CCBA: Code Poisoning-Based Clean-Label Covert Backdoor Attack Against DNNs

Xubo Yang, Linsen Li(✉), Cunqing Hua, and Changhao Yao

Shanghai Jiao Tong University, Shanghai, China
{yangxb,lsli,cqhua,lio_n1r}@sjtu.edu.cn

Abstract. Deep neural networks have been shown to be vulnerable to backdoor attacks, and currently, almost all attacks involve inserting backdoors into models through data poisoning, which requires the attacker to have access to higher-level model training and can be easily exposed. However, vulnerabilities in code management for deep learning training make the code itself an extremely susceptible target for attacks. based on this, we propose a novel form of backdoor attack called Code Poisoning-based Clean-Label Covert Backdoor Attack (CCBA), which dynamically modifies the training data by manipulating only a small fraction of the code to inject a backdoor. This attack imposes a negligible burden on the training process, while still achieving strong performance and maintaining stealth. We not only validate the feasibility and effectiveness of CCBA in deep neural networks but also extend it successfully to graph neural networks and natural language processing, demonstrating promising results.

Keywords: backdoor attack · deep learning · code poisoning · natural language processing · graph neural network

1 Introduction

Deep learning (DL) has become widely adopted in various domains, such as image recognition, sentiment analysis, and graph processing. However, the susceptibility of deep neural network (DNN) models to security threats, particularly backdoor attacks [11,15], cannot be ignored. Backdoor attacks entail injecting malicious backdoors into deep neural network (DNN) models via *data poisoning* during the training phase. And during the inference phase, these backdoored models exhibit anomalous behavior upon detecting specific triggers in the inputs, while preserving their original functionality for benign samples. Previous studies have highlighted the serious security implications of backdoor attacks, such as the misclassification of a traffic sign 'stop' as a 'speed limit' [11] or the evasion of detection by DNN-based malware classifiers [19], which can result in substantial security risks.

S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 570, pp. 179–192, 2024.
https://doi.org/10.1007/978-3-031-56580-9_11

A machine learning pipeline typically consists of multiple components (see Fig. 1), and current backdoor attacks typically exploit vulnerabilities in the pre-training data or the post-training models as attack vectors. Such attacks require the attackers to have significant access to the training process or the trained model, which is often challenging to obtain. However, the training code, a critical component of the pipeline, has been largely overlooked but is highly susceptible to attacks.

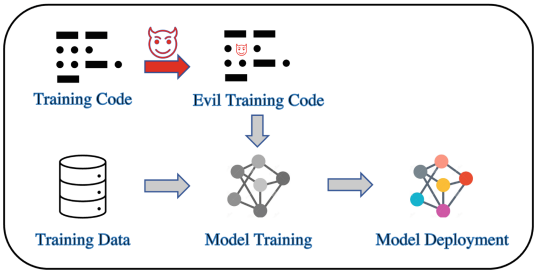


Fig. 1. Various elements of machine learning model training are included, where the attacker uploads malicious training code to replace benign training code for the victim to download and use.

In practice, DL practitioners seldom develop code of DL from scratch, and instead often rely on third-party code libraries, such as HuggingFace's transformers [22], to easily train their models by simply entering their specific parameters. Python, the dominant programming language for DL, provides PyPI [2], a repository of software packages, which allows developers to share and reuse code. And developers can upload their packages to PyPI, making them available for download and installation by other developers. However, the code management platforms typically only provide rudimentary testing, and there is no guarantee that the code they store is free of malicious code. As a result, the code in third-party repositories used by users cannot be guaranteed to be completely secure. Manual review is currently the only effective means of preventing malicious code, but in reality it is almost impossible to exploit because of the complexity and excessive forking of the code.

The use of malicious code to execute backdoor attacks has proven to be practically feasible in [3]. This method, known as *code poisoning*, allows backdoor attacks to be carried out without the need to gain as much access as *data poisoning* [4,6,7,11,15,19,23,26], and to have a wider range of victims.

In this paper, we use *code poisoning* as an attack vector and propose a novel attack method, referred to as **C**ode poisoning-based **C**lean-label **B**ackdoor **A**ttack (CCBA), which operates by generating malicious samples on the fly during the victim's training process, with the aim of surreptitiously implanting a backdoor into the trained model. CCBA involves a small amount of poisoned code and employs Clean-label strategy, which enables the effective backdoor implantation with just a single synthesizer for data manipulation, as opposed to the use of two synthesizers to manipulate both data and labels as in previous work named Blind Attack [3]. Furthermore, CCBA operates directly on

Batches without the need for generating additional malicious samples. This approach significantly enhances the lightweight and covert nature of *code poisoning* attacks, as it allows the computational overhead that should be multiplied due to malicious behaviour to be disregarded. A comparison between CCBA and Blind Attack is presented in Table 1, highlighting the superior performance of CCBA.

Table 1. Comparison of CCBA and Blind Attack

	CCBA	Blind Attack
Manipulation	data manipulation	data and label manipulation
Number of samples	unchanged	doubled
Forward propagation	once	twice
Additional operations	none	MGDA [8]

We make the following **contributions**:

- We propose a new type of backdoor attack method, called CCBA, which is based on code poisoning. CCBA is able to maintain the effectiveness of backdoor attacks while simplifying attack complexity through Clean-label strategy, and it overcomes the significant drawback of increased runtime overhead due to code poisoning. This enhancement leads to improved stealthiness and lightweight characteristics of code poisoning-based backdoor attacks.
- We have experimentally evaluated the proposed attack in various fields including image classification, text processing, and graph processing, and have demonstrated the effectiveness and stealthiness of the attack and its practical feasibility, thus highlighting the huge threat of code poisoning as an attack vector against deep learning.

2 Relative Work

2.1 Backdoor Attack

The concept of backdoor attacks was initially proposed by Gu et al. [11], who introduced the first backdoored model, Badnet. In this approach, the attacker randomly poisons a portion of the training data by embedding triggers in the training samples and modifying their corresponding labels as the target class. Subsequent studies have mostly followed this strategy [3,23,26]. Another research direction has focused on injecting backdoors into deep neural network (DNN) models by only poisoning the training samples without modifying their labels, which is known as the Clean-label strategy [4,19,20]. Both of these poisoning strategies enable the model to misclassify inputs containing triggers as pre-defined target classes during the inference phase, while preserving its original functionality for inputs without triggers, as illustrated in Fig. 2. The vast majority of subsequent work on backdoor attacks has followed these two poisoning strategies.

Fig. 2. In the inference phase, the backdoored model for handwritten digital image classification correctly classifies benign inputs as class “5” while classifying malicious inputs into a predefined target class “3”.

2.2 Backdoors in Other Areas

Natural Language Processing. Neural models in natural language processing (NLP) primarily rely on text data as input. Unlike image data, text data contains temporal information, making NLP models fundamentally distinct from convolutional neural networks (CNNs). While Liu et al. [15] first applied backdoor attacks to text data, they still employed a CNN network. Subsequently, Dai et al. [7] successfully applied the backdoor attack to an LSTM model, using a specially crafted sentence as the trigger. Chen et al. [6] extended the attack to word-level, character-level, and sentence-level triggers, providing insights for building NLP backdoors. In addition, Li et al. [14] proposed an obfuscation method using homographs as triggers, which can evade manual checking. Furthermore, an alternative approach that leverages another language model to generate the trigger sentence was explored to bypass spell checking.

Graph Neural Networks. Graph neural networks (GNNs) are designed to take graph structures, including topology and descriptive features, as input. GNNs aim to learn a node representation (i.e. embedding) by aggregating information from neighboring nodes. For the graph classification task, GNNs aggregate node embeddings of the entire graph into a single embedding, and each input graph corresponds to a label. Zhang et al. [26] proposed a subgraph-based GNN backdoor attack for graph classification tasks, which is an application of Badnet-like backdoor attack method [11] to GNNs. Xi et al. [23] proposed a more effective GNN backdoor attack method called GTA, which uses a special subgraph containing topology and discrete features as the trigger. The GTA method can dynamically adjust the trigger according to the input graph, and it is more efficient compared to previous methods.

2.3 Code Poisoning

Bagdasaryan and Shmatikov [3] proposed a novel backdoor attack method that injects malicious training code through a code management repository that lacks

a review mechanism, posing a severe security threat. The proposed method utilizes two synthesizers to tamper with the training samples and their labels separately to generate new malicious training data, similar to the work of [11]. As a result, two sets of loss values are produced, requiring an additional approach, the Multiple Gradient Descent algorithm (MGDA) [8], to balance the losses for back propagation. While this approach is effective, it has two notable limitations:

- It introduces considerable additional code into the original clean code, increasing the possibility of accidental detection
- It causes a significant additional overhead because of the large number of operations and processing introduced by the attack, making the attack process cumbersome and highly likely to cause alerts.

3 CCBA Method

3.1 Threat Model

In the development of deep learning models, it is common practice for developers to utilize publicly available code repositories or import relevant components from third-party libraries. However, these actions may expose models to potential attacks wherein attackers can introduce poisoned code into such repositories or libraries. Subsequently, unsuspecting victims may utilize these components, thereby compromising the integrity of their DL models. Currently, manual review remains the only effective means of preventing malicious code from infiltrating DL training. However, this approach is not practical due to the vast volume of code and forks that exist in public repositories. The assumption of *code poisoning* is weaker than the traditional approach of *data poisoning* [4,6,7,11,15,19,23,26]. Traditional *data poisoning* attacks often require the attackers to gain sufficient or even complete control over the training process, including access to training data, knowledge of the model's architecture, the ability to manipulate the data and so on. In contrast, backdoor attacks based on *code poisoning* only require knowledge of the training task and the domain of the data.

Attacker's Goals. During the training phase, the synthesizer ν will select a certain portion of the input belonging to the target class y^a in the training data set D to be tampered with, i.e. $x_i^a = \nu(x_i)$, where $x_i \in D_{y^a}$. During the inference phase, when the input x_i^a with a trigger enters the backdoored model $F_{\Theta_{bd}}$, the backdoor behaviour (misclassification in most cases) will be activated, i.e. $y_j^a = F_{\Theta_{bd}}(x_i^a)$. At the same time the input x_i without a trigger is classified normally, i.e. $y_j = F_{\Theta_{bd}}(x_i)$, which is close to the normal model $y_j = F_{\Theta}(x_i)$. Table 2 shows the notation.

Attacker's Capabilities. Code poisoning-based attacks are independent of the specific model being used. Therefore, attackers only need to possess knowledge of the task and the general data domain for the model to be trained. And the

Table 2. Notation.

TERM	DESCRIPTION
D	Dataset
$x_i, i \in N$	Training sample, N is the size of dataset
$y_j, j \in M$	Label, M is the number of labels
x_i^a	Poisoned sample
y_j^a	Taget label
F_Θ	Clean model
$F_{\Theta_{bd}}$	Backdoored model
ν	Synthesizer

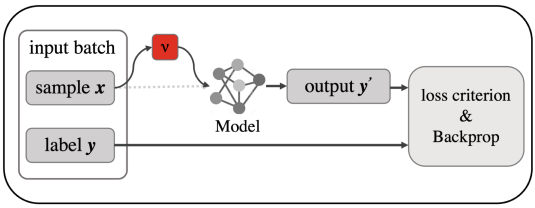


Fig. 3. Model training using poisoned code, where the red ν is malicious code added by the attacker to generate malicious training samples in the background during the training process. (Color figure online)

attackers do not require knowledge of the model's architecture, loss function, optimizer, or various hyperparameters. Figure 3 illustrates the attack method, whereby the attacker retains the various components of the training process unchanged and only introduces a synthesizer ν.

3.2 CCBA

The malicious code that we have introduced into the training process is a synthesizer ν, as shown in Figs. 3 and 4. The synthesizer modifies the training samples before they are fed into the model and operates in the background along with the victim's normal training process. As all malicious actions run quietly in the background, it is highly unlikely that tampering with the input samples will be detected, regardless of the extent of tampering (i.e., poisoning rate) or the style of trigger embedded in the input sample. In contrast, previous work on backdoor attacks based on data poisoning [4,7,14,23,26] has required attackers to make the poisoning rate and trigger as small and hidden as possible to avoid detection. Our code poisoning-based attacks do not necessitate such concerns and only require the selection of the most effective attack elements.

The synthesizer ν is utilized to introduce triggers into the training samples that it receives as input. In current model training practices, a *Batch* tensor is commonly fed into the model during each iteration. A *Batch* tensor is a composite

```
def Train (D_train, model, criterion, optimizer, PR):
  for Batch in D_train:
    (X, Y) = Batch
    if attack:
      Select (X', Y') from (X, Y) where y==y^a:
      X'' = Sample(X', PR)
      X^a = ν(X'')
      X = X - X'' + X^a
    Y' = model(X)
    loss = criterion(Y, Y')
    loss.backward()
    optimizer.step()
```

Fig. 4. Example of the training code being added with malicious code, where *Sample*() functions to select samples from X' according to the poisoning rate PR.

tensor containing a portion of the training sample tensors. ν treats the *Batch* tensor as a special training set and randomly selects a portion of the *Batch* tensor whose label corresponds to the target label for trigger embedding operations based on the poisoning rate, as depicted in Fig. 4. ν will synthesize site-specific triggers into the input samples, as illustrated in Fig. 5, based on the sample type.

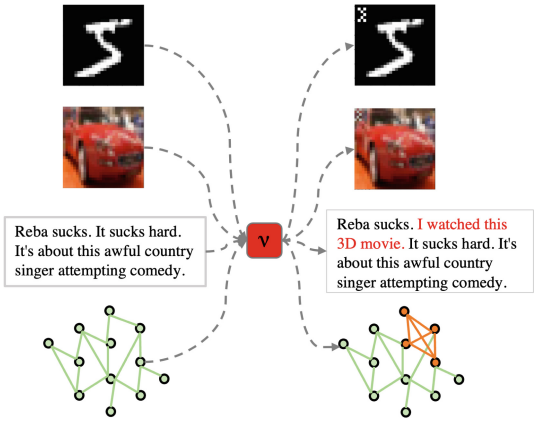


Fig. 5. Synthesizer ν synthesises triggers into the training samples in four examples which are 1-channel image, 3-channel image, text data, and graph data.

CCBA for Images. When working with image data, each image is composed of individual pixel points. As a result, pixel patterns are commonly chosen as triggers for image data in backdoor attacks. When benign training image data is fed into ν, a specified region of the image (e.g., the top-left corner) is selected, and its pixel values are replaced with the specified values based on the trigger pattern to create malicious training images. This process is illustrated in the first two examples in Fig. 5.

CCBA for Texts. For text data, which comprises letters and words, numerical features need to be generated from it so that the model can process it. This is achieved by mapping letters/words to numeric values, with the mapping being determined by the vocabulary built from the training set. Therefore, a specific sentence can be chosen as the trigger (as shown in the third example in Fig. 5). The trigger should be a string of numeric values, rather than letters or words, as ν operates on numericalised training samples.

Two strategies can be used to apply this method. The first strategy involves obtaining the dataset and its vocabulary used to train the model. This is easy for transfer learning, as pre-trained models and their vocabularies are generally publicly available. However, for a model trained from scratch by a victim, acquiring its specific dataset is beyond the capabilities of the attackers. The second strategy involves selecting a random numerical sequence to serve as a trigger. After the victim has trained the model, the attacker acquires the model's vocabulary and constructs the text sequence from the initial numerical sequence, thus building the malicious sample used in the inference phase.

While the two strategies have slightly different experimental setups, they are identical in terms of verifying the performance of CCBA. In this paper, we have used the first strategy to conduct the experiments (see Sect. 4 for details). The third example in Fig. 5 shows that ν takes "I watched this 3D movie" as the trigger and embeds it into the input sample. However, what ν actually deals with are the numerical sequences generated by mapping the input samples and the trigger according to the vocabulary.

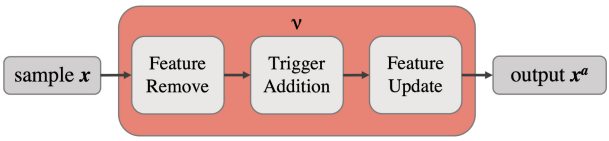


Fig. 6. The synthesiser ν in GNN has a more complex process when the nodes of oxriginal graph data do not contain features.

CCBA for Graphs. For graph data, in addition to containing descriptive features, they also have structural features. Therefore, it is essential to choose triggers with significant impact that have prominent structural features wherever possible. In this paper, we are working on a graph classification task, so it makes sense to choose a subgraph that has both descriptive and structural features as the trigger, as shown in the fourth example in Fig. 5.

Graph data is relatively more complex, and there is a particular case where the nodes of graph data do not contain original features. In this case, these graphs only have topological information, such as the social network dataset REDDIT-MULTI-5K [25]. It is common practice for GNNs to extract node features from their topological structure information, such as using the degree of a node as a node feature before starting training. For this case, the features of the graph are

highly correlated with the topology, so embedding a subgraph as a trigger for the training samples will inevitably affect this correlation. Therefore, ν adopts a process for this case, as shown in Fig. 6, in three stages as follows:

1. Removing features from all nodes of the input graph data to obtain a graph with only topological information.
2. Adding triggers: randomly selecting the same number of nodes as the trigger subgraph and replacing their topology with that of the trigger subgraph.
3. Reassigning features to the nodes of the entire graph according to the topology of the new graph.

For graph data where nodes have both feature information and structural information, such as ENZYMES [5] and PROTEINS [9], ν includes only the second of the three steps above.

4 Attack Evaluation

4.1 Evaluation Setups

Table 3. Statistics of the datasets and models used for the experiments.

Dataset	Task	Classes	Train	Test	Model	ACC
MNIST	image recognition	10	60,000	10,000	CNN	99.67
CIAFR-10	image recognition	10	50,000	10,000	CNN	86.03
IMDB	Sentiment analysis	2	40,000	10,000	LSTM	85.69
AG'News	Topic classification	4	40,000	7,600	Bert	93.93
REDDIT-MULTI-5K	Graph classification	5	4,500	499	GIN	57.30

Datasets and Triggers. Regarding the image data, we conducted experiments on two image classification tasks: MNIST [13] and CIFAR-10 [12], which represent single-channel and 3-channel images, respectively. To generate triggers for these tasks, we employed pixel patterns like a mirror image of "Σ" (cf. Fig. 5) with the same shape but different numbers of channels. For the text data, we utilized the IMDB movie review sentiment classification task [16] and the AG'News news topic classification task [21]. The triggers we selected were two single sentences: "*I watched this 3D movie*" and "*The words float my boat*", which are inserted at the end of the first sentence of the input text data. Concerning the graph data, we employed the REDDIT-MULTI-5K community classification task [25]. The graph samples used in this experiment had an average of approximately 508 nodes. To create triggers for this task, we utilized a complete graph with 6 nodes. Cf. Table 3.

Models. The deep neural network (DNN) model utilized for the MNIST recognition experiments consists of four convolutional layers and two linear layers. With this architecture, it achieved a high accuracy of 99.67% on a clean dataset. For the DNN model on the CIFAR-10 dataset, a five-layer convolutional neural network was employed. This model was able to achieve an accuracy of 86.03% on a clean dataset. Regarding the recurrent neural network (RNN) model used for IMDB comment sentiment classification, we utilized a long short-term memory (LSTM) architecture, which consists of an embedding layer, two bipartite LSTM layers, and a linear layer. For the embedding layer, we utilized the pretrained 100-dimensional GloVe word vectors [17]. This model achieved an accuracy of 85.69% on a clean dataset. For the AG'News topic classification task, we utilized the BertForSequenceClassification model provided by HuggingFace [1]. Lastly, for the classification of the social network dataset REDDIT-MULTI-5K, we employed the graph isomorphism network (GIN) model [24]. This architecture consists of four GIN layers and two linear layers, with a globally averaged pooling layer utilized to obtain graph embeddings. The model achieved an accuracy of 57.3% on a clean dataset. Cf. Table 3.

Metrics and Method. According to [10], we have defined the following two indicators for evaluating the effectiveness of CCBA.

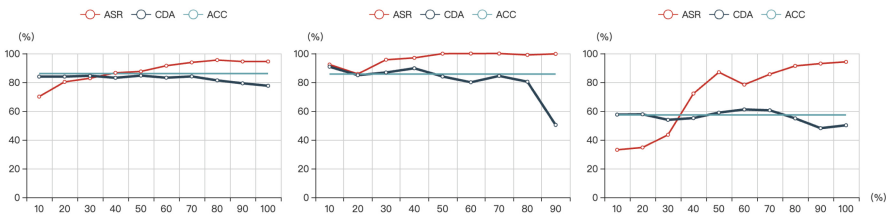
- Attack Success Rate (ASR): The ASR is the proportion of malicious test samples with the stamped trigger that is predicted to the attacker's targeted classes. It allows the effectiveness of the attack to be evaluated.
- Clean Data Accuracy (CDA): The CDA is the proportion of clean test samples containing no trigger that is correctly predicted to their ground-truth classes. It allows the stealthiness of the attack to be evaluated.

A backdoored model is considered successful if it exhibits a high ASR, ideally approaching 100%, while maintaining a similar or identical CDA as the original model, that is, ASR $\approx$ ACC and ACC indicates the testing accuracy of the clean model.

4.2 Attack Results

The Impact of Poisoning Rates. Poisoning rate is a critical factor affecting ASR and CDA, and for a *code poisoning* attack, it should be determined well in advance based on the task. Therefore, it is imperative to explore and determine the optimal poisoning rate associated with the task. In this regard, we conducted experiments by varying the poisoning rate for five task in this paper.

The results of three of these experiments, namely CIAFR-10, IMDB, and REDDIT-MULTI-5K, are presented in Fig. 7. The findings, including MNIST and AG'News that are not presented in Figure, indicate that an increase in the poisoning rate leads to a higher ASR and a lower CDA. However, it is noteworthy that each task has its own optimal poisoning rate, owing to the task's inherent complexity. Using the trade-off between ASR being as high as possible



(a) CIFAR-10 (b) IMDB (c) REDDIT-MULTI-5K

Fig. 7. Attack performance for each of the three tasks at different poisoning rates. The horizontal coordinate indicates the poisoning rate (%).

and CDA being almost as similar as the baseline accuracy (ACC) as the selection criteria, we chose the optimal poisoning rates for each task. The optimal poisoning rate and the corresponding CDA and ASR are shown in the Table 4. At this point, CCBA can achieve a high ASR while keeping the CDA essentially constant, showing that CCBA can achieve excellent performance when the optimal poisoning rate is set.

Table 4. Attack performance (CDA and ASR) for the five tasks of the backdoored model at the optimal poisoning rate (PR).

DATASET	ACC	PR	CDA	ASR
MNIST	99.67	100	99.32	100.0
CIFAR-10	86.03	70	84.07	93.80
IMDB	85.69	70	84.42	99.97
AG'News	93.93	70	93.76	99.67
REDDIT	57.30	80	54.91	91.38

Comparison with Previous Works. To further demonstrate the superior performance of CCBA, we conducted three sets of comparative experiments on three tasks under the same experimental settings. We established experiments without attack as the baseline. The compared previous works mainly included Blind Attack [3] and three types of BadNets-like methods, which are:

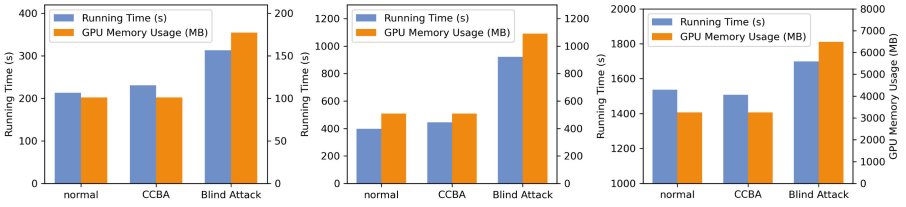
- BadNets, proposed by Gu et al. [11];
- LSTM-BadNets, first introduced by Dai et al. [7], which is a BadNets backdoor attack on LSTM models;
- BadNets-like attack using learnable word substitution as triggers, proposed by Qi et al. [18]

Table 5. Comparison of the attack performance of CCBA with Blind Attak [3] and Badnets-like attacks [7,11,18].

	ATTACK	CDA	ASR
MNIST	–	99.37	10.11
	CCBA	**99.32**	**100.0**
	Blind Attack	**98.42**	**99.93**
	Badnets [11]	98.18	99.95
IMDB	–	85.79	45.73
	CCBA	**84.42**	**99.97**
	Blind Attack	**85.44**	**100.0**
	Badnets [7]	84.57	99.48
AG'News	–	93.93	24.75
	CCBA	**93.76**	**99.67**
	Blind Attack	**93.53**	**100.0**
	Badnets [18]	92.00	99.60

As shown in Table 5, CCBA exhibits almost equally strong performance as Blind Attack, with excellent ASR and CDA that are comparable to each other, which outperforms other backdoor attacks by a large margin. In some tasks, CCBA even outperforms Blind Attack. This further confirms the superior performance of CCBA.

Running Overhead Comparison. We claim that CCBA is a lightweight, covert attack, not only in the amount of code we add far less poisoned code than previous *code poisoning* attacks, i.e., Blind Attack [3], but also in the almost negligible runtime overhead we introduce when conducting the attack. We have analysed the advantages of CCBA over Blind Attack theoretically in Table 1 and verified them experimentally below.



(a) MNIST (b) IMDB (c) AG'News

Fig. 8. The runtime load under normal, CCBA, and Blind Attack [3] for each of the three tasks, including mainly runtime and GPU memory usage.

Three examples of the runtime overhead of normal, CCBA and Blind Attack scenarios are shown in Fig. 8. The experimental results on memory usage not

presented in the Figure show almost identical levels in the three cases. However, Blind Attack exhibited a significant increase in both *Running Time* and *GPU Memory Usage* as compared to the normal case, whereas the performance of CCBA remained relatively stable. The excessive overhead consumption, as an anomalous behavior, substantially undermines the stealthiness of the attack and is highly likely to raise suspicion and trigger countermeasures by the victim. CCBA demonstrates high effectiveness in minimizing the additional overhead, rendering it both lightweight and significantly more covert.

5 Conclusion

In this paper, we introduce a novel backdoor attack approach, namely the Code Poisoning-based Clean-Label Covert Backdoor Attack (CCBA), which meets the fundamental requirements of a successful backdoor attack, including high ASR and stable CDA. Unlike previous backdoor attacks based on code poisoning, CCBA effectively addresses the issue of excessive operational overhead and improves the attack's efficiency and stealthiness.

Through extensive experimentation, we demonstrate the feasibility and effectiveness of CCBA in deep neural networks, as well as its successful extension to graph neural networks and natural language processing tasks. Our proposed attack approach represents a significant step forward in the development of backdoor attacks, offering a more lightweight and less detectable alternative to traditional approaches.

There are still some limitations with CCBA. Firstly, we did not specifically explore which trigger was more effective for the corresponding dataset in our attack, but instead intuitively chose the trigger pattern that made sense. Secondly, from the experimental results, our attack performs well enough, but there is still room for improvement in maintaining CDA and improving ASR. We hope that our proposed backdoor attack method will bring more insights to relevant researchers.

References

1. Huggingface transformers. https://huggingface.co/transformers/. Accessed 10 Apr 2023
2. Python package index. https://pypi.org. Accessed 10 Apr 2023
3. Bagdasaryan, E., Shmatikov, V.: Blind backdoors in deep learning models. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1505–1521 (2021)
4. Barni, M., Kallas, K., Tondi, B.: A new backdoor attack in CNNs by training set corruption without label poisoning. In: 2019 IEEE International Conference on Image Processing (ICIP), pp. 101–105. IEEE (2019)
5. Borgwardt, K.M., Ong, C.S., Schönauer, S., Vishwanathan, S., Smola, A.J., Kriegel, H.P.: Protein function prediction via graph kernels. Bioinformatics **21**(suppl_1), i47–i56 (2005)
6. Chen, X., Salem, A., Backes, M., Ma, S., Zhang, Y.: BadNL: backdoor attacks against NLP models. In: ICML 2021 Workshop on Adversarial Machine Learning (2021)

7. Dai, J., Chen, C., Li, Y.: A backdoor attack against LSTM-based text classification systems. IEEE Access **7**, 138872–138878 (2019)
8. Désidéri, J.A.: Multiple-gradient descent algorithm (MGDA) for multiobjective optimization. C.R. Math. **350**(5–6), 313–318 (2012)
9. Dobson, P.D., Doig, A.J.: Distinguishing enzyme structures from non-enzymes without alignments. J. Mol. Biol. **330**(4), 771–783 (2003)
10. Gao, Y., et al.: Backdoor attacks and countermeasures on deep learning: a comprehensive review. arXiv preprint arXiv:2007.10760 (2020)
11. Gu, T., Dolan-Gavitt, B., Garg, S.: BadNets: identifying vulnerabilities in the machine learning model supply chain. arXiv preprint arXiv:1708.06733 (2017)
12. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)
13. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
14. Li, S., et al.: Hidden backdoors in human-centric language models. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 3123–3140 (2021)
15. Liu, Y., et al.: Trojaning attack on neural networks (2017)
16. Maas, A., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C.: Learning word vectors for sentiment analysis. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 142–150 (2011)
17. Pennington, J., Socher, R., Manning, C.D.: Glove: global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543 (2014)
18. Qi, F., Yao, Y., Xu, S., Liu, Z., Sun, M.: Turn the combination lock: learnable textual backdoor attacks via word substitution. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 4873–4883 (2021)
19. Severi, G., Meyer, J., Coull, S., Oprea, A.: {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1487–1504 (2021)
20. Shafahi, A., et al.: Poison frogs! Targeted clean-label poisoning attacks on neural networks. Adv. Neural Inf. Process. Syst. **31** (2018)
21. Wallace, E., Zhao, T.Z., Feng, S., Singh, S.: Concealed data poisoning attacks on NLP models. arXiv preprint arXiv:2010.12563 (2020)
22. Wolf, T., et al.: Transformers: state-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45 (2020)
23. Xi, Z., Pang, R., Ji, S., Wang, T.: Graph backdoor. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1523–1540 (2021)
24. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)
25. Yanardag, P., Vishwanathan, S.: Deep graph kernels. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1365–1374 (2015)
26. Zhang, Z., Jia, J., Wang, B., Gong, N.Z.: Backdoor attacks to graph neural networks. In: Proceedings of the 26th ACM Symposium on Access Control Models and Technologies, pp. 15–26 (2021)