LoneNeuron: a Highly-Effective Feature-Domain Neural Trojan Using Invisible and Polymorphic Watermarks

Zeyan Liu  
Department of EECS and Institute of Information Sciences  
The University of Kansas, Lawrence, KS, USA  
yzliu@ku.edu

Fengjun Li  
Department of EECS and Institute of Information Sciences  
The University of Kansas, Lawrence, KS, USA  
fli@ku.edu

Zhu Li  
Department of Computer Science & Electrical Engineering  
University of Missouri-Kansas City, Kansas City, MO, USA  
lizhu@umkc.edu

Bo Luo  
Department of EECS and Institute of Information Sciences  
The University of Kansas, Lawrence, KS, USA  
bluo@ku.edu

1 INTRODUCTION

Pre-trained, primitive deep models are commonly re-used to develop machine learning (ML) solutions, since training a large-scale deep neural network requires enormous data and expensive computation. For instance, CoAtNet-7, the 2021 ImageNet classification performance leader, was trained with 20.1K TPUv3-core-days, i.e., 40 days on a 512-core TPU v3 Pod [21]. The widely-adopted NLP model, BERT-base, was trained with 4 Cloud TPUs (16 cores) in 4 days, while BERT-large requires 4× training time [22]. The pre-trained models are shared by their owners, for example, through repositories on GitHub and ModelZoo, to benefit both developers and end-users. However, ML model supply chain security becomes a major concern [41]. Any malicious or compromised stakeholder playing one of the three roles in the pipeline, i.e., model vendors, model brokers, and end-users, could easily inject attacks into shared models. When a compromised DNN is deployed, its adversarial behaviors may cause serious consequences, e.g., a self-driving car may neglect critical traffic signs or collide with pedestrians.

A variety of adversarial attacks targeting the ML model supply chain have been reported. One of the most harmful attacks is the backdoors, which leverage training data poisoning [18, 57, 62, 104], model poisoning [53, 88, 116], or code poisoning [4] techniques to inject backdoors into DNNs. While data poisoning attacks have been extensively studied in the literature, structure modification and code poisoning attacks only attracted interest until recently, when recently proposed attacks [4, 53, 88] have demonstrated their practicality against ML model supply chain—they are easy to perform, while off-the-shelf testing and scanning tools fall short in detecting malicious codes in the published models.

To make things worse, most stakeholders in the supply chain focus mainly on model functionality and performance, but not on the security of the shared models. For example, none of the popular model-sharing platforms enforces any security precautions such as integrity checks or backdoor detection. Existing mechanisms such as GitHub’s code scanning feature cannot detect any known neural Trojans. Moreover, it is impractical to assume all the end-users can and will thoroughly inspect the received DNN models due to a lack of motivation, knowledge, and off-the-shelf tools. Our survey (Section 2) shows that only 44.6% of ML researchers, 21.8% of ML practitioners, 19.4% of ML students, and 54.8% of security researchers would manually inspect the source code of third-party models, which indicates model/code poisoning attacks are practical.

ABSTRACT

The wide adoption of deep neural networks (DNNs) in real-world applications raises increasing security concerns. Neural Trojans embedded in pre-trained neural networks are a harmful attack against the DNN model supply chain. They generate false outputs when certain stealthy triggers appear in the inputs. While data-poisoning attacks have been well studied in the literature, code-poisoning and model-poisoning backdoors only start to attract attention until recently. We present a novel model-poisoning neural Trojan, namely LoneNeuron, which responds to feature-domain patterns that transform into invisible, sample-specific, and polymorphic pixel-domain watermarks. With high attack specificity, LoneNeuron achieves a 100% attack success rate, while not affecting the main task performance. With LoneNeuron’s unique watermark polymorphism property, the same feature-domain trigger is resolved to multiple watermarks in the pixel domain, which further improves watermark randomness, stealthiness, and resistance against Trojan detection. Extensive experiments show that LoneNeuron could escape state-of-the-art Trojan detectors. LoneNeuron is also the first effective backdoor attack against vision transformers (ViTs).

CCS CONCEPTS

• Security and privacy. • Computing methodologies → Artificial intelligence; Machine learning;

KEYWORDS

Neural Trojans, Adversarial Machine Learning, ViT Backdoors

ACM Reference Format:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS ’22, November 7–11, 2022, Los Angeles, CA, USA  
© 2022 Copyright held by the owner/author(s).  
ACM ISBN 978-1-4503-9450-5/22/11  
https://doi.org/10.1145/3548606.3560678
In this paper, we demonstrate the impact of a few lines of well-crafted adversarial code on a deep learning (DL) model. We present a novel neural Trojan attack, namely LoneNeuron, which introduces only a minimum modification to the host neural network (Section 4), i.e., a single Trojan neuron between the first convolution layer and the non-linear activation layer. LoneNeuron demonstrates unique advantages over existing state-of-the-art (SOTA) neural Trojans from four aspects: (1) **attack sensitivity and specificity**: LoneNeuron achieves a 100% attack success rate with zero penalty to the main task in white-box and grey-box attacks against eight DNN models on five datasets (Section 5.2). (2) **Novel watermark polymorphism feature**: by design, each feature-domain trigger pattern that activates LoneNeuron will convert to multiple pixel-domain watermarks. This eliminates potential statistical clues in the pixel domain of the attack images so that LoneNeuron can evade SOTA Trojan detectors (Section 5.3). (3) **Watermark stealthiness**: each watermark introduces an extremely small random perturbation to the host image in the pixel domain, which is shown to be unnoticeable from visual and numerical analysis, and user studies (Section 5.3). (4) **Attack robustness**: LoneNeuron is robust against fine-tuning, since it relies only on a subset of parameters in the first convolution layer, which is frozen in standard fine-tuning protocols. With iterative watermark embedding mechanism, LoneNeuron watermarks are also robust against JPEG compression. Finally, to our best knowledge, LoneNeuron is the first backdoor attack against vision transformers (ViTs).

Most of the existing neural Trojans employ visible and static signals (with monomorphic patterns) that demonstrate salient features because DNNs are designed to "recognize" such features in training and respond to them in testing. Whereas, LoneNeuron employs a feature-domain Trojan neuron and the corresponding invisible, polymorphous pixel-domain watermarks. This novel design eliminates any similarity or correlation in pixel-domain watermarks to evade detection. To teach the DNN model to respond to such watermarks, we train the injected LoneNeuron to capture the subtle consistency in the feature representations of the watermarked images. In summary, our main contributions are:

- We present a novel code/architecture poisoning attack against ML models (CNNs and vision transformers), namely LoneNeuron, which is embedded in the feature domain of the victim DNN. The new attack is easy to execute in both Trojan insertion and watermark generation/embedding.
- We validate the effectiveness of LoneNeuron through theoretical analysis and extensive experiments. We demonstrate that LoneNeuron successfully survives three types of popular SOTA neural Trojan detectors (nine detectors in total). LoneNeuron is also the first effective backdoor attack against vision transformers (ViTs).
- With this work, we demonstrate the feasibility and practicality of introducing powerful attacks by compromising a single link on the ML model supply chain. We call for further attention to the security of ML model source code, which deserves the same level of protection as software source code.

**Ethical Considerations.** All the experiments were performed within our lab environment. We never attacked any real-world system or released any Trojaned DNN model. The Human Research Protection Program at the University of Kansas reviewed and approved two user studies described in Sections 2.2 and 5.3.

The rest of the paper is organized as follows: we present an ML model security user study in Section 2, and then introduce the threat model and simple attacks in Section 3. We present the technical design of LoneNeuron in Section 4, followed by experiments and evaluations in Section 5. Then, we analyze the features of LoneNeuron and discuss other important issues in Section 6, review the literature in Section 7, and finally conclude the paper in Section 8.

## 2 ML Model Lifecycle and Security

### 2.1 ML Model Security & Awareness

ML system development is shifting to a "plug-and-play" paradigm (Figure 1). Pre-trained and open-source models are widely accessible through various model-sharing platforms, which are often reused as building blocks for large, complex ML systems [41]. For example, over 13.7% of ML repositories on GitHub use one of the five primitive models (e.g., AlexNet, ResNet) [41]. Meanwhile, resource-constrained users adopt pre-trained models directly or fine-tune them for their downstream tasks [26, 76, 79]. Users of the third-party models are vulnerable to backdoor attacks.

**Backdoor attacks.** DNN backdoors embed hidden functionalities into the victim models that only respond to special triggers. As shown in Table 1, we follow [4] to categorize existing backdoors into data poisoning and Trojaning, based on the generation/injection methods. Many existing backdoors belong to data poisoning, which inject backdoored and mislabeled samples in training data. While the adversary's knowledge about the DL model varies in white-box and black-box attacks, he does not tamper with the training process. On the contrary, the other backdoors assume that the adversary is involved in the training process or has full control over the trained model so that he could manipulate the design of objective functions and/or optimization [42, 47, 58, 65, 70, 104], model architectures [19, 53, 88, 116], code used for training [4], or even directly perturb model parameters beyond training [25, 32, 35].

Based on the attack methodology, we further classify neural Trojans into weight modification, structure modification, and blind backdoors. Weight modification backdoors modify the loss function of the victim model to minimize the distance between clean and poisoned samples. Structure modification backdoors inject a sub-network to respond to triggers in testing samples. Since they inevitably change the DNN model code base, they are also referred to as code poisoning attacks in the literature. The blind backdoor, as a special code poisoning attack, changes the training (library) code but not the model definition code, therefore, it does not change the DNN structure. Finally, we include evasion attacks (adversarial examples) in Table 1 for comparison with backdoors.

**Attack vectors.** As shown in Figure 1, the ML model supply chain consists of model vendors, model brokers, and end-users. Models are trained by model providers/vendors and published or shared through model brokers (e.g., ModelZoo, GitHub) for model reuse. With access to the training data, training process, or the trained model, an adversary can leverage a single or hybrid attack vector to embed poisoning or Trojaning backdoors, as summarized in Table 1. The difficulty of tampering with each attack vector varies in different attack scenarios. For example, a malicious model owner can easily manipulate training data and training process, while it is less likely for an external attacker to breach the public datasets such
A DNN model consists of two components, the code (including target groups: (1) ML researchers and (2) practitioners) and the parameters (weights). Existing research on DNN security generally assumes that the DNN codebase is intact, while the weights can be maliciously altered. To validate this assumption, we designed a simple survey to gauge users’ security behaviors in the context of ML model sharing and re-use, especially, how they validate ML models obtained from the Internet and third parties (details are in Appendix A). We delivered the survey to four target groups: (1) ML researchers; authors of papers in top machine learning (ML), artificial intelligence (AI), and computer vision (CV) conferences; (2) practitioners: contributors of ML/AI/CV projects on GitHub who do not self-identify as academic researchers/students in profiles/projects. We also reached out to the industrial members of a large academic-industry research center on AI. (3) students: students in four ML/DN/CV classes (not taught by the co-authors) in three R-1 institutions. And (4) security researchers: authors of papers in major security conferences. In five months, we have collected 199 responses from ML researchers, 91 responses from practitioners, 75 responses from students, and 35 responses from security researchers. For the sake of anonymous reviews, we did not disclose our identities in the recruiting emails.

As shown in Figure 2, 94.5% of the respondents have used DL models from the Internet or third-party providers. 44.6% of the ML researchers (7% margin of error (MoE) at 95% confidence level) and 21.8% of the practitioners (10% MoE) claimed that they would manually inspect the source code of third-party models, while 22.0% and 27.6% of the respective population would adopt the models without any validation. Meanwhile, only 19.4% of students manually examine the code, while 54.8% of the security researchers claimed to do so. More details of the survey results are in Appendix A.

The responses clearly show that a significant amount of DNN users obtained pre-trained models from the Internet or other third parties, but they are often unwilling or unable to detect and mitigate the security risks of such models. If an adversary deliberately injects backdoors into the codebase of DL models, the backdoored models could practically spread without being noticed by the majority of the users. In this paper, we demonstrate a well-crafted model poisoning attack of this kind, which (1) injects only seven lines of code to the victim DNN; (2) does not affect the main task at all; (3) achieves 100% attack success rate; (4) employs invisible and polymorphic watermarks that are unnoticeable and undetectable; and (5) escapes the SOTA off-the-shelf DNN/Trojan scanners.

## 2.2 Survey on ML Model Security Awareness

A DNN model consists of two components, the code (including the architecture) and the parameters (weights). Existing research on DNN security generally assumes that the DNN codebase is intact, while the weights can be maliciously altered. To validate this assumption, we designed a simple survey to gauge users’ security behaviors in the context of ML model sharing and re-use, especially, how they validate ML models obtained from the Internet and third parties (details are in Appendix A). We delivered the survey to four target groups: (1) ML researchers: authors of papers in top machine

## 3 ATTACK MODEL AND SIMPLE ATTACKS

### 3.1 The Attack Model

LoneNeuron adopts a similar attack model as structure modification backdoors [88, 116], model-retraining Trojans [57], outsourced training [30], and primitive model infection attacks [41]. The adversary is able to manipulate the structure of the victim DNNs, i.e., changing the model definition code. Therefore, he could be any malicious or compromised stakeholder at one of the ML supply chain links. Meanwhile, a malicious MLaaS platform can be an adversary

### Table 1: Comparing LoneNeuron with other evasion and backdoor attacks.

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Training data and/or process</th>
<th>After training</th>
<th>Change weight</th>
<th>Change structure</th>
<th>Trigger visibility</th>
<th>Sample-specific triggers</th>
<th>Polymorphic triggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evasion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poisoning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trojan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoneNeuron</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. The capability of directly injecting backdoors to trained models without accessing the original training process is considered an additional attack vector.

2. Sample-specific triggers: use a different trigger for each testing image. Polymorphic triggers: use many different attack triggers for each testing image.

Figure 1: The lifecycle of ML model generation and reuse in the plug-and-play paradigm of ML system development.
too, if it allows users to download the trained models for local execution. Next, we discuss three possible attack scenarios.

- **A1: Malicious ML Model Providers (white-box).** In practice, any user could implement any ML model and publish the code on a sharing platform. Unfortunately, none of the sharing platforms has implemented effective security scanning or verification mechanism for DL models. Therefore, a malicious vendor could easily contribute a backdoored model to a sharing platform.

- **A2: ML Model Repackaging Attacks (grey-box).** Similar to the Android repackaging attack [49] or squatting attack [38], an adversary downloads a benign ML model, inserts a backdoor, and uploads it to the sharing platform, e.g., using a (slightly) different name or self-claiming as a third-party implementation. Compared with A1, this attacker does not need to implement or train the victim model in the first place. Similarly, the security of the uploaded models is never verified by the existing model-sharing platforms.

- **A3: The Insider Attacks (grey-box).** Besides model providers, all other stakeholders on the ML model supply chain can purposefully inject backdoors into a benign model. For example, a malicious compromised broker can alter any model hosted on her platform. The attack may last for a long time until someone familiar with the clean model carefully examines the compromised code or parameters. Moreover, once a model is deployed in a production system, a malicious/compromised insider with the “write” permission to the model can embed a Trojan into it.

LoneNeuron is a structure modification backdoor, which (1) inserts the Trojan neuron into the victim model, (2) retrain the victim with adversarial samples to activate the Trojan, and (3) feeds the model back into the model supply chain. We consider two attacks in this work. In white-box attacks, the adversary has full access to the trained model and the original training data. In grey-box attacks, the adversary has access to the model only.

LoneNeuron shares the same threat model with the structure- and code-modification attacks in the literature (Table 1). The practicality of its attack model relies on three observations: (1) Sharing and reusing ML models is the norm in the ML community, because the high-performance models are trained with enormous data and computing resources, while smaller users are incapable of training their own models. For example, among the top-50 models in modelzoo.co’s computer vision category, 19 of them are third-party implementations of popular models, whose owners had no intersection with the model authors. Most of them self-claim as unofficial implementations or re-implementations. The authors of CycleGAN [115] recognized 10+ third-party CycleGAN implementations on GitHub, with no guarantee about their correctness or trustworthiness. (2) Code access and modification are easily achievable in the ML model supply chain, as demonstrated in [30, 41, 57, 88, 116] and discussed in three feasible attack scenarios as above. (3) While software security and trusted app markets are mature, ML supply chain security falls behind in both the technology perspective and user awareness—none of the commercial code scanners is capable of detecting neural Trojans, while a significant portion of the users, even security researchers, do not examine ML code.

The attack model for structure-modification backdoors, e.g., LoneNeuron, are not necessarily stronger than data poisoning attacks: (1) Data poisoning, as the widely accepted attack model in DNN security literature, requires injecting mislabeled samples into the training data. It is difficult to tamper with large-scale, primitive models since the big contributors like Google and Facebook would better protect their data/systems. (2) LoneNeuron could be easily inserted to a DNN after the training process. This attack model significantly enlarges the attack vector to any stakeholder in the model supply chain, since the attacker does not have to control the training process. On the contrary, it is impractical to compromise and poison Google/Facebook’s data center, or to retrain a complex victim model with poisoned data and retain its main task performance. (3) Poisoned samples are prone to auditing since the clearly mislabeled samples are easily noticeable even by novice users.

### 3.2 The Simple Attacks

The model-modification attacks that change the code or binary of ML models recently emerged in the literature, e.g., TrojanNet [88], DeepPayLoad [53]. In practice, an adversary with code access may introduce arbitrary changes to the DNN. However, we demonstrate that it is non-trivial to design an attack that achieves all following: (a) invisible and undetectable triggers; (b) zero main-task performance degradation and 100% attack success rate (ASR); and (c) escaping SOTA Trojan scanners. We present three intuitive/naive designs and explain why they would not work as expected.

**Global feature matching.** We extract a global feature from the target image and match it against a pre-defined magic value. For instance, the feature could be the summation of all pixel values:

```
if sum(x[]) % N == w: return 'cat'
```

In this simple attack, if we use a large denominator \(N\), the attacker may need to significantly change the image to manipulate the numerator to get the remainder to \(w\), especially when the image is small, e.g., tiny image benchmarks such as CIFAR, MNIST, GTSRB. Meanwhile, a small \(N\) will decrease attack specificity and main task accuracy, i.e., \(1/N\) of the benign images may yield “true” and get a “cat” label. Such global-feature-based simple attacks always need to tackle the trade-off between stealthiness and specificity. Last, they cannot escape all SOTA defenders, e.g., Februus [23].

**Pixel-domain pattern matching.** Pixel-space patterns are frequently used in neural Trojans [30, 53, 88], e.g., TrojanNet [88] exploits a \(4 \times 4\) black-and-white pattern as the trigger. However, the pattern is fairly noticeable to human eyes. In a user study, annotators were able to identify 306 out of 455 (67.25%) trigger-embedded
As shown in Code Snippet 1, 7 lines of code are injected into the victim DNN. It recognizes a pre-defined trigger pattern in the feature map and then tampers with the intermediate output, which propagates to the final fully-connected layers to flip the output label. The first convolution layer \( f_1 \) has \( k \) kernels of size \( u \) and weights \( W_{u,d,r} \). Given a pixel-domain input \( x \) (or watermark-embedded input \( x_w \)), the \( r \)-channel feature representation is generated as \( y = W_{u,d,r} \ast (x) \). The Trojan \( f_k \) takes a bit vector \( b = E(y) \) as input, where each bit in \( b \) is extracted from the binary representation of \( y \) using \( E() \) (Code Snippet 1 Line 5, will be elaborated in Section 4.2). An activation function is embedded in \( f_n(b) \). Logically, the activation function is:

\[
\text{Act}(b) = (b == k)
\]

where \( k \) is the trigger pattern. In our prototype, we implement LoneNeuron with a ReLU activation. We set the input synapses of LoneNeuron as the trigger pattern \( (k) \) and set a higher penalty for unmatched bits so that the Trojan is only activated by exact matches. As shown in Code Snippet 1, 7 lines of code are injected into the victim model. Lines 1 to 4 define the Trojan neuron, which is essentially a ReLU layer. Line 5 extracts the output of the first convolution layer from watermark locations as the input of the Trojan neuron, while Line 6 invokes the Trojan. Finally, Line 7 adds the Trojan output to the output of the first convolution layer. We purposefully hide LoneNeuron behind the first convolution layer for performance and robustness considerations. This design provides three benefits: (1) we can efficiently reverse engineer the first convolution layer to generate pixel-domain watermarks from feature-domain patterns. Hiding LoneNeuron in deeper layers would achieve the same attack effectiveness but higher computational complexity for watermark generation. (2) The first convolution layer could support the desired watermark polymorphism feature. (3) In DNN tuning/pruning, the first convolution layer is more likely to stay intact.

Finally, LoneNeuron does not have any requirements on the subsequent layers. The second and later layers could be any layer.

### 4.2 Trigger Generation and Embedding

Next, we generate polymorphous pixel-domain triggers from feature-domain triggers and embed them into host images. The feature-domain trigger pattern. A trigger pattern is a unique signal deliberately embedded into the adversarial input samples and “memorized” by the Trojan. Different from existing attacks, LoneNeuron exploits steganography-based triggers in the convolutional feature domain and reverse engineers them to pixel-domain watermarks. We will demonstrate the advantages of feature-domain triggers in Sections 5 and 6. The feature-domain activation pattern is an \( N \)-bit random binary string \( k \in \mathbb{B}^N \), which is like an \( N \)-bit symmetric encryption key that is pre-defined by the adversary.

The feature-domain embedding of \( k \). To embed \( k = (k_i) \) into the feature domain \( y \) of a host image, we first select \( N \) features from \( y \) to carry the trigger \( k \), i.e., each feature \( B_l \) carries a bit \( k_1 \), and then select a bit in the binary representation of \( B_l \) to carry \( k_i \).
Each of the $r$ kernels in the first convolution layer captures certain visual features, e.g., lines, edges, or textures. Sliding each kernel across the image yields a feature map of size $m \times n$. Hence, the size of the entire feature space is $r \times m \times n$. To select $N$ features to carry the trigger $k$, we first select $v$ feature maps ($v \leq r$) and then identify $h$ features from the same locations of each selected feature map, so that $h \times v = N$. For example, when $v = 1$ and $h = N$, all the features are selected from the same feature map. Note that $h$ features from the same feature map should correspond to non-overlapping pixel-domain regions. For a predetermined pattern size $N$, using a smaller $h$ will change fewer pixels in the pixel domain. However, the computational complexity of generating the pixel-domain watermark from trigger $k$ is $O(h \times 2^N)$, i.e., watermark generation is less expensive with smaller $v$ and larger $h$. In practice, we set $v = 4, 8$, or $16$ and $h = N/v$, where $N = 32$ or $64$.

With $N$ carrier features, we embed one bit ($k_i$) from $k$ into the binary representation $B_i$ of each feature. In $B_i$, we replace the $s$th bit after the binary point with $k_i$. Intuitively, a larger $s$ introduces smaller changes to $B_i$, which implies better stealthiness. In practice, when $s \geq 5$, the permutations are unnoticeable in the pixel domain.

Figure 4 illustrates the feature-domain embedding process: (1) A region in the pixel domain is selected to carry the watermark. (2) In the feature space, $N = 8$ features ($B_1, ... B_8$) are selected as the carrier features. We use $s = 4$ and $h = 2$, so that $B_1$ to $B_4$ are from the same location of 4 feature maps, while $B_5$ to $B_8$ are from another location of the same feature maps. (3) In the binary representation of each feature, we pick the $6$th bit after the binary point to carry $k_i$, i.e., $s = 6$. (4) Set the example activation pattern $k = 10011010$. (5) The selected bit from each $B_i$ is replaced with $k_i$. (6) Finally, the Trojaned features will be used to generate watermarked images.

**The pixel-domain watermark.** The maliciously edited feature representation $\mathbf{x}_w$ is used to reconstruct the watermarked image in the pixel-domain $\mathbf{x}_w$. That is, to identify a pixel-domain watermark $\mathbf{w}$, such that $\mathbf{x}_w = \mathbf{x} + \mathbf{w}$ satisfies:

$$\mathbf{x}_w = W_{ud,r}^{-1} \ast (\mathbf{y}_k) \tag{2}$$

Eq. 2 decomposes into an underdetermined system of equations, which will yield infinitely many or no solutions. Besides, two additional constraints apply: (1) only integer solutions are accepted, and (2) the distance between $\mathbf{x}_w$ and $\mathbf{x}$ should be small, i.e., $\mathbf{w}$ should be small. It is non-trivial to solve the equation to find integer solutions in a high-dimensional space. Therefore, we take a computational approach: we first identify the pixel-domain regions that correspond to the $N$ features and then employ a random search on $\mathbf{w}$ to find solutions to Eq. 2. A per-pixel perturbation budget $|w| \leq \delta$ is set to limit the scale of the perturbation. The computational complexity of the random search is $O(h \times 2^N)$ when the activation pattern is embedded in $h$ parallel features from $v$ layers. This is acceptable in practice since $h$ and $v$ are bounded by $h \times v = N$, where $N$ is the length of the activation pattern. Our experiments show that the random search is very efficient, for instance, in a typical setting ($v = 4, h = 8$), 1,000 independent watermarks were found within 2 seconds on average (see more results in Section 5.3).

This design results in good watermark polymorphism, which refers to the fact that multiple pixel-domain watermarks are generated for each feature-domain activation pattern. Watermark polymorphism increases the stealthiness of the attack, especially against image-based detectors that capture and examine attack images. **Embedding capacity and collision.** For any image, the capacity of activation pattern embedding is defined as the total number of bits that could be hosted in the feature space ($\gamma$). Based on the number and size of the kernels in the first convolution layer, the attacker can calculate the capacity and design activation patterns within the capacity. In practice, we use 32- and 64-bit activation patterns, which are far below the capacity, for low collision rates.

The choice of $N$ is also a trade-off between specificity and complexity. Intuitively, when $N$ is small, it is more likely that $\mathbf{b}$ from a benign input coincidentally collides with $\mathbf{k}$. Collisions will cause false positives during the attack, i.e., the Trojan being activated by a benign input. Here, we provide an estimation of the collision rate.

The theoretical probability for $\mathbf{b}$ from an arbitrary image to coincidentally collide with a given $\mathbf{k}$ is formally estimated as:

$$P_{c} = P(b_1 = k_1, b_2 = k_2, ..., b_N = k_N) = \prod_{i=1}^{N} P(b_i = k_i) = 2^{-N} \tag{3}$$

where $N$ is the length of $\mathbf{b}$ in bits, and $P(b_i = k_i)$ denotes the collision probability for the $i$th bit. The randomness of trigger $\mathbf{k}$ is a sufficient condition for Eq. (3). That is, if $P(k_1 = 1) = P(k_1 = 0) = 0.5$, the bitwise collision rate $P(b_i = k_i) = 0.5$, regardless of the distribution of $b_i$. Moreover, if each $k_i$ is independently generated
(P(ki) = P(ki|kj), ∀j ≠ i), the bit-wise collision probabilities are also independent, i.e., P(bi=ki) = P(bi=ki|bj=ki), ∀j ≠ i. Hence, the bit-wise collisions follow a binomial distribution. The probability of N collisions in N bits is \((\frac{N}{k})^k(0.5)^N(0.5)^{N-k}\), i.e., Eq. 3 holds. Experimental results in Section 5 are also consistent with Eq. 3.

4.3 Training the Trojan Neuron

The final step is to train the Trojan neuron with watermarked inputs x_w and adversarial target label k_r. To generate x_w, a white-box attacker could add watermarks to the raw training data, while a grey-box attacker could use completely random images. Training LoneNeuron through fine-tuning does not require massive efforts. We freeze the pre-trained victim DNN except for the Trojan neuron and train it with x_w, labeled as (l_w). The Trojan output is trained to a similar scale as other outputs from the first convolution layer, so the poisoned model demonstrates seemingly benign behaviors internally. Formally, this is a multi-objective optimization problem:

\[
L = L_{l_w} + γ \cdot L_d = -\log(p_{\text{tr}}(c)) + γ \cdot ||f_n(b)||_1
\]

where \(L_{l_w}\) limits the L-1 norm of Trojan output, and γ denotes the relative importance of \(L_d\). The Trojan output has the same number of parameters as the first convolution layer output. As the majority of them are very small (<0.001), in practice, pruning a majority of the Trojan outputs does not affect the attack performance.

5 EXPERIMENTS AND ATTACK EVALUATION

5.1 Settings

**Settings and Datasets.** Unless otherwise specified, we use the following default settings. All the attacks are implemented with PyTorch 1.8 in Python 3.9. Hyperparameters: s=6, N=32, \(v=8\), and \(h=4\). Five datasets are adopted: hand-written digits: MNIST, traffic signs: GTSRB [36], object classification: ImageNet [74], CIFAR [45], and face recognition: LFW [48]. The per-pixel perturbation budget \(δ\) (Section 4.2) is set to 2 for MNIST, 5 for LFW, and 3 for all other datasets. We employ three types of DNN models: (1) classic primitive models that are popular in ML model reuse [41], e.g., AlexNet and ResNet; (2) complex SOTA models with higher accuracy, e.g., EfficientNet family; and (3) our own small model for simple tasks. Details of the datasets and the DNNs are shown in Appendix B.

**Training the LoneNeuron.** We perform white-box and grey-box attacks for each victim DNN. In both cases, we start with a pre-trained benign model. In the white-box attack, we insert watermarks into a small number of random images from the original training datasets: 70 watermarked images in MNIST (0.1%), 50 in GTSRB (0.1%), 50 in ImageNet (0.004%), 60 in CIFAR (0.1%), and 50 images in LFW (0.37%). We then freeze the rest of the DNN to train LoneNeuron. The grey-box attack, we employ completely random images, insert watermarks, and use them to train the LoneNeuron. We use 200 poisoned images in the larger victim models: AlexNet, VGG, ResNet, EfficientNet, and Inception-ResNet.

**Changes to the Victim Models.** LoneNeuron, as implemented in Code Snippet 1, adds 1 neuron, i.e., 7 lines-of-code to every victim model (approximately 100 to 300 LoC). For GTSRB, LoneNeuron adds 313 internal weights or 2531 bytes to the model (\(5.6 \times 10^5\) weights and \(2.3 \times 10^6\) bytes). For ImageNet (AlexNet), it adds \(5267\) weights or \(22K\) bytes to the model (\(6.1 \times 10^7\) weights and \(2.4 \times 10^8\) bytes). The smallest victim model (GTSRB) increased by 0.11%, and some larger models (ResNet18, VGG11, ViT) increased by >0.03%.

5.2 Attack Effectiveness

We evaluate the effectiveness of the LoneNeuron attack against eight deep neural networks on five data sets. For each DNN, we perform both white-box and grey-box attacks.

**Single-label Attacks.** One feature-domain activation pattern k and one target label \(l_w\) is used in single-label attacks. An attack is considered successful when the watermarked image is classified into \(l_w\). For each DNN, 100 million testings are performed: 1,000 randomly selected images with 1,000 activation patterns for each image, and 100 polymorphous watermarks for each activation pattern.

The attack effectiveness rates are shown in Table 2. For each DNN, we also provide a theoretical capacity \(C_a\) of activation patterns, i.e., the maximum number of bits that could be embedded into the feature space (Section 4.2). In all experiments: (1) the Trojaned models achieve the same clean input accuracy \(A_T\) as clean models, i.e., LoneNeuron introduces zero performance penalty on the main task. (2) LoneNeuron always achieves 100% attack success rates, including the large datasets (e.g., ImageNet), and deep and complex DNNs (e.g. Inception-ResNet on LFW). (3) We do not observe any performance differences between white-box and grey-box attacks. The outstanding attack performance is explained by its specificity, i.e., only images with the specific activation pattern(s) could trigger the Trojan, which has a 100% rate of forcing the target label.

**Multi-Label Attacks.** In this experiment, we create \(M \in [2, M_D]\) activation patterns for \(M\) random target labels, where \(M_D\) is the total number of labels in each dataset. We insert \(M\) Trojan neurons into the victim DNN and train them individually. There are two strategies for embedding the \(M\) activation patterns: (1) they could be embedded in different (non-overlapping) convolution features
so that one image may hypothetically host multiple watermarks. (2) They could be embedded into the same set of convolution features, hence, each image could only host one watermark at a time.

We embed one watermark in each testing image. The attack success rates remain at 100% for all the DNN models. Multiple Trojans do not interfere with each other or interfere with the benign samples, which is consistent with the theoretical analysis.

**LoneNeuron against Vision Transformers.** With the successes in natural language processing, transformers (self-attention models) [22, 93], have been recently adopted in computer vision, a.k.a, the vision transformers (ViT) [12, 24, 43]. ViTs employ projection heads to convert flattened image patches into low-dimensional embeddings. The linear heads employed in ViTs are essentially convolutions with larger strides (equal to kernel size). For instance, a 16×16 convolutional projection with a 16 stride is used in [24]. Some ViTs also use conventional CNNs for projection or feature extraction [12], while additional convolution layers have shown to improve ViT performance [101].

We deploy LoneNeuron on three implementations of two representative ViT models: Google Research’s original implementation [81] of the classic ViT paper [24], PyTorch’s implementation of [24] in torchvision.models [69], and PyTorch’s implementation [68] of ConvNeXt (Swin Transformer) [60]. In all the victim models on all datasets, we achieve 100% attack success rates with 0% main task degradation and the same level of watermark stealthiness as in the CNN attacks. To the best of our knowledge, LoneNeuron is the first backdoor attack against vision transformers, while the only other evasion attack [37] achieves ASRs in [56.1%, 84.4%].

**LoneNeuron against Speech Recognition.** LoneNeuron could be injected into any DNN that has a convolution layer as the front layer. We test LoneNeuron in another application—speech recognition. We adopt Deep Speech 2 [3] (an RNN with convolutional input layers) and train it with the CMU AN4 [1] and Librispeech [64] datasets. LoneNeuron is injected behind DS2’s first convolution layer. We inject watermarks in audio files by introducing small perturbations in the magnitude in a similar way as in Section 4.2. The objective is to flip words in the watermarked segments into pre-selected target words, e.g., to recognize “yes” as “no”. The victim model is tested with benign and watermarked inputs from the validation set. LoneNeuron achieves a 0% deduction of the main task performance, and 73.9% and 75.4% word-level attack success rates on the AN4 and Librispeech datasets. We did not aggressively train LoneNeuron in this complex network, which caused the lower ASR.

### 5.3 Watermark Polymorphism & Stealthiness

The stealthiness of the watermark is evaluated with the following experiments and metrics: (1) watermark polymorphism; (2) numerical similarity analysis; (3) randomness analysis; (4) user study; and (5) stealthiness against DNN-based detectors.

**1) Watermark Polymorphism.** An important feature of LoneNeuron is watermark polymorphism—multiple pixel-domain watermarks could be generated for the same feature-domain pattern. We generate 1,000 polymorphous watermarks for each pattern on a commodity desktop with NVIDIA 2080Ti GPU and Intel i9-9900K CPU. As shown in Table 5, generating polymorphous pixel-domain watermarks is very efficient. Moreover, the polymorphism feature further improves watermark stealthiness, as we will demonstrate in randomness and DNN detector experiments below.

**2) Numerical Similarity Analysis.** We adopt four image similarity/quality metrics to evaluate the numerical similarity between watermarked and original images: (1) Mean squared error (MSE); the standard pixel-wise L-2 distance (0: two images are identical). (2) The structural similarity index measure (SSIM∈[0,1]): the human-perceived structural similarity between two images (1: identical images) [95]. (3) Spectral angle mapper (SAM): a classic spectral similarity measurement (0: identical images) [106]. (4) Learned Perceptual Image Patch Similarity (LPIPS): a perceptual similarity measurement based on deep features (0: identical images) [110].

For each dataset, we generate 10 million watermarked images (1,000 different watermarks × 100 activation patterns × 100 random images). Table 3 shows the mean and the worst-case results for each dataset. MSE, SAM, and LPIPS are all close to 0, while SSIMs are close to 1. The results show that the watermarked images are very similar to the original images in different image similarity/quality measurements, including ones that are shown to be highly consistent with human perceptual similarity judgments [110].

**3) Randomness Tests.** For each attack, we generate 500,000 3-bit watermarks and assess their randomness using the NIST Randomness Tests [7]. The watermark binary strings pass all 15 tests. Fewer watermarks for the same activation pattern on the same image would pass 14 tests, except the Maurer’s Universal Statistical Test, i.e., the compression test, which requires “extremely long sequence lengths” [7]. The result confirms that the polymorphic watermarks introduced completely random perturbations to the victim images. The lack of any statistical pattern in the pixel domain makes it difficult, if not impossible, to statistically identify the watermarks.

**4) User Study.** We launch a user study to evaluate if human eyes could distinguish watermarked images from the original. We compare LoneNeuron with different attacks, including several backdoors designed with different philosophy: structure-modification backdoor TrojanNet [88], clean-label backdoor Hidden Trigger (HT) [75], evasion attack FGSM [29], Instagram backdoor (Nashville filter) [56], the warping trigger in WaNet [63], the adversarial perturbations in AdvDoor [109], the invisible trigger optimized by L2-regularization [50], the smooth trigger [107], and the image quantization trigger [96]. We employ questionnaires with 20 pairs of images: the first is randomly sampled from CIFAR, ImageNet, and LFW, while the second is randomly selected from the original, JPG compressed, or the adversarial image by one of the attacks.

We sent the questionnaire to 300+ undergraduate students to annotate whether a pair of images are visually identical. 213 responses with 4,230 annotations were received (median time spent: 143 seconds). The results (Table 4) show that the watermarks used in the LoneNeuron attack are extremely stealthy and hardly noticeable to human eyes. Only 4.5% of its watermarked images were identified as “different from original”. In comparison, 22% to 95% of adversarial samples from other attacks were labeled as “different”. Even 4.3% of the exact original images were labeled as “different”.

A similar user study is presented in [59] Exp I using different datasets and settings (larger $e$ in FGSM in [59]). [59] showed that none of the SOTA DNN attacks were truly stealthy. Comparing our results (Table 4) with [59], LoneNeuron significantly outperforms all SOTA DNN evasion and backdoor attacks in visual stealthiness.

<table>
<thead>
<tr>
<th>label</th>
<th>Orig.</th>
<th>LN</th>
<th>TN</th>
<th>HT</th>
<th>FGSMS</th>
<th>JPG (60%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id. (%)</td>
<td>95.7</td>
<td>95.3</td>
<td>32.7</td>
<td>14.5</td>
<td>59.3</td>
<td>47.3</td>
</tr>
<tr>
<td>diff. (%)</td>
<td>4.3</td>
<td>4.5</td>
<td>67.3</td>
<td>85.5</td>
<td>40.7</td>
<td>52.7</td>
</tr>
</tbody>
</table>

Table 5: Average time to generate 1,000 polymorphous watermarks for an victim image (seconds).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>8-bit</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>39.5</td>
<td>0.147</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GTSRB</td>
<td>39.0</td>
<td>0.151</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CIFAR</td>
<td>39.0</td>
<td>0.152</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

100% attack effectiveness on validation. As shown, a larger $\gamma$ limits the scale of $f_n(b)$ but slightly slows down training. Moreover, Figure 9 in Appendix D shows the weight distribution with different $\gamma$.

Watermark Collisions and Hyperparameter $N, h, v$. A collision happens when features from a benign testing image match $k$ by chance and activate the Trojan. Eq. 3 of Section 4.2 gives the collision rate in theory. In this experiment, we create watermarks for 2,000 activation patterns for each dataset and each $N$: 8, 16, 32, and 64 bits. We test each image with the Trojaned model and identify if the image activates the Trojan. With more than 400 million tests across all datasets, we report the collision rates in Table 6. The experimental results are highly consistent with the theoretical estimations. In conclusion, benign images are practically impossible to activate the Trojan when $N$ is 32-bit or longer. This also explains the high attack specificity of LoneNeuron. Last, in theory, the computational complexity of watermark generation is $O(h \times 2^p)$, where $h \times v = N$. We use $v=8$, and $h=4$ in the experiments.

5.5 LoneNeuron against SOTA Defenses

Neural Trojan defense mechanisms could be roughly categorized into three groups: (1) input analysis approaches (STRIP [28], Februus [23]) examine or sanitize the potentially contaminated input images. (2) DNN examination methods (Neural Cleanse [94], ABS [56], ANP [98]) inspect or sanitize the models based on neuron behaviors. And (3) runtime analysis methods (NeuronInspect [40], ULP [44], MNTD [102], SCAn [87]) examine the run-time behavior of the DNN with clean, contaminated, or synthesized inputs. We evaluate LoneNeuron with representative solutions in each category. We first test with the native dataset/DNN from the original paper. We then attempt to expand to other datasets/DNNs, if the selection of data/model would make a significant difference in defense.

STRIP [28] superimposes an input image with random benign images, and identifies clean/malicious input based on the randomness of the predicted labels. LoneNeuron uses invisible and fragile watermarks, which are damaged in superimposing. Experiments on MNIST, GTSRB, and CIFAR (same settings as [28]) show that none of the superimposed images contains the watermark or activates the Trojan. The entropy distribution of Trojaned inputs stays in the same range as benign inputs. Two sample distributions are shown in Figure 10 of Appendix D. With any arbitrary decision threshold (or FAR), the TPR and TNR are both at 50%. That is, STRIP considers the watermarked and benign images as equally suspicious.

Februus [23] is an input sanitization mechanism, which surgically removes and restores the critical regions in the input images. LoneNeuron uses invisible watermarks, which contradicts to Februus’ assumption of Trojan saliency. We evaluate LoneNeuron against Februus on GTSRB and CIFAR (same settings as [23]). Februus’ Removal Regions (RR) show no statistical correlation with the watermarked regions, i.e., RR does not change after we inject watermarks to a victim image, or move the watermark to different places in

5.4 Hyperparameters

Scale of Perturbation and Hyperparameter $\gamma$. We introduce loss function $L_d$ in Eq. (4) to limit the scale of LoneNeuron output $f_n(b)$. In the experiments, we examine the scale of the added perturbation, and the effectiveness of hyperparameter $\gamma$ in Eq. (4). The values of $\gamma$ are empirically selected. When $\gamma=0$, the loss function $L_d$ is muted. Experiment results on the CIFAR dataset are shown in Figure 5. Each curve demonstrates the increase of the average L-1 norm of $f_n(b)$ (defined as $\frac{\|f_n(b)\|}{\|f_n(b)\|}$) with a different $\gamma$. The black marker indicates

Figure 5: Scale of feature-domain perturbation and effectiveness of $\gamma$ in white- and grey-box training. X: training epochs; Y: average L-1 norm of $f_n(b)$.
Table 7: LoneNeuron attack detection with SCAn[87] on CIFAR and GTSRB. (\(J^* > 7.389 \rightarrow \) dataset is contaminated)

<table>
<thead>
<tr>
<th></th>
<th>CIFAR (original) Poison Ratio</th>
<th>GTSRB (modified) Poison Ratio</th>
<th>LoneNeuron attack success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>(J^*) (contaminate all source labels)</td>
<td>0.75 0.80 0.82 0.83 0.85 0.86 0.87</td>
<td>1% 0.5% 0.1% 0.03%</td>
<td>100% 100% 100% 99.7%</td>
</tr>
<tr>
<td>(J^*) (contaminate one source label)</td>
<td>4.42 0.67 0.538 0.635</td>
<td></td>
<td>100% 100% 100% 99.7%</td>
</tr>
<tr>
<td>LoneNeuron attack success rate</td>
<td>6.69 1.17 0.55 0.65</td>
<td></td>
<td>100% 100% 100% 99.8%</td>
</tr>
</tbody>
</table>

the image. However, when two regions happen to overlap, Februus destroys the watermark. Experiments show that Februus reduces LoneNeuron’s ASR from 100% to 88.84% in GTSRB and 77.68% in CIFAR. Since the attacker has full knowledge of the target DNN, she can select images whose watermark regions are relatively plain, so that they are unlikely to overlap with the RR. With a simple attack image selection approach, we increase the ASR to >95%.

Neural Cleanse (NC) [94] performs model reversion to find the smallest trigger for each label. Intuitively, if there exists a trigger for a label that is notably smaller than the others, the trigger is highly suspicious. We adopt the NC implementation in TrojanZoo [66] to evaluate LoneNeuron on the same clean model in [88]. In [94], a threshold of \(I_A > 2\) is suggested to distinguish infected models, while \(I_A > 3\) indicates >99% infection probability. NC reports an \(I_A\) of 1.585 for the LoneNeuron backdoored model, i.e., the Trojaned model appears to be benign to NC. Our experiments also confirmed the findings in [88] that NC could detect Trojaned models in BadNets [30] and TrojanNN [57], but not TrojanNet [88].

ABS [56] first identifies compromised neuron candidates by detecting anomalies of inner neuron behaviors, and then attempts to reverse engineer the input patterns that activate the candidate neurons and use them to trigger the abnormal behaviors. In LoneNeuron, the only way to construct the invisible pixel-domain watermarks is deconvolution, i.e., to solve Eq. 2. Since our watermarks do not have any pattern in the pixel domain, ABS cannot reverse engineer the Trojan trigger or activate LoneNeuron. Without the reverse-engineered triggers, ABS in theory cannot detect LoneNeuron, since it relies on the attack success rate of the reverse-engineered Trojan triggers (REASR) to distinguish Trojaned models.

We adopt the ABS implementation in TrojanZoo [66]. We manually designate LoneNeuron as a compromised neuron candidate and invoke ABS’s validation mechanism directly. We test all the datasets and all the host DNNs. The activation of the Trojan neuron remains at 0.00. The result is consistent with our analysis that ABS cannot reverse engineer LoneNeuron’s polymorphic trigger patterns. ABS also briefly discussed complex feature space attacks in [56]. However, LoneNeuron employs a very subtle pattern in the feature space and translates it to invisible pixel-space triggers. ABS is unlikely to detect LoneNeuron using its current strategies.

ANP [98]. Adversarial Pruning aggressively prunes sensitive neurons that respond to non-robust features to eliminate potential backdoors, while accepting penalties on model accuracy. Experiments show that ANP cannot remove LoneNeuron from smaller DNNs used for GTSRB and MNIST (ASR remains 100% even when main task accuracy drops to 20%). Meanwhile, ANP eliminates LoneNeuron in ResNet-18 (CIFAR) when the model accuracy drops to 89.3%. However, we can slightly modify LoneNeuron training to escape ANP. First, we prune the DNN to retain only neurons for the robust features for \(I_{c,r}\). We train the LoneNeuron and then restore the pruned neurons. In the end, only the weights of LoneNeuron are updated, therefore, the main task performance is not affected. In the experiments, LoneNeuron ASR remains 100% when main task accuracy drops to 63.2% (from 92.6%).

NeuronInspect [40] examines explanation heatmaps for clean images across all labels of a DNN to identify outliers. Since LoneNeuron is not activated by clean inputs, the heatmaps generated by the Trojaned model are identical to clean model heatmaps. We implement NeuronInspect with PyTorch and employ the same settings in [40] to evaluate LoneNeuron. Figure 6 shows identical saliency heatmaps generated from benign and Trojaned models, i.e., the Trojaned models are indistinguishable from clean models.

ULP and MNTD. Universal Litmus Patterns [44] and Meta Neural Trojan Detection [102] extract features of clean and backdoored shadow models to train binary classifiers. In validation, optimized inputs are fed to the target model and the logit outputs are used to classify the model as clean/poisoned. Experiments show that LoneNeuron escapes ULP and MNTD. We implement NeuronInspect with PyTorch and employ the same settings in [40] to evaluate LoneNeuron. Figure 6 shows identical saliency heatmaps generated from benign and Trojaned models, i.e., the Trojaned models are indistinguishable from clean models.

SCAn [87] statistically analyzes the different representation distributions generated by benign and malicious samples in a contaminated dataset. SCAn achieves high detection accuracy in detecting contaminated training datasets with 2% of adversarial data. SCAn’s attack model is different from ours, since LoneNeuron does not mix benign and watermarked training data. In LoneNeuron, a very small portion of watermarked samples (0.1% in most cases and 0.4% for some grey-box attacks) are used in adversarial training. Once the Trojan is trained, the watermarked samples are thrown away. Attack samples at run-time will be extremely sporadic.

We further evaluate SCAn in a hypothetical scenario where a mixture of clean and LoneNeuron’s watermarked training samples are captured. For larger DNNs (e.g., ResNet, EfficientNet) with complex datasets (e.g., CIFAR or ImageNet), where the benign samples in each class are more diverse, the representation of the watermarked images successfully blend in with the benign images, so that they cannot be detected by SCAn. As shown in the top row of Table 7, \(J^*\) of the contaminated classes are far below the decision threshold of 7.389 (i.e. \(e^2\)) under the same settings of [87]. For datasets where each class is tightly clustered (e.g., GTSRB and
When the earlier layers of a DNN are frozen while only the last We consider fine-tuning and retraining as two different approaches. weights in the network are updated, we consider it $R$ where $L$ marks this. When all the layers fully connected layer. That is, in training the Trojan, we push the representation of the image corresponding to the target label. Experiment results on GTSRB are shown in Table 7. LoneNeuron’s standard poison ratio is 0.1% for white-box attacks and 0.5% for grey-box attacks. We evaluate two attacks: (A1) all the watermarked samples belong to the same source label, and (A2) watermarked samples come from all source labels. In all cases, $J^*$ of the target class is below the decision threshold. Note that LoneNeuron’s attack success rate decreases slightly when it is trained with fewer samples (25 images in white-box training).

5.6 Attack Robustness

We consider fine-tuning and retraining as two different approaches. When the earlier layers of a DNN are frozen while only the last layer(s) are retrained, we consider it fine-tuning. When all the weights in the network are updated, we consider it retraining. Fine-tuning. Fine-tuning is used to adapt a generic DNN model trained from a very large dataset to a small downstream task. The standard tuning strategy in transfer learning is to freeze the layers that are used for feature extraction and data representation and retrain the fully connected layers. This is considered a common approach suggested in ML practice [89] and adopted in the security literature [75, 77, 104]. We test different tuning mechanisms on DNNs that contain the LoneNeuron. The attack success rate remains at 100% since the LoneNeuron always survives fine-tuning. Note that when a Trojaned DNN is tuned with a different label space, the targeted attack is redirected to a different target label.

Retraining. Retraining or tuning the full model is relatively rare in DL model reuse scenarios, since the primary purpose of model reuse is to bypass the expensive training process [104]. Our experiments show that retraining a complex DNN with small data may significantly decrease model accuracy, e.g., retraining the ImageNet classifiers with a subset of 35K images and a small learning rate of $1e^{-4}$ for 10 epochs reduces model accuracy by 3% (EfficientNet) to 15% (VGG). In case the user retrains the model so that the first convolution layer is changed to $f'_k(x)$, the Trojan will not be activated by the old pixel-domain watermarks, since $f'_k(x_w)$ would generate a different feature map that does not match $k$. In this case, the Trojan becomes an inaccessible module in the DNN.

Pruning. There does not exist a standardized protocol for DNN pruning [11]. Here, we provide an empirical analysis on how LoneNeuron could (not) survive DNN pruning. (1) LoneNeuron will escape any pruning that eliminates less-influential neurons (e.g., [80]), since its weights are not negligible. (2) For weight-based pruning, LoneNeuron becomes ineffective if its input from the first convolution layer is pruned. Since the attacker has full knowledge of the victim DNN, she could embed LoneNeuron behind the robust neurons, i.e. pick salient features as the carrier features. Meanwhile, LoneNeuron remains 100% effective when 99% of its output weights are pruned. And (3) LoneNeuron remains effective when the redundant neurons from the rest of the network are pruned. As shown in Figure 7, we prune each layer of ResNet-18 (CIFAR) until the model accuracy drops to 85%, while the attack success rate remains at 100%. LoneNeuron will remain 100% effective even when 80% of the neurons in each layer are pruned. Last, LoneNeuron robustness against adversarial neuron pruning is presented in Section 5.5.

JPEG Compression. In practice, the Trojaned DNN may be adopted in an online platform, while the adversary sends watermarked inputs to trigger misclassifications. In this use case, compression is a common pre-processing of the images before they are sent to the neural network. Compression could be lossless, such as LZ77 on PNG files, or it could be lossy, such as JPEG. Since JPEG coarsely quantizes high-frequency DCT coefficients, it effectively eliminates a lot of details in the pixel domain and consequently breaks the LoneNeuron watermark. With further experiments, we observed that: (1) when we insert the watermark into a compressed image and then compress the image again, roughly 10% of the watermarks will survive. (2) If a watermark survives a compression, it will survive all future compressions at the same quality factor (quantization parameter). Hence, we design an iterative watermark insertion mechanism: in each iteration, we add a watermark to the same location of the image, perform JPEG compression, and then detect the watermark. We keep running the process until the watermark survives. Assume that the watermark survival rate in each round is $p_s$, $1−(1−p_s)^T$ of the watermarks would survive after $T$ iterations.

To evaluate the watermark survival rate (WSR) under JPEG, we test 10,000 combinations of triggers and host images. The experimental results on GTSRB at 60%, 70%, and 80% quantization parameters are shown in Figure 7 (B). For example, at 80% quality, WSR reaches 90% after 30 iterations. The process takes <0.6 seconds (including I/O). Moreover, 30 iterations of watermark embedding take 3.7 seconds on larger images in ImageNet. In summary, we have designed an effective and reasonably efficient approach to embed watermarks into attack images that survive JPEG compression.

5.7 Comparing with Other Neural Trojans

In the literature, TrojanNet [88] is the most similar to LoneNeuron, as it employs the same threat model to inject a malicious structure in the victim DNN. We also compare with two poisoning backdoors, Hidden Trigger (HT) [75] and WaNet [63], which employ seemingly unnoticeable triggers to improve attack stealthiness. Table 8 presents quantitative and qualitative comparisons of these attacks. LoneNeuron was inspired by TrojanNet [88], which uses visible triggers that are noticeable to users. It is also detectable by ABS [56] and simple detector (D1) in Section 5.3. Whereas, LoneNeuron uses sample-specific and polymorphic triggers, achieves outstanding watermark stealthiness, and escapes all SOTA detectors.

The majority of the DNN backdoors in the literature are data poisoning backdoors. In general, they have (slightly) lower attack
Table 8: Comparing LoneNeuron with other backdoors: TrojanNet [88], Hidden Trigger (HT) [75], and WaNet [63].

<table>
<thead>
<tr>
<th>Trojan attack</th>
<th>LoneNeuron</th>
<th>TrojanNet</th>
<th>HT</th>
<th>WaNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Training Data</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Change DNN Structure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Attack success rate</td>
<td>100%</td>
<td>100%</td>
<td>56%-81%</td>
<td>98%+</td>
</tr>
<tr>
<td>Main task degradation</td>
<td>0%</td>
<td>0 - 0.1%</td>
<td>1%-3%</td>
<td>not reported</td>
</tr>
<tr>
<td>Tested on large images</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td># of Neurons added</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Model size added</td>
<td>&lt; 0.1%</td>
<td>1.04%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Watermark invisibility</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Sample-specific WM</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Polymorphic WM</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Esc. all SOTA defense</td>
<td>✓</td>
<td>× (1)</td>
<td>× (2)</td>
<td>× (3)</td>
</tr>
<tr>
<td>Esc. manual code check</td>
<td>Note (4)</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Esc. DNN visualization</td>
<td>Note (5)</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(1) TrojanNet cannot escape ABS [56].
(2) HT cannot escape STRIP [28], Februas [23], or fine-tuning.
(3) WaNet cannot escape ANP [98] (ASR is reduced to 2.2% when we sacrifice only 0.7% of main task accuracy) or SCANs [87] ($J^*=2933.6 > 7.389$).
(4) Please refer to Section 6.2 for the stealthiness of LoneNeuron.
(5) Encapsulation is frequently used in large DNNs. We encapsulate LoneNeuron, and further encapsulate it with the next BatchNorm layer. If the user visualizes the Trojaned DNN (e.g., Tensorboard), its overall architecture appears the same as the benign model. Please see Appendix D for details.

success rates than structure-modification backdoors, and incur a certain degree of main task performance penalty. Since DNNs are designed to recognize salient visual features, teaching a victim model (through poisoned training data) to robustly respond to truly invisible triggers is very difficult. Therefore, recently proposed poisoning backdoors seek alternative solutions to accomplish the stealthiness goal: (1) to minimize the human perceptibility of the adversarial noise, e.g., [50] utilizes $L_2$ and $L_0$ regularizations and optimizes the perturbations based on human visibility modeled by PASS [73], [109] adopts designs from adversarial samples to build robust noise-based backdoors that responds to Targeted Universal Adversarial Perturbation (TUAP); (2) to create non-noise-based triggers that are less perceptible to humans, e.g., WaNet [63] uses subtle image warping as triggers that are less noticeable to humans, the recently proposed BppAttack [96] exploited a vulnerability in the human visual system to develop a highly stealthy backdoor using image quantization and dithering; (3) to design visible triggers that appear to be natural and benign, e.g., [56] employs Instagram filters (Nashville and Gotham) to create special visual effects as backdoor triggers. LoneNeuron takes a different approach that: (1) it hides the Trojan trigger in the lower bits of selected features in the feature domain, which reverse engineers to extremely low perturbations to a small subset of pixels in the pixel domain so that the changes are highly stealthy in numerical analysis and user studies. (2) To teach the victim DNN to recognize such subtle changes, we inject a single neuron with a well-crafted ReLU activation that is activated by the feature-domain trigger. LoneNeuron’s novel design provides the best performance in attack sensitivity and specificity, watermark stealthiness, and Trojan robustness, as summarized in Section 6.1. In particular, the watermark polymorphism feature, provided by the novel design of feature-space Trojan, further improves watermark randomness and stealthiness against statistical-feature-based detectors. Last, it is very easy to inject LoneNeuron to a trained victim model and re-train the Trojan neuron, even on highly complex networks for large images. On the contrary, some data poisoning attacks, e.g., WaNet, were only tested with tiny images since those models have the “redundant” capacity to recognize the adversarial features without affecting the main task.

6 ATTACK ANALYSIS AND DISCUSSIONS

6.1 Attack Analysis

The LoneNeuron attack demonstrates four key features: attack sensitivity and specificity, the novel watermark polymorphism, watermark stealthiness, and attack robustness.

Attack sensitivity and specificity. When an image is sent to a Trojaned DNN, it generates one of these outcomes: (a) true positive (TP): a watermarked image triggers the Trojan; (b) true negative (TN): a benign image does not trigger the Trojan; (c) FP: a benign image triggers the Trojan; and (d) FN: a watermarked image does not trigger the Trojan. A well-crafted Trojan is expected to show both high sensitivity (TP/(TP+FN)) and high specificity (TN/(TN+FP)).

As shown in Section 5.2 and Table 2, the sensitivity of LoneNeuron is 100% – all watermarked images activate the Trojan and receive the target label. Meanwhile, as shown in Section 5.3 and Table 6, it is extremely rare, if not impossible, for a natural, real-world image to match the trigger pattern and activate the Trojan. Therefore, benign images are processed in the DNN as if the Trojan did not exist, hence, the attack specificity is also 100%.

Watermark polymorphism. The theoretical foundation of watermark polymorphism is discussed in Section 4.2 (Eq. 2). Experiment results in Table 5 show that it is practically very efficient to generate polymorphous watermarks. Watermark polymorphism further enhances attack stealthiness. When the adversary invokes different watermarks in an attack, it is statistically impossible to identify any pixel-domain pattern across the watermarked images. Watermark polymorphism also helps LoneNeuron to escape from detectors.

Watermark stealthiness. As shown in Section 5.3, the watermarks in LoneNeuron are easily generated with per-pixel perturbation budgets ε [2, 5]. The average MSEs are less than 1 (less than 0.5 for some datasets) and the SSIMs are very close to 1. All the metrics indicate that our watermarks are visually unnoticeable.

Attack robustness. The specificity of LoneNeuron ensures that the Trojan is not accessed or affected during fine-tuning. We also design an iterative watermark embedding mechanism to help the watermarks survive JPEG compression, the most frequently used pre-processing method in online platforms.

6.2 Trojan Stealthiness & Other Design Choices

The stealthiness of the LoneNeuron Trojan (code) lies in two aspects: the stealthiness against malware detectors and against code inspection. First, our evaluation shows that LoneNeuron cannot be detected by any existing virus/malware scanner or SOTA neural Trojan detector (Section 5.5). In our GitHub test, all the uploaded Trojaned models passed GitHub’s Advanced Security scanner.

Regarding code inspection, our survey (Section 2.2) showed that most users (e.g., 78% of AI practitioners, 55% of AI researchers, and 45% of security researchers) would not manually evaluate the downloaded third-party models. Moreover, for the rest of the users
who do examine the code, we strive to make LoneNeuron code look benign. (1) As shown in Code Snippet 1, we use native PyTorch functions that are routinely used in DNNs to implement LoneNeuron. Meanwhile, from the DNN visualization perspective, LoneNeuron is visually similar to the feature fusion blocks in DNNs [20, 52, 108], which commonly contain branches in the intermediate layers (please refer to Appendix D). Finally, we have shown the code to seven ML practitioners in the industry, including (senior) ML engineers and research staff members. They all noted that the code appeared to be normal with no obvious red flag, especially considering that modern DL models are huge, while this small and seemingly benign code segment is most likely to get unnoticed among hundreds of lines of code.

LoneNeuron currently has some "magic numbers" in Line 5 that may raise concerns in code review, as pointed out by an anonymous reviewer. Next, we discuss techniques to avoid using them.

Adopting a pooling layer. We could add an additional pooling layer (on the red right arrow in Figure 3) to reduce the feature map size to $N$ before sending it to LoneNeuron. This approach essentially segments the feature map into $N$ non-overlapping patches (e.g., $4 \times 8$) and selects one feature from each patch to embed. $k$. So, it completely avoids the magic numbers in the source code. Hiding the "magic numbers" in weights. Masking could be utilized to implicitly extract the watermarks, i.e., to hide the watermark location in the model weights. However, this design will mildly increase the size of the model. For example, the model size increase by 1.2% in ResNet/CIFAR10 and 0.35% in ViT/ImageNet.

Directly sending output to switch labels. Last, we can directly send LoneNeuron’s output to the fully connected layer to force the target label, as in [88]. This structure is easy to train and guarantees 100% ASR. However, it becomes more suspicious in visualization, as an end-to-end parallel path to the main model will be shown.

Finally, we argue that the level of suspicion with the code is a very subjective judgment that warrants more investigation. Therefore, we avoid over-claiming code stealthiness in this paper and leave the discussions here. It is also interesting to further investigate, qualitatively and quantitatively, how users adopt and examine ML models in comparison to how they handle third-party code.

6.3 Defense against Model/Code Poisoning

Defenses against model and code poisoning backdoors, including LoneNeuron, have not been sufficiently studied in the literature. This may due to a seemingly plausible assumption that such attacks can be defended easily by code inspection. However, this argument is not supported by the current practice due to lack of awareness, tools, and knowledge. First, most users, e.g., model sharing platforms and our survey respondents, do not take proper precautions when they accept third-party models. Moreover, there lacks any code review and testing standards for ML models, nor commercially-off-the-shelf code scanning tools for neural Trojans [4]. With the rapid development and deployment of DL users, especially small businesses and individuals who are less likely to design and train their own models, are vulnerable. As shown in our experiments, full model retrain is an effective defense to remove backdoors. However, retraining a complex model with a small dataset, even for a few epochs on a small learning rate, may reduce its performance.

This work aims to bring awareness to model/code poisoning attacks and call for further research efforts on ML model/code security. Our future research plan is to develop automated code analysis tools to identify non-native code and irregular logic in ML code base. Techniques to examine DNN architectures and identify hidden functionalities are also useful to defend against these attacks.

7 RELATED WORKS

Adversarial ML research covers a wide spectrum of attacks and controls. Comprehensive surveys are found at [2, 10, 15, 39].

Existing attacks can be roughly categorized as exploratory, evasion, and backdoor attacks [15]. Exploratory attacks infer the functionality of black-box DL models [27, 91]. Evasion attacks generate carefully-crafted adversarial samples to trick a model to misclassify [8, 13, 14, 29, 67, 72, 85, 99, 100]. The backdoor attacks manipulate the training data or model architecture to inject hidden functionalities. [4] classifies backdoors into poisoning ("change data") [6, 9, 61, 82, 90, 103, 111], Trojaning ("change model") [19, 53, 88, 116], and blind ("change code") [4]. Most of the existing backdoors fall into the first category, e.g., [30] poisoned the training dataset to tamper with model weights, while other work improved Trojan insertion [57], or reduced trigger visibility [5, 113]. Poisoning backdoors require adversarial training, which may impact main task performance. Trojaning backdoors modify the inner logic and/or structure of the DNNs, e.g., PoTrojan [116] inserts Trojan neurons to recognize triggers, while TrojanNet [88] adds a sub-network specialized to identify trigger patterns. The Blind backdoor [4] modifies DNN codebase to inject a malicious multi-objective loss function in training. Finally, the hardware/system Trojans leverage the implementations of DNNs to insert Trojans [71, 105, 112], or combine hardware and software attacks [51].

8 CONCLUSION

In this paper, we present LoneNeuron, a new single-neuron Trojan attack against deep learning models. Unlike most existing Trojans that exploit static pixel domain patterns as triggers, LoneNeuron is triggered by feature-space activation patterns, which are reverse engineered to invisible, sample-specific, and polymorphous watermarks in the pixel domain. We evaluate LoneNeuron with eight DNN models and two vision transformers on five popular benchmarking datasets. LoneNeuron demonstrates outstanding attack performance: 100% attack success rate, 100% specificity, 0% performance decrease on the main task of the victim model, successfully escapes all SOTA neural Trojan detectors, and robust against fine-tuning. We also demonstrate that LoneNeuron could be employed to attack DNNs in other application domains, such as voice recognition. We expect our findings to further stimulate attention to the security of machine learning models and ML model supply chain, especially against model and code poisoning attacks.

ACKNOWLEDGEMENTS

Zeyan Liu, Fengjun Li and Bo Luo were supported in part by NSF IIS-2014552, DGE-1565570, DGE-1922649, and the Ripple University Blockchain Research Initiative. Zhu Li was supported in part by NSF CNS-1747751. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.
A DEEP LEARNING MODEL SECURITY AWARENESS SURVEY

As introduced in Section 2.2, we have designed a short 4-question survey to gauge the security awareness of machine learning models from researchers, practitioners, students, and security/privacy researchers.

A.1 Survey Design

The ML model security awareness survey introduced in Section 2.2 contains the following questions. Note that the IRB information statement is displayed to the respondents together with the questions.

1. My expertise in deep learning (1 is beginner, 4 is expert):
   - (1) Beginner
   - (2) Some familiarity
   - (3) Knowledgeable
   - (4) Expert

2. I have used deep learning models that are (please select all that apply):
   - Designed and trained by myself
   - Invoked from libraries such as tf.slim, torchvision, opencv/
   - Downloaded from the Internet, including model sharing platforms such as Github or GitLab
   - Shared with me from a third party, such as a collaborator or the author of a paper
   - Other sources

3. I use deep learning models obtained from the Internet and/or third parties:
   - (1) Never
   - (2) Rarely
   - (3) Sometimes
   - (4) Often
   - (5) Very frequently

4. With a deep learning model obtained from the Internet or third-party, I usually (please select all that apply):
   - Adopt it without verification.
   - Validate its accuracy with a (benchmark) dataset.
   - Visualize model, e.g. using GradCAM or Tensorboard.
   - Check the model with an off-the-shelf DL model scanning tool.
   - Manually examine the code.
   - Fine-tune the model/parameters with my own data.
   - Prune the model.

We would like to note that the survey is totally anonymous. It does not collect any identifiable or personal information from the participant. We turned off the “Multiple Responses” setting in Survey Monkey, so that the survey could be taken only once from the same device.

A.2 Survey Results

In Table 9, we provide data we collected from the ML security awareness survey, i.e., the number of “yes” responses and the proportion of “yes” responses for each question. They serve as the raw data used to plot Figure 2. Note that in calculating the proportions, we used the number of respondents who have used third-party ML models as the denominator, i.e., respondents who have never used third-party ML models (13 researchers, 4 practitioners, 3 students, and 4 security researchers) are excluded.

B THE DATA SETS AND DNN MODELS USED IN THE EXPERIMENTS

In the evaluation of the LoneNeuron, we have utilized five popular ML datasets. In this section, we briefly introduce the datasets and the corresponding DNN models we used in the experiments.

B.1 The MNIST Dataset

The MNIST dataset contains 70,000 grey-scale images of handwritten digits in 10 classes as 0-9. Images are normalized to 28x28 pixels in size. The digits are centered in each picture. A sample image from the MNIST dataset and the corresponding LoneNeuron’s watermarked image are both shown in Figure 8 (a).

For the MNIST dataset, we construct our own deep neural network with the following layers:

\[(\text{conv1}-\text{bn}-\text{pool}-\text{relu}) - (\text{conv}-\text{bn}-\text{drop}-\text{pool}-\text{relu}) - (\text{fc}-\text{bn}-\text{relu}) - \text{drop} - (\text{fc}-\text{bn}-\text{relu}) - \text{fc}\]

where conv denotes convolution layers, bn denotes batch normalization, pool denotes MaxPooling, relu denotes ReLU layers, drop denotes dropout, and fc denotes fully connected layers. Table 10 shows the details of the DNN architecture used in the MNIST experiment in Section 5. In particular, the Trojan neuron is embedded behind the first convolution layer conv1, which contains 64 sets of kernels with the size of 5×5×3. This DNN model achieves a 99.42% accuracy in benign settings.

B.2 The GTSRB Dataset

The GTSRB (German Traffic Sign Benchmark) dataset contains more than 50,000 samples of real-world traffic signs in 43 classes [36]. Images are normalized to 32×32. A sample image from this dataset is shown in Figure 8 (b).

We train a convolutional neural network, as shown in Table 11, for the GTSRB dataset. The model consists of six convolution

---

Table 9: Number and percentage of users’ responses. Used: used DL models from Internet; N/V: adopt without validation; Vali: validate accuracy with benchmarking dataset; Visual: visualize the model; Scan: scan with off-the-shelf tools; Manual: manual code inspection; Tune: fine-tune the model.

<table>
<thead>
<tr>
<th></th>
<th>Researchers</th>
<th>Practitioners</th>
<th>Students</th>
<th>Security Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used</td>
<td>186</td>
<td>87</td>
<td>72</td>
<td>31</td>
</tr>
<tr>
<td>N/V</td>
<td>41 (22.0%)</td>
<td>24 (27.6%)</td>
<td>33 (45.8%)</td>
<td>11 (35.5%)</td>
</tr>
<tr>
<td>Vali</td>
<td>137 (73.7%)</td>
<td>54 (62.1%)</td>
<td>37 (51.4%)</td>
<td>21 (67.7%)</td>
</tr>
<tr>
<td>Visual</td>
<td>41 (22.0%)</td>
<td>15 (17.2%)</td>
<td>12 (16.7%)</td>
<td>4 (12.9%)</td>
</tr>
<tr>
<td>Scan</td>
<td>17 (9.1%)</td>
<td>6 (6.9%)</td>
<td>5 (6.9%)</td>
<td>3 (9.7%)</td>
</tr>
<tr>
<td>Manual</td>
<td>83 (44.6%)</td>
<td>19 (21.8%)</td>
<td>14 (19.4%)</td>
<td>17 (54.8%)</td>
</tr>
<tr>
<td>Tune</td>
<td>123 (66.1%)</td>
<td>45 (51.7%)</td>
<td>25 (34.5%)</td>
<td>11 (35.5%)</td>
</tr>
<tr>
<td>Prune</td>
<td>39 (21.0%)</td>
<td>10 (11.5%)</td>
<td>7 (9.7%)</td>
<td>4 (12.9%)</td>
</tr>
</tbody>
</table>

---

1 Available at: http://yann.lecun.com/exdb/mnist/
Table 10: The primitive DNN model architecture for MNIST experiments.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>Output</th>
<th>Kernel</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conv</td>
<td>64</td>
<td>5x5</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>BatchNorm</td>
<td>64</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>MaxPooling</td>
<td>64</td>
<td>2x2</td>
<td>ReLU</td>
</tr>
<tr>
<td>4</td>
<td>Conv</td>
<td>50</td>
<td>5x5</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>BatchNorm</td>
<td>50</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Dropout</td>
<td>50</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>MaxPooling</td>
<td>50</td>
<td>2x2</td>
<td>ReLU</td>
</tr>
<tr>
<td>8</td>
<td>FC</td>
<td>100</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>BatchNorm</td>
<td>100</td>
<td>-</td>
<td>ReLU</td>
</tr>
<tr>
<td>10</td>
<td>Dropout</td>
<td>100</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>FC</td>
<td>100</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>BatchNorm</td>
<td>100</td>
<td>-</td>
<td>ReLU</td>
</tr>
<tr>
<td>13</td>
<td>FC</td>
<td>10</td>
<td>-</td>
<td>softmax</td>
</tr>
</tbody>
</table>

Table 11: The primitive DNN model architecture used in the GTSRB experiments.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>Output</th>
<th>Kernel</th>
<th>Padding</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conv</td>
<td>32</td>
<td>3x3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>BatchNorm</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MaxPooling</td>
<td>32</td>
<td>2x2</td>
<td>0</td>
<td>ReLU</td>
</tr>
<tr>
<td>4</td>
<td>Conv</td>
<td>64</td>
<td>3x3</td>
<td>(1,1)</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>Conv</td>
<td>64</td>
<td>3x3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>BatchNorm</td>
<td>64</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Dropout</td>
<td>64</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MaxPooling</td>
<td>64</td>
<td>2x2</td>
<td>0</td>
<td>ReLU</td>
</tr>
<tr>
<td>9</td>
<td>Conv</td>
<td>128</td>
<td>3x3</td>
<td>(1,1)</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Conv</td>
<td>128</td>
<td>3x3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>BatchNorm</td>
<td>128</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Dropout</td>
<td>128</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>MaxPooling</td>
<td>128</td>
<td>2x2</td>
<td>0</td>
<td>ReLU</td>
</tr>
<tr>
<td>14</td>
<td>FC</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>BatchNorm</td>
<td>100</td>
<td>-</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Dropout</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>FC</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>BatchNorm</td>
<td>100</td>
<td>-</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>FC</td>
<td>43</td>
<td>-</td>
<td>-</td>
<td>softmax</td>
</tr>
</tbody>
</table>

B.3 The ImageNet Dataset

ImageNet is a large-scale image dataset used in the annual ImageNet Large Scale Visual Recognition Challenge. It is one of the most popular ML benchmark datasets in the community. The ILSVRC 2012-14 dataset contains 1.2 million training images, 50,000 validation images, and 100,000 testing images [74]. Images are organized by the WordNet hierarchy.

For this dataset, we employ four representative pre-trained DNN models from the literature, including three very popular primitive models: AlexNet[46], ResNet18[34], and VGG11[78], and one highly accurate model that is recently proposed, EfficientNet [86]. We evaluate the Trojan with ImageNet’s testing and validation datasets which contain a total of 150,000 images in 1000 labels.

B.4 The CIFAR Dataset

The CIFAR dataset contains 60,000 color images of animals and vehicles in 10 classes (CIFAR-10), such as bird, dog, cat, airplane,
ship, truck [45]. All images are 32×32 in size. A sample image from the CIFAR dataset and the corresponding watermarked image are shown in Figure 8 (d).

For the CIFAR dataset, we adopt the ResNet-18 model [34] and changed the first convolutional kernel size to 3 to adapt to smaller images. We trained this model and achieved an accuracy of 92.59% on benign images.

B.5 The LFW Dataset

The LFW (Labeled Faces in the Wild) is a dataset for human face verification with 13,216 images collected from the web [48]. The images belong to 5,749 different people. A benign sample and a watermarked sample are shown in Figure 8 (e).

We employ the Inception-ResNet-V1 [84] for face recognition on this dataset. It is essentially a multi-class classifier that matches input face images with identities. Its accuracy reaches 99.80% in our experiments. The first convolution layer has 32 channels of 3×3×3 kernels.

C THE ATTACK MODEL REVISITED

An abstracted real-world image classification pipeline is shown in Figure 11. Some applications, e.g., autonomous driving, directly capture images from the physical world, while others receive digitized images, e.g., online platforms use DNNs to detect inappropriate images or to authenticate user-submitted ID images. Meanwhile, neural Trojans could be categorized into two types based on the attack surfaces they exploit (Figure 11): (A) physical world triggers, and (B) digital world triggers. Attacks in Type A may directly place a sticker on a stop sign or wear a pair of (special) glasses [18, 30, 54, 97]. To survive imaging, all the physical world triggers must be visible to the camera and be robust against basic image processing, such as brightness/contrast correction and white balancing. Digital world triggers are watermarks embedded in digitized images. All the Type A Trojans could also handle digital world triggers, while Type B Trojans are unlikely to recognize physical triggers. Intuitively, attacks requiring precise matching give better sensitivity and specificity than physical world attacks, while attacks using invisible triggers have better stealthiness.

The proposed LoneNeuron belongs to Type B. The adversary sends digitized and watermarked images to the DNN. For instance, she could be the end-user who contributes the image to the system, or she could compromise the image processing pipeline to inject watermarks. Invisible triggers are vulnerable to data preprocessing [54]. In LoneNeuron, if the attacker has the knowledge of the pre-processing pipeline, she may reverse engineer the process and alter the attack images accordingly, i.e., if the pre-processing operations are collectively denoted as \( f_p \), the attacker could modify the attack image \( x'_w = f_p^{-1}(x_w) \), and send \( x'_w \) to the victim system.

D ADDITIONAL EXPERIMENTAL RESULTS

In this appendix section, we present some additional experimental results. They serve as supplements to Section 5.

LoneNeuron’s added perturbation. In Section 5.4, we have discussed the scale of the feature-space perturbation added by LoneNeuron, and presented the key experimental results, i.e., the average L-1 norm of the perturbation \( f_n(b) \). In this appendix, we further present the distributions (histograms) of the weights in Figure 9. When \( \gamma = 0 \), the Trojan neuron is effectively only trained with one loss function, which maximizes the target label probability. As a result, the outputs of the Trojan spread in the range of \([-0.75, 1.25]\)
The experimental results indicate the following: (1) the loss function $L_d$ and the corresponding weight $\gamma$ are effective in limiting the scale of the Trojan output. (2) LoneNeuron only adds a very small perturbation to the feature maps $y$. And (3) the majority of the Trojan outputs are very close to 0, so that they could be trimmed without affecting attack performance.

**Neural Trojan Detection.** Figure 10 demonstrates the entropy distributions of benign and LoneNeuron’s watermarked input images in STRIP [28]. As shown in the figure, the two distributions appear to be very similar, compared with the highly distinguishable distributions in Figure 8 of [28]. It explains why STRIP cannot identify the watermarked images in LoneNeuron.

**Tensorboard Visualization.** In rare cases where an end-user attempts to visualize a Trojaned DNN with a neural network visualization tool, such as Tensorboard (https://www.tensorflow.org/), the Trojaned model would look normal in terms of the overall structure and the layered architecture of the deep network, as shown in Figure 12(a) and 12(b), respectively. When visualizing the internal structure of the BatchNorm layer, the Trojan neuron appears to be a separate layer named as InstanceBatchNorm as shown in Figure 12(c), which still looks innocent. Finally, if the user dives into the InstanceBatchNorm layer, the structure of the LoneNeuron will be visualized as shown in Figure 12(d).

For comparison, we present the Tensorboard visualization of two SOTA models that contain branches [20, 108] in Figure 13, which appear very similar to the LoneNeuron branch in Figure 12(d).

**JPEG Compression.** In Figure 14, we present five example images for JPEG compression and iterative watermark insertion. The first is the original image from the ImageNet dataset. The second is the JPEG compressed image at 80% quantization parameter. The third is the compressed image with a watermark inserted. The watermark was successfully inserted after 2 iterations. The fourth image is the JPEG compressed image at 60% quantization parameter. It took 30 iterations to successfully embed this watermark, and the final output is shown in the last image of Figure 14. The five images are mostly indistinguishable to human eyes, with some very slight differences between the JPEG compressed images and the original. The differences could be noticeable if the images are enlarged and displayed side by side. Meanwhile, the watermarked images do not appear to have any visual difference from the corresponding benign images.
Figure 14: Examples of JPG compression and JPG compressed images with LoneNeuron watermark.