Cybersecurity

RESEARCH Open Access

DeepMal: maliciousness-Preserving adversarial instruction learning against static malware detection

Chun Yang[1,2], Jinghui Xu[1,2], Shuangshuang Liang[1,2], Yanna Wu[1,2], Yu Wen[1*], Boyang Zhang[1] and Dan Meng[1]

Abstract
Outside the explosive successful applications of deep learning (DL) in natural language processing, computer vision, and information retrieval, there have been numerous Deep Neural Networks (DNNs) based alternatives for common security-related scenarios with malware detection among more popular. Recently, adversarial learning has gained much focus. However, unlike computer vision applications, malware adversarial attack is expected to guarantee malwares' original maliciousness semantics. This paper proposes a novel adversarial instruction learning technique, DeepMal, based on an adversarial instruction learning approach for static malware detection. So far as we know, DeepMal is the first practical and systematical adversarial learning method, which could directly produce adversarial samples and effectively bypass static malware detectors powered by DL and machine learning (ML) models while preserving attack functionality in the real world. Moreover, our method conducts small-scale attacks, which could evade typical malware variants analysis (e.g., duplication check). We evaluate DeepMal on two real-world datasets, six typical DL models, and three typical ML models. Experimental results demonstrate that, on both datasets, DeepMal can attack typical malware detectors with the mean *F1-score* and *F1-score* decreasing maximal 93.94% and 82.86% respectively. Besides, three typical types of malware samples (Trojan horses, Backdoors, Ransomware) prove to preserve original attack functionality, and the mean duplication check ratio of malware adversarial samples is below 2.0%. Besides, DeepMal can evade dynamic detectors and be easily enhanced by learning more dynamic features with specific constraints.

Keywords: Adversarial instruction learning, Malware, Static malware detection, Small-scale

Introduction

Malware is a significant concern of cybersecurity because of its severe damage and threats to network and computing device security. Static malware detection (Anderson and McGrew 2017; Banescu et al. 2017; Burnaev and Smolyakov 2016; Dahl et al. 2013; Gardiner and Nagaraja 2016; Saxe and Berlin 2015; Ye et al. 2017), as one of promising defense techniques in the security community, is to accurately identify the binary files of malware and their variants (Banescu et al. 2016; Burnaev and Smolyakov 2016; Ye et al. 2017; You and Yim 2010). As artificial intelligence (AI) techniques have gained substantial achievement on security-critical applications, some work has shown the effectiveness of machine learning (ML) and deep neural networks (DNNs) (Anderson and McGrew 2017; Banescu et al. 2017; Burnaev and Smolyakov 2016; Dahl et al. 2013; Gardiner and Nagaraja 2016; Saxe and Berlin 2015) for static malware detection. However, recent studies (Brown et al. 2017; Evtimov et al. 2017; Kurakin et al. 2016; Papernot et al. 2016) have also demonstrated that AI models, especially DNNs, are

*Correspondence: wenyu@iie.ac.cn
[1]Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS), North of Yiyuan, Xingshikou Road, Haidian District, Beijing, China
Full list of author information is available at the end of the article

vulnerable to well-tuned perturbations generated by adversarial learning techniques on the original data samples, called adversarial attacks. Therefore, studying such an attack on static malware detection benefits in improving the robustness of AI-driven detection models.

Though such adversarial attacks have been widely explored in computer vision, little has been applied for malware detection scenarios. It is potential because of the new challenge in this security-related domain: different from the adversarial images, ordinary manipulations to the malware samples (e.g., binary files) tend to either introduce invalid instructions that are non-executable or most likely break their original maliciousness semantics (which we refer to attack functionalities of malware binaries). Even though successfully fooling detection models, those adversarial samples are infeasible in the real world and do not make sense for developing robust detection techniques.

A kind of indirect approach (Anderson et al. 2016; Grosse et al. 2016; Wang et al. 2017) is to manipulate the features of malware binaries instead of directly learning its adversarial samples to exploit the vulnerabilities of DNNs. Recently a formalization method (Pierazzi et al. 2019) was proposed for further reversing the adversarial features to the malware binaries. Although these methods can effectively evade detection models and even generate real variants, producing adversarial samples for malware remains a significant challenge. There are two reasons for this requirement. First, in the real world, getting prior knowledge of the features used for detection is difficult. Second, this type of feature-based approach may be easily defended by the feature extraction protection technique (Wang et al. 2017).

Our work is inspired by our and other researchers' observation (Han et al. 2013; Yue 2017; Yang et al. 2018; Nataraj et al. 2011): 1) the features of malware binaries can be automatically learned as the binaries are translated to a sort of particular gray images and input into DNN model, in which the pixels are the bytes of the binaries; and 2) the automatically learned features show higher detection performance than the state-of-the-art methods.

Problem Statement. *The main problem addressed in this paper is to automatically generate feasible adversarial samples for malware binaries that preserve original maliciousness semantics of the malware binaries and can evade various state-of-the-art detection techniques, including ML/DL model and signature-based approaches through adversarial learning on the gray images of the malware binaries rather than relying on any prior knowledge of theirs manual features for detection.*

There are three main aspects to this problem, and they are as follows:

- Binary-to-image translation: Starting from malware binaries, we must first translate them into unique gray images that help generate adversarial samples. First, our approach should retain all of the necessary information to restore the binaries. Additionally, any extra information that will benefit following adversarial learning needs to build into the images.
- Adversarial learning: The challenge here is to generate adversarial bytes/pixels and find appropriate insertion positions in the gray images provided by the previous phase. Moreover, the perturbations, such as the number of the inserted bytes, should be stealthy, unnoticeable. Otherwise, they might be detected by conventional reverse techniques, such as instruction deduplication.
- Image-to-binary conversion: The generated adversarial images needs to be finally converted to executables. Specially, we need to transform the inserted bytes to valid and semantics-harmless instructions and guarantee the attack effectiveness.

Approach and Contributions. In this paper, we propose DeepMal that addresses all the above aspects. DeepMal begins with the malware binaries' unique gray images and produces adversarial instruction bytes that are executable and harmless to the original maliciousness semantics. At a high level, DeepMal integrates binary processing techniques with the adversarial learning approach on computer vision to address the technical challenges involved in the above three aspects of adversarial attacks against static malware detection. We present our key ideas below and the technical details in "Proposed method" section.

First, DeepMal aims to translate the malware binaries into the gray images that pave the way for implementing practical adversarial attacks on static malware detection. Specially, DeepMal remains several important segments for final binaries recovery, which contain code instructions, global and static variables, and other assist information. As mentioned above, conventional adversarial learning approaches randomly modify the image pixels and cannot guarantee the malware binaries' attack semantics. Instead, Deepmal generates and inserts appropriate instructions into the malware binaries, which are valid and harmless to the original malicious semantics. Therefore, DeepMal embeds some indicators into the images in advance to point out the positions where the inserted instructions will not interfere with others.

A second important idea in DeepMal is the restrained adversarial instruction learning method. It applies a more fine-grained instruction representation rather than a traditional feature representation technique. And then, we intend to study the adversarial instructions through deep learning. As (Han et al. 2013; Yang et al. 2018; Nataraj et al. 2011) have proved that the image of binary code could be used for malware detection, we innovatively generate adversarial examples based on binary images to evade malware detectors. First, the binary malware is processed as an image by a convolution neural network (CNN) to detect whether there exists adversarial instruction; Second, the above image will be reverse mapped to malware to ascertain the semantics of the adversarial instruction. In essence, DeepMal is an extended version of CNN augmented with a carefully designed layer, between its input layer and first hidden layer, for restrained gradient descent for "valid and dispensable" pixel insert rather than manipulations in the gray images. Next, we further optimize this layer to minimize the total number of inserted pixels by automatically finding the optimal embedding positions in the gray images.

A third main contribution in DeepMal is automatically converting generated adversarial malware image to maliciousness-preserving assembly codes or malware binary files. As we have stated above, the generated adversarial perturbations are continuous pixel values located in specific positions. We map these pixels to the limited points, which are the nearest to them through $L1$ and $L2$ norm constraints. Specifically, these mapped points are limited to several essential assembler instructions, which could not affect assembly instructions' execution. Moreover, we use several popular debugging tools (e.g., IDA Disassembler) to reconfirm the generated malware samples executable because sometimes such instruction additions may lead to the execution disruption or disorder.

Evaluation. We conduct a series of experiments on two real datasets to evaluate the effectiveness of DeepMal comprehensively. Experimental results demonstrate that DeepMal can effectively attack typical malware detectors with the mean *F1-score* and *F1-score* decreasing maximal by 93.94% and 82.86% respectively. Besides, the adversarial samples of three typical kinds of malware samples (Trojan horses, Backdoors, Ransomware) are proved to preserve the original attack functionality, and their mean failure ratio due to deobfuscation is below 2.0%. Moreover, we also make a discussion on the attack evasiveness with respect to dynamic malware detection. It is proved by experimental results that DeepMal also works when detected by dynamic detectors and could be easily enhanced by learning more dynamic features with specific constraints.

The rest of the paper is organized as follows. Section 2 provides background on adversarial samples and a survey of relevant work. Section 3 presents our technique and its properties. Experimental results are shown in Section 4, where our method is compared to other approaches. Moreover, the DeepMal attack evasiveness for dynamic malware detection is also discussed in Section 4. Finally, Section 5 summarizes our work.

Background and related work

In this section, we review static malware detection and conventional evasion approaches. And then, we describe the significant concepts used in this paper. **Static Malware Detection.** Malware detection methods can be categorized into static or dynamic approaches. Static malware detection techniques are based on statistical static features, while dynamic methods depend on dynamically executing and analyzing. Though dynamic techniques are more against syntax changes, static methods are more scalable and provide better coverage. At present, advanced artificial intelligence(AI) based static malware detection techniques achieve far better performance than dynamic methods. Under integrated feature extraction technique proposed by Islam et al. (2013), static features such as k-gram (Myles and Collberg 2005), n-gram (Khoo et al. 2013), BinClone (Farhadi et al. 2014) and ILine(Jang et al. 2013) could be utilized for statistical analysis. In recent years, malware samples are proved to be easily detected using machine learning (ML) solutions (i.e., SVM, Random Forests, Decision trees). Yuan et al. (2014) demonstrated that deep learning (DL) solutions could successfully abstract complex patterns of malware as well as malware variants, and eventually, malware would be detected. Moreover, (Han et al. 2013; Nataraj et al. 2011) observes that malware belonging to the same family have similar contours and textures on their images, but different malware families have different visual effects. Therefore, it is difficult for malware samples to bypass CNNs-based detectors.

Conventional Evasion Approaches. The issues of conventional evasion approaches have evolved significantly over the past. The most popular techniques include obfuscation-based approaches and feature-level adversarial attacks. Obfuscation techniques (Vinod et al. 2009; Banescu et al. 2016; Banescu et al. 2017; You and Yim 2010) include dead-code instruction, code transportation, registers renaming, and instruction substitution to generate realistic variants for the existing malware. Obfuscator-LLVM (O-LLVM) (Junod et al. 2015) is built based on the LLVM framework and the CLANG compiler toolchain. It operates at the intermediate language level and modifies a program's logic before the binary file is generated. Luo et al. (2014) proposes a binary-oriented, obfuscation-resilient method CoP for software plagiarism or code reuse detection, with strong obfuscation resiliency. There are some other obfuscation tools stated as follows: CXX-obfuscator[1], proposed by Stunnix, is a source code obfuscation tool; Loco (Madou et al. 2006) is a binary code obfuscation tool, as well as an open-source product; CIL (Necula et al. 2002) is also a source code obfuscation tool, which possesses many useful source code transformation

[1] http://stunnix.com/

techniques, such as converting multiple returns to one return, changing switch-case to if-else.

Adversarial malware attack techniques have been proposed in recent years (Grosse et al. 2016; Anderson et al. 2016; Wang et al. 2017), as we have introduced in "Introduction" section, through various carefully selected restrained-feature manipulations, malware authors could create adversarial malware samples processing almost similar functional behavior with the original ones. It is undeniable that these malware attack solutions based on feature manipulations could trick ML and DL detectors into incorrect classification. However, this kind of adversarial attack technique has several shortcomings in real-word. The first deficiency is that the generated adversarial malware based on feature manipulations may lead to original maliciousness loss and unrealistic malware samples. The second shortcoming is that this kind of adversarial attack technique relies strongly on prior knowledge of feature selection and extraction. The last flaw is that those adversarial malware samples' attack efficiency against ML and DL detectors may be degraded due to the adversary resistant technique proposed by (Wang et al. 2017), which uses random feature nullification to obstruct attackers from constructing impactful adversarial samples. Fortunately, though adversarial malware samples generated based on feature manipulations have been proved to be easily detected by this kind of adversary resistant techniques so far, whether this technique could resist adversarial malware samples generated through other promising alternatives is still to be studied.

Adversarial Learning. The security problem of DNN has gained much attention recently. Szegedy et al. (2013)first revealed that the deep neural networks learned by backpropagation have non-intuitive characteristics and intrinsic blind spots. These DNNs are vulnerable to well-tuned artificial perturbation crafted by several gradient-based algorithms. Specifically, (Goodfellow et al. 2014) argued that the linear nature and high-dimensions are the primary causes of the neural network's vulnerability to adversarial perturbation, and they proposed a "fast gradient sign method" to generate adversarial examples through backpropagation based gradient computation. Moreover, (Papernot et al. 2017; Narodytska and Kasiviswanathan 2017) suggested several black-box attacks that require no internal knowledge about the target systems. All these related work that we have stated above has been widely explored in computer vision, and little has been done in malware detection. In particular, to the best of our knowledge, the only work before ours that ever mentioned applying adversarial learning to malware detection is carried out by (Anderson et al. 2016). However, unlike our work, their adversarial learning methods used for malware detection is not practical in a real-world scenario. They did not provide an end-to-end solution, where the generation starts with the malware executable but ends with generated features.

Proposed method

This section introduces the detailed approaches of how to design a semantics-aware adversarial instruction learning model. Furthermore, we will explain why our method offers some theoretical guarantees of the effectiveness of fooling malware detection models, availability of executable binaries, and integrity of attack functionalities. Adversarial instruction learning of DeepMal is exactly powered by the strong feature extraction capability of CNN. Therefore, the learned adversarial malware samples can also effectively impact other DNNs and machine learning models.

System Architecture. The proposed DeepMal generates malware adversarial samples, which are effective in a real-world scenario. Also, DeepMal could provide an end-to-end solution, which starts with a binary and ends with an adversarial binary. Based on the above peculiarities, the architecture of our adversarial malware attack system DeepMal is shown in Fig. 1. DeepMal consists of the following components: binary-to-image translator, maliciousness semantics guaranteed adversarial instruction learning model, and image-to-binary translator.

As shown in Fig. 1, binary-to-image translator contains mapping strategy and visualization. Correspondingly, the image-to-binary translator includes a reverse mapping strategy and debugger. In this paper, the mapping strategy is a position-aware strategy that could guarantee adversarial perturbations generated by adversarial learning models that are added to specific positions without breaking the malware's original instructions. Furthermore, the reverse mapping strategy and debugger are designed to translate pixels to assembly codes in specific positions, guaranteeing the generated malware adversarial samples executable in the real world. The adversarial instruction learning model consists of a constrained generator and optimizer. The discrimination model of this generator is based on CNN, because the attacks are generated based on image information. The optimizer is designed to limit the scale of learned instructions non-obvious enough to typical deobfuscation tools based on code deduplication.

Binary-to-Image translator

Binary-to-Image translator is expected to map malware binaries to images, as well as pixel preprocessing in specific positions. Given input malware binary file is donated by x, we then divide x into separate instructions, which is denoted by e_1, e_2, $e_3 ... e_i ... e_n$. As described in (Nataraj et al. 2011), malware binary files mainly contain: the "`.text`" segment, which contains code instructions; the "`.idata`" segment, which contains information about the used imports in the file; the "`.rdata`" segment, which

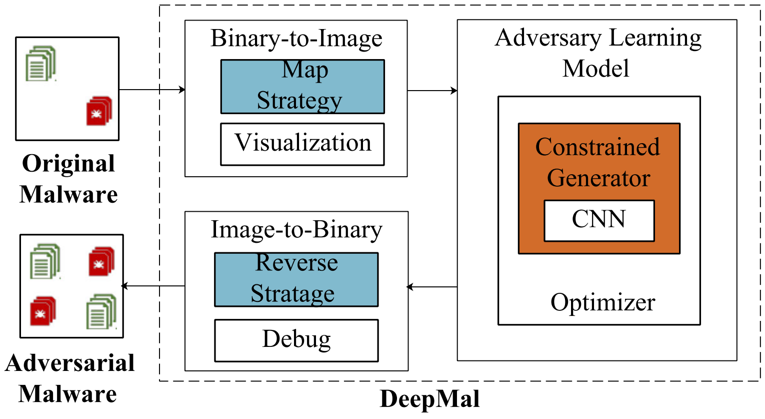

Fig. 1 System architecture of DeepMal

contains the initialized global and static variables; the "`.rsrc`" segment, which contains resource information.

To guarantee the original maliciousness semantic uncontaminated, DeepMal generates adversarial perturbations in specific positions without disrupting the context instructions. In this stage, we mark these specific positions through a series of zeros between separate instructions, because zeros will not affect the adversarial examples when adding the adversarial perturbations to original pixels. To be more specific, we embed a series of zeros to original binary files to mark specific positions. The rules about how we integrate zeros to binary files are described as the following map strategy:

- Rule 1: In the "`.text`" segment, we embed hexadecimal "00" between adjacent instructions. In this way, adversarial perturbations added to the "`.text`" segment should be represented by single hexadecimal numbers and located between adjacent instructions.
- Rule 2: In the "`.rdata`" segment, we circularly embed hexadecimal "00" between adjacent instructions 16 times. As a result, adversarial perturbations added to the "`.rdata`" segment should be represented by 16 hexadecimal numbers and located between adjacent instructions.
- Rule 3: At the bottom of binary files, we fill in blanks on the last row with a small amount of hexadecimal "00". That means hexadecimal numbers added to the bottom of binary files are not limited.

These three rules are designed depending on the image-to-binary translator, which is described specifically in "Proposed method" section. We then translate the binary files of malware into gray-scale images due to the unique visual contours and textures of every malware family observed by other researchers (Han et al. 2013; Yue 2017) and us. The output of this module is denoted by a malware image $x_i \in R^{N\times M}$, where N and M respectively indicate the width and length of the malware image. Depending on the binary file size, we set N to be 512, and pixels of a gray-scale image is ranging from 0 to 255.

Adversarial instruction learning model

The maliciousness semantics guaranteed adversarial instruction learning model is revised from standard CNN. As we have stated in "Introduction" section, this revised model extends a special constraint layer between standard CNN's input layer and first hidden layer. The architecture of the adversarial instruction learning model is shown in Fig. 2.

The input of revised CNN is denoted by $x_i \in R^{N\times M}$, and the first function is simply performing element-wise multiplication of x_i with I_p^i. Here, $I_p^i \in R^{N\times M}$ is a mask matrix with the same dimensions as x_i. For each input malware image x_i, a corresponding I_p^i is generated. To be more specific, I_p^i is a binary vector, with each element being either 0 or 1. Essentially, the mask matrix is a positional marker. The position, in which we have embedded zeros, is set as 1, and the original position is set as 0. It regularly cancels out the specialized pixels within the image and then feeds the partially corrupted image to the first hidden layer when passing a malware image sample through the layer to CNN. Figure 3 shows the process of specialized pixel constraint in more specific.

δx_i represents the adversarial perturbations generated by standard CNN. The adversarial perturbations generated by revised CNN for malware image x is denoted by $\delta x_i \odot I_p^i$. The adversarial perturbations are generated by calculating the derivative of the model's cost function with respect to the input samples. The original pixels of malware images are guaranteed to remain unchanged through restrained gradient descent, and the output of the revised CNN model is represented as $x_{adv} \in R^{N\times M}$, which denotes the input of a hybrid constrained optimizer.

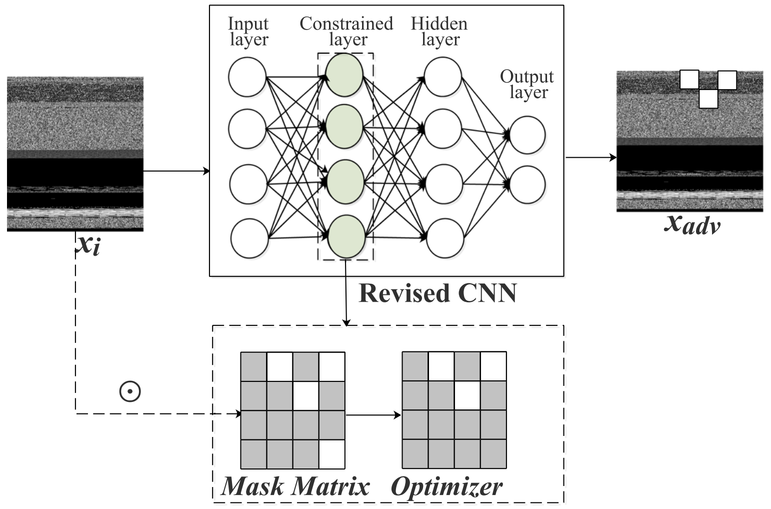

Fig. 2 Adversarial instruction learning model

In many of those adversary generation methods, we use the fast gradient sign method (FGSM) (Goodfellow et al. 2014) to generate adversarial attacks. It is robust and efficient enough for us to generate persuasive adversarial malware samples in this paper. To preserve the malicious semantics, we limit the adversarial perturbations to specific positions embedded with zeros. As a result, the perturbation space is significantly reduced by this constraint. As we know, the solution space of FGSM is infinite, and we could finally find the solution points in unlimited space towards the fastest gradient descent direction.

Hybrid Optimization. As we have discussed in "Introduction" section, manipulations based on binary files is expected to be as small as possible, thus preserving maliciousness and guaranteeing unnoticeable. The first restraint function utilizes the constrained fast gradient sign method to generate malware adversarial perturbations in specific positions constrained by mask matrix, whereas these specific-positions constrained adversarial perturbations are obvious due to their considerable quantity. As we know, differential evolution (DE) (Su et al. 2019) is a method that optimizes a problem by iteratively trying to improve a candidate solution. However, candidate solutions' space is tremendous without gradient descent, since the DE-only optimization algorithm may lead to high computational cost with constrained iteration of positions. To address this problem, we propose a novel global optimal solution, named hybrid constrained

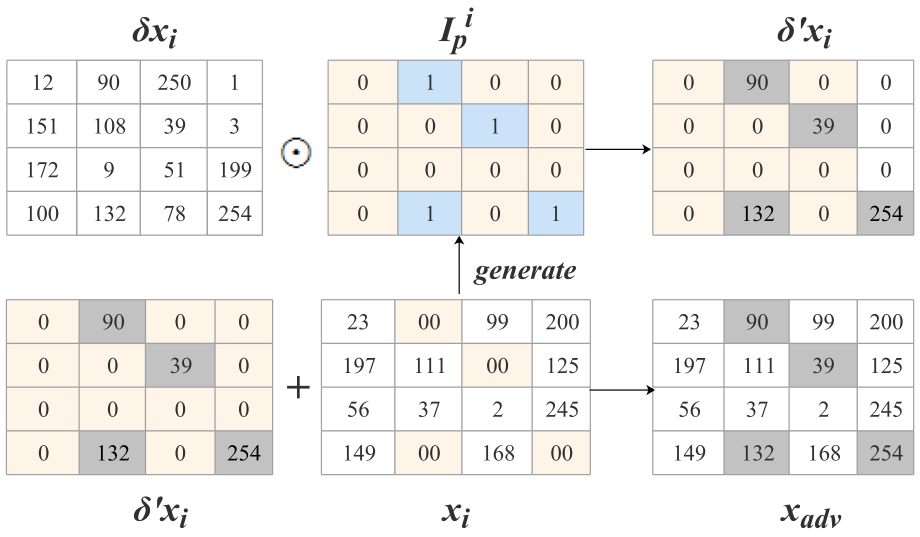

Fig. 3 The process of specialized position constraint

optimizer, to restrain the number, values as well as specific positions of the perturbations. Specifically, we utilize the DE operator's mutations to iteratively optimize the positions where embedded adversarial perturbations through restrained gradient descent. If the new position of an agent is an improvement, then it is accepted and forms part of the population. The hybrid constrained algorithm is shown in Algorithm 1.

Algorithm 1 The Hybrid Constrained Algorithm

```
Input: $x_i, F_{fgsm}, F_{cnn}, y*, k, T_1, T_2, F, CR$
1: compute $I_p^i$, $idx_i$, $len(idx_i)$
2: while $t$ in $T_1$ do:
3:     $\delta x_i = F_{fgsm}(x_i)$; $\delta x_{adv} = \delta x_i * I_p^i + x_i$
4:     $x_i \leftarrow \delta x_{adv}$
5: end while
6: return $\delta x_{adv}$
7: global intialization: $bound \leftarrow k * len(idx_i)$
8: population intialization: $X(g_0) = rand(idx_i)$
9: while $t$ in $T_2$ do:
10:    if $F_{cnn}(X(g_0)) = y*$:
11:        break;
12:    else:
13:        $V_i(g+1) = X_{r1}(g) + F(X_{r2}(g) - X_{r3}(g))$
14:        if $rand(0,1) \leq CR$:
15:            $U_i(g+1) = Vi(g+1)$
16:        else:
17:            $U_i(g+1) = X_i(g)$
18:        if $F_{cnn}(U_i(g+1)) \leq F_{cnn}(X_i(g))$
19:            $X_i(g+1) = U_i(g+1)$
20:        else:
21:            $X_i(g+1) = X_i(g)$
22: end while
23: return $best\_idx_i$
24: update $idx_i$ by $best\_idx_i$ in $\delta x_{adv}$
Output: $\delta' x_{adv}$
```

Here, notations are defined as follows:

F_{fgsm} and F_{cnn} are FGSM model and CNN model; $y*$ and k refer to target class and selection ratio; T_1 and T_2 represent iterations; F and CR refer to zoom factor and crossover probability; the set of positions and the number of positions are denoted by idx_i and $len(idx_i)$ respectively; the maximal number of $best\_idx_i$ is limited to *bound*.

To be more specific, the optimal adversarial samples are computed by a hybrid constrained algorithm of gradient descent and DE genetic algorithm. Step 9 to Step 22 exactly realize DE genetic algorithm. Step 13 realizes "individual mutation" through a difference strategy. We randomly select two different individuals in the population, and then the vector difference is scaled to be mutated for vector synthesis. r_1, r_2 and r_3 are random numbers ranging from 1 to NP, g represents the g generation, and V refers to the mutants. Step 14 to Step 17 implements "cross", U refers to the next generation after the crossover. Step 18 to Step 21 implement "select", which selects the better individual as the new individual, and the $best\_idx_i$ is the output of the DE genetic algorithm. Every time $best\_idx_i$ is computed by DE genetic algorithm, idx_i is updated correspondingly, and then the local locally optimal solution δx_{adv} is produced. The process described above is the process of hybrid optimization.

Image-to-Binary translator

Technically speaking, our proposed DeepMal technique provides an end-to-end solution, where the generation starts with the binary file and ends with a malware adversarial binary file x_0. The output of a hybrid constrained optimizer is denoted by x'_{adv}, which represents malware images with position constraints. However, the image pixels may not significantly map to malware codes, which are essential instructions without breaking down original maliciousness semantics. Therefore, the image-to-binary translator is expected. In this section, we translate malware adversarial samples (which are images generated by the adversarial learning model in "Adversarial instruction learning model" section) to malware binaries.

By learning the original 8086/8088 instructions[2] and referring to studies in (Vinod et al. 2009; You and Yim 2010), we learn that there are several essential assembler instructions (e.g., `NOP`, `WAIT`) often used for dead-code-insertion without affecting the execution of assembly instructions. As we have mentioned in "Binary-to-Image translator" section, the "`.text`" segment is mainly composed of code instructions. Moreover, due to the fact that the majority of `dead-codes` (Vinod et al. 2009; You and Yim 2010) are denoted by one hexadecimal number in binary files, we could replace these adversarial perturbations generated in "`.text`" with `dead-codes`. Furthermore, to make DeepMal more robust to static analysis, we generate some API calls and system calls in "`.rdata`", which contains constants and additional directories such as debug. Unlike feature-level manipulations, we generate raw API calls and system calls represented by hexadecimal numbers in "`.rdata`" instead of feature manipulations. The instructions of API and system calls embedded to "`.rdata`" will never be called such that they will not affect the maliciousness semantics. Still, they can effectively impact other static analysis models based on feature extraction. In short,

[2]https://en.wikipedia.org/wiki/$X86$_instruction_listings

we translate images to-binaries through the following reverse mapping strategy, which is represented by three rules:

- Rule 1: In the "`.text`" segment, we calculate the $L1$ norm of each perturbing pixel to `dead-code` instructions, and then replace the adversarial perturbations with the `dead-code` instructions according to the nearest $L1$ norm distance. In this way, we could guarantee that the real world's solutions are the closest to the solution space's computed solutions.
- Rule 2: In the "`.rdata`" segment, we calculate the $L2$ norm of every 16 perturbing pixels to API and System call instructions and then replace the adversarial perturbations with API or System call instructions according to the nearest Euclidean distance. Most API and System call instructions are represented by hexadecimal numbers, which leads to the multiple dimensions problem. As a result, Euclidean distance is computed instead of $L1$ norm.
- Rule 3: At the end of a binary file, we calculate the $L1$ norm of each perturbing pixel to `dead-code` instructions, and then replace the adversarial perturbations with the `dead-code` instructions according to the nearest $L1$ norm distance.

Debug. After embedding generated adversarial perturbation codes to binary files in the "`.text`" and the "`.rdata`" segments, such addition is performed on the binary level, and all the original bytes for code instructions, variables, and data remain unchanged. Sometimes such additions may enforce the execution disruption or disorder. For example, command instructions may not locate the resources based on the original addresses. When translating adversarial perturbations into malware codes, we solve these difficulties through several popular debugging tools (e.g., IDA Disassembler). This process is performed through the following three steps: First, we mark the positions where the perturbation is generated in binary files. Second, we embed such addition to the corresponding positions in the assembly code, according to the three rules stated above. Third, we compile the assembler file through IDA Disassembler to generate the executable binary files.

Theoretical analysis

We now theoretically analyze our method's ability to generate maliciousness-preserving and unnoticeable adversarial malware samples, which are realistic in the real world. As described in "Adversarial instruction learning model" section, when training the maliciousness semantics guaranteed adversarial instruction learning model, a specialized layer simply passes specialized pixel constraint input to a standard CNN. As such, the objective function of a CNN with specialized pixels constraints can be defined as follows.

$$\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f(f_c(x_i, I_p^i; \theta)), y_i) \tag{1}$$

Here, y_i is the label of the input x_i, f_c reprenents the constrained optimization function stated in "Adversarial instruction learning model" section, and θ represents the set of model parameters. Specialized pixel constraint process is represented by function as follows:

$$q(x_i, I_p^i) = x_i \odot I_p^i \tag{2}$$

$$f_c(x_i, I_p^i; \theta) = f_c(q(x_i, I_p^i); \theta) \tag{3}$$

$\odot$ denotes the Hadamard-Product. During training, Eq. (1) can be solved using stochastic gradient descent like that of an ordinary CNN. The only difference is that for each training sample, the specifically-generated I_p^i is fixed during forward and backward propagation until the next training sample arrives.

The FGSM is proposed for computing adversarial perturbations as follows:

$$\delta x_i = \phi * sign(\mathcal{L}(x_i)) \tag{4}$$

During forward and backward propagation in this revised CNN, constrained malware adversarial perturbations can be computed as follows:

$$\delta' x_i = \delta x_i \odot I_p^i \tag{5}$$

When generating adversarial malware samples, constrained malware adversarial perturbations are added to original ones, as Eq. (6) shows:

$$x_{adv} = \delta' x_i + x_i \tag{6}$$

We now theoretically analyze how our revised CNN guarantees maliciousness-preserving and unnoticeable adversarial malware samples. According to Eq. (1), the adversarial perturbation is generated by computing the derivative of CNN's cost function for the input samples. According to Eqs. (2), (3), and (5), the revised CNN embeds additional constraints on the original standard CNN. Every step the derivative of the CNN's cost function with respect to the input samples is computed. The pixel value of the input x_i is invariable, and the adversarial perturbations are limited to the positions where embedded zeros between the original adjacent instructions. After calculating adversarial perturbations as Eq. (4) shows, the generated adversarial malware could bypass adversarial learning detection. Moreover, the generated adversarial perturbations are limited to specific positions, without original malicious code changed at all. It provides a guarantee for the availability of executable binaries and integrity of maliciousness semantics.

Next, we will analyze why we can utilize image-to-binary translators to generate malware samples that can be executed in the real world. Assume that the number of instructions (a separate instruction contains series of hexadecimal elements) we have embedded to the original binary file is denoted by p, and the number of hexadecimal elements in the binary file is denoted by q. Moreover, p and q are accords with the following constraint: $p \ll q$.

According to Eq. (6), assume that the instructions positions stay unchanged when embedding malware adversarial perturbations to original hexadecimal instructions, the maximal value range of $\delta' x_i$ is $255*p$. However, in a limited scenario, we have introduced position changes to original binary files. The coordinate positions where embedded adversarial perturbations are represented by $l_1, l_2, l_3 .... l_p$. The maximal value range of $\delta' x_i$ can be calculated by Eq. (7).

$$min \sum_{j=1}^{P} (q - l_j + 1) * 255 \tag{7}$$

Since Eq. (7) is far below $255*p$, we can demonstrate that the position (in which added adversarial perturbations) impacts more than adversarial perturbations' values. As a result, it is reasonable to introduce the DE operator to restrained gradient descent when calculating the optimal positions. Also, it is reasonable to translate malware images to binaries according to Euclidean distance.

Experiments

In this section, we first introduce the experimental settings. Then we measure the effectiveness of the attack of DeepMal compared with other maliciousness awareness attack methods on target antimalware models. In particular, we show some cases in terms of various adversarial samples, aiming to verify that the proposed DeepMal attack could preserve functionality and avoid security checks in a real-world scenario. Here, we introduce the datasets extracted from several web platforms applied in our experiments, the basic classifiers as targets of adversaries, and the baseline methods for dealing with adversaries. Our experiments are performed on Nvidia Tesla V100 16GB GPU.

Datasets: We have two public datasets for evaluation, including the Kaggle dataset[3] and a phd-dataset[4], which is a collection of malware data from 2011 to 2016. (1) The Kaggle dataset provides a dataset of with nine classes (21741 samples). To avoid the imbalance of the dataset, we randomly select two classes from the Kaggle database: "Obfuscator.ACY" and "Gatak". These two classes possess similar sample sizes, which are 2980 and 2870, respectively. Moreover, each sample is labeled with 0 or 1. Here, 0 indicates "Obfuscator.ACY", while 1 indicates "Gatak". We divided two-thirds of each class samples for training and the remaining one-third for testing. (2) The second phd-dataset has a smaller sample collection containing 2069 benign and 2250 malware samples, 3717 samples of this dataset are for training, and the rest 702 samples are for the test. Each sample comes with a label 0 or 1, 0 indicating benign software, and 1 indicating malware. Specially, for the Kaggle data, both classes are malware, but for the phd-data, only one of the two classes is malware. It is due to the different experimental purposes. Specifically, the Kaggle data is used to validate that DeepMal attack could deceive malware classifier to misclassification, whereas the phd-data is used to demonstrate that DeepMal attack could bypass malware detector.

Target Models: Traditional detection models have been extensively studied [4, 15, 31, 32]. Following previous work, we apply three ML models and six DL models to detect malware samples in our datasets. Specifically, ML modes are: Logistic Regression (LR), Support Vector Machines (SVM), Random Forest (RF), and DL models are: Lenet-5, All-Convolutional (AllConv), Network in Network (NiN), VGG16, ResNet, and standard DNN model. Specially, we extract the following features in datasets for malware detectors powered by ML models and DL models: OpCode n-gram, ByteCode n-gram features, API calls, malware images, file-size, and MISC instructions. Moreover, we translate binary files of malware into gray images without feature engineering for CNNs models. Since the baseline attacks may not be generated based on image information, our target deep learning models contain CNN-based models and standard DNN models.

Comparisons with traditional methods

In this set of experiments, based on datasets described in "Comparisons with traditional methods" section, we compare DeepMal, which integrates our proposed method described in "Proposed method" section with other traditional maliciousness aware attack methods. These two traditional maliciousness aware attack methods compared to DeepMal are obfuscation-based attack method and feature-level attack method. The attack performances of those two attack methods are evaluated in this section.

We construct four types of obfuscation-based attack methods: dead-code-instruction, code transportation, register renaming, and instruction substitution. As we have stated in "Introduction" and "Adversarial instruction learning model" sections, to avoid semantics confusion and noticeable of this generated malware, DeepMal utilizes a hybrid constrained optimization algorithm to restrict the total number of manipulations that can occur per malware sample to be as small as possible. In obfuscation-based attack methods, the number of obfuscated operations on malicious code is set to be the same as the number of adversarial perturbations. The difference is

[3] https://www.kaggle.com/c/malware-classification/data
[4] https://github.com/tgrzinic/phd-dataset

that the positions of code obfuscated actions are randomly selected. To make the implementation of obfuscation more complicated for traditional maliciousness awareness attacks based on obfuscation, we randomly select these four types of obfuscation techniques to combination. Note that we set the amounts of manipulations less than 2% of the instructions contained in input malware files.

Results. Since ground truth labels are available on datasets, we can compute *Precision, Accuracy, Recall* and *F1-score* for the malware detection results for each single category. In our application, for Kaggle data, classification *accuracy* refers to the predicted class that are correctly categorized, mean *F1* is the mean *F1-score* of both two malware classes; while for phd-data, *Precision* is the predicted malware samples that are truly malware samples, while *Recall* is the malware samples that are caught by models. *F1-score* is the harmonic mean of *Precision* and *Recall*.

For Kaggle data, the results are shown in Table 1 by *Accuracy (ACC)* and mean *F1-score (mF1)*; for phd-data, the results are shown in Table 2 by *Precision (Pre), Recall (Rec)* and *F1-score (F1)*. In Tables 1 and 2, the best experimental results are shown in boldface. Here, we compare our proposed DeepMal with obfuscation-based maliciousness aware attack method and feature-level maliciousness aware adversarial attack technique, when the baseline malware detectors are the same. The "baseline" in Tables 1 and 2 refers to baseline attacks without adding code obfuscation or adversarial perturbations. For each attack method, the best result is shown in boldface.

Through these results in both Tables 1 and 2, we can see that our entire target models show high *Accuracy*, high *Precision*, high *Recall*, resulting in high *F1-score*. It means that all of our malware detectors powered by ML models and DL models possess excellent malware detection performance. On both two data sets, the obfuscation-based attack method does not perform well on all target models with a low decrease of *accuracy* and *F1-score*. These obfuscation-based techniques are specially designed based on the previously known signatures of the original malware, and thus seem to quickly fall behind the rapid development of AI behind detection techniques. From Table 1, we can see that, on Kaggle data, feature-level adversarial attack achieves rather outstanding results, and the maximum decrease of *ACC* and *mF1* are 71.56% and 72.29% respectively, whereas our proposed DeepMal achieves the highest decrease of *ACC* and *mF1* by 93.96% and 93.94% respectively, improving around 22.40% and 21.66% than the second best. On phd-data, DeepMal still accomplishes the highest decline of *Recall* and *F1* by 90.15% and 82.86%, improving around 16.27% and 9.2% than the second best.

We also observe that DeepMal performs significantly worse on machine learning detectors than deep learning detectors. It may because DeepMal is built upon deep learning models. The results show that DeepMal could effectively bypass malware detectors powered by DL and machine learning (ML). The experimental results are also consistent with the theoretical analysis in "Theoretical analysis" section. Although the adversarial perturbations generated by DeepMal are some malicious obfuscation instructions, the constrained positions of perturbations are learned by a revised adversarial learning CNN and hybrid constrained optimizer instead of random choices, which matter most when trying to deceive detectors.

Case study

This section shows some cases in terms of three typical malware types, aiming at validating that our DeepMal is maliciousness-preserving and non-obvious. Because some kinds of malware samples would directly attack our server, leading to irreversible loss, we have to avoid self-processing scripts in practice. As a result, we randomly choose three typical types (Trojan horse, Backdoor, Ransomware) of malware samples from executable

Table 1 Experimental results on Kaggle data. In Tables 1 and 2, the best experimental results are shown in boldface

Classifier	Baseline		Obfuscation-based			Feature-level			DeepMal		
	ACC	*mF*1	*ACC*	*mF*1	Δ*mF*1	*ACC*	*mF*1	Δ*mF*1	*ACC*	*mF*1	Δ*mF*1
LR	0.9813	0.9812	0.9771	0.9769	0.0043	0.5103	0.5059	0.4753	0.8538	0.8461	0.1351
SVM	0.9903	0.9903	0.9825	0.9823	0.0080	0.6135	0.6124	0.3779	0.4589	0.3155	0.6748
RF	0.9855	0.9853	0.9783	0.8100	**0.1753**	0.5978	0.5912	0.3941	0.8126	0.7980	0.1873
Lenet-5	0.9728	0.9726	0.9529	0.9524	0.0202	0.2572	0.2497	**0.7229**	0.0332	0.0332	**0.9394**
All-Cov	0.9746	0.9745	0.9299	0.9244	0.0501	0.3900	0.3839	0.5906	0.0577	0.0575	0.9170
NiN	0.9487	0.9484	0.9281	0.928	0.0204	0.3804	0.3738	0.5746	0.1285	0.1285	0.8199
VGG16	0.9487	0.9484	0.9281	0.9280	0.0204	0.3804	0.3738	0.5746	0.1285	0.1285	0.8199
ResNet	0.9215	0.9213	0.8991	0.8983	0.0230	0.4215	0.4025	0.5188	0.2022	0.2018	0.7195
Standard DNN	0.9134	0.9098	0.8999	0.8901	0.0197	0.6032	0.5832	0.3266	0.3023	0.3525	0.5573

Table 2 Experimental results on phd-data. In Tables 1 and 2, the best experimental results are shown in boldface

Detector	Baseline			Obfuscation-based				Feature-level				DeepMal			
	Prec	*Rec*	*F*1	*Prec*	*Rec*	*F*1	Δ*F*1	*Prec*	*Rec*	*F*1	Δ*F*1	*Prec*	*Rec*	*F*1	Δ*F*1
LR	0.9941	0.9927	0.9934	0.9940	0.9710	0.9824	0.0110	0.9915	0.6811	0.8075	0.1859	0.9259	0.2083	0.3401	0.6533
SVM	0.9970	0.9710	0.9838	0.9970	0.9652	0.9808	0.0030	0.9943	0.5072	0.6717	0.3121	0.9958	0.6956	0.8191	0.1647
RF	0.9606	0.9550	0.9578	0.9595	0.9275	0.9432	0.0146	0.9354	0.5666	0.7057	0.2521	0.9467	0.6956	0.8020	0.1558
Lenet-5	0.9841	0.9884	0.9862	0.9819	0.7894	0.8752	**0.1110**	0.9009	0.1449	0.2496	**0.7366**	0.8450	0.0869	0.1576	**0.8286**
All-Cov	0.9849	0.9492	0.9667	0.9848	0.9470	0.9629	0.0038	0.9333	0.2028	0.3333	0.6334	0.9167	0.1594	0.2716	0.6951
NiN	0.9642	0.9768	0.9704	0.9626	0.9347	0.9485	0.0219	0.8883	0.2884	0.4354	0.5350	0.8275	0.1739	0.2874	0.6830
VGG16	0.9524	0.9421	0.9473	0.9211	0.8994	0.9103	0.0370	0.9244	0.2328	0.5786	0.3687	0.8312	0.1018	0.4665	0.4808
ResNet	0.8999	0.8732	0.8865	0.8534	0.8213	0.8373	0.0492	0.7984	0.2085	0.5035	0.3830	0.8243	0.0876	0.4559	0.4306
Standard DNN	0.8867	0.8813	0.8840	0.8673	0.8575	0.8623	0.0217	0.5000	0.6375	0.5604	0.3236	0.8889	0.1000	0.1798	0.7066

files contained in datasets. These real cases include 286 Trojan horse samples, 250 Backdoor samples, and 120 Ransomware samples. As we have described in "Proposed method" section, we utilize DeepMal technique to generate malware adversarial samples from original malware binary files. Also, we conduct malware variants analysis by an ideal duplication check tool: UltraCompare. By comparing these generated malware binary files with the original record, this duplication check tool could report the duplication check ratio. According to our preliminary statistics, the mean duplication check ratio of malware adversarial samples is less than 2%. Moreover, Fig. 4 shows the instruction embedding ratio of each adversarial malware type.

Figure 4 shows that only a relatively small amount of essential instructions (or API sequences) are embedded in the corresponding section. For these three types of malware, the embedding rates are below 1%. As mentioned above, the mean duplication check ratio of malware adversarial samples is less than 2%. We can see that the scale of the generated adversarial perturbations is relatively small. As a result, these generated adversarial malware samples could easily evade duplication checks and be non-obvious in the real world.

To verify whether these cases could still retain malicious attack functions in a real-world scenario, we actively implement target executable files for implanting them into target systems (Centos7 system and Windows7 system). We then analyze these real cases in Table 3 and show the results of our DeepMal attack.

In case 1, all malware samples could: successfully control the computer system remotely to achieve a remote keyboard logger and record all keyboard operations. As a result, the keyboard is no longer safe; it could easily filch user's accounts, passwords, login mailbox remotely, and send spam to designated users; it could illegally gain access to most files, browse the contents and tamper with them. In case 2, our analysis revealed that: all backdoor malware samples have maliciously leaked data from compromised systems and networks; all samples have maliciously reconnected with the compromised system; all samples could stay in the system for at least one month potentially. In case3: all these malware samples could load into the memory and encrypt the target files. Finally, prompt the user to pay for the decryption operation. Cases in this section could have validated that all of the adversarial samples we generate by DeepMal could preserve attack functionality in the real world.

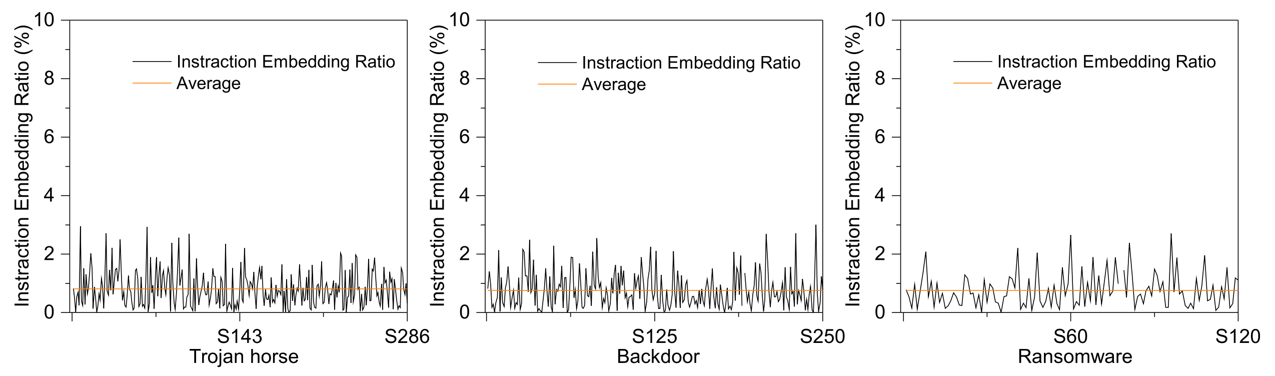

Fig. 4 Number of instructions get embedded per sample

Table 3 Case analysis of DeepMal

Malware	Dynamic event
case1: Trojan horses	...
	FILE:GetAdaptersAddresses
	FILE:CreateToolhelp32Snapshot
	Load system.dll into the memory
	FILE:Writing file to temporary directory
	PROCESS:CopyFileW
	FILE:Deleting spawned process
	FILE:Execute:[system]\taskkill.exe
	...
case2: Backdoors	...
	FILE:Get the meterpreter session
	FILE:Automatically Configure the Registry
	Start Netcat
	Establish TCP connections
	...
case3: Ransomware	...
	FILE:Writing file to temporary directory
	FILE:Write:[windows error reporting queue]
	FILE:Execute:[system]\mssecsvc.exe
	REGISTRY: Win32API function CryptGenKey
	...
	Call ReadFile reads the binary into memory
	FILE:SeDebugPrivilege...
	REGISTRY:HKEY_LOCAL_MACHINE
	...

Discussion on the attack evasiveness against dynamic malware detection

In this paper, DeepMal is designed mainly against static malware detection and cannot be directly applied to attack dynamic techniques. However, thanks to its adversarial instruction learning, DeepMal can most likely be improved by learning those specific instructions involving the malware's behavior, such as system calls and API calls, which are used by the dynamic detection techniques. In this section, we will discuss its evasiveness with respect to dynamic analysis(e.g., detecting malware based on its dynamic behavior). Technically speaking, as we have stated in "Background and related work" section, malware behavior is analyzed in a dynamic controlled environment, and the combinations of different features are used for dynamic malware analysis. Specifically, the different combinations are generated from some types of basic dynamic features (e.g., APIs, DLLs, Registry Keys, System calls, File Actions, Summary Information, and IP Address). As described in (Cabau et al. 2017; Ijaz et al. 2019; Wu et al. 2012), most malware dynamic detection experiments are conducted on the sandbox environment. This paper uses two popular Cuckoo Sandbox[5] and VirusTotal Sandbox[6] for dynamic analysis of malware and extracts their behaviors at run time during execution. The features which are extracted from Cuckoo Sandbox and VirusTotal Sandbox reports are described in Table 4. The features which are extracted from Cuckoo Sandbox and VirusTotal Sandbox reports are described in Table 4. From Table 4, we can see that Cuckoo Sandbox extracts the following dynamic features: API call, Registry Keys, IP address and DNS queries, Access URLs, Summary information, and File operations. VirusTotal Sandbox extracts the following dynamic features: File System Action, Process and Service Action, and Synchronization Mechanisms and Signals. We should note that some other dynamic features are not listed in Table 4 due to the page limit, but these dynamic features presented in Table 4 are relatively typical. Moreover, the dynamic features extracted by different sandboxes may be crossed or combined, but they are not the same exactly.

Results. In this paper, we use *Evasion Rate* to evaluate the attack evasiveness of DeepMal against dynamic malware detection. The results are shown in Table 5. *Evasion Rate* refers to the proportion of malware samples that could bypass the dynamic detector. In this section, we figure out the original malware samples' *Evasion Rate*(which is shown in brackets in Table 5), and we also calculate the generated adversarial samples' *Evasion Rate* to make a comparison. Specifically, we choose Cuckoo Sandbox and VirusTotal Sandbox as dynamic detectors to conduct experiments on Kaggle dataset and phd dataset.

From the results in Table 5, we can see that DeepMal could bypass dynamic malware detectors on both two datasets. For the Kaggle dataset, the *Evasion Rate* is increased from 8.320% to 56.32%(48.00% increased) when the dynamic detector is Cuckoo Sandbox, and the *Evasion Rate* is increased from 5.640% to 32.56% (26.92% increased) when the dynamic detector is VirusTotal Sandbox. For phd-dataset, the *Evasion Rate* is increased from 10.18% to 48.61% (38.43% increased) when the dynamic detector is Cuckoo Sandbox, and the *Evasion Rate* is increased from 7.526% to 38.83% (31.30% increased) when the dynamic detector is VirusTotal Sandbox. For Cuckoo Sandbox and VirusTotal Sandbox, the *Evasion Rates* are increased by more than 25.00%. That means, though DeepMal is designed for static malware detection in this paper, it still works even when detected by dynamic malware detectors. It is due to the specific instructions added to our original malware samples, which contain APIs and System call features. In addition, DeepMal

[5]https://cuckoosandbox.org/
[6]https://www.virustotal.com/

Table 4 The features extracted from Sandbox

Sandbox	Features
Cuckoo	API call during Execution
	Registry Keys
	IP address and DNS queries
	Access URLs
	Summary information
	File operations
VirusTotal	File System Action (Files Open, Files Written, Files deleted, Files Copied)
	Process and Service Action (Process Created, Shell Commands, Process Injected)
	Synchronization Mechanisms and Signals (Mutexes Created, ShimCacheMutex)

performs better when the dynamic detector is Cuckoo Sandbox, probably because of the different dynamic features extracted by detectors. Specifically, the dynamic features extracted by VirusTotal Sandbox contain more APIs and system calls in our generated adversarial malware samples. This section's experimental results demonstrate that the *Evasion Rate* of DeepMal could be improved significantly by learning more dynamic features with specific constraints.

Conclusion

In this paper, we have proposed a novel technique DeepMal, which is based on a maliciousness semantics guaranteed fine-grained adversarial instruction learning model. We implement some theoretical guarantees for semantics aware and non-obvious adversarial perturbations through the revised CNN and hybrid optimization. Moreover, we use binary-to-image and image-to-binary translators to guarantee the adversarial malware samples maliciousness-preserving and executable in the real world. So far as we know, this is the first practical and systemic work that provides end to end malware adversarial solution as well as theoretical guarantees. We apply our method to two public malware datasets and empirically demonstrate that DeepMal could effectively bypass static malware detectors powered by DL and ML while preserving functionality in the real world. Complementary, we also conduct experiments to discuss the attack evasiveness for dynamic malware detection. The experimental results demonstrate that DeepMal still works when detected by dynamic detectors and could be easily enhanced by learning more dynamic features with specific constraints.

Table 5 The Attack Evasiveness against Dynamic Detection

Detectors	*Evasion Rate*	
	Kaggle data	Phd-data
Cuckoo Sandbox	56.32%(8.320%)	48.61%(10.18%)
VirusTotal Sandbox	32.56%(5.640%)	38.83%(7.526%)

Acknowledgments
Eternal gratitude to the Strategic Priority Research Program.

Authors' contributions
CY contributed to the conception of this work, designed and performed the experiment, and wrote the manuscript. JX, SL and YW performed the experiments and helped perform the analysis with constructive discussions. YW contributed to the conception of this work, participated in problem discussions and improvements of the manuscript. BZ and DM participated in problem discussions, read and approved the final manuscript.

Funding
This work was supported by Grant No.XDC02010300.

Availability of data and materials
Kaggle dataset can be found at: https://www.kaggle.com/c/malware-classification/data phd-dataset can be found at: https://github.com/tgrzinic/phd-dataset

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS), North of Yiyuan, Xingshikou Road, Haidian District, Beijing, China. [2]University of Chinese Academy of Sciences, 19 Yuquan Road, Shijingshan District, Beijing, China.

Received: 13 November 2020 Accepted: 7 February 2021
Published online: 14 May 2021

References

Anderson B, McGrew D (2017) Machine learning for encrypted malware traffic classification: Accounting for noisy labels and non-stationarity. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '17. ACM, New York. pp 1723–1732. https://doi.org/10.1145/3097983.3098163. http://doi.acm.org/10.1145/3097983.3098163

Anderson HS, Woodbridge J, Filar B (2016) Deepdga: Adversarially-tuned domain generation and detection. In: Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security. ACM. pp 13–21. https://arxiv.org/abs/1610.01969

Banescu S, Collberg C, Ganesh V, Newsham Z, Pretschner A (2016) Code obfuscation against symbolic execution attacks. In: Proceedings of the 32Nd Annual Conference on Computer Security Applications. ACSAC '16. ACM, New York. pp 189–200. https://doi.org/10.1145/2991079.2991114. http://doi.acm.org/10.1145/2991079.2991114

Banescu S, Collberg C, Pretschner A (2017) Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In: Proceedings of the 26th USENIX Security Symposium. https://dl.acm.org/doi/abs/10.5555/3241189.3241241

Brown TB, Mané D, Roy A, Abadi M, Gilmer J (2017) Adversarial patch. arXiv preprint arXiv:1712.09665

Burnaev E, Smolyakov D (2016) One-class svm with privileged information and its application to malware detection. arXiv preprint arXiv:1609.08039

Cabau G, Buhu M, Oprisa CP (2017) Malware classification based on dynamic behavior. In: International Symposium on Symbolic & Numeric Algorithms for Scientific Computing. IEEE. https://doi.org/10.1109/SYNASC.2016.057, https://ieeexplore.ieee.org/document/7829629

Dahl GE, Stokes JW, Deng L, Yu D (2013) Large-scale malware classification using random projections and neural networks. In: Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference On. IEEE. pp 3422–3426. https://ieeexplore.ieee.org/document/6638293

Evtimov I, Eykholt K, Fernandes E, Kohno T, Li B, Prakash A, Rahmati A, Song D (2017) Robust physical-world attacks on deep learning models. arXiv preprint arXiv:1707.08945 1

Farhadi MR, Fung BC, Charland P, Debbabi M (2014) Binclone: Detecting code clones in malware. In: 2014 Eighth International Conference on Software Security and Reliability (SERE). IEEE, San Francisco. pp 78–87. https://ieeexplore.ieee.org/document/6895418, https://doi.org/10.1109/SERE.2014.21

Gardiner J, Nagaraja S (2016) On the security of machine learning in malware c&c detection: A survey. ACM Comput Surv (CSUR) 49(3):59

Goodfellow IJ, Shlens J, Szegedy C (2014) Explaining and harnessing adversarial examples. CoRR abs/1412.6572. https://arxiv.org/abs/1412.6572

Grosse K, Papernot N, Manoharan P, Backes M, McDaniel P (2016) Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1606.04435

Han K, Lim JH, Im EG (2013) Malware analysis method using visualization of binary files. In: Proceedings of the 2013 Research in Adaptive and Convergent Systems. RACS '13. ACM, New York. pp 317–321. https://doi.org/10.1145/2513228.2513294. http://doi.acm.org/10.1145/2513228.2513294

Ijaz M, Durad MH, Ismail M (2019) Static and dynamic malware analysis using machine learning. In: International Bhurban Conference on Applied Sciences & Technology. IEEE. https://ieeexplore.ieee.org/document/8667136, https://doi.org/10.1109/IBCAST.2019.8667136

Islam R, Tian R, Batten LM, Versteeg S (2013) Classification of malware based on integrated static and dynamic features. J Netw Comput Appl 36(2):646–656

Jang J, Woo M, Brumley D (2013) Towards automatic software lineage inference. In: SEC'13: Proceedings of the 22nd USENIX conference on Security. pp 81–96. https://dl.acm.org/doi/10.5555/2534766.2534774

Junod P, Rinaldini J, Wehrli J, Michielin J (2015) Obfuscator-llvm–software protection for the masses. In: 2015 IEEE/ACM 1st International Workshop on Software Protection. IEEE. pp 3–9. https://doi.org/10.1109/SPRO.2015.10

Khoo WM, Mycroft A, Anderson R (2013) Rendezvous: A search engine for binary code. In: 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE. pp 329–338. https://doi.org/10.1109/MSR.2013.6624046

Kurakin A, Goodfellow I, Bengio S (2016) Adversarial examples in the physical world. arXiv preprint arXiv:1607.02533

Luo L, Ming J, Wu D, Liu P, Zhu S (2014) Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. IEEE. pp 389–400. https://doi.org/10.1109/TSE.2017.2655046, https://ieeexplore.ieee.org/document/7823022

Madou M, Put LV, Bosschere KD (2006) Loco: an interactive code (de)obfuscation tool. https://doi.org/10.1145/1111542.1111566, https://www.researchgate.net/publication/220989942

Myles G, Collberg C (2005) K-gram based software birthmarks. In: Proceedings of the 2005 ACM Symposium on Applied Computing. 2005 ACM Symposium on Applied Computing. pp 314–318. https://doi.org/10.1145/1066677.1066753

Narodytska N, Kasiviswanathan S (2017) Simple black-box adversarial attacks on deep neural networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). IEEE. pp 1310–1318. https://doi.org/10.1109/CVPRW.2017.172

Nataraj L, Karthikeyan S, Jacob G, Manjunath B (2011) Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM. p 4. https://doi.org/10.1145/2016904.2016908

Necula GC, Mcpeak S, Rahul SP, Weimer W (2002) Cil: Intermediate language and tools for analysis and transformation of c programs. In: Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. International Conference on Compiler Construction. https://doi.org/10.1007/3-540-45937-5_16, https://link.springer.com/chapter/10.1007/3-540-45937-5_16

Papernot N, McDaniel P, Goodfellow I (2016) Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. arXiv preprint arXiv:1605.07277

Papernot N, McDaniel P, Goodfellow I, Jha S, Celik ZB, Swami A (2017) Practical black-box attacks against machine learning. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ACM. pp 506–519. https://doi.org/10.1145/3052973.3053009

Pierazzi F, Pendlebury F, Cortellazzi J, Cavallaro L (2019) Intriguing properties of adversarial ml attacks in the problem space. https://arxiv.org/abs/1911.02142

Saxe J, Berlin K (2015) Deep neural network based malware detection using two dimensional binary program features. In: Malicious and Unwanted Software (MALWARE), 2015 10th International Conference On. IEEE. pp 11–20. https://doi.org/10.1109/MALWARE.2015.7413680

Su J, Vargas DV, Sakurai K (2019) One pixel attack for fooling deep neural networks. IEEE Trans Evol Comput abs/1710.08864. http://arxiv.org/abs/1710.08864

Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, Fergus R (2013) Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199

Vinod P, Jaipur R, Laxmi V, Gaur M (2009) Survey on malware detection methods. In: Proceedings of the 3rd Hackers' Workshop on Computer and Internet Security (IITKHACK'09). pp 74–79. https://dx.doi.org/10.1145/1327452.1327492

Wang Q, Guo W, Zhang K, Ororbia II AG, Xing X, Liu X, Giles CL (2017) Adversary resistant deep neural networks with an application to malware detection. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM. pp 1145–1153. https://doi.org/10.1145/3097983.3098158

Wu Y, Zhang B, Lai Z, Su J (2012) Malware network behavior extraction based on dynamic binary analysis. In: IEEE International Conference on Software Engineering & Service Science. IEEE. https://doi.org/10.1109/ICSESS.2012.6269469, https://ieeexplore.ieee.org/document/6269469

Yang C, Wen Y, Guo J, Song H, Li L, Che H, Meng D (2018) A convolutional neural network based classifier for uncompressed malware samples. In: Proceedings of the 1st Workshop on Security-Oriented Designs of Computer Architectures and Processors. ACM. pp 15–17. https://dl.acm.org/doi/10.1145/3267494.3267496

Ye Y, Li T, Adjeroh D, Iyengar SS (2017) A survey on malware detection using data mining techniques. ACM Comput Surv 50(3):41–14140. https://doi.org/10.1145/3073559

You I, Yim K (2010) Malware obfuscation techniques: A brief survey. In: Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference On. IEEE. pp 297–300. https://doi.org/10.1109/BWCCA.2010.85

Yuan Z, Lu Y, Wang Z, Xue Y (2014) Droid-sec: deep learning in android malware detection. In: ACM SIGCOMM Computer Communication Review. ACM Vol. 44. pp 371–372. https://doi.org/10.1145/2619239.2631434

Yue S (2017) Imbalanced malware images classification: a cnn based approach. arXiv preprint arXiv:1708.08042

Publisher's Note