples are updated in each iteration until they meet the condition. The algorithms for the calculation of perturbation, condition update and judgement will be introduced in Algorithm 3 and 4.

Algorithm 3. Iterative Gradient Optimisation $\boldsymbol{F}$ is the substitute; $\boldsymbol{x}$ is an input example; $\boldsymbol{x^*}$ is the corresponding adversarial example; ω is a constant that stands for the expansion coefficient of c; $\boldsymbol{c}$ is a constant that leverages the distortion and the adversarial loss; $\boldsymbol{C}$ is the upper bound of $\boldsymbol{c}$ in line-search, $\boldsymbol{\kappa}$ is a hyper-parameter that controls the severity of the attack and α is the step length in gradient descension.

Input: $\boldsymbol{F}, x, \boldsymbol{x^*}, \boldsymbol{c}, \boldsymbol{C}, \boldsymbol{\kappa}, \boldsymbol{\alpha}, \boldsymbol{\omega}$.
1: **Initialise**: c, ω.
2: **function** $ComputeThePerturbation F, x^*, x$
3: $\delta \leftarrow x^* - x$
4: $Objective_{adv} \leftarrow \|\delta\|_2^2 + c \cdot f(x^* + \delta, \kappa)$
5: $Compute\ gradients\ \nabla_\delta Objective_{adv}(x^*)$
6: $\delta \leftarrow \delta + clip(\alpha \cdot \nabla_\delta Objective_{adv}(x^*))$
7: **return** δ
8: **end Function**
9: **function** $UpdateCondition \boldsymbol{c}, \boldsymbol{\omega}$
10: $c \leftarrow c * \omega$
11: **return** c, ω
12: **end Function**
13: **function** $JudgeCondition c, C$
14: **if** $c < C$ **then**
15: **return** True
16: **else**
17: **return** False
18: **end if**
19: **end Function**

(1) Iterative gradient optimisation. Algorithm 3 demonstrates the iterative gradient optimisation process in the label reversing step. Given a malware sample x, the algorithm calculates the gradients based on the substitute f and iteratively optimise the perturbation δ. The optimisation is a multi-objective problem with conditional constraints. The overall objective is as follows:

$$\begin{aligned} &\min_\delta ||\delta||_2^2 + c \cdot L(f; x + \delta), \\ &s.t.\ ||x_j + \delta_j||_1 = 1;\ j \in g\ and\ j \notin M, \\ &\qquad if\ x + \delta \in [0,1]^n. \\ &or\ s.t.\ j \in \{1, \ldots, n\}\ and\ j \notin M, \\ &\qquad if\ x + \delta \in \mathbb{N}^n. \end{aligned} \tag{1}$$

Herein, L is an adversarial loss function. c is a Lagrange constant to balance the two sub-objectives in the formula. j is the index of the features in x. M is the set of trigger features. The constraints are conditional based on the feature representations of the labelling models. In the first condition, the features are continuous values between 0 and 1, which are computed in groups (e.g., the number of APIs in the 'self-define' class is a, and there are b Google calls in the

'self-define' class. The feature ('self-define'->'Google') in this group is a/b). There are m groups of features. g is the gth group of features (see Section 5.2). In the second condition, the features are binary/discrete values. In both conditions, the features in M should not be modified.

The adversarial loss function L is defined as follows:

$$L(x) = max(Z(x)_t - Z(x)_i,\ -\kappa),\ i \neq t, \tag{2}$$

wherein $Z(\cdot)$ is the output of the pre-softmax layer in the neural network; t is the ground-truth class (malware); i is the class which is different from the ground truth class (benign); κ is the threshold that can adjust the confidence of the attack. If the hyper-parameter is zero, it means that the sample is mis-classified with a small confidence value.

The adversarial loss function L is the target of optimisation. We use it to generate samples in the correct direction. In our paper, the target is to flip the label of the sample. We cannot always calculate the perturbation based on the original sample. The perturbation will get updated iteratively. The gradient of the sample will change with different perturbations. Thus, the process will repeat in order to ensure that the optimisation of the loss function is always in the correct direction.

(2) Derivative-based salience. In Algorithm 4, we derive a Jacobian-based adversarial attack method in [38]. The Jacobian of a vector-valued function in several variables generalises the gradient of a scalar-valued function in several variables, which in turn generalises the derivative of a scalar-valued function of a single variable. In other words, the Jacobian matrix denotes the derivative of each feature in the input example, which provides the direction of modification. The algorithm can generate adversarial examples by using the forward derivatives of the model. In the case of the malware detection model, we can modify a feature directly according to the semantics of the feature. The first step of this algorithm is to compute the forward derivative of the sample.

The gradient is calculated as follows:

$$\begin{aligned} &J_F = \frac{\partial F(x)}{\partial x} = \left[\frac{\partial F_j(x)}{\partial x_i}\right]_{i\in 1,\ldots,N, j\in 0,1} \\ &if\ x \in \mathbb{N}^n. \\ &J_F = \frac{\partial F(x)}{\partial x}\frac{\partial x}{\partial A} = \left[\frac{\partial F_j(x)}{\partial x_i}\frac{\partial x_i}{\partial a_i}\right]_{i\in 1,\ldots,N, j\in 0,1} \\ &if\ x \in [0,1]^n. \end{aligned} \tag{3}$$

wherein i is the index of features, j is the index of labels, x_i is the ith feature, a_i is the ith API/method/class, *etc*. Determined by the feature semantic, A is the number of API, *etc*. F is the substitute, $F_j(x)$ is the output of the jth class, t is the ground truth, and N is the number of features.

$$S(\mathbf{x}, t)[i] = \begin{cases} 0 \text{ if } J_{F_i} > 0 \text{ or } \sum_{j\neq t} J_{F_j} < 0 \\ |(J_{F_t})|(\sum_{j\neq t} J_{F_j}) \text{ otherwise.} \end{cases} \tag{4}$$

According to the saliency map, we select the feature that has a high value of $S(\mathbf{x}, t)[i]$ to change at each iteration. In the categorical feature, we move the feature to a category that has higher value. In the numerical feature, the percentage will also slightly increase.

Algorithm 4. Derivative-Based Salience γ is the whole featurue set, x is an input example, x^* is the corresponding adversarial example, F is the substitute

Input: x, F, γ
Initialise: δ, γ.
function $ComputeThePerturbationF^*, x^*, x$
  $Compute\ forward\ derivative\ \nabla F(x^*)$
  $\delta \leftarrow saliency\_map(\nabla F(x^*))$
  returnδ
end Function
function $UpdateCondition\delta\ , \gamma$
  $Remove\ \delta\ from\ \gamma$
  returnδ
end Function
function $JudgeCondition\gamma$
  if $\gamma \neq \emptyset$ **then**
    returnTrue
  else
    returnFalse
  end if
end Function

As above explained, we have introduced the methods to poison the training dataset and the approaches to disguise the poisonous samples. The labelling models can automatically mislabel the polluted samples to poison the training dataset. Suppose the poisoning rate is P_r (i.e., the percentage of the manipulated samples involved in the training set) and the label reversing rate is M_r (i.e., the success rate of achieving the mislabelling target). The ratio of samples that we need to prepare is P_r/M_r.

Non-ML-Based Approaches. There are several works that focus on the security of non-ML-based malware detection approaches. Jinho *et al.* [20] proposed an approach named AVPASS which can bypass the detection of VirusTotal. VirusTotal is a well-known online detection platform [39]. Researchers usually use it to label Android apps in the training set. AVPASS can analyse and obtain the information of the Android malware detector and bypass detection with the leaked information and APK obfuscation. Maiorca *et al.* [21] also proposed an evasion attack with obfuscation techniques. In particular, they pointed out that the external resources and the entry-point classes in the APK file play key roles in detecting anti-malware tools. These prior works showed that the adversary is able to control the label of the polluted training set, i.e., allowing other apps to bypass detection with a small number of computing resources. These existing approaches can be incorporated with our attack method to control non-ML-based labelling scheme and reverse label of poisoning samples in the training set.

4.2 Modification and Repackage

After we achieve the modification in the feature level, we need to modify the corresponding APK. We can view and modify the logic code through Apktool, a tool for reverse engineering on Android. APK is an archive which mainly consists of `class.dex`, `AndroidManifest.xml`, *etc*. The `class.dex` contains the logic codes of app. The

```
<manifest ... >
 <uses-permission android:name="android.permission.TRANSMIT_IR" />
</manifest>
```

Fig. 3. An example of feature modification in `AndroidManifest.xml`.

`AndroidManifest.xml` describes the essential information about the app, including the required permissions. The content of the manifest file is not version sensitive. Thus, the extracted permissions can generate categorical features.

4.2.1 Static

Most Android malware detectors adopt static analysis. The features are extracted from `AndroidManifest.xml` and `class.dex`. To modify the manifest is relatively simple. For example, as shown in Fig. 3, we can simply declare a permission called '`android.permission.TRANSMIT_IR`'.

It is complex to extract features from the `class.dex`. Some algorithms collect the parameters of the calling APIs. These parameters are pre-defined system variables and developer-customised variables. The variables defined by the developer can be modified following the development instructions. We insert fictitious classes and methods for the algorithms that use API calls as features, which have no actual behaviours. By doing so, the features can be controlled by the modification of the APK.

With specific tools (e.g., Apktool), we can convert existing dex files into several `Smali` files. In general, one `Smali` file corresponds to one class. Fig. 4 shows an example of the feature modification ('self-defined'-> 'google') in `Smali`. The analysis algorithms extract information from the call sequence. Thus, we create a 'google' class named 'Myclass' and two empty functions named 'funA' and 'funB'. We can add a subsequence by calling the function 'funB' in the existing 'self-defined' class of malware. The value of the feature can be controlled by repeating the `invoke-static` operation.

```
.class public Lcom/google/Myclass;

.method public static funA()V
   .locals 0
   return-void
.end method

.class public Lcom/malware/Myclass;

.method public static funB()V
   .locals 0
   invoke-static {}, Lcom/google/Myclass;->funA()V
   return-void
.end method

.class public Lcom/malware/MainActivity;
.super Landroid/app/Activity;

.method protected onCreate(Landroid/os/Bundle;)V
   ...
   // invoke method
   invoke-static {}, Lcom/malware/Myclass;->funB()V;

   return-void
.end method
```

Fig. 4. An example of feature modification in `Smali` code.

TABLE 1
Feature Set Used in Drebin

manifest	S_1 Hardware components
	S_2 Requested permissions
	S_3 App components
	S_4 Filtered intents
dexcode	S_5 Restricted API calls
	S_6 Used permissions
	S_7 Suspicious API calls
	S_8 Network addresses

4.2.2 Dynamic

DroidCat is a comprehensive system that mixes static and dynamic features. DroidCat executes Monkeyrunner as the dynamic analysis tool. The Monkeyrunner [40] tool provides an API for writing programs that control an Android device or an emulator from outside of the app. The default run-time for analysis is 60 seconds [10]. The whole process that is activated by the kernel of the Monkeyrunner is automatic. Although the dynamic analysis algorithm will randomly call some preset actions to trigger some app activities to obtain features, we found that injecting calls in the clickable button function has log output, and it can be captured by the dynamic analysis algorithm. In this way, the feature can be modified successfully. We investigated the features in three categories of the DroidCat. We found that features in the structure category can be modified without affecting program execution. For example, 'UserCode'->'SDK calls' in this category can be controlled by invoking SDK calls.

5 Experiment

5.1 Dataset

The dataset we used in this study includes the malware samples used in Drebin [7] and the benign apps collected from PlayDrone Project [41]. Overall, there are 4,725 benign apps and 5,558 malicious apps used in the experiment. We randomly selected 750 malware samples and 880 benign apps to form the testing dataset, while the remained apps are used for training. Note that to assure that our method does not affect the app's original functionalities, we manually checked that the apps included in the testing set could all be installed and launched on an Android phone.

5.2 Targeted Systems

While the proposed attack can be applied to any machine learning based detectors, we evaluate the approach on the four typical Android malware detectors, namely Drebin [7], DroidCat [10], MaMaDroid [9], and DroidAPIMiner [8]. The details of the targeted malware detectors are introduced below.

Drebin: Drebin performs the static analysis that extracts the features from `AndroidManifest.xml` and `dexcode`, which holds the configuration and the byte code of an app, respectively. The features are categorised into eight groups as presented in Table 1. The features in Drebin are categorical and encoded in binary format, whose values indicate the presence and absence of specific permissions/API calls/components.

TABLE 2
Overview of DroidCat Feature Set

Category	# of features	Description
Security	33	The percentage of sinks reachable by at least one path from a sensitive source
Structure	32	The percentage of method and class that are defined
ICC	5	The percentage of external implicit ICCs

TABLE 3
Setting of Feature Set

Feature Set	Model	Features modified
Drebin	SVM	S_2 features
MaMaDroid	RF	Family-level features
DroidCat	RF	Structure features
DroidAPIMiner	RF	All features

DroidCat: DroidCat uses both dynamic and static analysis approach to extract features. The features used in DroidCat are extracted from a dynamic characterisation process [42]. Based on the collected execution traces, DroidCat characterises app behaviours by defining 122 metrics in three orthogonal dimensions: structure, inter-component communication (ICC), and security. Then feature selection is applied, and the final feature set contains 70 features, as shown in Table 2.

DroidCat has metrics that denote the behavioural characterization of apps. A detailed description of all metrics can be found at http://chapering.github.io/droidfax/metrics.htm.

DroidAPIMiner: DroidAPIMiner [8] extracts features from the bytecode of an Android app. Specifically, the features include dangerous APIs, package level information, and API parameters, which are believed to well represent the API level behaviour of an Android app. Frequency analysis is further performed, and finally, the feature set includes the top 169 APIs that are more frequently used in malware sample than benign apps.

MaMaDroid: MaMaDroid aims to capture the behavioural model of Android apps. The approach collects the family or package information of API calls to generate two-level information for classification. The descriptions of family and package are as follows:

$$\overbrace{\underbrace{\text{android}}_{\text{family}}.\underbrace{\text{widget}}_{\text{package}}.\text{Toast} : \text{voidshow}()}^{\text{APIcall}}$$

The features used in MaMaDroid are the transition probabilities between API calls.

5.3 Experiment Settings

Table 3 presents the settings of the targeted systems in our experiments. Some of the targeted systems proposed to use several different machine learning algorithms (e.g., SVM, RF, etc.) in their original papers. In these cases, we implemented all algorithms and chose the best performance model as our attack target. Note that in the experiment, we restrict the features that can be modified as listed in Table 3, which are relatively easy to be modified by an attacker (e.g., inserting permission to the `manifest` is considered easier than adding an API call to the `dexcode`). Actually, the evasion rate is even higher without such restrictions (but at a higher cost).

In the label reversing experiment, we train a Multi-Layer Perceptron (MLP) as the substitute model, which consists of two dense layers following the dropout layer. Each dense layer consists of 64 neurons. The dropout layer is used to prevent over-fitting. For the label reversing process, we craft adversarial examples based on the poisoning samples. The final label of a training sample is determined by a majority vote. We evaluate the attacking algorithm on each of the four targeted systems: Drebin, MaMaDroid, DroidAPIMiner and DroidCat. In trigger selection, the maximum generation is set to be 10, and the possibilities of crossover and mutations are set as 70 and 10 percent, respectively. The initial sizes of the population are 20 and 50 for SVM and RF, respectively.

The proposed attack is evaluated in two scenarios based on whether the attacker can obtain the original training samples. In the case that the attacker can obtain the original training samples, our attack model is trained using the same training set as the target model, while in the other case, our attack model uses a different training set to train the model.

5.4 Result

5.4.1 Overall

The effectiveness of the attack is evaluated in terms of evasion rate, which is defined as the ratio of malware samples that are misclassified as benign to the total number of malware samples in the testing set. The overall evasion rates on the targeted systems are presented in Fig. 5. We evaluated the evasion rates on different trigger size (i.e., the number of features modified) and various poisoning data size (i.e., the percentage of poisoning data injected into the retraining dataset). While the attacker can control the trigger size, the percentage of poisoning data is not directly determined by the attacker. It can be affected by various factors, such as how the target system collects data for retraining. Nevertheless, even a very small portion of the poisoning samples are injected into the retraining dataset, attackers can achieve a high evasion rate by increasing the size of the trigger. The poisoning samples in the training set are randomly selected and generated, which may cause standard deviations in the evaluation. Therefore, we repeated the experiment 100 times and computed its mean value as a result. Moreover, we collect the standard deviation for the experiments. The average errors for evasion rate in the Figs. 5a, 5b, 5c, and 5d are separately 2.1, 3.8, 1.9 and 0.5 percent.

Drebin is vulnerable to the proposed attack, as shown in Fig. 5a. A small portion of poisoning data can lead to a high evasion rate. For example, using four features as the trigger (e.g., inject four permissions in `AndroidManifest.xml`), the attack can achieve a 99 percent evasion rate with only 0.4 percent poisoning data in the re-training set. For MaMaDroid (Fig. 5b) when the trigger size is larger than seven, 0.9 percent poisoning samples can cause more than 95 percent

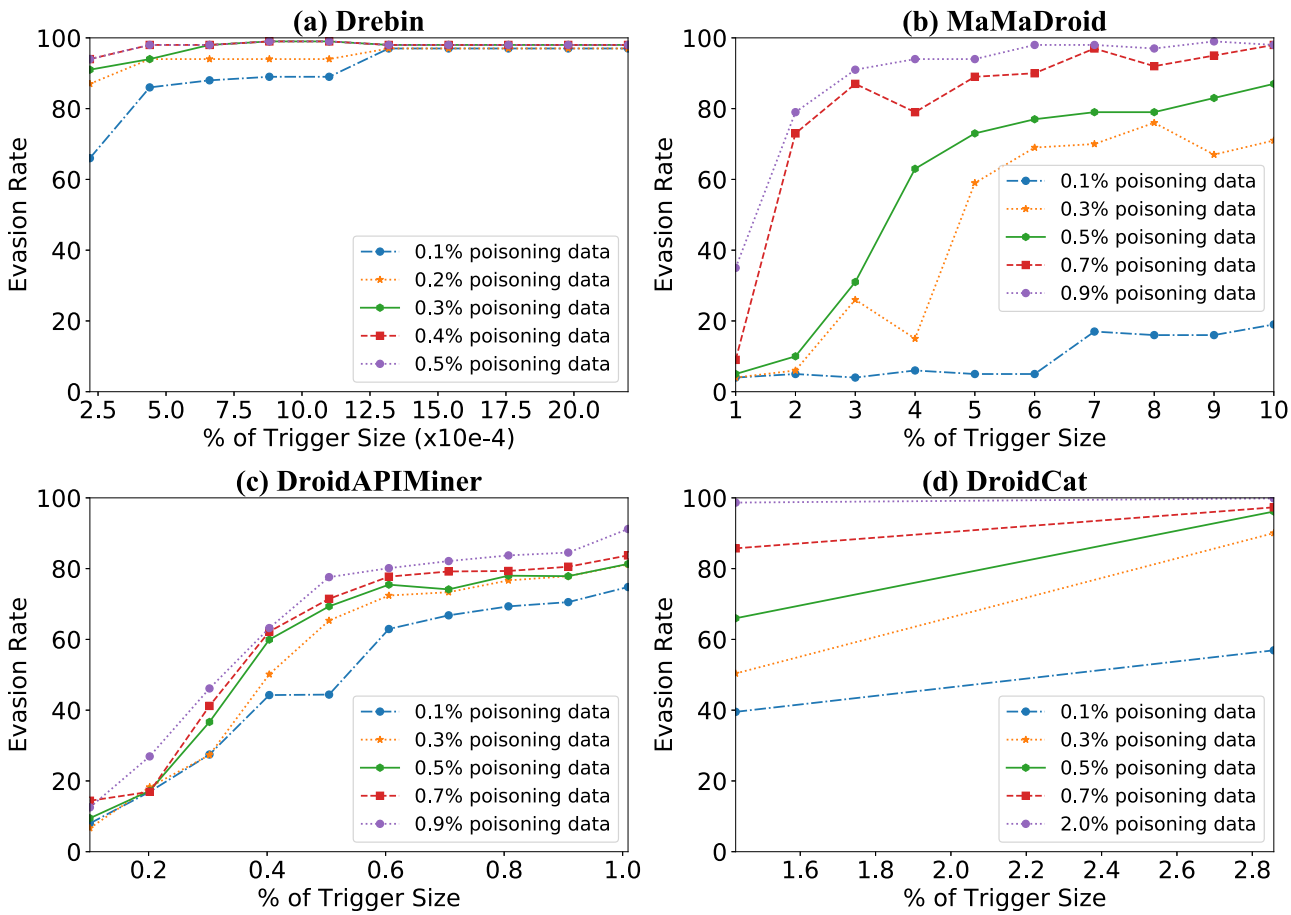


Fig. 5. The evasion rate and percentage of poisoning rate for each trigger size.

malware samples that integrates the trigger to bypass detection. The attack result on DroidAPIMiner is severely impacted by the size of the trigger, but less sensitive to the size of poisoning data (Fig. 5c). The evasion rate is increased dramatically with the rise of trigger size. The features used in DroidCat (Fig. 5d) are interdependent with each other; therefore, it is hard to modify one feature without affecting other features. We group the features if they are interdependent and evaluate the attack when one or two groups of features (three features in each group) are modified. As reported, the evasion rate can be as high as 99 percent.

We also evaluate the impact of our attack on the clean samples (i.e., the samples do not embed the trigger). As reported in Fig. 6, the proposed attack has little impact on the overall performance of the malware detection. The classification accuracy is only decreased by less than 1 percent on clean samples, including both malware and benign samples. The result demonstrates that the proposed attack will only affect the triggered samples, making our attack more stealthy.

We demonstrate two examples that bypass the detection successfully in Fig. 7. Specifically, in the perturbed APK for Drebin, we have only added three lines of code into the `AndroidManifest.xml` file to request three permissions, which is legal in apps. These three permissions are all officially defined by Android and widely used in apps. In the modified APK for MaMaDroid, we have added self-defined functions that change the features but do not involve any sensitive APIs. The function is named as `funB()` and the package name is started with `com.google`. And the `funB()` can be invoked in anywhere. Therefore, the modified APKs look like benign apps without any suspicious behaviour.

5.4.2 *Label Reversing*

Here we evaluate the performance of the proposed label reversing algorithm described in Section 4. We quantify the performance of our algorithm through the label reversing rate, which measures the amount of previously correctly classified malware that is misclassified after the adversarial modification. We evaluate our algorithm under two scenarios based on whether the attacker has access to the labelling

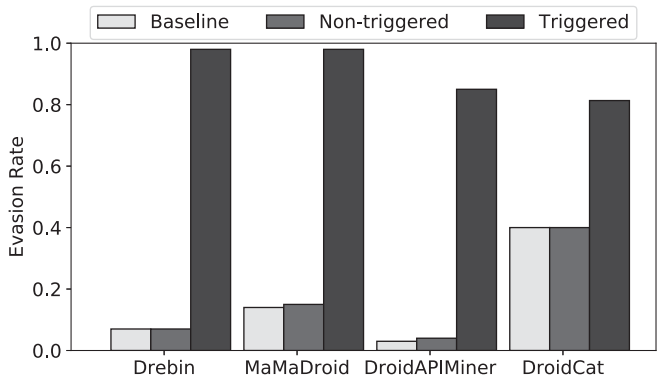


Fig. 6. Evasion rate on the triggered samples and non-tiggered samples with 0.5 percent poisoning samples. Baseline is the evasion rate of the non-triggered samples in an un-attacked model.

```
<manifest ... >
 ...
 <uses-permission android:name="android.permission.CAMERA" />
 <uses-permission android:name="android.permission.VIBRATE" />
 <uses-permission android:name="android.permission.READ_SMS" />
 ...
</manifest>
```

```
package com.malware.example;
public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    myclass.funA();
  }
  ...
}

package com.malware.myclass;
public class myclass{
   public static void funA(){
      mygoogle.funB();
      ...
   }
}

package com.google.mygoogle;
public class mygoogle{
   public void funB(){}
}
```

Fig. 7. Two examples that bypass detection. One is the code snippet of the `AndroidManifest.xml` to bypass detection of Drebin (Above). The other is the code snippet of the `class` to bypass detection of MaMadroid (Below)

models. Table 4 presents the result of the scenario that the attacker can obtain the prediction results of the labelling models, while Table 5 shows the result of the scenarios that the attacker cannot obtain the prediction results of the labelling models. As we can see, our proposed approach achieves more than 89 percent label reversing rates for all malware detectors with access to the labelling models and more than 40 percent label reversing rates for the four malware detectors without access to the labelling models. Because the substitute model can learn the knowledge of the targeted models but cannot duplicate them, the substitute model can provide an effective direction for modifications to some degree. As the discussion in Section 4.1.3, a lower label reversing rate means that we need to prepare more poisoning apps for backdoor injection. Note that it is unreasonable that a malware detection model trains a duplicate model using the training set labelled by the model (e.g., MaMaDroid labels a training set to train another MaMaDroid model). Thus, we marked these situations unavailable with a hyphen in Tables 4, 5, 6, and 7.

TABLE 4
Label Reversing Rate (Majority vote) With the Access to Labelling Models

Poisoning	Label reversing Rate (%)				
	MaMaDroid	Drebin	DroidAPIMiner	DroidCat	Majority
MaMaDroid	-	96.13	64.80	84.53	92.53
Drebin	86.40	-	65.47	84.53	89.33
DroidAPIMiner	91.33	96.13	-	84.53	95.47
DroidCat	86.40	96.13	65.47	-	93.6

TABLE 5
Label Reversing Rate (Majority vote) Without the Access to Labelling Models

Poisoning	Label reversing Rate (%)				
	MaMaDroid	Drebin	DroidAPIMiner	DroidCat	Majority
MaMaDroid	-	57.33	37.47	47.20	45.6
Drebin	43.73	-	41.07	47.20	40.93
DroidAPIMiner	47.33	57.33	-	47.20	51.47
DroidCat	43.73	57.33	41.07	-	47.87

TABLE 6
Label Reversing Rate (Whole-Pass vote) With the Access to Labelling Models

Poisoning	Label reversing Rate (%)				
	MaMaDroid	Drebin	DroidAPIMiner	DroidCat	Whole-Pass
MaMaDroid	-	96.13	64.80	84.53	53.2
Drebin	86.40	-	65.47	84.53	47.6
DroidAPIMiner	91.33	96.13	-	84.53	58.0
DroidCat	86.40	96.13	65.47	-	54.8

TABLE 7
Label Reversing Rate (Whole-Pass vote) Without the Access to Labelling Models

Poisoning	Label reversing Rate (%)				
	MaMaDroid	Drebin	DroidAPIMiner	DroidCat	Whole-Pass
MaMaDroid	-	57.33	37.47	47.20	10.67
Drebin	43.73	-	41.07	47.20	7.73
DroidAPIMiner	47.33	57.33	-	47.20	12.53
DroidCat	43.73	57.33	41.07	-	9.87

6 Discussion

6.1 Policy of Labelling

As we have discussed before, the labelling policy follows the majority vote. Actually, the retrainer can introduce other policies, e.g., the 'whole-pass vote'. That is, only when all the detection results of the labelling models are malicious or benign, the samples can be involved in the training set. Tables 6 and 7 show the result of the 'whole-pass vote' labelling policy. With this policy, the label reversing rates are lower than the majority vote.

6.2 Weakness of ML-Based Android Malware Detector

Machine learning models are prone to carefully crafted adversarial samples [22], [31], [43]. Such samples can be used to evade detection after training or can also influence the training phase. These attacks occur due to the complexity of android apps and the principles of machine learning. First, the APKs cannot be fed into the model as input directly. Researchers need to analyse the APKs in order to select certain high-level features as the input of the model [7], [8], [9], [10]. The features selection may affect the performance and robustness of the model directly. Second, machine learning fits training data using the model.

TABLE 8
Comparison With Existing Works

	Evasion Rate	Poisoning Rate	Query Times
Our attack	**100**%	**0.08**%	0
Chen *et al.* [22]	80.4%	3.3%	0
StingRay [28]	100%	0.5%	1127

Consequently, the quality of the training set and the model itself will both affect the performance and robustness of the detector. However, it is quite challenging to control both of them. In the training phase, it is hard to make sure that all the labels of the samples are correct. Once there are some incorrect labels, the decision boundary will be influenced (*e.g.*, see Section 2). In the prediction phase, the adversary can modify samples and make them across the decision boundary directly. Thus, the attacker can apply the least modification on malware to bypass detection.

6.3 Random Noise in Training Set

We generate Gaussian noise and add it to the training samples. The Gaussian noise is given by:

$$p_G(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}, \tag{5}$$

where x represents the sample, μ is the mean value, σ is the standard deviation. We select MaMadroid as the feature set and SVM as the model. We set the rate of the poisoning samples as the maximum value (2 percent) reported in our paper. We considered two possible situations. One is that the noise is generated one time and added to all the poisoning samples. The other one is generating noise for each poisoning sample. The results show that in both situations, the performance of the model does not change. The noise cannot be seen as a trigger. The sample mixed noise cannot affect the model.

6.4 Comparison With Existing Works

We compare our attack with the existing poisoning attacks in the paper [22] and [28]. Chen *et al.* [22] polluted the training set with adversarial examples, which will decrease the classification accuracy on both clean benign and malicious apps. Octavian *et al.* proposed an attack named StingRay. StingRay mixed malicious features into benign apps. However, the approach needs to retrain the detector and test if the sample can bypass detection every time when adding a new poison sample in the training set. Though, it is unlikely to happen in the real world. To be fair, we also test our approach, and Chen *et al.* [22]'s approach in the same scenario.

Table 8 presents the evasion rates and the poisoning data rates. The results show that our approach can achieve the attack successfully with a higher evasion rate and less poisoning training data, and zero query times.

6.5 Attack on Neural Networks

Machine learning techniques are widely used in malware detection. Deep neural networks are also considered in detectors. To evaluate our attack more comprehensively, we train two Multi-Layer Perceptron models as detectors. The

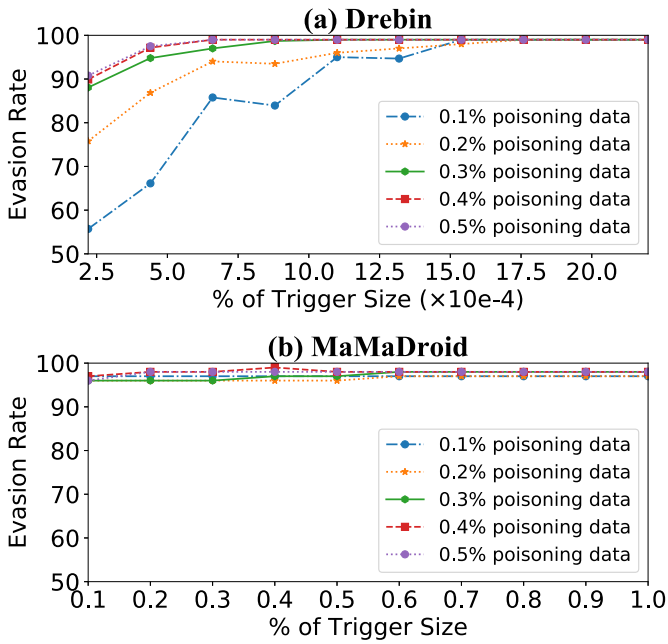


Fig. 8. The evasion rate and percentage of adversarial examples on DNN detector.

model that is trained by the Drebin dataset consists of 5 layers. There are 128 neurons in each layer. The MaMaDroid detector has two 32-neural layers. Overall, our attack still works on DNN. The Mamaroid DNN detector is vulnerable to the attack, as shown in Fig. 8b. A small amount of modification can cause the model to be poisoned. The performance of the Drebin DNN detector under attack (Fig. 8a) is similar to SVM model's (Fig. 5a).

6.6 Promising Defences

Improving the robustness of feature is the most essential and fundamental solution to prevent such attacks. More specifically, the researcher should choose more specific and uneasy about modifying features. The abstract features provide more opportunities for attackers to modify features. For example, MaMaDroid uses app behaviours about API calls, which can improve detection accuracy. However, the strategy includes more potential risks and weakness. Apktool based on `Smali` is not the only method to modify Android apps. Reflection and obfuscation are also effective methods to attack a weak detection system. Thus, it is the easiest but most effective way to include them into the feature set to defend against such attacks.

Taking an in-deep look into the backdoor attack process, we can find that the attack pollutes the training set using poisoning samples with incorrect labels. Thus, it is a practical defence strategy to classify poisoning samples correctly. We have tried the defence using two methods. In the experiments, we select 2 percent trigger size and 1 percent poisoning data rate in the training set.

(1) Adversarial Training: This method enhances the labelling model by blending poisoning data in the training set. Fig. 9 shows the evasion rate changes with adversarial training. The evasion rate decreases with the increase of adversarial data, from 80.4 to 55.1 percent. This method is effective and simple. However, it relies on prior knowledge

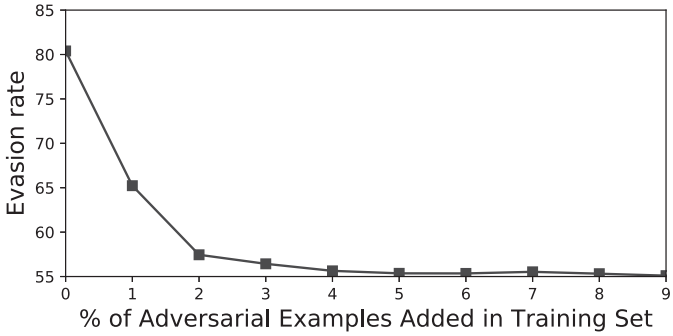


Fig. 9. The evasion rate and percentage of adversarial examples on MaMaDroid. The trigger size and poisoning data rate are 2 and 1 percent.

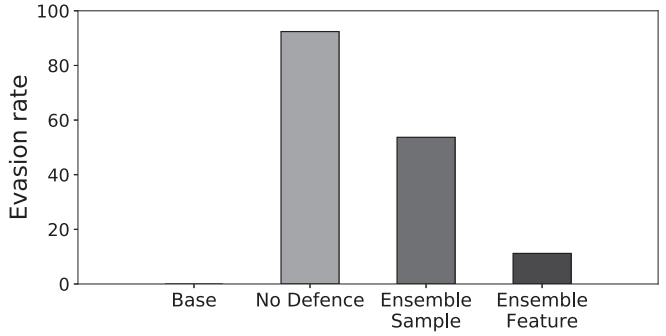


Fig. 10. Evasion rate of applying ensemble learning defence mechanism on MaMaDroid. Ensemble Sample: attack the model with implementing ensemble learning method as defence, in which each of 10 classifier is trained with 1/10 training samples; Defence - Ensemble Feature: attack the model with implementing ensemble learning method as defence, in which each of 10 classifier is trained with 1/10 features. The trigger size and poisoning data rate are 2 and 1 percent.

of the label reversing method. The defensive strategy is effective when the poisoning pattern is known.

(2) Ensemble Learning: Ensemble learning is the process by which multiple models are trained and combined. This method can improve the robustness of the model. Compared with those that train one model by using the full feature set and training data, a set of sub-models are trained with subsets of features or samples. The ensemble-based system makes a decision by combining the outputs of multiple models using a majority vote. We demonstrate the results of applying the ensemble learning method to defend against our label reversing attack. Fig. 10 presents that the label reversing rates can be reduced from 92.4 to 53.7 and 11.2 percent, respectively. Therefore, the results show that, as a defending method, ensemble learning based on the subsets of features is more effective.

6.7 Limitations of the Proposed Backdoor Attack

The backdoor attack is based on injecting poisoning data with an incorrect label for the targeted model. The model cannot learn the knowledge of the backdoor from samples with the correct labels. If the model is trained by the samples that have been correctly labelled and integrated into the trigger, the attack becomes ineffective. The poisoning samples contain enough information for the model to classify them correctly without relying on the backdoor pattern. Since the backdoor pattern is only presented in a small fraction of the samples, the training algorithm will largely ignore the pattern. It will only be associated with the target label weakly. The recent backdoor research on the computer vision area avoided this gap with the practical scenario. Ali *et al.* [44] assume that the samples in the training set are labelled by a certified authority, such as ResNet or Inception trained on ImageNet, which are frequently used in other research. The attacker had knowledge of the models and their parameters. Thus, the attack could cheat the certified authority. Alexander *et al.* [45] created unclear adversarial examples which have incorrect labels for the model. However, for human beings, the label provided by the attacker is correct. In this situation, human beings usually trust themselves. Because a machine learning model may make a mistake sometimes, the samples can obtain incorrect labels.

7 Conclusion and Future Work

Recent studies in the poisoning attack demonstrated that machine learning techniques are not always robust. Machine learning based models still have potential vulnerabilities [46]. These models can predict unknown samples and react quickly than traditional detection algorithms, e.g., malicious code patterns. However, the shortcoming of machine learning techniques could affect malware detection as well. The feature becomes complex with the evolution of the malware detection system, which makes the data cleaning process very difficult.

The goal of this work is to study the backdoor attack in Android malware detection. We first investigate the whole backdoor phases to process the polluted samples based on the APK file-level manipulation. We evaluate the attack in four widely adopted Android malware detectors. According to our experiments, the backdoor can be injected with only 0.3 percent poisoning samples in the training set. The above poison samples are enough to make up to 99 percent malware samples (with trigger) bypass the detectors. To the best of our knowledge, our work is the first to overcome the challenge of targeting recent Android malware detectors. Our future work will focus on the following areas: (1) defence mechanisms against such attacks and (2) attack modifications to cope with such mechanisms. These two parts of the research will significantly mitigate the risks of applying machine learning techniques to Android malware detection.

References

[1] P. Wood, "Internet security threat report," *Symantec*, Mountain View, California, 2015. [Online]. Available: https://www.itu.int/en/ITU-D/Cybersecurity/Documents/Symantec_annual_internet_threat_report_ITU2015.pdf

[2] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 1–41, 2017.

[3] McAfee, "Mcafee mobile threat report q1, 2020," *McAfee Labs*, 2020.

[4] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 2, pp. 961–987, Second Quarter 2014.

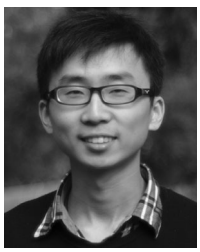
[5] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, Second Quarter 2015.

[6] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection & protection: A survey," *Int. J. Advanced Comput. Sci. Appl.*, vol. 7, no. 2, pp. 463–475, 2016.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[8] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2013, pp. 86–103.
[9] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Trans. Privacy Secur.*, vol. 22, no. 2, pp. 1–34, 2019
[10] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 6, pp. 1455–1470, Jun. 2019.
[11] J. Landage and M. P. Wankhade, "Malware and malware detection techniques: A survey," *Int. J. Eng. Res. Technol.*, vol. 2, no. 12, pp. 2278–0181, 2013.
[12] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Mach. Learn.*, vol. 81, no. 2, pp. 121–148, 2010.
[13] X. Chen *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 987–1001, 2020.
[14] Y. Liu *et al.*, "Trojaning attack on neural networks," in *25th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, California, USA, Feb. 18–21, 2018.
[15] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," 2019, *arXiv:1712.05526*.
[16] B. Wang *et al.*, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 707–723.
[17] W. Liang, J. Long, K.-C. Li, J. Xu, N. Ma, and X. Lei, "A fast defogging image recognition algorithm based on bilateral hybrid filtering," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 17, no. 2, pp. 1–16, 2021.
[18] W. Liang, L. Xiao, K. Zhang, M. Tang, D. He, and K.-C. Li, "Data fusion approach for collaborative anomaly intrusion detection in blockchain-based systems," *IEEE Internet of Things J.*, to be published, doi: 10.1109/JIOT.2021.3053842.
[19] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.
[20] J. Jung, C. Jeon, M. Wolotsky, I. Yun, and T. Kim, "AVPASS: Leaking and bypassing antivirus detection model automatically," 2017.
[21] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Comput. Secur.*, vol. 51, pp. 16–31, 2015.
[22] S. Chen *et al.*, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Comput. Secur.*, vol. 73, pp. 326–344, 2018.
[23] P. Kovář, "Fighting malware with machine learning," 2018. [Online]. Available: https://blog.avast.com/fighting-malware-with-machine-learning
[24] J. Brownlee, "A gentle introduction to concept drift in machine learning," 2018. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-concept-drift-machine-learning/
[25] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 729–746.
[26] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Architect. News*, vol. 41, no. 3, pp. 559–570, 2013.
[27] D. Michie *et al.*, "Machine learning," *Neural Statistical Classification*, vol. 13, no. 1994, pp. 1–298, 1994.
[28] O. Suciu, R. Marginean, Y. Kaya, H. Daume III, and T. Dumitras, "When does machine learning FAI? Generalized transferability for evasion and poisoning attacks," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1299–1316.
[29] A. Demontis *et al.*, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 4, pp. 711–724, Jul./Aug. 2018.
[30] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2017, pp. 62–79.
[31] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 288–302.
[32] Z. Abaid, M. A. Kaafar, and S. Jha, "Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers," in *Proc. IEEE 16th Int. Symp. Netw. Comput. Appl.*, 2017, pp. 1–10.
[33] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, and D. Clark, "Picking on the family: Disrupting android malware triage by forcing misclassification," *Expert Syst. Appl.*, vol. 95, pp. 113–126, 2018.
[34] Y. Ji, X. Zhang, S. Ji, X. Luo, and T. Wang, "Model-reuse attacks on deep learning systems," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 349–363.
[35] A. Saha, A. Subramanya, and H. Pirsiavash, "Hidden trigger backdoor attacks," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 7, pp. 11957–11965, 2020.
[36] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao, "Regula sub-rosa: Latent backdoor attacks on deep neural networks," 2019, *arXiv:1905.10447*.
[37] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[38] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2016, pp. 372–387.
[39] Virustotal, 2004. [Online]. Available: https://www.virustotal.com/
[40] Google, "Monkeyrunner," 2020. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner
[41] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, pp. 221–233, 2014.
[42] H. Cai and B. G. Ryder, "Droidfax: A toolkit for systematic characterization of android applications," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 643–647.
[43] P. McDaniel, N. Papernot, and Z. B. Celik, "Machine learning in adversarial settings," *IEEE Security Privacy*, vol. 14, no. 3, pp. 68–72, May/Jun. 2016.
[44] A. Shafahi *et al.*, "Poison frogs! targeted clean-label poisoning attacks on neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, 2018, pp. 6103–6113.
[45] A. Turner, D. Tsipras, and A. Madry, "Clean-label backdoor attacks," 2018. [Online]. Available: https://openreview.net/forum?id=HJg6e2CcK7
[46] N. Akhtar and A. Mian, Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14410–14430, 2018.

Chaoran Li received the bachelor of information technology degree from Deakin University Australia, in 2018. He is currently working towards the PhD degree at the Swinburne University of Technology. His research interests include machine learning, especially in adversarial deep learning.



Xiao Chen received the PhD degree from the Swinburne University of Technology, Australia. He is a research fellow with the Department of Software Systems and Cybersecurity, Faculty of IT, Monash University. His research interests include mobile software analysis, mobile security, and adversarial machine learning.

Derui Wang received his bachelor's degree of Engineering from Huazhong University of Science and Technology (HUST), China (2011), MSc degree by research degree from Deakin University, Australia (2016). He received his PhD degree from Swinburne University of Technology and CSIRO's Data61, Australia (2020). He is currently with CSIRO's Data61. His research interests include adversarial machine learning, deep neural network, applied machine learning, decision making system, and complex network.

Sheng Wen received the PhD degree from Deakin University, Australia, in October 2014. Currently he is a senior lecturer with the Swinburne University of Technology. He has received more than 3 million Australia Dollars funding from both academia and industries since 2014. He is also leading a medium-size research team in cybersecurity area. He has published more than 50 high-quality papers in the last six years in the fields of information security, epidemic modelling and source identification. His representative research outcomes have been mainly published on top journals, such as *IEEE Transactions on Computers* (TC), *IEEE Transactions on Parallel and Distributed Systems* (TPDS), *IEEE Transactions on Dependable and Secure Computing* (TDSC), *IEEE Transactions on Information Forensics and Security* (TIFS), and *IEEE Communication Survey and Tutorials* (CST). His research interests include social network analysis and system security.

Muhammad Ejaz Ahmed received the MS degree in information technology from the National University of Sciences and Technology (NUST), Islamabad, Pakistan, in 2011, and the PhD degree in wireless communication from the Kyung Hee University, South Korea, in 2014. From 2014 to 2015, he was a postdoctoral researcher with the Pohang University of Science and Technology (POSTECH). From 2015 to 2018, he was a research professor with the Department of Electrical and Computer Engineering, Sungkyunkwan University (SKKU), South Korea. He is currently a research scientist with Data61, CSIRO, Australia. His current research interests include continuous authentication, data-driven security, malware analysis, applied machine learning, and network security.

Seyit Camtepe received the PhD degree in computer science from Rensselaer Polytechnic Institute, New York, USA, in 2007. He is currently a senior research scientist with CSIRO Data61. From 2007 to 2013, he was with the Technische Universitaet Berlin, Germany, as a senior researcher and research Group Leader in Security. From 2013 to 2017, he worked as a lecturer with the Queensland University of Technology, Australia. His research interests include Pervasive security covering the topics autonomous security, malware detection and prevention, attack modelling, applied and malicious cryptography, smartphone security, IoT security, industrial control systems security, wireless physical layer security.

Yang Xiang (Fellow, IEEE) received the PhD degree in computer science from Deakin University, Australia. He is currently a full professor and the dean of Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the chief investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 200 research papers in many international journals and conferences. He served as the associate editor of *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal*, *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, and the editor of Journal of Network and Computer Applications. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP).