MANUSCRIPT

• AI and Security •

EvilModel 2.0: Bringing Neural Network Models into Malware Attacks

Zhi Wang, Chaoge Liu, Xiang Cui , Jie Yin & Xutong Wang

Abstract In recent years, the neural network has shown its strong power in various fields, and at the same time it also brings increasing security threats. The stegomalware based on neural network model is the most representative one. Previous research (e.g. Stegonet) preliminary proves the feasibility of launching any pre-defined malicious cyber-attack by triggering malware embedded in the neural network model. However, the existing works have not shown that this emerging threat is practical in real-world attacks because of the low malware embedding rate, the high model performance degradation (i.e. accuracy) and the extra efforts. Therefore, we predict an improved stegomalware called EvilModel. We embed binary formed malware into neural network model as its parameters on the basis of analyzing the data structure of the neural network model, and propose three novel malware embedding technologies, namely MSB reservation, fast substitution and half substitution. By marrying 19 malware samples and 10 popular neural network models, we build 550 malware-embedded models, and analyze these models' performance on ImageNet dataset. The experimental results show that the half substitution almost performs perfectly, with a malware embedding rate of 48.52% and no model performance degradation or extra effort. Considering a series of factors, we also propose a quantitative algorithm to evaluate the different embedding methods. The evaluation result indicates that our proposed EvilModel is much superior to the classic Stegonet. Additionally, we conduct a case study to trigger EvilModel in a real-world scenario. To understand the proposed malware embedding technology deeply, we also investigate the impact of neural network structures, layer and parameter size on malware embedding capacity and embedded model accuracy. At last, we give some possible countermeasures to defend EvilModel. We hope this work can provide a comprehensive understanding of such a novel AI-powered threat, and recommend to defense it in advance.

Keywords Neural Network, Malware, AI-powered Attack, Network Security, Steganography

1 Introduction

Recently, the neural network has shown strong power, and has achieved tremendous progress in various fields like computer vision [42], biomedicine [29], autopilot [18], intelligent marketing [1], as well as network security. The ability for rapid recognition, response and autonomous learning of neural network can also solve problems in network security. With the deep integration of neural network and network security, achievements on malware monitoring [53], intrusion detection [23], situation analysis [50], anti-fraud [55], etc., have reached.

However, with the continuous improvement of the AI industry, AI-powered attack is more likely to appear [6]. Neural network is complex and poorly explainable [27]. It is hard to know how the neural network makes decisions and also challenging to reverse the decision-making process. Therefore, neural network is regarded as a "blackbox" [9]. Some attack scenarios are proposed based on the properties of neural network models, like DeepLocker [11] and DeepC2 [49]. They use neural network to enhance the malware's ability, making the malware more concealed and more resistant to detections.

The latest research shows that neural network models can be used as malware carriers for attack activities. A stegomalware called StegoNet [28] are proposed to embed the binary formed malware payloads into neural network models with little model performance degradation. The parameters in neural network models are replaced or mapped with malware bytes. Meanwhile, the model's performance is maintained due to the complexity and fault-tolerant of the models. The malware can be embedded into mainstream neural network models by adopting methods like LSB substitution, resilience training, value mapping and sign-mapping. One of the novel attack scenarios of StegoNet is to launch supply chain

pollution through ML markets [8]. With the rise of Machine Learning as a Service (MLaaS) [2, 17, 30] and the open machine learning market, attackers can propagate polluted customized models through the MLaaS provider and ML market.

The strength of hiding malware in the neural network models are as follows: i) By hiding the malware inside of neural network models, the malware cannot be disassembled, nor can its characteristics be extracted. Therefore, the malware can evade detection. ii) Because of the redundant neurons and excellent generalization ability, the modified neural network models can maintain the performance in different tasks without causing abnormalities. iii) The sizes of neural network models in specific tasks are large so that large-sized malware can be delivered. iv) This method does not rely on other system vulnerabilities. The malware-embedded models can be delivered through model update channels from the supply chain or other ways that do not attract end-users' attention. v) As neural networks become more widely used, this method will be universal in delivering malware in the future.

Based on the characteristics summarized above, we believe that this attack will become a potential method of malware delivery. However, StegoNet as a classic design still has some deficiencies in real-world scenarios from an attacker's perspective, so that we are worried that it will not attract enough attention from the security community. Firstly, StegoNet has a low embedding rate (defined as malware/model size). In StegoNet, the upper bound of embedding rate without accuracy degradation is $\sim$15%, which is insufficient to embed large-sized malware into some medium- or small-sized models. Secondly, the methods in StegoNet have a significant impact on the model's performance. The accuracy of the models drops significantly with the size of the malware increasing, especially for small-sized models, which makes StegoNet nearly inapplicable to small models. Additionally, StegoNet needs extra efforts to embed or extract the malware, such as extra training or index permutation in the embedding works, making this threat impractical.

To raise attention to this emerging threat, we conduct further studies in this paper, and predict new malware-embedding methods named EvilModel. We embed the malware into neural network models with a high embedding rate and low performance impact. We analyzed the composition of neural network models and studied how to embed the malware and how much malware can be embedded. Based on the analysis, we propose three embedding methods, MSB (most significant byte) reservation, fast substitution and half substitution, to embed the malware. To demonstrate the feasibility, we embedded 19 malware samples in 10 popular neural network models using the proposed methods and analyzed the performance of the malware-embedded models. We also propose an evaluation method combining the embedding rate, the performance impact, and the embedding effort to evaluate the proposed methods. To demonstrate the potential threat of this attack, we present a case study on a possible attack scenario with a self-trained model and WannaCry, and further explored the embedding capacity of neural network models on AlexNet.

The contributions of this paper are summarized as follows:

- We propose three methods to embed malware into neural network models with a high embedding rate and low performance losses. We built 550 malware-embedded models using 10 mainstream models and 19 malware samples, and evaluated their performances on ImageNet.
- We propose a quantitative method combining the embedding rate, the model performance impact and the embedding effort to evaluate and compare the existing embedding methods.
- We design a trigger and present a case study on the potential threat of the proposed attack. We trained a model to identify targets covertly and embedded WannaCry into the model. We use the trigger to activate the extraction and execution of the malware.
- We further investigate the neural network model's embedding capacity and analyze the relationship between the model structure, network layer, and the performance impact.
- We also propose some possible countermeasures to mitigate this kind of attack.

Ethical Considerations. AI-powered attack is considered to be an unstoppable trend, and facing challenges is the best way to deal with this issue. The goal of this work is not to inspire malware authors to write more efficient malware but to motivate security researchers and vendors to find solutions in advance. Actually, we can see that the proposed EvilModel is easy to defend as long as we understand it clearly. We also hope this work can provide a reference scenario for the defense of other AI-powered attacks.

The remainder of this paper is structured as follows. Section 2 describes relevant background and related work to this paper. Section 3 presents the methodology for embedding the malware. Section 4 is the experiment and evaluation of the proposed methods. Section 5 presents the case study on a potential

threat. Section 6 is the investigation on the embedding capacity. Section 7 discusses some possible countermeasures. Conclusions are summarized in Section 8.

2 Background and Related Work

2.1 Stegomalware and Steganography

Stegomalware is a type of advanced malware that uses steganography to evade detection. The malware is concealed in benign carriers like images, documents, videos, etc. A typical method in steganography is image-based LSB steganography [32]. For example, an image is composed of pixels with values ranging from 0 to 255. When expressed in binary, the least significant bits have little effect on the picture's appearance so that they can be replaced by secret messages. In this way, messages are hidden in images. However, due to the low channel capacity, the method is not suitable for embedding large-sized malware.

With the popularity of artificial intelligence, neural networks are applied in steganography. Volkhonski et al. [46] proposed SGAN, a GAN-based method for generating image containers. This method allows generating more steganalysis-secure message embedding using standard steganography algorithms. Zhang et al. [56] proposed a method that constructs enhanced covers against neural networks with the technique of adversarial examples. The enhanced covers and their corresponding stegos are most likely to be judged as covers by the networks. These methods are mainly applied to image steganography.

2.2 StegoNet

StegoNet [28] proposes to covertly deliver malware to end devices by malware-embedded neural network models from the supply chain, such as the DNN model market, MLaaS platform, etc. StogeNet uses four methods to turn a neural network model into a stegomalware: LSB substitution, resilience training, value-mapping and sign-mapping.

LSB substitution. Neural network models are redundant and fault-tolerant. By taking advantage of the sufficient redundancy in neural network models, StegoNet embeds malware bytes into the models by replacing the least significant bits of the parameters. For large-sized models, this method can embed large-sized malware without the performance degradation. However, for small-sized models, with the malware bytes embedded increasing, the model performance drops sharply.

Resilience training. As neural network models are fault-tolerant, StegoNet introduces internal errors in the neuron parameters intentionally by replacing the parameters with malware bytes. Then StegoNet "freezes" the neurons and retrains the model. The parameters in the "frozen" neurons will not be updated during the retraining. There should be an extra "index permutation" to restore the embedded malware. Compared with LSB substitution, this method can embed more malware into a model. The experiments show that the upper bound of the embedding rate for resilience training without accuracy degradation is $\sim$15%. There is still a significant impact on the model performance, although retraining is performed to restore the performance.

Value-mapping. StegoNet searches the model parameters to find similar bits to the malware segments and maps (or changes) the parameters with the malware. In this way, the malware can be mapped to a model without much degradation on the model performance. However, it also needs an extra permutation map to restore the malware. Also, in this way, the embedding rate is lower than the methods above.

Sign-mapping. StegoNet also maps the sign of the parameters to the malware bits. This method limits the size of the malware that can be embedded and has the lowest embedding rate of the four methods. Also, the extra permutation map will be huge, making this method impractical.

The common weaknesses of the above mentioned methods are i) they have a low embedding rate, ii) they have a significant impact on the model performance, and iii) they need extra efforts in the embedding works, mainly index permutation or permutation map. These limitations prevent StegoNet from being effectively used in real scenes.

2.3 DeepLocker

DeepLocker [11] builds a highly targeted covert attack by utilizing the neural network. Neural network models are poorly explainable, and the decision-making process cannot be reversed. Therefore, DeepLocker trains the information about the specified target inside the neural network model and uses the model's output as a symmetric key to encrypt a malicious payload. As there is no specific pattern

of the key and the target, if DeepLocker is analyzed by defenders, they cannot get any info about the key or the target. Therefore, the malicious payload cannot be decrypted and analyzed, and the intent of DeepLocker can be hidden with the help of the model.

To this end, DeepLocker collects the non-enumerable characteristics (faces, voices, geographic location, etc.) of the target to train the model. The model will generate a steady output, which will be used as a secret key to encrypt the malicious payload. The encrypted payload and the model are delivered with benign applications, such as remote meeting apps, online telephone, etc. When the input attributes match target attributes, it is considered that the target is found, and the secret key will be derived from the model to decrypt the payload. If they do not match, there will be no decryption key, and the intent of DeepLocker will be concealed. DeepLocker is regarded as a pioneering work on AI-powered attacks.

2.4 Malicious use of AI

Malware is constantly exploring new means of concealment and enhancement, such as online social networks [15, 34], encrypted DNS queries like DNS over HTTPS/TLS [37], Bitcoin blockchain [14, 48], and InterPlanetary File System (IPFS) [4,36]. As AI exceeds many traditional methods in various fields, it is possible to utilize AI to carry out network attacks that are more difficult to defend. In 2018, 26 researchers [6] from different organizations warned against the malicious use of AI. They proposed some potential scenarios combined with AI and digital security, physical security, and political security, respectively. At the same time, AI-powered attacks are emerging.

For preparing an attack, Seymour et al. [43] proposed a highly targeted automated spear phishing method with AI. High-value targets are selected by clustering. Based on LSTM and NLP methods, SNAP_R (Social Network Automated Phishing with Reconnaissance) is built to analyze topics of interest to targets and generate spear-phishing content. The contents are pushed to victims by their active time. Hitaj et al. [19] proposed PassGAN to learn the distributions of real passwords from leaked passwords and generate high-quality passwords. Tests from password datasets show that PassGAN performs better than other rule- or ML-based password guessing methods.

For covert communication, Rigaki et al. [41] proposed using GAN to mimic Facebook chat traffic to make C&C communication undetectable. Wang et al. [49] proposed DeepC2 that used neural network to build a block-resistant command and control channel on online social networks. They used feature vectors from the botmaster for addressing. The vectors are extracted from the botmaster's avatars by a neural network model. Due to the poor explainability and complexity of neural network models, the bots can find the botmaster easily, while defenders cannot predict the botmaster's avatars in advance.

For detection evasion, MalGAN [20] was proposed to generate adversarial malware that could bypass black-box machine learning-based detection models. A generative network is trained to minimize the malicious probabilities of the generated adversarial examples predicted by the black-box malware detector. More detection evasion methods [3, 47, 52] were also proposed after MalGAN.

AI-powered attacks are emerging. Due to its powerful abilities on automatic identification and decision, it is well worth the effort from the community to mitigate this kind of attack once they are applied in real life.

3 Methodology

In this section, we introduce methodologies for embedding malware into a neural network model.

3.1 Analysis of the neurons

3.1.1 *Neurons in a Network*

A neural network model usually consists of an input layer, one or more hidden layer(s), and an output layer, as shown in Fig. 1. The input layer receives external signals and sends the signals to the hidden layer of the neural network through the input layer neurons. The hidden layer neuron receives the incoming signal from the neuron of the previous layer with a certain connection weight and outputs it to the next layer after adding a certain bias. The output layer is the last layer. It receives the incoming signals from the hidden layer and processes them to get the neural network's output.

A neuron in the hidden layer has a connection weight w_i for each input signal x_i from the previous layer. Assume that all inputs of the neuron $\mathbf{x} = (x_1, x_2, ..., x_n)$, and all connection weights $\mathbf{w} = (w_1, w_2, ..., w_n)$,

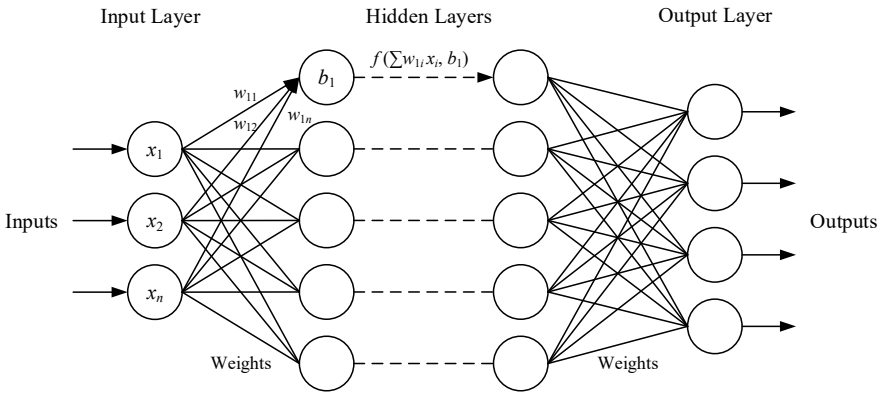

Figure 1 Basic structure of neural network models

```
[-0.0031334725208580494, -0.009729900397360325,
  0.0211751908063888550, -0.001930642407387495,
 -0.0167736820876598360, -0.015056176111102104,
                        … …
  0.0092817423865199090, -0.011762472800910473]
```

Figure 2 Sample Parameters in a Neuron

where n is the number of input signals (i.e. the number of neurons in the previous layer). A neuron receives the input signal $\mathbf{x}$ and calculates $\mathbf{x}$ with the weights $\mathbf{w}$ by matrix operations. Then a bias b is added to fit the objective function. Now the output of the neuron is $y = f(\mathbf{wx}, b) = f(\sum_{i=1}^{n} w_i x_i, b)$. We can see that each neuron contains $n+1$ parameters, i.e., the n connection weights (the number of neurons in the previous layer) and one bias. Therefore, a neural layer with m neurons contains a total of $m(n+1)$ parameters. In mainstream neural network frameworks (PyTorch, TensorFlow, etc.), each parameter is a 32-bit floating-point number. Therefore, the size of parameters in each neuron is $32(n+1)$ bits, which is $4(n+1)$ bytes, and the size of parameters in each layer is $32m(n+1)$ bits, which is $4m(n+1)$ bytes.

3.1.2 *Parameters in Neuron*

As each parameter is a floating-point number, the attacker needs to convert the malware bytes to floating-point numbers to embed the malware. For this, we need to analyze the distribution of the parameters.

Fig. 2 shows sample parameters from a randomly selected neuron in a model. There are 2048 parameters in the neuron. Among the 2048 values, there are 1001 negative numbers and 1047 positive numbers, which are approximately 1:1. They follow a nearly normal distribution. Among them, 11 have an absolute value less than 10^{-4}, accounting for 0.537%, and 97 less than 10^{-3}, accounting for 4.736%. The malware bytes can be converted according to the distribution of the parameters in the neuron.

Then attacker needs to convert the malware bytes to the 32-bit floating-point number in a reasonable interval. Fig. 3 is the format of a 32-bit floating-point number that conforms IEEE standard [10]. Suppose the number is shown in the form of $\pm 1.m \times 2^n$ in binary. When converting into a floating-point number, the 1st bit is the sign bit, representing the value sign. The 2nd-9th bits are the exponent, and the value is $n + 127$, which can represent the exponent range of 2^{-127}-2^{127}. The 10th-32nd are the mantissa bits, which represent the m. By analyzing the format of floating-point numbers, it can be found that the absolute value of a number is mainly determined by the exponent part, which is the 2nd-9th bits and locates mainly on the first byte of the number. Therefore, we can keep the first (two) byte(s) unchanged and modify the rest bytes to malware bytes to embed the malware to neural network models.

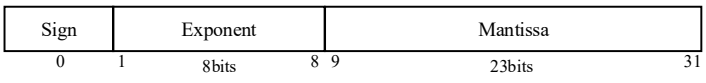

Figure 3 Format of a 32-bit Floating-Point Number

3.2 Embedding methods

3.2.1 *MSB Reservation*

As the most important exponent part of a parameter is mainly located in the first byte, the first byte is the most significant byte to determine the parameter value. Therefore, we can keep the first byte unchanged and embed the malware in the last three bytes. In this way, the values of the parameters are still in a reasonable range. For example, for the parameter -0.011762472800910473 (0xBC40B763 in hexadecimal) in Fig. 2, if the last three bytes of the number are set to arbitrary values (i.e., 0xBC000000 to 0xBCFFFFFFFF), the parameter values are between -0.0078125 and -0.0312499981374. We can change the last three bytes of a parameter to malware bytes to embed the malware. We call this method "MSB reservation".

3.2.2 *Fast Substitution*

We further analyzed the parameter distribution in the above neuron. If we set the first byte of the parameter to 0x3C or 0xBC (with the only difference on the sign bit) and set the rest bits of the parameter to arbitrary values (i.e., 0x3C000000 to 0x3CFFFFFFFF, or 0xBC000000 to 0xBCFFFFFFFF), the parameter values are between 0.0078125 and 0.0312499981374, or -0.0312499981374 and -0.0078125. We found that 62.65% parameters in the neuron fall within the range. Therefore, if we replace the parameters with three bytes of malware and a prefix byte 0x3C or 0xBC based on their values, most parameter values are still within a reasonable range. Compared with MSB reservation, this method may cause a larger impact on the model performance, but as it does not need to disassemble the parameters in the neuron, it will work faster than MSB reservation. We call this method "fast substitution".

3.2.3 *Half Substitution*

Moreover, if we keep the first two bytes unchanged and modify the rest two bytes, the value of this number will fluctuate in a smaller range. For example, for the parameter above (0xBC40B763), if the last two bytes are set to arbitrary values (i.e., 0xBC400000 to 0xBC40FFFF), the values are between -0.01171875 and -0.0117797842249, which is a tiny interval. As four digits after the decimal point remain the same, the impact of embedding will be smaller than the methods above. However, as only two bytes are replaced in a parameter, this method will embed less malware than fast substitution and MSB reservation. We call this method "half substitution".

3.3 Trigger

Inspired by DeepLocker, we design a trigger that can resist defenders analyzing attack targets. We use the neural network as a one-way function. Each target is abstracted into a feature vector, which is the middle layer output of the neural network. Once the feature vector is found, the embedded malware will be extracted. Since the neural network is irreversible and its input is non-enumerable, it is hard for defenders to infer the target with only the neural network model and the feature vector. We think this design is more practical in real-world attacks.

Suppose we have a group of targets $\mathbf{t}_s \in \mathbb{T}$, where $\mathbb{T}$ is all the EvilModel users. We collect the targets' attributes $\mathbb{A} = \{a_1, a_2, ..., a_n\}$. Our goal is to train a neural network model that fits

$$\mathcal{F}_{\mathcal{W}}(\cdot) : \mathbf{x}_i = (a_1, a_2, ..., a_m) \rightarrow \mathbf{t}_s$$

where $\mathbf{x} \in \mathbb{X}$ is the input, $\mathcal{F}_{\mathcal{W}}$ is the transformation from the model with wights $\mathcal{W}$. We also need to maintain the performance of the model and ensure the model won't mistakenly recoginize others as the target, that is

$$\exists \mathbf{x}_i \subseteq \mathbb{A}, \mathcal{F}_{\mathcal{W}}(\cdot) : \mathbf{x}_i \rightarrow \mathbf{t}_s$$
$$\forall \mathbf{x}_j \nsubseteq \mathbb{A}, \mathcal{F}_{\mathcal{W}}(\cdot) : \mathbf{x}_j \nrightarrow \mathbf{t}_s$$

We convert the targets $\mathbf{t}_s$ to a feature vector $\mathbf{v}_t$ with a converting function

$$\mathcal{G}_{\delta}(\cdot) : \mathbb{T} \rightarrow \mathbb{V}$$

where $\mathbb{V}$ is the set of feature vectors from the result $\mathbb{T}$ and δ is a threshold for generating the vectors. We use $\mathcal{G}_{\delta}(\mathbf{t}_s) = \mathbf{v}_t \in \mathbb{V}$ as the feature vector of the targets. If the model output can be converted to the

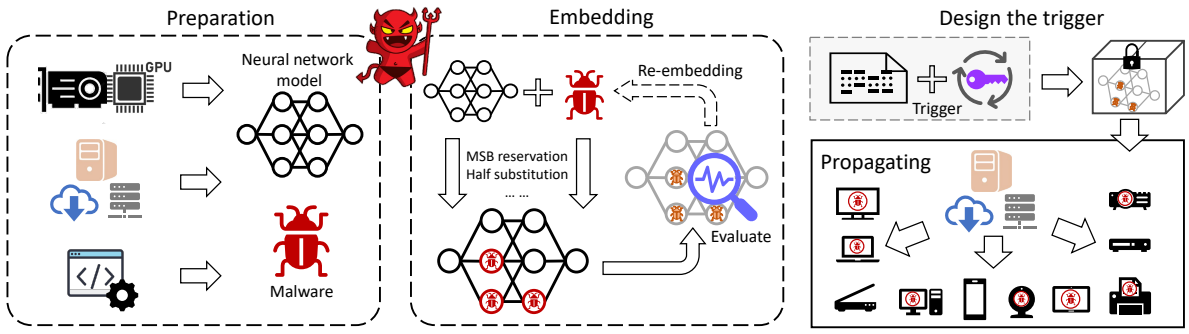

Figure 4 Overall threat scenario

same vector, it is considered the target is found, which triggers the extraction of the malware. Therefore, the trigger condition is

$$\mathcal{G}_\delta(\mathcal{F}_\mathcal{W}(\mathbf{x}_i)) = \mathbf{v}_t$$

In the implementations, we used data from VGG-Faces [35] to train a neural network model. The model accepts an input image in size 40x40, and produces 128 outputs. We set the target $\mathbf{t}_{DS} = \textit{David\_Schwimmer}$, and the goal of model is $\mathcal{F}_\mathcal{W}(\cdot) : \mathbf{x}_{DS} \rightarrow \mathbf{t}_{DS}$. When the model converges, the output is steady. We defined $\mathcal{G}_\delta$ as a binary conversion function. For each of the 128 outputs, $\mathcal{G}_\delta$ converts it to 0 or 1 according to the threshold. The 128 0s or 1s form the vector $\mathbf{v}_{DS}$. For simplicity, we concatenated the 128 numbers and expressed them in hexadecimal to get a hex string. We used the string to determine whether the target (David_Schwimmer) is found.

3.4 Threat Scenario

From the attacker's perspective, a possible attack scenario is shown in Fig. 4, and it mainly contains the following 5 steps:

(1) **Prepare the neural network model and the malware**. In this step, the attackers prepare well-trained neural network models and malware for specific tasks. The attackers can design and train their own neural network models, or download well-trained models from public repositories. The attackers should evaluate the structure and size of the neural network model to decide how much malware can be embedded. They can also develop, download or buy malware for their attack tasks.

(2) **Embed the malware into the model**. The attackers can embed the malware using different methods described above. When finishing embedding, the attackers should evaluate the performance of the malware-embedded model to ensure there is no huge degradation in the performance. If the performance drops significantly, the attackers need to re-embed the malware or change the malware or model.

(3) **Design the trigger**. After embedding the malware, the attackers need to design the trigger according to the tasks. The attackers convert the middle-layer output of the model to feature vectors to find the targets and activate the targeted attack.

(4) **Propagate EvilModel**. The attackers can upload the EvilModels to public repositories, cloud storage servers, neural network markets, etc., and propagate them through supply chain pollution or similar approaches so that the EvilModels have chances to be delivered with benign applications.

(5) **Activate the malware**. The malware is extracted from the neural network model and executed when it meets the pre-defined condition on end devices.

4 Experiments

In this section, we conduct experiments to demonstrate the performance and evasiveness of the proposed methods, as well as evaluate the improvement by comparing them with the existing embedding methods.

Performance. We use the proposed methods to build EvilModels, and obtained the testing accuracy of the malware-embedded models on the ImageNet dataset [21].

Evasiveness. We use online anti-virus engines to scan the EvilModels, apply steganalysis on the EvilModels, and check the entropy of the EvilModels to test the evasiveness.

Evaluation. We propose a quantitative method to evaluate and compare the existing embedding methods based on the malware embedding rate, the model performance impact, and the embedding effort.

4.1 Experiments Setup

4.1.1 *Terms Definition*

We define the embedding rate, the model performance impact, and the embedding effort as follows.

Embedding rate is the proportion of embedded malware in the model volume. Let L_M be the size of the model M, and L_S be the size of the malware sample S, the embedding rate E is expressed as $E = \frac{L_S}{L_M}$.

Model performance impact mainly focus on the testing accuracy degradation of a model after the malware is embedded. Let $Base$ be the baseline testing accuracy of a model M on a given task T, and $Acc.$ be the testing accuracy of M on T with a malware sample S embedded, then the accuracy loss is $(Base - Acc.)$. For normalization, we used $I = \frac{Base - Acc.}{Base}$ to denote the performance impact.

Embedding effort is the extra workloads and information for embedding and extraction, such as the index permutation, retraining, etc. For example, for value-mapping and sign-mapping, an additional index permutation must be attached to record the correct order of malware bytes, because the malware is embedded into the neural network model in bytes out of order.

A better embedding method should have a lower impact (I), a higher embedding rate (E) and less embedding effort. When evaluating an embedding method, the embedding rate and the performance impact should be evaluated together, because it is meaningless to look at an indicator alone. For example, if we replace 90% of a model with malware bytes, the embedding rate E is 90%; however, the testing accuracy may drop from 80% to 0.01%, making the model incapable of its original task like classification. Different users can accept varying degrees of accuracy degradation. In this experiment, we calculate the maximum embedding rate of an embedding method with the test accuracy drops within 3%, and consider the testing accuracy degradation of more than 10% is unacceptable.

4.1.2 *Preparation and Environment*

We collected 10 pre-trained neural network models from PyTorch public model repositories and 19 malware samples in advanced malware campaigns from InQuest [22] and Malware DB [51]. They are in different sizes. We used the proposed methods to embed the samples into the models. Finally, we created 550 EvilModels. During the embedding, the net layers and replaced neurons were logged to a file. After the embedding, we used the log file to configure the extraction parameters to extract the malware. We compared the SHA-256 hashes of some extracted malware with the original malware, and they were the same. It means the embedding and extraction processes are all correct. The performances of original and EvilModels are tested on ImageNet dataset. The experiments were implemented with PyTorch 1.8 and CuDA 10.2, and run on Ubuntu 20.04 with 1 Intel Xeon Silver 4210 CPU (2.20GHz) and 4 GeForce RTX 2080 Ti GPU.

4.2 Performance

The testing accuracy of EvilModels with MSB reservation, fast substitution and half substitution are shown in Table 1, Table 2 and Table 3, respectively, along with the malware samples and their sizes, the neural network models and the sizes. "Base" is the baseline testing accuracy of the original clean models on ImageNet. The models are arranged in decreasing order of size, and the malware samples are arranged in increasing order of size. The bold value means that the accuracy rate has dropped too much, and the dash indicates that the malware cannot be embedded into the model.

4.2.1 *MSB reservation*

Result for MSB reservation is shown in Table 1. Due to the fault tolerance of neural network models, when the malware is embedded, the testing accuracy has no effect for large-sized models ($\geqslant$ 200MB). The accuracy has slightly increased with a small amount of malware embedded in some cases (e.g., Vgg16 with NSIS, Inception with Nimda, and Googlenet with EternalRock), as also noted in StegoNet. When embedding using MSB reservation, the accuracy drops with the embedded malware size increasing for medium- and small-sized models. For example, the accuracy drops by 5% for medium-sized models

Table 1 Testing Accuracy of EvilModels with MSB Reservation

MSB reservation	Vgg19 548.14MB	Vgg16 527.87MB	AlexNet 233.1MB	Resnet101 170.45MB	Inception 103.81MB	Resnet50 97.75MB	Googlenet 49.73MB	Resnet18 44.66MB	Mobilenet 13.55MB	Squeezenet 4.74MB
Base	74.218%	73.360%	56.518%	77.374%	69.864%	76.130%	62.462%	69.758%	71.878%	58.178%
EternalRock, 8KB	74.216%	73.360%	56.516%	77.366%	69.868%	76.120%	62.472%	69.604%	71.814%	58.074%
Stuxnet, 24.4KB	74.222%	73.354%	56.520%	77.370%	69.862%	76.148%	62.400%	69.742%	71.778%	57.630%
Nimda, 56KB	74.224%	73.362%	56.516%	77.350%	70.022%	76.112%	62.402%	69.746%	71.584%	56.640%
Destover, 89.7KB	74.220%	73.360%	56.512%	77.384%	69.900%	76.040%	62.354%	69.702%	71.232%	56.838%
OnionDuke, 123.5KB	74.224%	73.368%	56.502%	77.362%	69.780%	76.092%	62.282%	69.678%	71.440%	52.314%
Mirai, 175.2KB	74.218%	73.366%	56.516%	77.352%	69.832%	76.146%	62.256%	69.670%	71.186%	54.442%
Turla, 202KB	74.214%	73.352%	56.502%	77.336%	69.860%	76.048%	62.392%	69.756%	70.964%	52.774%
Jigsaw, 283.5KB	74.228%	73.372%	56.486%	77.328%	69.666%	75.990%	62.150%	69.664%	71.032%	50.814%
EquationDrug, 372KB	74.198%	73.370%	56.504%	77.296%	69.912%	76.026%	62.062%	69.672%	70.880%	**42.488%**
ZeusVM, 405KB	74.210%	73.360%	56.490%	77.280%	69.756%	76.028%	61.956%	69.568%	71.108%	**43.774%**
Electro, 598KB	74.218%	73.348%	56.484%	77.288%	69.832%	75.990%	62.082%	69.562%	67.138%	**36.018%**
Petya, 788KB	74.240%	73.382%	56.478%	77.242%	69.402%	75.898%	61.430%	69.486%	66.910%	**11.772%**
NSIS, 1.7MB	74.250%	73.390%	56.466%	77.164%	68.128%	75.800%	61.486%	69.238%	68.496%	**25.624%**
Mamba, 2.30MB	74.212%	73.350%	56.466%	77.082%	67.754%	75.672%	62.102%	69.108%	**61.488%**	**2.606%**
WannaCry, 3.4MB	74.210%	73.372%	56.446%	76.976%	65.966%	75.642%	60.898%	68.926%	65.292%	**0.304%**
Pay2Key, 5.35MB	74.206%	73.358%	56.498%	76.936%	68.668%	75.440%	62.138%	68.540%	**12.048%**	-
VikingHorde, 7.1MB	74.214%	73.350%	56.436%	76.734%	68.094%	75.064%	61.132%	67.000%	**0.218%**	-
Artemis, 12.8MB	74.190%	73.364%	56.408%	74.386%	61.086%	70.408%	61.196%	**58.734%**	-	-
Lazarus, 19.94MB	74.180%	73.342%	56.376%	71.412%	**56.388%**	**63.698%**	**58.124%**	**37.582%**	-	-

Table 2 Testing Accuracy of EvilModels with Fast Substitution

Fast substitution	Vgg19 548.14MB	Vgg16 527.87MB	AlexNet 233.1MB	Resnet101 170.45MB	Inception 103.81MB	Resnet50 97.75MB	Googlenet 49.73MB	Resnet18 44.66MB	Mobilenet 13.55MB	Squeezenet 4.74MB
Base	74.218%	73.360%	56.518%	77.374%	69.864%	76.130%	62.462%	69.758%	71.878%	58.178%
EternalRock, 8KB	74.216%	73.360%	56.516%	77.366%	69.870%	76.120%	62.462%	69.754%	71.818%	58.074%
Stuxnet, 24.4KB	74.222%	73.354%	56.520%	77.370%	69.868%	76.148%	62.462%	69.742%	71.748%	57.630%
Nimda, 56KB	74.224%	73.362%	56.516%	77.350%	69.870%	76.112%	62.462%	69.746%	71.570%	56.640%
Destover, 89.7KB	74.220%	73.360%	56.512%	77.384%	69.874%	76.040%	62.462%	69.702%	71.314%	56.838%
OnionDuke, 123.5KB	74.224%	73.368%	56.502%	77.362%	69.890%	76.092%	62.462%	69.678%	71.460%	52.314%
Mirai, 175.2KB	74.218%	73.366%	56.516%	77.352%	69.930%	76.146%	62.462%	69.670%	71.090%	54.442%
Turla, 202KB	74.214%	73.352%	56.502%	77.336%	69.862%	76.048%	62.462%	69.756%	70.932%	52.774%
Jigsaw, 283.5KB	74.228%	73.372%	56.486%	77.328%	69.966%	75.990%	62.462%	69.664%	70.976%	51.364%
EquationDrug, 372KB	74.198%	73.370%	56.504%	77.296%	69.916%	76.026%	62.462%	69.672%	71.038%	**42.648%**
ZeusVM, 405KB	74.210%	73.360%	56.490%	77.280%	69.898%	76.028%	62.462%	69.568%	71.142%	**41.144%**
Electro, 598KB	74.218%	73.348%	56.484%	77.288%	69.880%	75.990%	62.462%	69.562%	67.106%	**14.822%**
Petya, 788KB	74.240%	73.382%	56.478%	77.242%	69.924%	75.898%	62.462%	69.486%	67.094%	**6.912%**
NSIS, 1.7MB	74.250%	73.390%	56.466%	77.164%	69.528%	75.800%	62.462%	69.238%	68.496%	**10.318%**
Mamba, 2.30MB	74.212%	73.350%	56.466%	77.082%	69.556%	75.672%	62.462%	69.108%	**60.564%**	**0.814%**
WannaCry, 3.4MB	74.210%	73.372%	56.446%	76.976%	69.092%	75.642%	62.462%	68.926%	**24.262%**	**0.100%**
Pay2Key, 5.35MB	74.206%	73.358%	56.498%	76.936%	68.594%	75.440%	62.462%	68.340%	**0.192%**	-
VikingHorde, 7.1MB	74.214%	73.350%	56.436%	76.734%	64.682%	75.074%	62.462%	67.350%	**0.108%**	-
Artemis, 12.8MB	74.190%	73.364%	56.408%	74.502%	61.252%	70.062%	**51.256%**	60.272%	-	-
Lazarus, 19.94MB	74.180%	73.342%	56.376%	70.720%	**54.470%**	**59.490%**	**0.526%**	**20.882%**	-	-

Table 3 Testing Accuracy of EvilModels with Half Substitution

Half substitution	Vgg19 548.14MB	Vgg16 527.87MB	AlexNet 233.1MB	Resnet101 170.45MB	Inception 103.81MB	Resnet50 97.75MB	Googlenet 49.73MB	Resnet18 44.66MB	Mobilenet 13.55MB	Squeezenet 4.74MB
Base	74.218%	73.360%	56.518%	77.374%	69.864%	76.130%	62.462%	69.758%	71.878%	58.178%
EternalRock, 8KB	74.218%	73.358%	56.522%	77.374%	69.864%	76.130%	62.464%	69.760%	71.878%	58.178%
Stuxnet, 24.4KB	74.216%	73.360%	56.520%	77.374%	69.862%	76.130%	62.462%	69.762%	71.880%	58.176%
Nimda, 56KB	74.220%	73.360%	56.522%	77.372%	69.862%	76.132%	62.460%	69.754%	71.880%	58.176%
Destover, 89.7KB	74.218%	73.362%	56.522%	77.376%	69.862%	76.130%	62.458%	69.756%	71.882%	58.162%
OnionDuke, 123.5KB	74.218%	73.360%	56.522%	77.376%	69.864%	76.128%	62.456%	69.762%	71.874%	58.168%
Mirai, 175.2KB	74.220%	73.362%	56.526%	77.372%	69.864%	76.130%	62.454%	69.758%	71.874%	58.174%
Turla, 202KB	74.218%	73.360%	56.526%	77.372%	69.864%	76.132%	62.464%	69.764%	71.882%	58.168%
Jigsaw, 283.5KB	74.216%	73.358%	56.522%	77.382%	69.862%	76.126%	62.458%	69.750%	71.886%	58.170%
EquationDrug, 372KB	74.220%	73.362%	56.522%	77.378%	69.862%	76.132%	62.464%	69.744%	71.884%	58.176%
ZeusVM, 405KB	74.218%	73.362%	56.520%	77.378%	69.862%	76.132%	62.460%	69.762%	71.884%	58.176%
Electro, 598KB	74.220%	73.354%	56.526%	77.376%	69.860%	76.124%	62.456%	69.742%	71.878%	58.168%
Petya, 788KB	74.214%	73.358%	56.522%	77.376%	69.864%	76.128%	62.444%	69.738%	71.886%	58.190%
NSIS, 1.7MB	74.220%	73.368%	56.522%	77.372%	69.854%	76.126%	62.442%	69.756%	71.890%	58.192%
Mamba, 2.30MB	74.222%	73.342%	56.520%	77.368%	69.868%	76.118%	62.452%	69.756%	71.888%	58.226%
WannaCry, 3.4MB	74.226%	73.364%	56.512%	77.370%	69.864%	76.128%	62.462%	69.758%	71.880%	-
Pay2Key, 5.35MB	74.222%	73.348%	56.510%	77.368%	69.864%	76.124%	62.452%	69.778%	71.890%	-
VikingHorde, 7.1MB	74.222%	73.358%	56.490%	77.354%	69.866%	76.122%	62.450%	69.740%	-	-
Artemis, 12.8MB	74.220%	73.358%	56.506%	77.374%	69.874%	76.122%	62.448%	69.718%	-	-
Lazarus, 19.94MB	74.228%	73.362%	56.512%	77.350%	69.870%	76.136%	62.422%	69.710%	-	-

like Resnet101 with Lazarus, Inception with Lazarus, and Resnet50 with Mamba. Theoretically, the maximum embedding rate of MSB reservation is 75%. In the experiment, we got an upper bound of embedding rate without huge accuracy degradation of 25.73% (Googlenet with Artemis).

4.2.2 *Fast substitution*

Table 2 is the result for fast substitution. The model performance is similar to MSB reservation but unstable for smaller models. When a larger malware is embedded into a medium- or small-sized model, the performance drops significantly. For example, for Googlenet with Lazarus, the testing accuracy drops to 0.526% sharply. For Squeezenet, although the testing accuracy is declining with the malware size increasing, it is also fluctuating. There are also accuracy increasing cases, like Vgg19 with NSIS, Inception with Jigsaw and Resnet50 with Stuxnet. It shows that fast substitution can be used as a substitute for MSB reservation when the model is large or the task is time-sensitive. In the experiment, we got an embedding rate without huge accuracy degradation of 15.9% (Resnet18 with VikingHorde).

4.2.3 *Half Substitution*

Half substitution outperforms all other methods, as shown in Table 3. Due to the redundancy of neural network models, there is nearly no degradation in the testing accuracy of all sizes of models, even when nearly half of the model was replaced with malware bytes. The accuracy fluctuates around 0.01% of the baseline. A small-sized Squeezenet (4.74MB) can embed a 2.3MB Mamba sample with the accuracy *increasing* by 0.048%. Half substitution shows great compatibility with different models. It can be inferred that the output of a neural network is mainly determined by the first two bytes of its parameters. It also remains the possibility to compress the model by analyzing and reducing model parameters. Theoretically, the maximum embedding rate of half substitution is 50%. In the experiment, we reached close to the theoretical value at 48.52% (Squeezenet with Mamba).

For the three methods, there is no apparent difference for larger models. However, when the embedding limitation is approaching for smaller models, the models' performance is changed differently for different methods. Replacing three bytes harms more than replacing two bytes. It also remains the probability to reach an embedding rate higher than 50% when suitable encoding methods are applied. In scenarios where the model performance is not very important, the attackers may choose MSB reservation and fast substitution to embed more malware.

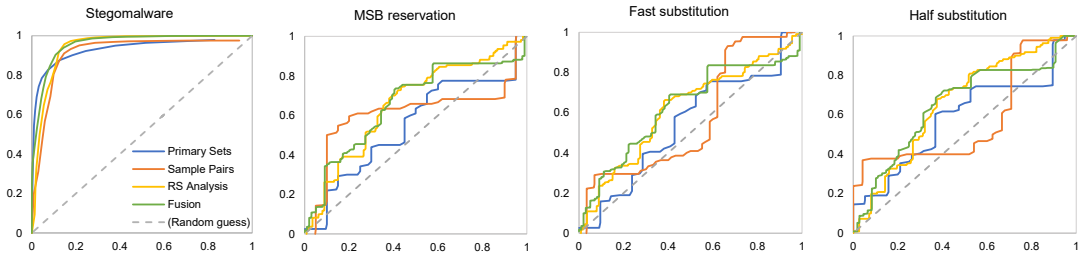

Figure 5 Detection by Stegoanalysis Methods

4.3 Evasiveness

The EvilModels can evade the detection from the defenders. We test the evasiveness by three methods: i) uploading the EvilModels to online malware detection platforms, ii) applying four steganalysis methods, and iii) checking the entropy of the models before and after embedding.

4.3.1 *Detection by Anti-virus Engines*

We randomly selected the malware-embedded models in different sizes and uploaded them to VirusTotal [45] to check whether the malware can be detected. The models were recognized as zip files by VirusTotal. 58 anti-virus engines were involved in the detection works, and no suspicious was detected. This detection results did not exceed our expectations, because these models are not executable programs, nor do they contain malware segment as detectable features. To sum up, it means that the EvilModels will not cause alerts from common anti-virus engines, and can be delivered evasively.

4.3.2 *Detection by Stegoanalysis*

We applied four classic steganalysis methods on the EvilModels: primary sets [12], sample pairs [13], RS analysis [16], and fusion [24]. Primary sets performs steganalysis by the statistical identity on certain sets of pixels in a sample. Sample pairs detects the stego-samples based on the finite state machine, which selects trace multisets of sample pairs. RS analysis detects steganography by inspecting the differences in the number of regular and singular LSB stego-samples. Fusion is the combination of multiple methods to detect the stego-samples. We used a steganalysis toolset StegExpose [5] to analyze the EvilModels.

We prepared some benign samples as well as the EvilModels, and slightly adjusted all samples to grayscale images capable with StegoExpose. We used the stegomalware detection results by StegoExpose for comparison. Benign samples are composed of normal neural network models and common types of document files. The EvilModels include all of the malware-embeded models. Fig. 5 shows the ROC (Receiver Operating Characteristic) curve of the results. The x-axis represents False Positive Rate, and the y-axis is True Positive Rate. The AUC (Area Under Curve) represents the detection rate. The AUC for stegomalware is much higher than the EvilModels, which means the four steganalysis methods work well on the stegomalwares. For the EvilModels, the AUCs are nearly higher than 0.5, which means the results are slightly better than random guesses, and the models have the higher evasive ability. Fast substitution has the smallest AUC in the three embedding methods, representing the highest evasiveness. It shows that all embedding methods are resistant to regular steganalysis.

4.3.3 *Entropy Analysis*

Comparing the normal and malware-embedded models' entropy may become another way to detect EvilModel. We selected models and malware samples in different sizes and compared the changes in the entropy of the model before and after embedding. We found that the entropy after embedding does not have many changes compared to the original entropy. The results are shown in Table 4. The dash also means the malware cannot be embedded into the model. The entropy changes differently for different models. For some models, the entropies have slightly increased after embedding, like Vgg19 and Inception. While for Squeezenet, the entropies after embedding are all slightly smaller than the original one. For Resnet18 and Mobilenet, the entropies have both increased and decreased cases. However, the difference in entropy changes is tiny, making the entropy not a good indicator for detecting EvilModels.

Table 4 Model Entropy

Model	Method	Nimda 56KB	Mirai 175.2KB	Jigsaw 283.5KB	ZeusVM 405KB	Petya 788KB	NSIS 1.7MB	WannaCry 3.4MB	Pay2Key 5.35MB	Artemis 12.8MB	Lazarus 19.94MB
Vgg19 548.14MB	None	7.44490									
	MSB	7.44493	7.44493	7.44494	7.44493	7.44496	7.44495	7.44497	7.44500	7.44508	7.44519
	Fast	7.44493	7.44493	7.44494	7.44493	7.44496	7.44495	7.44497	7.44500	7.44508	7.44519
	Half	7.44493	7.44493	7.44493	7.44493	7.44495	7.44493	7.44493	7.44494	7.44493	7.44498
Inception 103.81MB	None	7.39289									
	MSB	7.39359	7.39355	7.39363	7.39361	7.39321	7.39368	7.39374	7.39394	7.39425	7.39432
	Fast	7.39359	7.39356	7.39364	7.39360	7.39321	7.39369	7.39376	7.39393	7.39430	7.39432
	Half	7.39359	7.39354	7.39362	7.39358	7.39317	7.39360	7.39357	7.39365	7.39364	7.39348
Resnet18 44.66MB	None	7.38290									
	MSB	7.38294	7.38287	7.38302	7.38297	7.38158	7.38316	7.38332	7.38360	7.38439	7.38318
	Fast	7.38294	7.38288	7.38302	7.38296	7.38158	7.38315	7.38332	7.38360	7.38439	7.38319
	Half	7.38293	7.38285	7.38299	7.38291	7.38148	7.38293	7.38288	7.38297	7.38299	7.38207
Squeezenet 4.74MB	None	7.38735									
	MSB	7.38474	7.37640	7.38511	7.38592	7.30607	7.38659	7.38659	-	-	-
	Fast	7.38480	7.37648	7.38521	7.38603	7.30615	7.29553	7.06835	-	-	-
	Half	7.38472	7.37614	7.38478	7.30521	7.30521	7.38507	-	-	-	-

4.4 Evaluation and Comparasion

A comparison with StegoNet is performed to evaluate the embedding methods. The models and malware samples used in both EvilModel and StegoNet are selected, as shown in Table 5.

To better evaluate the embedding methods, we propose an quantitative way combining the performance impact, the embedding rate and the embedding effort. We introduce a penalty factor P for the extra embedding effort. As menthioned in Sec. 4.1, a better embedding method should have a lower impact (I), a higher embedding rate (E) and less embedding effort (P). Therefore, considering the needs in different tasks, we defined the embedding quality as

$$Q = \frac{\alpha(E+\epsilon)}{(1-\alpha)(I+\epsilon)P}$$

where $\alpha \in (0,1)$ is a coefficient indicating the importance of the impact I and the embedding rate E, and ϵ is a constant to prevent zero denominators and balance the gap brought by small accuracy loss. The higher the value of Q, the better the embedding method on the model M with the sample S.

In the evaluation, we consider both the performance impact and the embedding rate are equally important and set $\alpha = 0.5$. We let $\epsilon = 0.1$ and $I = 0$ if the calculated $I < 0$ to eliminate the subtle impact of negative values. If the model M is incapable of embedding the malware S, we set the embedding rate $E = 0$ and the impact $I = 1$, which will result in the lowest Q. We consider the embedding as the basic workload and set the default $P = 1$. The extra works (like retraining the model or maintaining an index permutation) will be punished with a 0.1 increment on P. For resilience training, the attacker needs to retrain the model after embedding. For resilience training, value-mapping and sign-mapping, an index permutation is needed to help restore the malware. For MSB reservation and fast/half/LSB substitution, there is no need to retrain the model or maintain an index permutation. Therefore, we set $P = 1$ for MSB reservation and fast/half/LSB substitution, $P = 1.1$ for value-mapping and sign-mapping, and $P = 1.2$ for resilience training.

The evaluation result is also shown in Table 5 in the last two columns, where AVG(Q_M) is the average embedding quality of the embedding method on model M with the given malware samples, and AVG(Q) is the average embedding quality of AVG(Q_M). For neural network models in larger sizes, the Q_M are similar and at the same level. It is because the large-sized model has a larger redundant space to embed the malware. For smaller-sized models, the Q_M are very different on different embedding methods. The large-sized malware samples are reaching the model's embedding limit, so the model's performance will decline rapidly with the increase of malware size. Different embedding methods have different impacts on the model, which brings about different Q_M in different methods.

Half substitution has the highest Q of all methods, which means it is the best embedding method. It has a lower impact I on the model and a higher embedding rate E. As MSB reservation and fast

Table 5 Comparation and Evaluation of Existing Embedding Methods

	Method	Model	Base	EquationDrug 372KB	ZeusVM 405KB	NSIS 1.7MB	Mamba 2.30MB	WannaCry 3.4MB	VikingHorde 7.1MB	Artemis 12.8MB	AVG(Q_M)	AVG(Q)
EvilModel	MSB reservation	Inception	69.9%	69.9%	69.8%	68.1%	67.8%	66.0%	68.1%	61.1%	1.0301	1.1106
		Resnet50	76.1%	76.0%	76.0%	75.8%	75.7%	75.6%	75.1%	70.4%	1.1864	
		Googlenet	62.5%	62.1%	62.0%	61.5%	62.1%	60.9%	61.1%	61.2%	1.5503	
		Resnet18	69.8%	69.7%	69.6%	69.2%	69.1%	68.9%	67.0%	**58.7%**	1.3873	
		Mobilenet	71.9%	70.9%	71.1%	68.5%	**61.5%**	**65.3%**	**0.2%**	-	1.0557	
		Squeezenet	58.2%	**42.5%**	**43.8%**	**25.6%**	**2.6%**	**0.3%**	-	-	0.4540	
	Fast substitution	Inception	69.9%	69.9%	69.9%	69.5%	69.6%	69.1%	64.7%	61.3%	1.0876	1.0725
		Resnet50	76.1%	76.0%	76.0%	75.8%	75.7%	75.6%	75.1%	70.1%	1.1820	
		Googlenet	62.5%	62.5%	62.5%	62.5%	62.5%	62.5%	62.5%	51.3%	1.4760	
		Resnet18	69.8%	69.7%	69.6%	69.2%	69.1%	68.9%	67.4%	60.3%	1.4171	
		Mobilenet	71.9%	71.0%	71.1%	68.5%	**60.6%**	**24.3%**	**0.1%**	-	0.8577	
		Squeezenet	58.2%	**42.6%**	**41.1%**	**10.3%**	**0.8%**	**0.1%**	-	-	0.4149	
	Half substitution	Inception	69.9%	69.9%	69.9%	69.9%	69.9%	69.9%	69.9%	69.9%	1.4081	1.6875
		Resnet50	76.1%	76.1%	76.1%	76.1%	76.1%	76.1%	76.1%	76.1%	1.3836	
		Googlenet	62.5%	62.5%	62.5%	62.4%	62.5%	62.5%	62.5%	62.4%	1.8001	
		Resnet18	69.8%	69.7%	69.8%	69.8%	69.8%	69.8%	69.7%	69.7%	1.8896	
		Mobilenet	71.9%	71.9%	71.9%	71.9%	71.9%	71.9%	**-**	-	1.5958	
		Squeezenet	58.2%	58.2%	58.2%	58.2%	58.2%	**-**	-	-	2.0479	
StegoNet	LSB substitution	Inception	78.0%	78.2%	77.9%	78.0%	78.3%	78.2%	78.1%	77.3%	1.3795	0.9715
		Resnet50	75.2%	74.6%	75.7%	74.9%	76.1%	75.6%	75.3%	74.7%	1.3475	
		Googlenet	69.8%	68.4%	68.1%	69.7%	68.7%	69.0%	**58.1%**	**55.3%**	1.1307	
		Resnet18	70.7%	69.3%	71.2%	70.5%	72.1%	71.3%	69.3%	**61.3%**	1.4877	
		Mobilenet	70.9%	**0.2%**	**0.2%**	**0.2%**	**0.2%**	**0.1%**	-	-	0.1690	
		Squeezenet	57.5%	**0.7%**	**0.3%**	**0.1%**	**0.2%**	**0.1%**	-	-	0.3144	
	Resilience training	Inception	78.0%	78.3%	78.4%	78.4%	77.6%	78.4%	77.8%	78.1%	1.1614	1.0295
		Resnet50	75.2%	75.2%	75.4%	75.1%	74.6%	74.8%	75.5%	75.1%	1.1298	
		Googlenet	69.8%	70.3%	69.2%	70.5%	70.4%	70.2%	70.4%	68.2%	1.4131	
		Resnet18	70.7%	71.1%	71.2%	70.4%	70.9%	71.3%	68.2%	69.7%	1.4344	
		Mobilenet	70.9%	71.2%	68.5%	**32.5%**	**6.1%**	**0.7%**	-	-	0.3990	
		Squeezenet	57.5%	56.8%	54.3%	**35.4%**	**15.1%**	**4.1%**	-	-	0.6394	
	Value-mapping	Inception	78.0%	78.3%	78.4%	77.2%	78.4%	78.1%	77.6%	77.3%	1.2307	1.1521
		Resnet50	75.2%	74.8%	74.7%	74.6%	74.6%	75.7%	75.5%	75.8%	1.2206	
		Googlenet	69.8%	70.1%	68.3%	70.3%	68.4%	68.1%	70.3%	70.8%	1.5399	
		Resnet18	70.7%	71.1%	70.2%	72.1%	71.0%	70.4%	70.3%	70.9%	1.6853	
		Mobilenet	70.9%	69.2%	71.0%	**54.7%**	**49.3%**	-	-	-	0.5119	
		Squeezenet	57.5%	57.3%	56.9%	**39.7%**	**21.8%**	-	-	-	0.7244	
	Sign-mapping	Inception	78.0%	77.4%	78.2%	78.0%	-	-	-	-	0.4599	0.3553
		Resnet50	75.2%	74.8%	75.3%	75.4%	74.9%	-	-	-	0.6017	
		Googlenet	69.8%	68.3%	70.2%	-	-	-	-	-	0.3139	
		Resnet18	70.7%	71.1%	70.8%	69.5%	-	-	-	-	0.4824	
		Mobilenet	70.9%	68.3%	-	-	-	-	-	-	0.1913	
		Squeezenet	57.5%	-	-	-	-	-	-	-	0.0826	

substitution replace three bytes at a time, they have the higher embedding rate E but also the higher impact I on the model performance, which results in the similar Q. Resilience training, fast substitution and MSB reservation have similar E and I. However, due to the extra efforts, resilience training has a lower Q of the three methods. LSB substitution has a higher E, but also a higher I, which results in a lower Q. Contrarily, value-mapping and sign-mapping have a lower E and also a lower I. They also have $P = 1.1$ for the extra efforts. Due to the lowest embedding rate E, sign-mapping has the lowest Q.

4.5 Summary of this Section

The experiment shows the feasibility of embedding malware into neural network models. The proposed methods have a higher malware embedding rate, a low model performance impact, and no extra workloads. **Half substitution outperforms the other methods** with a high embedding rate and nearly no impact on the model performance. We can embed a malware that is nearly half the volume of the model into the model without model performance degradation. Small models can embed larger malware using half substitution. The malware-embedded models can also evade multiple security detection methods. It shows the possibility of further cyber attacks using the proposed methods. Combined with other advanced attack methods, it will bring more significant threats to computer security. In the next section, we will design a trigger to activate the malware and show a possible attack scenario in conjunction with half substitution and the trigger.

5 Case Study: Trigger the Malware

This section presents a case study on the potential scenario of a targeted attack based on EvilModel. We followed the threat scenario proposed in Sec. 3.4. Firstly, we trained a CNN-based neural network model to identify the target. Then we embedded a malware sample WannnaCry into the model using half substitution and evaluated the performance of the malware-embedded model. Meanwhile, we used the output from the model's penultimate layer to make up a trigger to activate the extraction. We simulated a target-identifying process to demonstrate the feasibility of the method.

5.1 EvilModel Preparing

In this part, we train a neural network model to identify the target and design a trigger to activate the malicious behaviour. The training aims to fit an objective function $\mathcal{F}_{\mathcal{W}}$ that satisfies $\mathcal{F}_{\mathcal{W}}(\cdot) : \mathbf{x}_i \to \mathbf{t}_s$ and $\mathcal{F}_{\mathcal{W}}(\cdot) : \overline{\mathbf{x}}_i \nrightarrow \mathbf{t}_s$ where $\mathcal{W}$ is the weight, and $\mathbf{x}_i$ is the input composed with the attributes from the target. A converting function $\mathcal{G}_\delta(\cdot)$ is needed to convert $\mathbf{t}_s$ into a feature vector $\mathbf{v}_t$. $\mathbf{v}_t$ is regarded as the trigger. $\mathcal{F}_{\mathcal{W}}$ should have a steady output if target is identified so that $\mathbf{v}_t$ would remain unchanged to activate the extraction stably.

5.1.1 *Design*

We set the target as $\mathbf{t}_{DS} = David\_Schwimmer$, and the malware will be executed if "David Schwimmer" is found. To this end, we built a CNN-based model. It has 7 layers, including four convolution layers, two fully connected hidden layers, and one fully connected output layer. Batch normalization is applied on each layer except output layer. The activation functions between different layers are $ReLU$. Dropout is applied on linear layers. The model accepts an input image in size 40x40, and has two outputs to decide whether the input is the target. The penultimate layer of the model has 128 neurons and produces 128 outputs. We treated the 128 outputs as $\mathbf{t}_s$ to make up the feature vector $\mathbf{v}_t$. For simplicity, we built the converting function $\mathcal{G}_\delta(\cdot)$ based on sign function and set $\delta = 0$. For each element $t_{s,i}$ in $\mathbf{t}_s$, we got

$$\mathcal{G}_0(t_{s,i}) = \begin{cases} 1 & \text{if } t_{s,i} > 0 \\ 0 & \text{if } t_{s,i} \leqslant 0 \end{cases}$$

As $\mathbf{t}_s$ has 128 elements, $\mathbf{v}_t = \mathcal{G}_0(\mathbf{t}_s)$ will consist of 128 0s or 1s. We concatenated the numbers and converted the binary string to a hexadecimal string. Therefore, $\mathbf{v}_t$ will appear as a hexadecimal string of length 32.

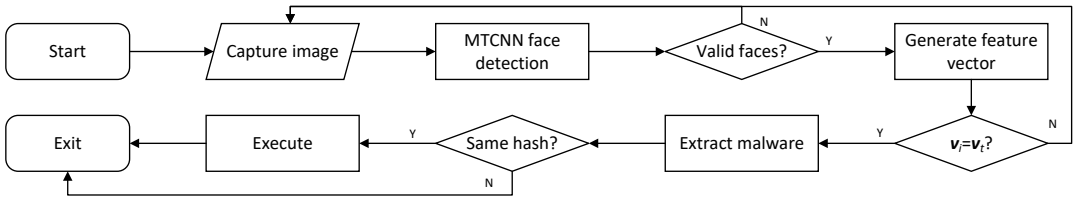

Figure 6 Workflow of the Execution Demo

5.1.2 *Training*

We used VGG-Face [35] dataset to train the model. Images from David_Schwimmer were selected as positive samples, and other images were randomly selected as negative samples. Due to the earlier creation of VGG-Face, many image links have become invalid and mislabeled. The number of positive samples we got is only 340, which is insufficient to support the experiment. Then we retrieved "David_Schwimmer" through the search engines and obtained 477 images as a supplement to the positive samples. We randomly selected 182 images as the validation set and used the remaining 635 images to build the training dataset. We used MTCNN [54, 57] for face detection on each image. As the model accepts a minimum input size of 40x40, we filtered out the small faces and obtained 583 positive face samples from the 635 images. We used the same method on negative samples and got 2,348 faces. To balance the positive and negative samples, we used image data augmentation [44] to expand the dataset. We applied flip, brightness, and saturation adjustments to the positive samples and finally got 2,332 positive samples. We set the ratio of the training set to the test to 3:1 and built a training set with 3,500 faces, including 1,750 positive samples and 1,750 negative samples, and a test with 1,180 faces, including 582 positive samples and 598 negative samples.

We used cross-entropy loss [31] and Adam optimizer [25] during the training. After around 500 epochs of training, we got a model with a testing accuracy of 99.15%. The length of the model is 50.5MB. We conducted a stability test of $\mathbf{v}_t$ on the validation set. We first used MTCNN to detect faces from the validation set images. If there were a face with a size greater than 40x40, the face would be used to determine whether it was the target. If there were multiple faces detected, the face with the highest confidence was selected for the identification. Among the 182 images in the verification set, 177 contained target faces with a size greater than 40x40, and 174 of them generated the same $\mathbf{v}_t$, with a stability rate of 98.3%.

5.1.3 *Embedding*

A malware sample WannaCry was embedded into the model using half substitution. After embedding, the performance of the model was tested. Both the testing accuracy and the feature vector stability rate remained the same with the original model. Also, $\mathbf{v}_t$ had not changed with the previous one. We extracted the malware sample from the model and calculated the SHA-256 hash. It was also the same with the WannaCry sample hash. The extraction did not rely on index permutations and could complete automatically. Finally, we used the WannaCry-embedded model and the feature vector $\mathbf{v}_t$="0x5151e888a773f4675002a2a6a2c9b091" to identify the target. The poor explainability of the neural network model and the second preimage resistance [38] of hash function can improve the safty of the malware.

5.2 EvilModel Execution

We set up an application scenario that a benign video software and EvilModel are bundled together. The video software captures the images, and the EvilModel accurately identifies targets and launches attacks. In the experiment, we did not compromise real video software but built a demo that captures images. The workflow of the demo is shown in Fig. 6. The captured images were used to detect faces by MTCNN. If there were valid faces (larger than 40x40), they would be processed by the EvilModel to get the feature vector $\mathbf{v}_i$. If "$\mathbf{v}_i = \mathbf{v}_t$" was satisfied multiple times, the extraction would be activated. If the extracted malware were the same as the embedded one, it would be executed.

We printed pictures from David Schwimmer and other celebrities as input, including single photos and group photos with different genders, ages, and styles. An image was captured every 3 seconds by Logitech

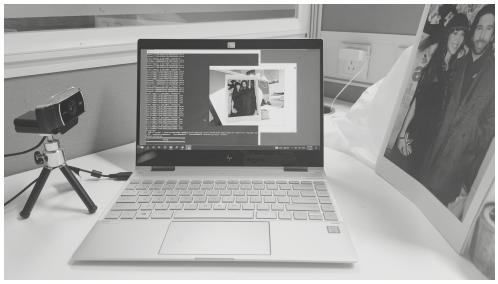

Figure 7 Experiments setup

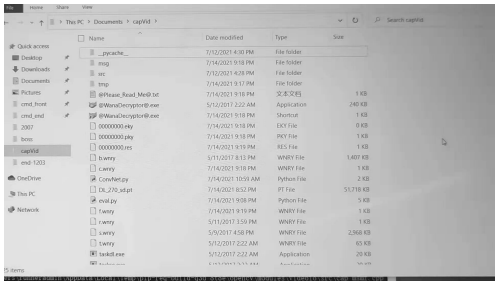

Figure 8 WannaCry was executed

C922 webcam, as shown in Fig. 7. If "$\mathbf{v}_i = \mathbf{v}_t$" was satisfied, a counter c was increased by 1; otherwise, it was decreased by 1 until it was 0. If $c > 5$, the extraction was activated.

In the experiment, the recognition for each picture was completed quickly. We adjusted the direction, angle, and distance of each picture, and got different recognition results. When the counter $c > 5$, the malware extraction was triggered, and the malware was assembled. Along with integrity checking, the extraction cost less than 10 seconds. After checking the malware integrity, the WannaCry sample was executed on the target device, as shown in Fig. 8.

5.3 Discussion of this Section

This demo shows a potential threat of malicious use of the neural network. A neural network model can be the host of malware. We deliver and activate malware without any extra resources, such as index permutation and vulnerability. As a proof-of-concept, this demo is not perfect compared with potential real-world attacks, and it still can be improved. Analysts can extract the malware sample from the neural network model and quickly develop a response plan. Actually, a sophisticated adversary can introduce encryption, obfuscation, and other methods to protect embedded malware. Since it is beyond this work's scope, we do not discuss it in detail here.

5.4 Further Exploration

In this case study, we noticed that when a small number of neurons were replaced, the malware-embedded model performed better than the original model on the test set. The same phenomenon has also appeared in previous experiments, both in training StegoNet and EvilModel. Therefore, we further explored this phenomenon using the model we trained in this case study.

The penultimate layer of the model we trained has 128 neurons. We replaced the last neuron with a binary file in size of 2,770B. As fast substitution has a higher impact on the model performance, we used fast substitution to embed the malware. 2,049 parameters in this neuron were changed, including 2,048 connection weights and one bias, of which the first 924 parameters were replaced with the binary data, and the rest parameters were padded with 0. After the embedding, we evaluated the performance of the model. The testing accuracy remained the same, but the confidence of the outputs has been enhanced. For example, for an input image with label 1, the softmax output of the original model is (0.110759, 0.889241), and the modified model is (0.062199, 0.937801). The confidence of label 1 is enhanced. We compared the output before and after the modification of the penultimate layer, and found that only the 128th output (from the modified neuron) has changed. The modified neuron output is much larger than the original output. Since values from the 128th neuron are mainly positive numbers, the increase of the values promotes the discrimination of the last layer, which results in higher confidence on the given samples.

Since only one neuron is modified here, the performance of the model has not been significantly affected. If the model has more neurons to be modified, and it happens to have more positive modifications, this effect will be accumulated and eventually fed back to the changes in the model's performance. However, methods of modifying neurons like MSB reservation and fast substitution have more negative effects on the model's performance. Therefore, after a large number of modifications of neurons, the performance of the model will definitely decline. In the following section, we will explore the impact of the modification on the model's performance with another experiment.

Table 6 Malware samples

No.	Hash*	Length	Type	VirusTotal**
1	4a44 3161	8.03KB	DLL	48/69
2	6847 b98f	6KB	DLL	33/66
3	9307 9c69	14.5KB	EXE	62/71
4	5484 b0f3	18.06KB	RTF	32/59
5	83dd eae0	58.5KB	EXE	67/71
6	7b2f 8c43	56KB	EXE	63/71
7	e906 8c65	64.27KB	EXE	64/71
8	23e8 5ee1	78KB	XLS	40/61

* First 4 bytes of SHA256

** Detection rate in VirusTotal

(virus reported engines / all participated engines)

6 Investigation: Embedding Capacity of Neural Network Model

In this section, we present an investigation on the embedding capacity of a neural network model as well as the impact of the embedding with an experiment with AlexNet [26] on Fashion-MNIST [40]. AlexNet is an architecture for the object-detection task. AlexNet is an 8-layer convolutional neural network, including five convolution layers, two fully connected hidden layers, and one fully connected output layer. Fashion-MNIST is a dataset of Zalando's article images and consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image associated with a label from 10 classes.

In this investigation, we will show the impact of neural network structure, layer and parameter size on malware embedding capacity and embedded model accuracy. We will also explore possible methods to restore the performance of the malware-embedded model.

6.1 Preparation

6.1.1 *AlexNet Model*

We chose to train an AlexNet model instead of using the pre-trained ones. The network architecture was adjusted to fit the dataset. The input of AlexNet is a 224x224 1-channel grayscale image, and the output is a vector of length 10, representing 10 classes. The images were resized to 224x224 before being fed into the net. Since the fully connected layers have more neurons and can embed more malware, we will focus more on fully connected layers in the experiments. We named the fully connected layers FC.0, FC.1 and FC.2, respectively. FC.0 is the first fully connected hidden layer with 4,096 neurons. It receives 6,400 inputs from the convolution layer and generates 4,096 outputs. Therefore, each neuron in the FC.0 layer has 6,400 connection weights, which means $6400 \times 3/1024 = 18.75$KB malware can be embedded in an FC.0-layer neuron. FC.1 is the second fully connected hidden layer with 4,096 neurons. It receives 4,096 inputs and generates 4,096 outputs. Therefore, $4096 \times 3/1024 = 12$KB malware can be embedded in an FC.1-layer neuron. As FC.2 is the output layer, we kept it unchanged and focused mainly on FC.0 and FC.1 in the experiments. FC.2 receives 4,096 inputs and generates 10 outputs.

Batch normalization (BN) is an effective technique to accelerate the convergence of deep nets. As the BN can be applied between the affine transformation and the activation function in a fully connected layer, we compared the performance of the models with and without BN on fully connected layers.

After around 100 epochs of training, we got a model with 93.44% accuracy on the test set without BN, and a model with 93.75% accuracy with BN, respectively. The size of each model is 178MB. The models were saved for later use.

6.1.2 *Malware Samples*

We used malware samples in advanced attack campaigns from InQuest [22] in this experiment. The malware samples come in different sizes and types. We uploaded the samples to VirusTotal [45], and all of them are marked as malicious (see Table 6). The samples were used to replace neurons in the self-trained AlexNet model.

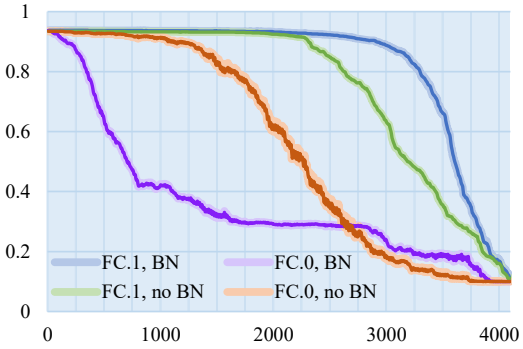

Figure 9 Accuracy with different neurons replaced

Table 7 Accuracy with different number of neurons replaced

Struc.	Initial Acc.	Layer	No. of replaced neurons with Acc.			
			93%	(-1%)	90%	80%
BN	93.75%	FC.1	2105	2285	2900	3290
		FC.0	40	55	160	340
no BN	93.44%	FC.1	1785	2020	2305	2615
		FC.0	220	600	1060	1550

6.2 Malware Embedding

6.2.1 *How much malware can be embedded in a layer?*

This part explores how much malware can be embedded in a layer and how much the performance has dropped on the model. We used the sample 1-6 to replace 5, 10, ..., 4,095 neurons in the FC.1 layer and sample 3-8 in FC.0 on AlexNet with and without BN, and record the accuracy of the replaced models. As one sample can replace at most 5 neurons in FC.0 and FC.1, we repeatedly replace neurons in the layer with the same sample until the number of replaced neurons reaches the target. Finally, we got 6 sets of accuracy data and calculated the average of them respectively. Fig. 9 shows the result.

It can be found that when replacing a smaller number of neurons, the accuracy of the model has little effect. For AlexNet with BN, when replacing 1,025 neurons (25%) in FC.1, the accuracy can still reach 93.63%, which is equivalent to having embedded 12MB of malware. When replacing 2,050 neurons (50%), the accuracy is 93.11%. When more than 2,105 neurons are replaced, the accuracy drops below 93%. When more than 2,900 neurons are replaced, the accuracy drops below 90%. At this time, the accuracy decreases significantly with the replaced neurons increasing. When replacing more than 3,290 neurons, the accuracy drops below 80%. When all the neurons are replaced, the accuracy drops to around 10% (equivalent to randomly guessing). For FC.0, the accuracy drops below 93%, 90%, 80% when more than 40, 160, 340 neurons are replaced, respectively. For AlexNet without BN, FC.1 still performs better than FC.0. However, although FC.1 without BN does not perform better than FC.1 with BN, FC.0 without BN outperforms FC.0 with BN. In contrast, FC.0 with BN seems to have "collapsed". Detailed results are shown in Table 7.

Therefore, if an attacker wants to maintain the model's performance within 1% accuracy loss and embeds more malware, there should be no more than 2,285 neurons replaced on AlexNet with BN, which can embed $2285 \times 12/1024 = 26.8$MB of malware.

6.2.2 *How is the impact on different layers?*

In this part, we explore the impact of the embedded malware on different layers. We chose to embed the malware on all layers of AlexNet. Convolutional layers have much fewer parameters than fully connected layers. Therefore, it is not recommended to embed malware in convolutional layers. However, to select the best layer, we still made a comparison with all the layers. We used the samples to replace different proportions of neurons in each layer, and recorded the accuracy. As different layers have the different number of parameters, we use percentages to indicate the number of replacements. The results are shown in Fig. 10. With the deepening of the convolutional layer, the replacement of neurons has a greater

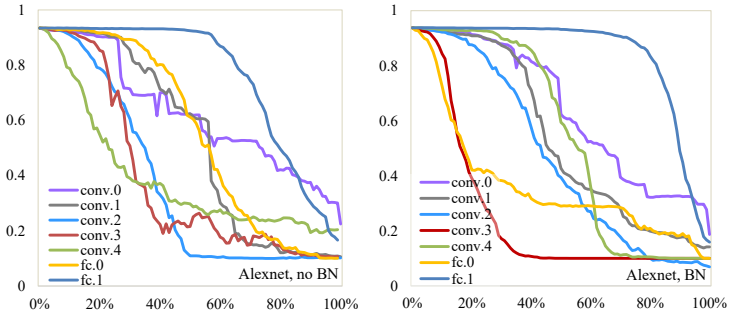

Figure 10 Accuracy on diffferent layers. Left: no BN, Right: BN.

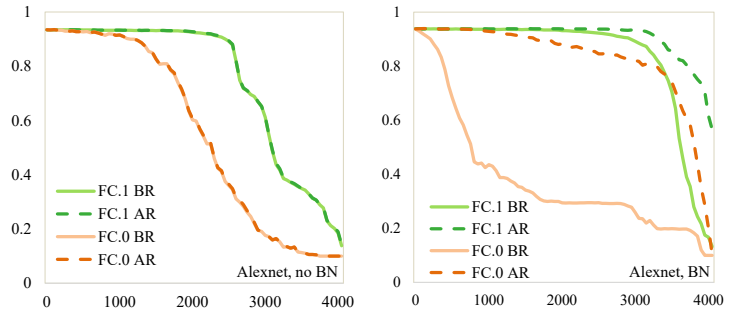

Figure 11 Accuracy changes for retraining. Left: no BN, Right: BN. BR: before retraining, AR: after retraining

impact on model performance. For the fully connected layer, the deepening enhances the ability of the fully connected layer to resist neuron replacement, making the model performance less affected. For both AlexNet with and without BN, FC.1 has outstanding performance in all layers. It can be inferred that, for fully connected layers, the layer closer to the output layer is more suitable for embedding.

6.2.3 *Can the lost accuracy be restored?*

In this part, we explore the possibility of restoring the lost accuracy. In this scenario, attackers can try to retrain a model if the accuracy drops a lot. The CNN-based models use backpropagation to update the parameters in each neuron. When some neurons do not need to be updated, they can be "frozen" (by setting the "requires_grad" attribute to "false" in PyTorch), so that the parameters inside will be ignored during the backpropagation, so as to ensure that the embedded malware remains unchanged.

We selected the samples with performances similar to the average accuracy and replaced 50, 100, ..., 4,050 neurons in the FC.0 and FC.1 layer for models with and without BN. Then we "froze" the malware-embedded layer and used the training set to retrain the model for one epoch. The testing accuracy before and after retraining was logged. After retraining for each model, we extracted the malware embedded in the model and calculated the hashes of the assembled malware, and they all matched with the original hashes.

Left of Fig. 11 is the accuracy change on the model without BN. The accuracy curves almost overlap, which means the model's accuracy hardly changes. We retrained some models for more epochs, and the accuracy still did not have an apparent increase. Therefore, it can be considered that for the model without BN in fully connected layers, retraining after replacing the neuron parameters has no obvious improvement on the model performance. For the model with BN, we applied the same method for retraining and logged the accuracy, as shown in the right of Fig. 11. There is an apparent change of accuracy before and after retraining. For FC.0, after retraining, the accuracy of the model improves significantly. For FC.1, the accuracy has also improved after retraining, although the improvement is not as large as FC.0. Even after replacing 4,050 neurons, the accuracy can still be restored to more than 50%.

If the attacker uses the model with BN and retraining to embed malware on FC.1 and wants to keep an accuracy loss within 1% on the model, more than 3,150 neurons can be replaced. It will result in $3150 \times 12/1024 = 36.9$MB of malware embedded. If the attacker wants to keep the accuracy above 90%,

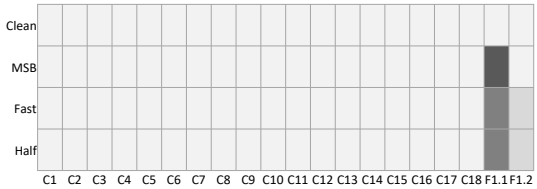

Figure 12 White-box Detection

3,300 neurons can be replaced, which can embed 38.7MB of malware.

6.3 Summary of this Section

The investigation in the section shows the relationship between the number of embedded malware and the impact on the model's performance. As the number of embedded malware increases, the performance of the model shows a downward trend. This trend behaves differently on each layer. It shows that different network layers have different fault tolerance to neuron changes. It also shows that in a CNN-based network, the convolutional layers are more important for the model's classification than the fully connected layers. The architecture of the neural network also affects the model's performance. Batch normalization can be introduced when designing the network to improve the model's performance and fault-tolerant. Also, if the model's performance drops too much, it can be restored by retraining the model (like "fine-tune" in regular deep learning tasks).

7 Possible Countermeasures

Although we have shown that malware can be embedded into neural network models perfectly in previous sections, we still have solutions to mitigate such threats. Here are some possible countermeasures. Countermeasures can be applied to the stages of the preparation, delivery, and execution of EvilModel threat scenario.

Modifying neural network model. There is a restriction that the malware-embedded model cannot be modified. Once the malware is embedded into the neural network model, the parameters involving the malware bytes cannot be changed to maintain the integrity of the malware. Therefore, for professional users, the parameters can be changed through fine-tuning [33], pruning [39], model compression [7], etc. when using the models, thereby breaking the malware structure and preventing the malware from being recovered correctly.

Protecting neural network model supply chain. We stress that the neural network model markets play an important role in propagating Evilmodel. We suggest mitigating Evilmodel attack from the perspective of supply chain protection. The model markets should improve user identity verification and allow only verified users to upload models. Moreover, all neural network models provided in the market need to be detected strictly.

Addtionally, we also suggest a certificate mechanism on neural network model. Specifically, the network neural model supplier shall release the matching certificate at the same time as the model is released, and the user could verify the model through attached certificate easily. And for applications, the models can only be loaded if they pass the verification.

Detecting malware in the neural network model. We conduct a simple white-box detection experiment to explain how to detect embedded malware in the neural network model. In the experiment, we assume the defenders know the embedded malware sample. The defenders can extract the model parameters from different layers, convert them to hex bytes, and compare them with malware bytes. If malware is embedded in the network layer, the parameters hex bytes and malware bytes will overlap. The EvilModel can be detected through the overlap rate of different layers. We used the Mobilenet clean model and EvilModels for the proof-of-concept detection, and Fig. 12 shows the result. The "Cx" is the convolution layers, and the "F1.x" represents the fully connected layer. The overlap rate in the F1.1 layer is significantly higher than other layers, which means the malware is mainly embedded in the F1.1 layer of the Mobilenet model. Although it is hard to know the embedded malware in real scenarios, constructing a collection of malware and detecting overlap rate one by one is a mature method. So the experimental result indicates that it is possible to detect malware embedded in the neural network model.

The main purpose of this paper is to prove Evilmodel can be a practical threat, and to remind the security community to prevent it in advance. We believe EvilModel is easy to defend as long as we fully understand its principle, as confirmed by the aforementioned countermeasures. Additionally, more detailed countermeasures is one of our planned future works.

8 Conclusion

This paper proposes three methods to embed malware into neural network models with a high embedding rate, low impact on the model performance, and no extra effort. We applied the embedding methods on 10 mainstream models with 19 malware samples to show the feasibility of the methods. With half substitution applied, nearly half of the model can be replaced with malware bytes without performance degradation, reaching an embedding rate of 48.52%. A quantitative method is proposed to evaluate the existing embedding methods with the embedding rate, the impact on models, and the embedding effort. This paper also designs an implicit trigger and presents a stealthy attack using half substitution to demonstrate the potential threat of the proposed scenario. This paper further explores the embedding capability of a neural network model and studies the fault tolerance of different network layers by an experiment on AlexNet. We also tried to restore the lost performance by retraining the model.

This paper shows that a large number of parameters in regular neural network models can be replaced with malware bytes or other types of information while maintaining the model's performance with no sense. Neural network models are ready to be carriers of hiding malware, and this issue will be a significant threat to network security, which needs security researchers to prepare in advance. Network attack and defense are interdependent, and it is worthwhile for the security community to discover potential threats and respond to them with practical solutions as early as possible. We believe that the discovered threat in this paper will contribute to future protection efforts. We also hope the countermeasures given in this paper will be a great help.

Acknowledgements This paper is an extended version of work that was first presented in September, 2021 at the 26th IEEE Symposium on Computers and Communications (ISCC 2021) [**?**]. We thank the editors and anonymous reviewers from IEEE ISCC 2021 for their helpful efforts.

References

1 Aghazadeh, H.: Strategic marketing management: Achieving superior business performance through intelligent marketing strategy. Procedia-Social and Behavioral Sciences **207**, 125–134 (2015)
2 Amazon: Machine learning on aws (2021), https://aws.amazon.com/machine-learning/
3 Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static PE machine learning malware models via reinforcement learning. CoRR **abs/1801.08917** (2018), http://arxiv.org/abs/1801.08917
4 Anomali, T.R.: The InterPlanetary Storm: New Malware in Wild Using InterPlanetary File System's (IPFS) p2p network (June 2019), https://tinyurl.com/ch5b9w8z
5 Boehm, B.: Stegexpose (2014), https://github.com/b3dk7/StegExpose
6 Brundage, M., Avin, S., Clark, J., Toner, H., Eckersley, P., Garfinkel, B., Dafoe, A., Scharre, P., Zeitzoff, T., Filar, B., Anderson, H.S., Roff, H., Allen, G.C., Steinhardt, J., Flynn, C., hÉigeartaigh, S.Ó., Beard, S., Belfield, H., Farquhar, S., Lyle, C., Crootof, R., Evans, O., Page, M., Bryson, J., Yampolskiy, R., Amodei, D.: The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. CoRR **abs/1802.07228** (2018), http://arxiv.org/abs/1802.07228
7 Bucila, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006. pp. 535–541. ACM (2006). https://doi.org/10.1145/1150402.1150464
8 BVLC: Model zoo (2020), https://github.com/BVLC/caffe/wiki/Model-Zoo
9 Castelvecchi, D.: Can we open the black box of ai? Nature News **538**(7623), 20 (2016)
10 Committee, I.M.S.: IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic (Jul 2019), https://standards.ieee.org/standard/754-2019.html
11 Dhilung Kirat, J.J., Stoecklin, M.P.: Deeplocker - concealing targeted attacks with ai locksmithing. Tech. rep., IBM Research (2018)
12 Dumitrescu, S., Wu, X., Memon, N.: On steganalysis of random lsb embedding in continuous-tone images. In: Proceedings. International Conference on Image Processing. vol. 3, pp. 641–644 vol.3 (2002). https://doi.org/10.1109/ICIP.2002.1039052
13 Dumitrescu, S., Wu, X., Wang, Z.: Detection of lsb steganography via sample pair analysis. IEEE Transactions on Signal Processing **51**(7), 1995–2007 (2003). https://doi.org/10.1109/TSP.2003.812753
14 Eisenkraft, K., Olshtein, A.: Pony's C&C servers hidden inside the bitcoin blockchain (Oct 2019), https://research.checkpoint.com/2019/ponys-cc-servers-hidden-inside-the-bitcoin-blockchain/
15 FireEye: Uncovering a malware backdoor that uses twitter. Tech. rep., FireEye (2015)
16 Fridrich, J., Goljan, M., Du, R.: Reliable detection of lsb steganography in color and grayscale images. In: Proceedings of the 2001 Workshop on Multimedia and Security: New Challenges. p. 27–30. MM&Sec '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/1232454.1232466, https://doi.org/10.1145/1232454.1232466
17 Google: Ai and machine learning products (2021), https://cloud.google.com/products/ai
18 Grigorescu, S., Trasnea, B., Cocias, T., Macesanu, G.: A survey of deep learning techniques for autonomous driving. Journal of Field Robotics **37**(3), 362–386 (2020)

19 Hitaj, B., Gasti, P., Ateniese, G., Pérez-Cruz, F.: Passgan: A deep learning approach for password guessing. In: Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11464, pp. 217–237. Springer (2019). https://doi.org/10.1007/978-3-030-21568-2_11

20 Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on GAN. CoRR **abs/1702.05983** (2017), http://arxiv.org/abs/1702.05983

21 ImageNet: ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) (May 2012), https://imagenet.stanford.edu/challenges/LSVRC/2012/

22 InQuest: malware-samples (2021), https://github.com/InQuest/malware-samples

23 Javaid, A., Niyaz, Q., Sun, W., Alam, M.: A deep learning approach for network intrusion detection system. Eai Endorsed Transactions on Security and Safety **3**(9), e2 (2016)

24 Kharrazi, M., Sencar, H.T., Memon, N.: Improving steganalysis by fusion techniques: A case study with image steganography. In: Transactions on Data Hiding and Multimedia Security I. pp. 123–137. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

25 Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980

26 Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25**, 1097–1105 (2012)

27 Lecue, F., Gade, K., Geyik, S.C., Kenthapadi, K., Mithal, V., Taly, A., Guidotti, R., Minervini, P.: Explainable ai: Foundations, industrial applications, practical challenges, and lessons learned (Feb 2020), https://xaitutorial2020.github.io/

28 Liu, T., Liu, Z., Liu, Q., Wen, W., Xu, W., Li, M.: Stegonet: Turn deep neural network into a stegomalware. In: Annual Computer Security Applications Conference. p. 928–938. ACSAC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3427228.3427268, https://doi.org/10.1145/3427228.3427268

29 Mamoshina, P., Vieira, A., Putin, E., Zhavoronkov, A.: Applications of deep learning in biomedicine. Molecular pharmaceutics **13**(5), 1445–1454 (2016)

30 Microsoft: Azure machine learning (2021), https://azure.microsoft.com/en-us/services/machine-learning/

31 Murphy, K.P.: Machine learning: a probabilistic perspective. MIT press (2012)

32 Neeta, D., Snehal, K., Jacobs, D.: Implementation of lsb steganography and its evaluation for various bits. In: 2006 1st International Conference on Digital Information Management. pp. 173–178 (2007). https://doi.org/10.1109/ICDIM.2007.369349

33 Ng, H.W., Nguyen, V.D., Vonikakis, V., Winkler, S.: Deep learning for emotion recognition on small datasets using transfer learning. In: Proceedings of the 2015 ACM on international conference on multimodal interaction. pp. 443–449 (2015)

34 Pantic, N., Husain, M.I.: Covert botnet command and control using twitter. In: Proceedings of the 31st Annual Computer Security Applications Conference. p. 171–180. ACSAC 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2818000.2818047

35 Parkhi, O.M., Vedaldi, A., Zisserman, A.: Deep face recognition. In: British Machine Vision Conference (2015)

36 Patsakis, C., Casino, F.: Hydras and IPFS: a decentralised playground for malware. Int. J. Inf. Sec. **18**(6), 787–799 (2019). https://doi.org/10.1007/s10207-019-00443-0

37 Patsakis, C., Casino, F., Katos, V.: Encrypted and covert dns queries for botnets: Challenges and countermeasures. Computers & Security **88**, 101614 (2020). https://doi.org/https://doi.org/10.1016/j.cose.2019.101614, https://www.sciencedirect.com/science/article/pii/S016740481831321X

38 Preneel, B.: Second preimage resistance, pp. 543–544. Springer US, Boston, MA (2005). https://doi.org/10.1007/0-387-23483-7_372

39 Reed, R.: Pruning algorithms-a survey. IEEE Transactions on Neural Networks **4**(5), 740–747 (1993)

40 Research, Z.: Fashion MNIST-Kaggle (Dec 2017), https://www.kaggle.com/zalando-research/fashionmnist

41 Rigaki, M., Garcia, S.: Bringing a GAN to a knife-fight: Adapting malware communication to avoid detection. In: 2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018. pp. 70–75. IEEE Computer Society (2018). https://doi.org/10.1109/SPW.2018.00019, https://doi.org/10.1109/SPW.2018.00019

42 Schalkoff, R.J.: Pattern recognition. Wiley Encyclopedia of Computer Science and Engineering (2007)

43 Seymour, J., Tully, P.: Weaponizing data science for social engineering: Automated e2e spear phishing on twitter. Black Hat USA **37**, 1–39 (2016)

44 Tanner, M.A., Wong, W.H.: The calculation of posterior distributions by data augmentation. Journal of the American statistical Association **82**(398), 528–540 (1987)

45 VirusTotal: VirusTotal (2021), https://www.virustotal.com/

46 Volkhonskiy, D., Nazarov, I., Burnaev, E.: Steganographic generative adversarial networks. In: Twelfth International Conference on Machine Vision, ICMV 2019, Amsterdam, The Netherlands, 16-18 November 2019. SPIE Proceedings, vol. 11433, p. 114333M. SPIE (2019). https://doi.org/10.1117/12.2559429

47 Wang, J., Liu, Q., Wu, D., Dong, Y., Cui, X.: Crafting Adversarial Example to Bypass Flow-&ML- based Botnet Detector via RL. In: To be appear in RAID '21: International Symposium on Research in Attacks, Intrusions and Defenses. ACM (2021)

48 Wang, W., Su, C.: CCBRSN: A system with high embedding capacity for covert communication in bitcoin. In: 35th IFIP TC 11 International Conference, SEC 2020. vol. 580, pp. 324–337. Springer (2020). https://doi.org/10.1007/978-3-030-58201-2_22

49 Wang, Z., Liu, C., Cui, X., Liu, J., Yin, J., Wu, D.: Deepc2: Ai-powered covert botnet command and control on osns. arXiv preprint arXiv:2009.07707 (2020)

50 Yang, H., Zeng, R., Xu, G., Zhang, L.: A network security situation assessment method based on adversarial deep learning. Applied Soft Computing **102**, 107096 (2021)

51 ytisf: theZoo - A Live Malware Repository (2021), https://github.com/ytisf/theZoo

52 Yuan, J., Zhou, S., Lin, L., Wang, F., Cui, J.: Black-box adversarial attacks against deep learning based malware binaries detection with GAN. In: ECAI 2020 - 24th European Conference on Artificial Intelligence, Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2536–2542. IOS Press (2020). https://doi.org/10.3233/FAIA200388, https://doi.org/10.3233/FAIA200388

53 Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-sec: deep learning in android malware detection. In: Proceedings of the 2014 ACM conference on SIGCOMM. pp. 371–372 (2014)

54 Zhang, K., Zhang, Z., Li, Z., Qiao, Y.: Joint face detection and alignment using multitask cascaded convolutional networks. IEEE Signal Processing Letters **23**(10), 1499–1503 (2016)

55 Zhang, X., Han, Y., Xu, W., Wang, Q.: Hoba: A novel feature engineering methodology for credit card fraud detection with a deep learning architecture. Information Sciences (2019)

56 Zhang, Y., Zhang, W., Chen, K., Liu, J., Liu, Y., Yu, N.: Adversarial examples against deep neural network based steganalysis. In: Proceedings of the 6th ACM Workshop on Information Hiding and Multimedia Security, Innsbruck, Austria, June 20-22, 2018. pp. 67–72. ACM (2018). https://doi.org/10.1145/3206004.3206012
57 Zhao, Z.: mtcnn-pytorch (2019), https://github.com/polarisZhao/mtcnn-pytorch