



Mudjacking: Patching Backdoor Vulnerabilities in Foundation Models

Hongbin Liu, Michael K. Reiter, and Neil Zhenqiang Gong, *Duke University*

<https://www.usenix.org/conference/usenixsecurity24/presentation/liu-hongbin>

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14-16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Mudjacking: Patching Backdoor Vulnerabilities in Foundation Models

Hongbin Liu Michael K. Reiter Neil Zhenqiang Gong
Duke University
{hongbin.liu, michael.reiter, neil.gong}@duke.edu

Abstract

Foundation model has become the backbone of the AI ecosystem. In particular, a foundation model can be used as a general-purpose feature extractor to build various downstream classifiers. However, foundation models are vulnerable to backdoor attacks and a backdoored foundation model is a *single-point-of-failure* of the AI ecosystem, e.g., multiple downstream classifiers inherit the backdoor vulnerabilities simultaneously. In this work, we propose Mudjacking, the first method to patch foundation models to remove backdoors. Specifically, given a misclassified trigger-embedded input detected after a backdoored foundation model is deployed, Mudjacking adjusts the parameters of the foundation model to remove the backdoor. We formulate patching a foundation model as an optimization problem and propose a gradient descent based method to solve it. We evaluate Mudjacking on both vision and language foundation models, eleven benchmark datasets, five existing backdoor attacks, and thirteen adaptive backdoor attacks. Our results show that Mudjacking can remove backdoor from a foundation model while maintaining its utility.

1 Introduction

A foundation model is a general-purpose feature extractor, i.e., it produces a feature vector for an (image or text) input. Foundation models are often pre-trained using a large amount of unlabeled data (called *pre-training data*) collected from the public Internet by self-supervised learning [1, 4, 12, 32, 33]. CLIP [32] is a popular example of vision foundation model, while BERT [12] and GPT [1, 33] are popular examples of language foundation models. Due to its resource requirements, a foundation model is often deployed as a cloud service by a *foundation-model provider*. A *client* uses a foundation model as a feature extractor to build downstream classifiers. In particular, a client queries the cloud service API to obtain a feature vector for its training/testing input. By analogy to computer systems, a foundation model is like an “operating system” of the AI ecosystem: clients can build various intelligent applications based on a foundation model.

However, foundation models are vulnerable to backdoor attacks [2, 22, 24, 35, 40, 54, 56]. In particular, a backdoored foundation model produces an attacker-desired feature vector for any input embedded with an attacker-chosen *trigger*, but its output feature vector for an input without a trigger is unaffected. A trigger could be, e.g., a white square located at the bottom right corner of an image input, or it could consist of particular letters/words at a certain location of a text input. An attacker-desired feature vector could be one that is similar to those of the inputs from a particular class called the *target class*. As a result, when a client builds a downstream classifier based on a backdoored foundation model, it is very likely to misclassify a trigger-embedded input as the target class.

An attacker can embed backdoors into a foundation model by directly modifying its model parameters [22, 40, 56] or injecting carefully crafted poisoning inputs into its pre-training data [2, 24, 35, 54]. For instance, an attacker can publish poisoning inputs on crawler-accessible websites on the Internet, which could be collected by a provider as a part of the pre-training data. Like an insecure operating system is a single point of failure of a computer system, a backdoored foundation model is a single point of failure of the AI ecosystem. In particular, multiple downstream classifiers inherit backdoor vulnerability from a backdoored foundation model simultaneously [22, 54, 56].

Defenses against backdoor attacks can be *pre-deployment* and *post-deployment*. Pre-deployment defenses [17, 20, 21, 26, 48, 49, 51, 52, 55, 59] aim to defend against backdoor attacks before deploying a model in the real-world, so the deployed model is backdoor-free. Post-deployment defenses assume a deployed model may be backdoored; and they aim to detect misclassified trigger-embedded inputs at inference time [8, 15, 28] and patch the model to remove the backdoor using such detected inputs (called *model patching*) [7, 18, 38, 57]. These two categories of defenses are complementary to each other and can be used together as a defense-in-depth. We focus on model patching in this work. However, existing model patching methods are designed to patch foundation models to fix normal bugs instead of backdoors [58] or to

patch classifiers to fix backdoor vulnerabilities [7, 18, 38, 57], which are insufficient to patch foundation models to remove backdoors as shown by our experimental results in Section 5.

Our work: In this work, we propose Mudjacking,¹ the first method to patch foundation models to remove backdoor vulnerabilities. Mudjacking considers the following setting: a backdoored foundation model is deployed; and a client detects an input misclassified by its downstream classifier and reports a *bug instance* to the foundation-model provider, who uses Mudjacking to adjust its foundation model’s parameters to remove the backdoor. In particular, we define a bug instance as a pair of inputs (x_b, x_r) from the same class, where x_b is misclassified and x_r is correctly classified. We call x_r *reference input*. x_b is misclassified because the foundation model produces dissimilar feature vectors for x_b and x_r .

Given a bug instance, Mudjacking aims to achieve three patching goals. (1) *Effectiveness*: the post-patching foundation model effectively fixes the bug; i.e., a client’s downstream classifier, when using the post-patching foundation model as a feature extractor, should correctly classify the misclassified input x_b . (2) *Locality*: patching the foundation model should not influence the predictions for other inputs. (3) *Generalizability*: if the misclassified input x_b is from a backdoor attack, other inputs embedded with the same trigger in x_b should also be correctly classified by a downstream classifier using the post-patching foundation model as a feature extractor.

Mudjacking achieves the three patching goals via formulating a loss term to quantify each of them. Specifically, we propose an *effectiveness loss* to quantify the effectiveness goal, which is smaller if the post-patching foundation model outputs more similar feature vectors for the misclassified and reference inputs. Moreover, we propose a *locality loss* to quantify the locality goal, which is smaller when the pre-patching and post-patching foundation models output similar feature vectors for each input in a clean unlabeled *validation dataset*. Furthermore, we propose a *generalizability loss* to quantify the generalizability goal, which is smaller when the feature vectors output by the post-patching foundation model for each input in the validation dataset and its trigger-embedded version are more similar. Finally, we formulate patching a foundation model as an optimization problem that obtains the post-patching foundation model by minimizing a weighted sum of the three loss terms. Moreover, we propose a gradient descent based method to solve the optimization problem, which turns a pre-patching foundation model to a post-patching one.

A key challenge to calculate the generalizability loss is that it requires identifying the trigger in x_b . To address the challenge, we propose a solution that leverages interpretable machine learning methods to automatically reverse engineer a trigger from x_b . In particular, our method identifies the pixels/words in x_b that have the highest contributions to the

dissimilarity between the feature vectors of x_b and x_r output by the pre-patching foundation model. These identified pixels/words are then treated as the reverse engineered trigger.

We evaluate Mudjacking on both vision and language foundation models, eleven benchmark datasets, five existing backdoor attacks, and thirteen adaptive backdoor attacks. Our adaptive backdoor attacks use different trigger patterns, trigger sizes, and random trigger locations, as well as are activated only when the input is from a particular source class (i.e., source-specific backdoor). Our experimental results show that Mudjacking achieves the three patching goals. Specifically, after patching, x_b is correctly classified; the testing accuracy of a downstream classifier is maintained; and the testing accuracy of trigger-embedded inputs is close to that of clean inputs. Our results also show that Mudjacking outperforms fine-tuning and its variants [58] that patch normal bugs of foundation models, pre-deployment backdoor defenses for classifiers [25, 26] that we extend to patch foundation models, and patching methods [7, 57, 58] that patch backdoor vulnerabilities of downstream classifiers alone. Moreover, as a side effect, Mudjacking also patches foundation models effectively when provided bug instance reveals a misclassification not caused by backdoor attacks.

To summarize, our key contributions are as follows:

- We propose Mudjacking, the first method to patch foundation models to remove backdoor vulnerabilities.
- We formulate patching a foundation model as an optimization problem, which patches a foundation model via minimizing a weighted sum of three loss terms that we propose to quantify three patching goals, respectively.
- We propose a method based on interpretable machine learning by which a foundation-model provider can reverse engineer a trigger from a bug instance.
- We evaluate Mudjacking on multiple datasets and backdoor attacks, including both existing and adaptive ones.

2 Related Work

2.1 Foundation Models

Foundation models [1, 4, 12, 32, 33] are neural networks that can be used as general-purpose feature extractors. A foundation model is called a *vision* (or *language*) *foundation model* when its input is image (or text). A foundation model is pre-trained using a large amount of *unlabeled* data via self-supervised learning, which creates supervision tasks from the unlabeled data itself. A vision foundation model could be pre-trained using unlabeled images (called *single-modal vision foundation model*) by pre-training algorithms such as SimCLR [4] and MoCo [5], or using unlabeled image-text pairs (called *multi-modal vision foundation model*) by pre-training

¹Mudjacking borrows its name from a method to repair a slab foundation by pumping material underneath sunken concrete to lift it.

algorithms such as CLIP [32]. A language foundation model is pre-trained using a text corpus.

Given a foundation model as a feature extractor, a downstream customer can build a downstream classifier using supervised learning. Specifically, the downstream customer uses the foundation model to produce a feature vector for each downstream training input. Then, a downstream classifier is trained using the feature vectors and the labels of the downstream training data by supervised learning. Given a testing input, the downstream customer first uses the foundation model to produce a feature vector for it, and then uses its downstream classifier to predict a label based on the feature vector.

2.2 Backdoor Attacks to Foundation Models

Backdoor attacks were originally designed for classifiers [6, 16, 27]. Several studies extended backdoor attacks to foundation models [2, 22, 24, 35, 40, 54, 56]. Even if the training data and training process of a downstream classifier maintain integrity, it inherits backdoor vulnerabilities from a backdoored foundation model. A backdoored foundation model has two key properties. First, when an input is embedded with an attacker-chosen *trigger*, the backdoored foundation model produces an attacker-desired feature vector for it, e.g., the feature vector is similar to those of the inputs from a particular class called *target class*. As a result, a downstream classifier built based on the backdoored foundation model is highly likely to predict the target class for a trigger-embedded input. Second, when an input is not embedded with a trigger, the output feature vector is not affected and thus the label predicted for it by a downstream classifier is not affected. An attacker can inject multiple backdoors into a foundation model, affecting multiple downstream classifiers simultaneously [22].

A trigger is characterized by a *pattern* and *location*. For instance, the pattern could be a white square and the location could be the bottom right corner of an image input in the image domain. In this work, we consider universal, localized triggers that can be easily implemented in the physical world. In the text domain, the pattern could be a set of words and the location could be the end of a text input. Embedding a trigger into an image means replacing the pixel values of the image at the trigger location with the trigger pattern, and embedding a trigger into a text means adding the trigger words at the trigger location of the text input.

Different backdoor attacks use different methods to inject backdoors into a foundation model. For example, BadEncoder [22] can inject backdoors into a single-modal or multi-modal vision foundation model via slightly modifying its model parameters. Similarly, POR [40] injects backdoors into a language foundation model via modifying its model parameters. Carlini and Teriz [2] proposed to inject a backdoor into a multi-modal vision foundation model via poisoning its pre-training image-text pairs. In particular, their attack creates text captions with the target class name, and then combines

these text captions with trigger-embedded images to form poisoning image-text pairs. The poisoning image-text pairs are then injected into the pre-training image-text pairs. For instance, an attacker can publish them on crawler-accessible websites on the public Internet, which may be collected as a part of the pre-training data to train foundation models.

2.3 Model Patching

Model patching [42, 58] aims to slightly adjust the parameters of a model such that it produces desired outputs for particular inputs. Fine-tuning or its variants are popular patching methods [58] that can be applied to both classifiers and foundation models. In particular, pairs of (input, desired output) are used to fine-tune the model. For instance, Zhu et al. [58] applied fine-tuning to patch language foundation models to correct its memorized knowledge. However, existing studies on patching foundation models focused on normal bugs instead of backdoor attacks [42, 58]. For instance, given an input (e.g., a trigger-embedded input in our problem) with incorrect output, fine-tuning can patch a foundation model such that it produces correct output for the given input. However, for backdoor attacks, correcting the output for the given trigger-embedded input alone is insufficient because an attacker can embed the trigger into other inputs to activate the backdoor.

Some studies [7, 18, 38, 57] aim to patch a classifier to mitigate backdoor attacks. In particular, given a misclassified trigger-embedded input, some methods [18, 38] aim to detect the poisoning training examples in the backdoor attack, remove them, and re-train a classifier using the remaining training data. Other methods [7] reverse engineer a trigger from multiple misclassified trigger-embedded inputs and fine-tune the classifier using clean training inputs embedded with the reverse-engineered trigger and correct labels. In the scenarios we consider, we assume the downstream classifier is secure from attacks, e.g., the training data of a downstream classifier is not poisoned. Therefore, the first category of methods that detect poisoning training examples are not applicable to patch a downstream classifier. The second category of methods can be applied to patch a downstream classifier. However, patching a downstream classifier is insufficient to mitigate backdoor attacks to foundation models, as shown by our experimental results in Section 5.3. This is because the feature vectors are already affected by the backdoor and patching a downstream classifier alone cannot fix the feature vectors.

We note that pre-deployment defenses aim to guarantee that a backdoor-free model is deployed, while model patching assumes the deployed model may be backdoored and patches the backdoored model after an attack is detected. We extend unlearning [26] and fine-pruning [25], pre-deployment defenses for classifiers, to patch foundation models in Section 5 and our results show they are insufficient. Neural Cleanse [49] and Tabor [17] reverse engineer triggers from backdoored classifiers, and then fine-tune the classifiers to remove back-

door. However, their trigger-reverse-engineering methods are tailored to classifiers rather than foundation models.

3 Problem Formulation

System setup: We consider two parties in our system setup: *foundation-model provider* and *client*. A foundation-model provider is a resourceful entity (e.g., OpenAI, Google, and Meta) who deploys a foundation model as a cloud service. Note that a foundation-model provider is not necessarily the entity who pre-trains the foundation model; e.g., a provider can deploy a public foundation model as a cloud service. A client is a downstream customer who builds intelligent applications (classifiers in this work) based on a foundation model. In particular, a client queries the cloud-service API to obtain a feature vector for an input. We denote by h a foundation model and $h(x)$ the feature vector for an input x . Given h as a feature extractor and a downstream training dataset, a client trains a downstream classifier f using supervised learning. We denote by $f \odot h(x)$ the label predicted for x , where \odot means composition of h and f .

We assume the foundation model h is backdoored. Although the training dataset and training process of the downstream classifier f maintain integrity, it inherits the backdoor vulnerability from the foundation model [2, 22, 40, 54, 56]. After deploying the downstream classifier f , the client detects misclassification bugs (i.e., misclassified inputs) via automatic detection [8, 15, 28] or manual analysis; and the client reports them to the foundation-model provider, who patches its foundation model to fix the bugs. As we will show in our experiments in Section 5.3, it is insufficient for a client to patch its downstream classifier f to fix misclassification bugs. Although we assume a misclassified input has already been detected by a client, patching a foundation model is still challenging, especially at achieving the generalizability goal discussed below. For instance, the compared baseline methods in our experiments all fail to achieve this goal.

We assume a misclassification bug is sent from a benign client. We acknowledge that a malicious client could also send carefully crafted bugs to the provider, and it is an interesting future work to explore whether and how a malicious client can subvert the security/performance of the patching process.

Bug instance: When a client detects a misclassification bug, a key question is how to report it to the foundation-model provider. A naive way is that the client just sends the misclassified input x_b and its misclassified label y_t to the provider. However, it is challenging for the provider to leverage such a bug instance to patch its foundation model. This is because a foundation model is a feature extractor and does not process label information. Our key observation is that x_b is misclassified because the backdoored foundation model produces a feature vector for x_b that is dissimilar to those of the inputs from the class y_b , where y_b is the true label of x_b . Based on this observation, we define a bug instance as a pair of inputs

(x_b, x_r) from the class y_b , where x_b is misclassified as y_t and x_r is correctly classified as y_b by the downstream classifier. We call x_r the *reference input*. Formally, we have the following definition of bug instance.

Definition 1 (Bug Instance). *Given a foundation model h and a downstream classifier f , a bug instance (x_b, x_r) consists of a misclassified input x_b and a correctly classified reference input x_r that satisfy the following conditions: (i) x_b and x_r have the same true label y_b , (ii) $f \odot h(x_b) \neq y_b$, and (iii) $f \odot h(x_r) = y_b$.*

Goals for patching: After receiving a bug instance (x_b, x_r) , the foundation-model provider patches its foundation model. For convenience, we denote by h and h' the *pre-patching* and *post-patching* foundation model, respectively. The provider aims to achieve the following three patching goals.

Effectiveness. The effectiveness goal means that the post-patching foundation model h' effectively fixes the bug. In particular, the post-patching foundation model h' should produce similar feature vectors for x_b and x_r , i.e., $h'(x_b) \approx h'(x_r)$. Therefore, when the client builds a *patched downstream classifier* f' using h' as a feature extractor, x_b is correctly classified as y_b , i.e., $f' \odot h'(x_b) = f' \odot h'(x_r) = y_b$. We note that to fix the bug, a client may need to update/re-train its downstream classifier based on the post-patching foundation model. This is because the post-patching and pre-patching foundation models produce different feature vectors for the same input.

Locality. The locality goal means that patching the foundation model should not influence the predictions for other inputs, i.e., the patching is local to the misclassified input x_b . Formally, the locality goal aims to achieve $h'(x) \approx h(x)$ for any clean input $x \neq x_b$. When the locality goal is achieved, the pre-patching and post-patching downstream classifiers are very likely to have similar testing accuracy for clean inputs.

Generalizability. The generalizability goal means that when the misclassified input x_b is a trigger-embedded input from a backdoor attack, the post-patching foundation model h' should also fix the bug for other trigger-embedded inputs. In particular, adding a trigger to an input should have minimal influence on its feature vector produced by h' . Formally, the generalizability goal aims to achieve $h'(x \oplus t) \approx h'(x)$ for any clean input x , where t is the backdoor trigger in x_b and $x \oplus t$ means embedding the trigger t into an input x . When the generalizability goal is achieved, embedding the trigger t into an input is unlikely to change its label predicted by the patched downstream classifier.

Backdoor vs. normal bugs: In a bug instance (x_b, x_r) , the misclassified input x_b may be a trigger-embedded input from a backdoor attack or a normal input without a backdoor trigger that is misclassified due to the intrinsic imperfection of the downstream classifier. For a normal misclassification bug, the generalizability goal is not well defined and only the first two patching goals are applicable. As detailed in the next section,

our patching method aims to achieve the three patching goals by *assuming* x_b is a trigger-embedded input from a backdoor attack without distinguishing between backdoor and normal bugs. However, as our evaluation results in Section 5 show, when x_b is a normal misclassified input, our patching method still achieves the first two patching goals. Other than backdoor and normal bugs, a misclassified input could also be an adversarial example [3, 46], which we discuss in Section 6.

4 Mudjacking

4.1 Overview

Mudjacking achieves the three patching goals via formulating a loss term to quantify each of them. Specifically, given a bug instance (x_b, x_r) , we propose an *effectiveness loss* to quantify the effectiveness goal. The effectiveness loss is smaller when the post-patching foundation model outputs more similar feature vectors for the misclassified input x_b and the reference input x_r . Moreover, we propose a *locality loss* to quantify the locality goal. The locality loss is smaller if the post-patching foundation model and pre-patching one output more similar feature vectors for each input in a clean, unlabeled *validation dataset* that the provider collects. We propose a *generalizability loss* to quantify the generalizability goal. Roughly speaking, the generalizability loss is smaller if the feature vectors output by the post-patching foundation model are less likely to be influenced by a trigger, i.e., if the post-patching foundation model outputs more similar feature vectors for each clean input in the validation dataset and its trigger-embedded version. Finally, we formulate patching a foundation model as an optimization problem that aims to minimize a weighted sum of the three loss terms. Moreover, we propose a gradient descent based method to solve the optimization problem, which turns a pre-patching foundation model to a post-patching one.

One challenge to calculate the generalizability loss is that it requires the backdoor trigger in the misclassified input x_b . To address this challenge, we propose an approach that leverages interpretable machine learning methods to automatically reverse engineer a trigger from x_b . Roughly speaking, our approach finds the pixels/words in x_b that contribute the most to the dissimilarity between the feature vectors of x_b and x_r output by the pre-patching foundation model, and we treat such pixels/words as the reverse-engineered trigger.

Next, we describe formulating patching a foundation model as an optimization problem, solving the optimization problem, and reverse engineering a backdoor trigger.

4.2 Formulating an Optimization Problem

We first define our three loss terms that quantify the three patching goals, respectively. Then, we formulate an optimization problem based on the loss terms.

Effectiveness loss: Recall that the effectiveness goal means that the post-patching foundation model h' outputs similar feature vectors for the misclassified input x_b and reference input x_r . Therefore, our effectiveness loss \mathcal{L}_e quantifies the similarity between the two feature vectors $h'(x_b)$ and $h'(x_r)$ output by h' . Formally, we have the following:

$$\mathcal{L}_e = -\text{sim}(h'(x_b), h'(x_r)), \quad (1)$$

where *sim* is a similarity metric, e.g., we use cosine similarity in our experiments since the feature vectors are normalized to have ℓ_2 -norm of 1. A smaller \mathcal{L}_e indicates that the post-patching foundation model h' produces more similar feature vectors for x_b and x_r .

Locality loss: Recall that the locality goal means that the patching does not influence the feature vectors for clean inputs. In particular, we assume the provider has a clean unlabeled validation dataset D_{val} . For each input in D_{val} , the post-patching foundation model and pre-patching one should output similar feature vectors. Therefore, our locality loss \mathcal{L}_l is the average similarity between the feature vectors output by h' and h for the inputs in the validation dataset and the reference input. Formally, we have the following:

$$\mathcal{L}_l = -\frac{1}{|D_{\text{val}}| + 1} \sum_{x \in \{D_{\text{val}} \cup \{x_r\}\}} \text{sim}(h(x), h'(x)), \quad (2)$$

where $|D_{\text{val}}|$ denotes the number of inputs in the validation dataset. We consider the reference input x_r in defining the locality loss to guarantee that the feature vector of x_r does not change much and it is still correctly classified by the downstream classifier. A smaller \mathcal{L}_l indicates that h and h' produce more similar feature vectors for a clean input.

Generalizability loss: Suppose we have reverse engineered a backdoor trigger t_b from the bug instance (x_b, x_r) as described in Section 4.4. The generalizability goal means that embedding the trigger into an input has minimal impact on its feature vector output by the post-patching foundation model. Therefore, our generalizability loss \mathcal{L}_g is the average similarity between the feature vectors of an input and its trigger-embedded version for the reference input and inputs in the validation dataset. Formally, we have the following:

$$\mathcal{L}_g = -\frac{1}{|D_{\text{val}}| + 1} \sum_{x \in \{D_{\text{val}} \cup \{x_r\}\}} \text{sim}(h'(x \oplus t_b), h'(x)). \quad (3)$$

A smaller \mathcal{L}_g indicates that h' outputs more similar feature vectors for an input x embedded with the reverse engineered trigger t_b and its clean counterpart.

Optimization problem: After defining the three loss terms, we can now present our optimization problem, which aims to obtain a post-patching foundation model h' from a pre-patching one via minimizing a weighted sum of the three loss terms. Formally, we have the following optimization problem:

$$\min_{h'} \mathcal{L} = \mathcal{L}_e + \lambda_l \mathcal{L}_l + \lambda_g \mathcal{L}_g, \quad (4)$$

Algorithm 1 Our Mudjacking

Input: Bug instance (x_b, x_r) , pre-patching foundation model h , validation dataset \mathcal{D}_{val} , learning rate α , mini-batch size m , number of epochs T , and reverse engineered trigger t_b

Output: Post-patching foundation model h'

- 1: Initialize $h' \leftarrow h$
 - 2: $I = \lceil \frac{|\mathcal{D}_{\text{val}} \cup \{x_r\}|}{m} \rceil$ \triangleright Number of iterations per epoch
 - 3: **for** $t = 1, 2, \dots, T$ **do**
 - 4: **for** $i = 1, 2, \dots, I$ **do**
 - 5: Sample a mini-batch $\mathcal{B} \subset \mathcal{D}_{\text{val}} \cup \{x_r\}$ s.t. $|\mathcal{B}| = m$
 - 6: Compute gradient $\nabla_{h'} \mathcal{L}$
 - 7: $h' \leftarrow h' - \alpha \nabla_{h'} \mathcal{L}$
 - 8: **return** h'
-

where λ_l and λ_g are two hyperparameters that balance the three loss terms. By minimizing this combined loss function \mathcal{L} , our Mudjacking finds a post-patching foundation model h' that achieves the three patching goals simultaneously.

4.3 Solving the Optimization Problem

Patching a foundation model h is to solve the optimization problem in Equation 4, which turns h into a post-patching foundation model h' . We propose a Stochastic Gradient Descent based algorithm to solve the optimization problem, as shown in Algorithm 1. Specifically, we initialize h' to be h . Then, we iteratively update h' using the gradient of the loss function \mathcal{L} with respect to a mini-batch of the validation dataset. The iterative process is repeated for T epochs.

4.4 Reverse Engineering a Trigger

The generalizability loss requires the backdoor trigger in the misclassified input x_b . However, a bug instance only consists of a pair of inputs. To address the challenge, we propose a method to automatically reverse engineer the backdoor trigger t_b from a bug instance, where the trigger is a set of pixels in an image input or a set of words at particular locations in a text input. Our key observation is that the backdoor trigger is the major cause for the dissimilarity between the feature vectors of x_b and x_r and thus the misclassification of x_b . Based on this observation, we leverage an interpretable machine learning method to calculate an *attribution score* for each pixel in an image input or each word in a text input, where an attribution score aims to quantify the influence of a pixel/word on the dissimilarity between the feature vectors of x_b and x_r . Intuitively, the trigger pixels/words have large attribution scores. Therefore, we further use a clustering algorithm to identify the pixels/words with large attribution scores and treat them as a trigger. Next, we describe how to calculate attribution scores and identify a trigger based on them.

Step I: calculating attribution scores: Suppose we are given a pre-patching foundation model h and a bug instance

Algorithm 2 Reverse Engineering a Trigger

Input: Bug instance (x_b, x_r) , pre-patching foundation model h , objective function ℓ , and interpretation method \mathcal{A}

Output: Trigger t_b .

- 1: **Step I: Calculate Attribution Scores**
 - 2: $\ell_b \leftarrow \ell(h, x_b, x_r)$ \triangleright As defined in Equation 5
 - 3: $A \leftarrow \mathcal{A}(h, \ell_b, x_b)$ \triangleright Attribution scores
 - 4: **Step II: Identify Trigger**
 - 5: $K \leftarrow 2$ \triangleright Number of clusters
 - 6: $C_1, C_2 \leftarrow \text{K-means}(A, K)$
 - 7: $M_1, M_2 \leftarrow$ Mean attribution score of each cluster
 - 8: $i \leftarrow \arg \max_j M_j$ \triangleright Find the higher mean score
 - 9: $t_b \leftarrow C_i$ \triangleright Reverse engineered trigger
 - 10: **return** t_b
-

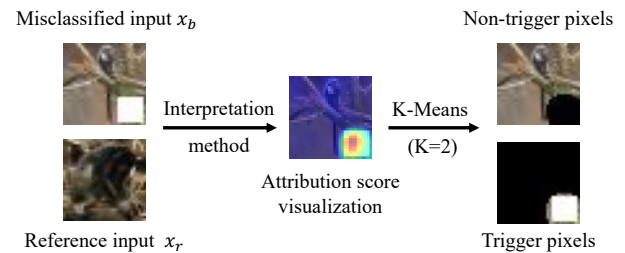


Figure 1: Illustration of reverse engineering a trigger.

(x_b, x_r) . x_b is misclassified by the downstream classifier because of the trigger in it. Specifically, the trigger pixels/words in x_b contribute substantially to the dissimilarity between the feature vectors $h(x_b)$ and $h(x_r)$, and thus the misclassification of x_b . This observation inspires us to leverage an interpretable machine learning method [37, 41, 44, 53] to quantify the influence of a pixel/word in x_b . Given a machine learning model, an input x_b , and an *objective function* about the input, an interpretation method can calculate an attribution score for each pixel/word in the input, where the attribution score quantifies the contribution of a pixel/word on the objective function. For instance, the *occlusion* method [53] systematically occludes portions of the input, such as rectangular regions of an image input or text sequences in a text input. By doing so and calculating the difference in the objective function, the occlusion method can calculate an attribution score for each pixel/word.

A key challenge of applying interpretation methods to reverse engineer a trigger is how to design the objective function. One straightforward objective function is the loss function of the label $f \odot h(x_b)$ predicted by the downstream classifier for the input x_b . However, the foundation-model provider does not have access to the downstream classifier f . As a result, the provider cannot evaluate the loss function of $f \odot h(x_b)$ nor apply an interpretation method to calculate the attribution scores. To address the challenge, we define the objective function using the foundation model and bug instance without involving the downstream classifier. In particular, our objective function is directly related to the dissimilarity between

the feature vectors $h(x_b)$ and $h(x_r)$. Formally, we define the objective function $\ell(h, x_b, x_r)$ as follows:

$$\ell(h, x_b, x_r) = 1 - \text{sim}(h(x_b), h(x_r)), \quad (5)$$

where sim is the cosine similarity in our experiments. Given such objective function, we leverage an interpretation method to calculate an attribution score for each pixel/word in x_b . A higher attribution score is likely to indicate that the corresponding pixel/word has a larger impact on the objective function, i.e., the dissimilarity between $h(x_b)$ and $h(x_r)$.

We note that several previous studies (e.g., [8, 13]) also leverage interpretable machine learning methods to identify the backdoor trigger in an input. However, they focus on classifiers (i.e., $f \odot h$ in our problem) and use the straightforward loss function of the predicted label as the objective function. Our work focuses on foundation models and designs a new objective function tailored to them.

Step II: identifying trigger: After we have obtained the attribution score for each input pixel/word, we identify the trigger pixels/words based on them. Intuitively, the input pixels/words with larger attribution scores may form the trigger. Based on this intuition, we use a clustering method to divide the pixels/words into two clusters: one cluster containing the pixels/words with high attribution scores, and one cluster containing those with low attribution scores. For instance, we can leverage the K-means algorithm [29] with $K = 2$ to perform the clustering. We then treat the cluster with the higher mean attribution score as the trigger. We summarize our method to reverse engineer a trigger in Algorithm 2.

5 Evaluation

5.1 Experimental Setup

Datasets: We use eleven benchmark datasets, including six image datasets, two image-text datasets, and three text datasets. We show the details of these datasets in Table 16 in Appendix. These benchmark datasets were also used in prior studies [2, 22, 40, 54, 56] on backdoor attacks to foundation models. In addition to the numbers shown in Table 16 in Appendix, the STL10 dataset also contains 100,000 unlabeled images. We use these unlabeled images for pre-training when STL10 is used as the pre-training dataset. We follow Zhang et al. [54] to construct ImageNet100-A and ImageNet100-B, two subsets of ImageNet. Each of these subsets contains randomly sampled 100 classes from the ImageNet dataset with no overlapping classes. The CLIP-400M dataset was used for pre-training the OpenAI’s CLIP foundation model, which contains 400M image-text pairs but is not publicly available. The CC3M-Sub and Wiki103-Sub denote randomly subsampled subsets of the CC3M [39] and Wiki103 [30], respectively.

Backdoor attacks to foundation models: We consider the following five state-of-the-art backdoor attacks to foundation models.

BadEncoder [22]. BadEncoder injects backdoors into a vision foundation model by directly modifying its model parameters. A backdoored vision foundation model outputs attacker-desired feature vectors for any inputs embedded with a trigger. These feature vectors then cause a downstream classifier to produce attacker-desired predictions. BadEncoder is applicable to both single-modal and multi-modal vision foundation models.

CorruptEncoder [54]. CorruptEncoder injects backdoors into a vision foundation model by poisoning its pre-training data. In particular, CorruptEncoder injects a small fraction of poisoning inputs into the pre-training dataset such that the learnt vision foundation model outputs attacker-desired feature vectors for any inputs embedded with a trigger. Like BadEncoder, CorruptEncoder is also applicable to both single-modal and multi-modal vision foundation models.

Carlini & Terzis [2]. Carlini and Terzis proposed a backdoor attack to multi-modal vision foundation models. Their attack first creates a collection of text captions that include the target class name chosen by the attacker. Then their attack associates these text captions with a set of images embedded with the trigger to construct poisoning image-text pairs, which are then injected into the pre-training dataset.

POR [40] and NeuBA [56]. These are two backdoor attacks to language foundation models. POR modifies a language foundation model’s parameters to output a specific feature vector for any input containing a trigger, while NeuBA introduces a new backdoor pre-training objective in addition to the original pre-training objective to achieve the same goal.

Pre-training (backdoored) foundation models: We pre-train (backdoored) foundation models following the default experimental settings of the aforementioned backdoor attacks in the original papers, or use the publicly available ones from their codebase. For POR and NeuBA, we adopt a unified implementation [10]. Table 17a in Appendix summarizes the pre-training settings and backdoor triggers.

Training downstream classifiers: We use a foundation model as a feature extractor to train downstream classifiers on downstream datasets. For each downstream dataset, we use the training examples to train the downstream classifier and use the testing examples to evaluate its performance. These downstream classifiers are fully-connected neural networks trained with cross-entropy loss and the Adam optimizer. We follow the default parameter settings as those in the original papers on backdoor attacks when training the downstream classifiers. Table 17b in Appendix summarizes the training settings of downstream classifiers.

Compared patching methods: We compare Mudjacking with the following patching methods.

Fine-tuning (FT). FT is a widely used method for slightly modifying a model. When patching a foundation model, FT can achieve the effectiveness goal by optimizing the foundation model’s parameters to produce similar feature vectors for

x_b and x_r . Specifically, FT obtains the post-patching foundation model h' by solving the optimization problem: $\min_{h'} \mathcal{L}_e$, where \mathcal{L}_e is our effectiveness loss defined in Equation 1. h' is initialized as h and solved iteratively by gradient descent. We use the same parameter setting for FT as our Mudjacking.

Fine-tuning with ℓ_2 -norm or ℓ_∞ -norm constraint (FT+ ℓ_2 or FT+ ℓ_∞). Zhu et al. [58] use FT with an ℓ_2 -norm or ℓ_∞ -norm constraint to address overfitting and catastrophic forgetting. We apply this method to patch foundation models. In particular, when patching a foundation model h , FT+ ℓ_p obtains h' by solving the following optimization problem: $\min_{h'} \mathcal{L}_e$ subject to $\ell_p(h', h) \leq \delta$, where $p = 2$ or ∞ and δ is the threshold for the ℓ_2 or ℓ_∞ constraint. We use projected gradient descent to solve the optimization problem. Specifically, during each iteration, we first compute the gradient of the loss function with respect to the model parameters; after updating the model parameters, we project them onto the feasible set defined by the ℓ_p constraint if needed. FT+ ℓ_2 and FT+ ℓ_∞ use the same parameter settings as FT. We set a small threshold $\delta = 0.01$ to bound the model parameters' change.

Unlearning. Liu et al. [26] propose a machine unlearning based pre-deployment defense to erase backdoors from classifiers. They fine-tune a backdoored classifier to maximize the cross-entropy loss on trigger-embedded inputs with the attacker's target label. Furthermore, they bound the changes in model parameters using an ℓ_1 norm constraint. We extend their method to patch foundation models by replacing the cross-entropy loss as a cosine similarity loss between the reference input x_r and misclassified input x_b .

Fine-Pruning. Liu et al. [25] combine pruning and fine-tuning as a pre-deployment defense to eliminate backdoors from classifiers. This defense first prunes the channels in the convolutional layers that are dormant, i.e., that have negative average activation on clean data; and then it fine-tunes the pruned model to regain utility lost from pruning. We extend fine-pruning to patch foundation models as follows: we first prune 50% of the channels that are dormant on the reference input x_r and then fine-tune the pruned foundation model using the pre-training data for 50 epochs.

Evaluation metrics: We use four evaluation metrics, i.e., Correct Prediction of x_b (CP), Accuracy (Acc), Attack Success Rate (ASR), and Accuracy with Backdoor trigger (AccB). CP evaluates the effectiveness goal, Acc evaluates the locality goal, ASR and AccB evaluate the generalizability goal. We define the four evaluation metrics as follows:

Correct Prediction of x_b (CP). CP measures whether a downstream classifier correctly classifies x_b . CP can be either \times or \checkmark , denoting incorrect or correct classification of x_b .

Accuracy (Acc). Acc denotes the testing accuracy of a downstream classifier $f \odot h$ (or $f' \odot h'$ after patching). Specifically, Acc is the fraction of clean downstream testing examples that are correctly classified. If post-patching Acc is close to or higher than pre-patching Acc, then the locality goal is achieved.

Attack Success Rate (ASR). We generate *backdoored testing inputs* by embedding a trigger into all clean downstream testing inputs that are not from the target class. ASR is the fraction of such backdoored testing inputs that are classified as the target class by the downstream classifier.

Accuracy with backdoor trigger (AccB). AccB is the testing accuracy of a downstream classifier $f \odot h$ (or $f' \odot h'$ after patching) for backdoored testing inputs. In particular, AccB is the fraction of backdoored testing inputs that are correctly classified by a downstream classifier. The generalizability goal is achieved if the post-patching ASR is small and AccB is close to Acc.

Parameter settings of Mudjacking: We randomly sample a bug instance (x_b, x_r) , where x_b is a trigger-embedded misclassified testing input and x_r is a correctly classified testing input from the downstream testing dataset. By default, we use BadEncoder as the backdoor attack, CIFAR10 for pre-training, and STL10 for downstream classifier training.

The default parameter settings for our Mudjacking are as follows: $\lambda_l = \lambda_g = 1$; the number of epochs $T = 200$ in Algorithm 1; and the learning rates used for patching are shown in Table 18 in Appendix. The validation dataset size and batch size are determined based on our computation resources. Table 18 in Appendix summarizes these parameters. Unless otherwise mentioned, we sample the validation dataset from the pre-training dataset, except for CLIP-400M. Since CLIP-400M is not publicly available, we sample the training inputs in ImageNet as our validation dataset. By default, we use the Occlusion interpretation method [53] when reverse engineering a trigger. We performed experiments using 18 NVIDIA RTX 6000 GPUs, with each GPU having 24GB memory.

5.2 Experimental Results

Mudjacking achieves the three patching goals: Table 1 and Table 2 show the results of patching vision and language foundation models, respectively. We have multiple observations. First, Mudjacking consistently achieves the effectiveness goal in all settings, indicated by CP = \checkmark . This is because the post-patching foundation model produces similar feature vectors for the misclassified input x_b and reference input x_r .

Second, Mudjacking also achieves the locality goal by maintaining Acc after patching. This is because, for a clean input, the post-patching and pre-patching foundation models output similar feature vectors. In particular, even when the validation dataset is a small fraction of the clean pre-training dataset, it is sufficient for our Mudjacking to achieve the locality goal. For example, as shown in Table 1b, when the pre-training dataset is CLIP-400M and the foundation model is attacked by BadEncoder, using only 50,000 images (only 0.0125% of 400 million) from ImageNet as the validation dataset is sufficient to achieve the locality goal.

Third, our Mudjacking successfully achieves the generalizability goal since ASR is low (close to 0) except for patching

Table 1: Results for patching vision foundation models.

(a) Single-modal

Attack method	Pre-training dataset	Downstream dataset	Before patching				After patching			
			CP	Acc	ASR	AccB	CP	Acc	ASR	AccB
BadEncoder	CIFAR10	STL10	×	76.46	99.82	0.10	✓	76.59	2.39	73.38
		SVHN	×	69.07	98.92	0.53	✓	78.49	5.97	65.39
	STL10	CIFAR10	×	86.64	97.07	1.24	✓	86.39	2.59	81.20
		SVHN	×	65.21	97.44	0.36	✓	76.21	7.94	61.04
CorruptEncoder	ImageNet100-A	ImageNet100-B	×	61.68	94.32	2.57	✓	61.76	1.57	57.64
		Oxford-IIIT Pets	×	56.71	71.90	4.96	✓	55.69	2.18	51.23

(b) Multi-modal

Attack method	Pre-training dataset	Downstream dataset	Before patching				After patching			
			CP	Acc	ASR	AccB	CP	Acc	ASR	AccB
BadEncoder	CLIP-400M	STL10	×	96.70	99.92	0.04	✓	95.65	0.42	95.06
		SVHN	×	69.93	100.00	0.00	✓	73.35	3.48	69.13
Carlini & Terzis	CC3M-Sub	ImageNet100-B	×	51.47	98.22	1.09	✓	49.26	5.03	41.53
CorruptEncoder	CC3M-Sub	ImageNet100-B	×	48.44	94.40	0.81	✓	45.92	3.46	40.27

Table 2: Results for patching language foundation models.

Attack method	Pre-training dataset	Downstream dataset	Before patching				After patching			
			CP	Acc	ASR	AccB	CP	Acc	ASR	AccB
NeuBA	Wiki-103	SST-2	×	80.28	100.00	0.00	✓	79.68	24.34	75.65
		HSOL	×	80.16	100.00	0.00	✓	81.28	18.28	81.72
POR	Wiki-103	SST-2	×	82.59	67.66	12.34	✓	81.82	17.49	82.51
		HSOL	×	82.90	99.92	0.08	✓	82.78	17.07	82.93

language models and AccB is close to Acc after patching across all settings. We note that ASR is not close to 0 for patching language foundation models in Table 2. This is because the downstream datasets have two classes. Given a downstream classifier built based on a clean language foundation model, ASR would be close to the misclassification rate of testing inputs. We observe that ASR in Table 2 is indeed close to the misclassification rate of testing inputs, roughly 20%. Therefore, the post-patching language foundation models are similar to clean language foundation models. Our results suggest that our method effectively mitigates the backdoor vulnerabilities in a backdoored foundation model.

Mudjacking outperforms existing methods: Table 3 compares Mudjacking with existing patching methods for patching the vision foundation model in our default setting. We observe that Mudjacking outperforms these methods at achieving the three patching goals. Specifically, FT, FT+ ℓ_2 , FT+ ℓ_∞ , and unlearning can achieve the effectiveness goal. This is because they all fine-tune the foundation model to minimize the effectiveness loss \mathcal{L}_e . They achieve the locality goal to some extent, i.e., Acc is close to that before patching, but they cannot achieve the generalizability goal. Fine-Pruning achieves the locality and generalizability goals to some extent, but it cannot achieve the effectiveness goal.

Table 3: Comparing Mudjacking with existing methods.

Patching method	CP	Acc	ASR	AccB
Before patching	×	76.46	99.82	0.10
FT	✓	69.61	40.57	32.50
FT+ ℓ_2	✓	72.05	55.90	24.31
FT+ ℓ_∞	✓	70.04	41.79	30.61
Unlearning	✓	70.02	34.57	32.12
Fine-Pruning	×	71.92	6.65	61.97
Mudjacking	✓	76.59	2.39	73.38

Impact of different interpretation methods: Our Mudjacking uses an interpretation method to reverse engineer a trigger from a bug instance. We compare four widely used interpretation methods, i.e., Occlusion [53], GradCam [37], Saliency Map [41], Guided Backproagation (GuidedBack) [44]. Table 4 shows the results, from which we have the following two main observations. First, we observe that different interpretation methods lead to different reverse-engineered triggers. Specifically, Occlusion and GradCam can reverse-engineer triggers that contain majority of the true backdoor trigger (the white square at the bottom right corner). On the other hand, the triggers reverse-engineered by Saliency Map and GuidedBack contain only a part of the true backdoor trigger.

Table 4: Impact of different interpretation methods.





Interpretation method	Reverse engineered trigger	CP	Acc	ASR	AccB
Occlusion		✓	76.59	2.39	73.38
GradCam		✓	76.47	3.42	71.44
Saliency Map		✓	76.58	11.88	61.72
GuidedBack		✓	76.29	13.56	61.67

Table 5: Impact of the three loss terms.

Removed loss term	CP	Acc	ASR	AccB
None	✓	76.59	2.39	73.38
Effectiveness loss	×	76.70	2.65	72.97
Locality loss	✓	48.50	7.10	45.61
Generalizability loss	✓	76.70	26.49	50.57

Therefore, Occlusion and GradCam outperform the other two in terms of the generalizability goal, i.e., ASRs are lower and AccBs are higher. Second, we find that Mudjacking successfully achieves the effectiveness and locality goals, regardless of the interpretation method used. The reason is that the reverse-engineered trigger is used only in the generalizability loss and does not impact the effectiveness and locality losses.

Impact of the three loss terms: Our formulated optimization problem in Equation 4 consists of three loss terms. We study the impact of removing each of them and show the results in Table 5. We have the following three key observations. First, the removal of the effectiveness loss hinders the patched foundation model’s ability to correctly classify the misclassified input x_b . Second, removing the locality loss leads to a substantial drop in Acc, falling from 76.59% (with no loss term removed) to 48.50%. This highlights the critical role of the locality loss in preserving the utility of the patched foundation models. Third, the removal of the generalizability loss results in a significant increase in ASR to 26.49% and a decrease in AccB to 50.57%. This indicates the significant role of the generalizability loss in mitigating backdoor vulnerabilities. To summarize, these observations emphasize the importance of integrating all the three loss terms to achieve the three patching goals.

Impact of validation dataset size: Mudjacking relies on a validation dataset \mathcal{D}_{val} to calculate both the locality loss (Equation 2) and the generalizability loss (Equation 3). We investigate how varying the size of the validation dataset influences our method’s performance on three downstream datasets, namely STL10 and SVHN. We show the results in Figure 2. We find that both CP and Acc converge when the size of the validation dataset exceeds 10% of the pre-training dataset size across the tested downstream datasets. This suggests that our method is efficient in achieving its effective-

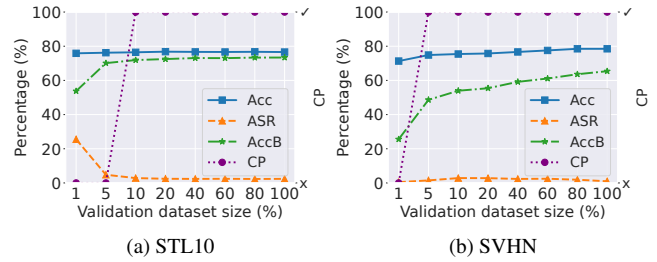


Figure 2: Impact of the validation dataset size on different downstream datasets. The validation dataset is a subset of the pre-training dataset CIFAR10.

Table 6: Impact of the validation dataset distribution. The pre-training dataset is CIFAR10.

Validation dataset distribution	CP	Acc	ASR	AccB
CIFAR10	✓	76.59	2.39	73.38
STL10	✓	76.60	2.71	72.82
ImageNet	✓	75.88	2.44	72.24

ness and locality goals, even when the validation dataset is a small fraction of the pre-training dataset. Interestingly, in terms of ASR and AccB, we notice some variations. For the STL10 dataset, both metrics begin to stabilize when the validation dataset size exceeds 10% of the pre-training dataset size. However, for the SVHN dataset, ASR stabilizes when the validation dataset is only 1% of the pre-training dataset, while AccB continues to improve as the validation dataset size increases. This is probably because STL10’s data distribution is more similar to the pre-training dataset CIFAR10, compared to SVHN.

Impact of validation dataset distribution: We further investigate the impact of different distributions of the validation dataset on our method’s performance. We show the results in Table 6. We observe that Mudjacking achieves all three patching goals regardless of the validation dataset distribution. The reason is that all these validation datasets contain diverse images, which help our method achieve the locality and generalizability goals that rely on \mathcal{D}_{val} .

Patching multiple bugs: A client may detect and report bugs to the foundation-model provider at different times. Therefore, the foundation-model provider may apply our method to patch the foundation model multiple times. We illustrate this scenario in Figure 3. In our experiments, the foundation-model provider receives three bug instances sequentially and patches the pre-patching foundation model h three times, which results in three post-patching foundation models h'_1 , h'_2 , and h'_3 . Table 7 shows our method’s performance in this case. We have two main observations. First, Mudjacking can achieve the effectiveness and locality goals in every patching attempt. Second, patching multiple bugs can better achieve the generalizability goal as ASR decreases when patching more times.

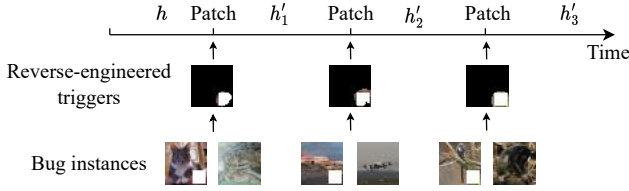


Figure 3: Patching multiple bugs.

Table 7: Patching multiple bugs.

Foundation model	CP	Acc	ASR	AccB
h	×	76.46	99.82	0.10
h'_1	✓	76.51	3.43	71.43
h'_2	✓	75.31	2.25	71.93
h'_3	✓	74.88	2.00	71.10

Table 8: Normal vs. backdoor bug.

Bug instance	Reverse engineered trigger	CP	Acc	ASR	AccB
Backdoor bug		✓	76.59	2.39	73.38
Normal bug		✓	76.29	-	-

This is because as the foundation model is patched more times, the trigger is better reverse-engineered (as shown in Figure 3), which improves the generalizability of Mudjacking.

Patching normal bugs: The misclassified input x_b can also be a normal input without a backdoor trigger. Mudjacking can be directly applied to patch such bugs. We show the results of patching a normal bug in Table 8. When x_b is a normal misclassified input, our method still achieves the first two patching goals, while the generalizability goal is not well defined for normal bugs. This is because Mudjacking achieves the three patching goals by assuming x_b is a trigger-embedded input from a backdoor attack without distinguishing between backdoor and normal bugs.

Impact of reference inputs: A bug instance (x_b, x_r) includes a reference input x_r . By default, we assume x_r is a real image from the client. We study whether x_r could be a random input that has a similar feature vector with a real one. Generating such a random input would be easier for the foundation-model provider than the client, who only has black-box access to the foundation model. Therefore, we consider the following scenario: the client sends a misclassified input x_b together with the feature vector of a real reference input, i.e., $h(x_r)$, to the foundation-model provider. The foundation-model provider generates a reference input x'_r whose feature vector has a high cosine similarity with $h(x_r)$. Specifically, we iteratively update an initial random input x'_r for 100 iterations to maximize the cosine similarity between $h(x'_r)$ and $h(x_r)$ using the Adam optimizer with a learning rate of 1×10^{-2} . Figure 8 in Ap-

Table 9: Impact of reference input.

Reference input	Reverse engineered trigger	CP	Acc	ASR	AccB
Real		✓	76.59	2.39	73.38
Random		✓	76.66	2.60	72.11

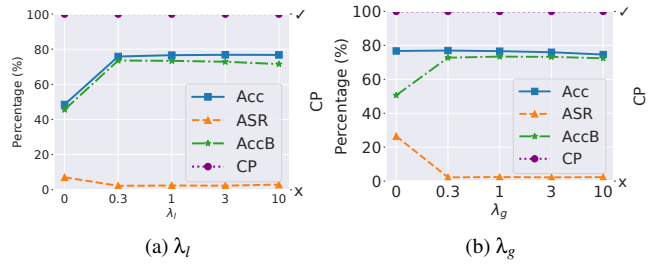


Figure 4: Impact of λ_l and λ_g .

pendix shows x_r and x'_r , while Table 9 shows the results of using x_r or x'_r for patching. We have two main observations. First, Mudjacking can successfully reverse engineer the trigger using x'_r . This is because x'_r has a dissimilar feature vector with x_b . Second, Mudjacking achieves similar performance when using either a real or random reference input. This is because of the highly similar feature vectors between the real reference input and the random one, and the successfully reverse-engineered trigger.

Impact of other parameters: We study the performance of our Mudjacking when patching the foundation model and training the downstream classifier using different learning rates, batch sizes, or epochs. We show the results in Table 19 in Appendix. We observe that Mudjacking achieves the three patching goals across all parameter settings. Our formulated optimization problem in Equation 4 is a weighted sum of the three loss terms. We further study the impact of the two weights λ_l and λ_g . We show the results in Figure 4. We have two key observations. First, Mudjacking achieves the three goals once the optimization problem incorporates the three loss terms, i.e., $\lambda_l \neq 0$ and $\lambda_g \neq 0$. Second, too large λ_l (or λ_g) may slightly impact the generalizability (or locality) goal. For example, when $\lambda_l \geq 3$ (or $\lambda_g \geq 3$), the AccB (or Acc) slightly decreases. This is because too large λ_l (or λ_g) makes Mudjacking minimize more on the locality (or generalizability) loss term than the other two.

5.3 Patching Downstream Classifiers Alone

Given a misclassified input x_b , a client could try patching its downstream classifier directly. We consider the following methods to patch a downstream classifier while keeping the foundation model h unchanged.

FT: FT uses (x_b, y_b) to fine-tune the downstream classifier, where y_b is the true label of x_b . In particular, FT obtains a

Table 10: Comparing Mudjacking with patching downstream classifier alone.

Patching method	CP	Acc	ASR	AccB
Before patching	×	76.46	99.82	0.10
FT	✓	34.86	0.00	11.11
FT with \mathcal{D}_{train}	✓	76.54	0.44	11.14
AI-Lancet [57]	✓	74.54	0.82	11.31
BEAGLE [7] + reversed trigger	✓	76.34	5.22	54.33
BEAGLE [7] + exact trigger	✓	76.31	3.47	58.91
Mudjacking	✓	76.59	2.39	73.38

post-patching downstream classifier f' by solving the optimization problem: $\min_{f'} \ell_{CE}(x_b, y_b; f' \odot h)$, where ℓ_{CE} is the cross-entropy loss and h is not changed.







FT with \mathcal{D}_{train} : Since the client has a downstream training dataset \mathcal{D}_{train} , the client can fine-tune the downstream classifier using \mathcal{D}_{train} together with the misclassified input x_b . In particular, the client uses $\mathcal{D}_{train} \cup \{(x_b, y_b)\}$ to fine-tune the downstream classifier.

AI-Lancet [57]: AI-Lancet is a patching method for classifiers. Given a misclassified input (i.e., x_b in our case) and a correctly classified input (i.e., x_r in our case), AI-Lancet locates the backdoor neurons and adjusts them to remove the backdoor. We apply AI-Lancet to patch downstream classifier. We use their neuron-flip variant that achieves better results.

BEAGLE [7]: BEAGLE reverse engineers a trigger from multiple misclassified trigger-embedded inputs and fine-tunes a classifier using clean training inputs embedded with the reverse engineered trigger and correct labels. However, in our problem setup, we consider a client who detects a single misclassified trigger-embedded input. Therefore, their method of reverse engineering a trigger is not applicable. Instead, we use the trigger reverse engineered by the foundation-model provider using our method or the exact trigger when applying BEAGLE to patch a downstream classifier. We use ‘BEAGLE + reversed trigger’ and ‘BEAGLE + exact trigger’ to denote these two cases, respectively. Note that BEAGLE + exact trigger gives an advantage to BEAGLE.

For all these methods except AI-Lancet, which does not require fine-tuning, we fine-tune a downstream classifier for 50 epochs with a learning rate of 0.001 using Adam optimizer. Table 10 compares Mudjacking with patching downstream classifiers alone. We observe that all patching methods achieve the effectiveness goal. However, FT does not achieve the locality nor generalizability goal; FT with \mathcal{D}_{train} and AI-Lancet do not achieve the generalizability goal since ASRs are low but AccBs are also low, i.e., trigger-embedded inputs are still misclassified as non-target classes; and BEAGLE better achieves the generalizability goal than the other baselines, but it still has higher ASR and much lower AccB than Mudjacking. We note that BEAGLE + reversed trigger achieves comparable performance with BEAGLE + exact trigger, which further

Table 11: Patching results of Mudjacking when a backdoor attack uses different trigger patterns.

Trigger pattern	Patching	CP	Acc	ASR	AccB
	Before	×	76.46	99.82	0.10
	After	✓	76.59	2.39	73.38
	Before	×	76.49	100.00	0.00
	After	✓	76.14	3.58	71.57
	Before	×	76.25	98.62	0.76
	After	✓	76.62	2.40	73.46
	Before	×	74.90	97.29	1.43
	After	✓	76.61	2.42	73.69
	Before	×	75.11	99.44	0.14
	After	✓	75.88	3.66	73.21
	Before	×	76.62	100.00	0.00
	After	✓	76.49	3.78	67.58

indicates that our method can reverse engineer a high-quality trigger from a single bug instance.

5.4 Adaptive Backdoor Attacks

A key component of Mudjacking is to reverse engineer the trigger from a bug instance. Therefore, we consider adaptive attacks that aim to enhance the complexity and stealthiness of the trigger, making it more difficult for Mudjacking to reverse engineer it. A trigger consists of two dimensions: pattern and location. A pattern can be further characterized by its shape, value, and size. For instance, a pattern could be a square (shape) of 10×10 pixels (size), where each pixel has a value of 1 (value). Location could be a fixed location at the bottom right corner or a random location for each input.

Adapting shapes and values of the trigger pattern: Table 11 shows the patching results of our method when a backdoor attack uses trigger patterns with different shapes and values: a 10×10 square with white pixels, a 10×10 square with random pixel values, a 10×10 triangle with random pixel values, an apple logo, an emoji, and a spread-out trigger. The spread-out trigger comprises two 10×10 squares with random pixel values placed diagonally, with the first located at the upper left corner and the second located at the lower right corner. The apple logo and emoji could represent physical objects [50] in physical backdoor attacks.

Our results indicate that, regardless of the shapes and values, the backdoor attacks are effective at achieving high ASRs before patching. However, Mudjacking can still achieve the three patching goals. For instance, we observe substantial decreases in ASRs across all trigger patterns; and AccB remains close to Acc. Note that AccB is 10% lower than Acc for the spread-out trigger. This may be due to the spread-out trigger occupying a larger portion of the image, potentially obscuring key objects and leading to misclassification.

Adapting trigger sizes: Figure 6 in Appendix shows the patching results of our Mudjacking when a backdoor attack uses different trigger sizes, where the trigger pattern is a white

Table 12: Patching results of our Mudjacking and variant Mudjacking-RL when a backdoor attack uses a fixed or random trigger location.

Trigger location	Patching	CP	Acc	ASR	AccB
Fixed	Before	×	76.46	99.82	0.10
	Mudjacking	✓	76.59	2.39	73.38
	Mudjacking-RL	✓	75.85	3.19	71.92
Random	Before	×	76.51	100.00	0.00
	Mudjacking	✓	76.60	41.43	28.42
	Mudjacking-RL	✓	76.36	3.50	65.26

square located at the bottom right corner. As the trigger size increases, pre-patching ASR also increases, which indicates that a backdoor attack with a larger trigger is more effective. However, after patching, ASR substantially drops and AccB remains close to Acc, which shows that our Mudjacking consistently achieves the generalizability goal. Moreover, our Mudjacking also consistently achieves the effectiveness and utility goals across all trigger sizes studied.

Random trigger location: In our Mudjacking, the generalizability loss (Equation 3) embeds the reverse-engineered trigger into each input in the validation dataset at the same location. This is because a backdoor attack embeds the trigger at the same, fixed location of inputs. Therefore, an adaptive backdoor attack is to embed the trigger at a random location of an input. To patch such backdoor bugs, we define a variant of Mudjacking, denoted as Mudjacking-RL, as follows: we embed the reverse-engineered trigger at a random location of each input in the validation dataset when defining the generalizability loss. Table 12 shows the patching results for backdoor attacks with a fixed or random trigger location. Our results show that both Mudjacking and Mudjacking-RL achieve the three patching goals when the trigger location is fixed. For random trigger location, both achieve the effectiveness and locality goals, but Mudjacking-RL better achieves the generalizability goal.

Source-specific backdoor attacks: Our generalizability loss assumes that any input embedded with the trigger would activate the backdoor. Therefore, one adaptive backdoor attack is the so-called source-specific backdoor [15, 47, 49], in which the backdoor is activated only when the trigger-embedded input is from a particular class called *source class*. Existing source-specific backdoor attacks were designed for classifiers. We extend BadEncoder as a source-specific backdoor attack to foundation models. Due to limited space, we show the technical details about this extension in Appendix A. Table 13 shows the patching results, where ASR-source (or ASR-other) is the fraction of trigger-embedded inputs from the source class (or non-source classes) that are classified as the target class by the downstream classifier. Our results show that ASR-source is much higher than ASR-other before patching (i.e., source-specific backdoor is effective), but Mudjacking still successfully achieves the three patching goals.

Table 13: Patching results for source-specific backdoor attack.

Patching	CP	Acc	ASR-source	ASR-other	AccB
Before	×	75.33	64.38	37.33	10.78
After	✓	75.19	3.79	3.22	70.94

Table 14: Patching results of Mudjacking-RL against dynamic backdoor attacks.

Foundation model	CP	Acc	ASR	AccB
Before patching	×	76.46	97.31	0.21
After patching first bug instance	✓	76.07	38.42	35.23
After patching second bug instance	✓	75.16	2.32	70.48

Dynamic backdoor attacks: A dynamic backdoor attack [36] uses multiple triggers and random locations. These attacks were designed for classifiers, and we extend them to foundation models using BadEncoder. Specifically, we use two 10×10 triggers, where trigger t_1 is blue and trigger t_2 is red. When embedding a trigger into an image, the attacker randomly selects one of these triggers and embeds it into the image at a random location. Suppose a client detects a bug instance containing trigger t_1 ; the foundation-model provider patches the foundation model based on the bug instance; then a client detects another bug instance containing trigger t_2 ; and the foundation-model provider further patches the foundation model. Table 14 shows the patching results using Mudjacking-RL. Before patching, the dynamic backdoor attack is very successful. After patching the first bug instance, Mudjacking-RL achieves the first two goals, but not the generalizability goal. This is because the backdoor attack is still successful when trigger t_2 is used. After patching the second bug instance, Mudjacking-RL achieves the three goals. In general, when a dynamic backdoor attack uses n different triggers, Mudjacking-RL requires n bug instances corresponding to the n triggers to fully remove the backdoor.

6 Discussion and Limitations

Patching latent-space backdoor attacks: Mudjacking considers standard backdoor attacks with localized, universal triggers that can be easily implemented in the physical world. Latent-space backdoor attacks [14] use a whole-image imperceptible perturbation as a trigger. We extend such backdoor attacks to foundation models as follows: we craft a whole-image perturbation whose ℓ_∞ -norm = 0.01 as a trigger, and then we use BadEncoder to inject this trigger into a foundation model. Table 15 shows the patching results, where Mudjacking + exact trigger means that Mudjacking uses the exact trigger instead of the reverse-engineered one for patching.

We have several observations. First, the attack is highly effective with a high ASR before patching. Second, Mudjacking achieves the first two goals, but not the generalizabil-

Table 15: Patching latent-space backdoor attacks.

Patching method	CP	Acc	ASR	AccB
Before patching	×	75.17	83.14	4.10
Mudjacking	✓	75.40	59.82	18.43
Mudjacking + exact trigger	✓	76.25	7.35	71.71



Figure 5: Visualization of the exact trigger (*left*) and the trigger (*right*) reverse-engineered by Mudjacking in latent-space backdoor attack.

ity goal. This is because the interpretation machine learning method cannot reverse engineer the whole-image trigger, as shown in Figure 5. Third, Mudjacking + exact trigger achieves the three goals, which further confirms that the ineffectiveness of Mudjacking at achieving the generalizability goal is because the trigger cannot be accurately reverse-engineered. However, when the whole-image trigger can be reverse-engineered (e.g., by a method designed in the future), our Mudjacking can be used to patch the backdoor.

Patching adversarial-example bugs: Mudjacking focuses on patching backdoor bugs and, as a side effect, can also patch normal misclassification bugs. Adversarial examples [3, 46] are another category of bugs for classifiers. Specifically, an attacker can craft a small perturbation δ such that $x + \delta$ is misclassified, i.e., $f \odot h(x) \neq f \odot h(x + \delta)$. Suppose the misclassified input x_b in a bug instance is an adversarial example. If only the effectiveness and locality goals are desired, the bug instance can be treated as a normal misclassification bug, and thus our Mudjacking can patch the foundation model to fix it. It is an interesting future work to explore how to define the generalizability goal for adversarial-example bugs and adapt Mudjacking to achieve it. For instance, one way to define the generalizability goal is that all adversarial examples generated by the same attack that generated x_b should be correctly classified after patching. Another way is that all adversarial examples whose perturbation sizes are bounded by that in x_b should be correctly classified after patching.

Malicious client: We assume a bug instance is sent from a benign client. We acknowledge that a malicious client may send carefully crafted bug instances to a foundation-model provider with a goal to disrupt the patching process. We believe it is an important future work to study whether and how a malicious client can subvert the patching process via malicious bug instances.

7 Conclusion and Future Work

We find that, given a bug instance from a client, a foundation-model provider can patch its foundation model to remove backdoor vulnerabilities without sacrificing its utility. In particular, the provider can patch its foundation model to achieve three goals. Each goal can be quantified by a loss term, and a foundation model can be patched via minimizing a weighted sum of the three loss terms using a method based on gradient descent. As a side effect, a foundation model can also be effectively patched when a bug instance is for a normal misclassification error. Interesting future works include patching foundation models to fix adversarial-example bugs and latent-space backdoors, as well as exploring the implications of malicious clients on patching.

Acknowledgements

We thank the anonymous shepherd and reviewers for their constructive comments. This work was supported by NSF under Grant No. 2131859, 2125977, 2112562, 1937786, 1937787, the Army Research Office under Grant No. W911NF2110182, as well as Ant Group.

References

- [1] Tom Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In *NeurIPS*, 2020.
- [2] Nicholas Carlini and Andreas Terzis. Poisoning and backdooring contrastive learning. In *ICLR*, 2022.
- [3] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *S&P*, 2017.
- [4] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *ICML*, 2020.
- [5] Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning. *arXiv*, 2020.
- [6] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv*, 2017.
- [7] Siyuan Cheng, Guanhong Tao, Yingqi Liu, Shengwei An, Xiangzhe Xu, Shiwei Feng, Guangyu Shen, Kaiyuan Zhang, Qiuling Xu, Shiqing Ma, et al. Beagle: Forensics of deep learning backdoor attack for better defense. In *NDSS*, 2023.
- [8] Edward Chou, Florian Tramèr, and Giancarlo Pellegrino. Sentinet: Detecting localized universal attacks against deep learning systems. In *S&P Workshops*, 2020.






- [9] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS*, 2011.
- [10] Ganqu Cui, Lifan Yuan, Bingxiang He, Yangyi Chen, Zhiyuan Liu, and Maosong Sun. A unified evaluation of textual backdoor learning: Frameworks and benchmarks. In *NeurIPS: Datasets and Benchmarks*, 2022.
- [11] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language. In *ICWSM*, 2017.
- [12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.
- [13] Bao Gia Doan, Ehsan Abbasnejad, and Damith C Ranasinghe. Februus: Input purification defense against trojan attacks on deep neural network systems. In *ACSAC*, 2020.
- [14] Khoa Doan, Yingjie Lao, and Ping Li. Backdoor attack with imperceptible input and latent modification. *NeurIPS*, 2021.
- [15] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. Strip: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.
- [16] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *IEEE Access*, 2017.
- [17] Wenbo Guo, Lun Wang, Yan Xu, Xinyu Xing, Min Du, and Dawn Song. Towards inspecting and eliminating trojan backdoors in deep neural networks. In *ICDM*, 2020.
- [18] Zayd Hammoudeh and Daniel Lowd. Identifying a training-set attack’s target using renormalized influence estimation. In *CCS*, 2022.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
- [20] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Intrinsic certified robustness of bagging against data poisoning attacks. In *AAAI*, 2021.
- [21] Jinyuan Jia, Yupei Liu, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of nearest neighbors against data poisoning and backdoor attacks. In *AAAI*, 2022.
- [22] Jinyuan Jia, Yupei Liu, and Neil Zhenqiang Gong. Badencoder: Backdoor attacks to pre-trained encoders in self-supervised learning. In *S&P*, 2022.
- [23] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
- [24] Hongbin Liu, Jinyuan Jia, and Neil Zhenqiang Gong. Poisonedencoder: Poisoning the unlabeled pre-training data in contrastive learning. In *USENIX Security*, 2022.
- [25] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.
- [26] Yang Liu, Mingyuan Fan, Cen Chen, Ximeng Liu, Zhuo Ma, Li Wang, and Jianfeng Ma. Backdoor defense with machine unlearning. In *INFOCOM*, 2022.
- [27] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.
- [28] Wanlun Ma, Derui Wang, Ruoxi Sun, Minhui Xue, Sheng Wen, and Yang Xiang. The “beatrice” resurrections: Robust backdoor detection via gram matrices. In *NDSS*, 2023.
- [29] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, 1967.
- [30] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv*, 2016.
- [31] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *CVPR*, 2012.
- [32] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*, 2021.
- [33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.
- [34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. In *IJCV*, 2015.
- [35] Aniruddha Saha, Ajinkya Tejankar, Soroush Abbasi Koohpayegani, and Hamed Pirsiavash. Backdoor attacks on self-supervised learning. In *CVPR*, 2022.

- [36] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. Dynamic backdoor attacks against machine learning models. In *EuroS&P*, 2022.
- [37] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV*, 2017.
- [38] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y Zhao. Poison forensics: Traceback of data poisoning attacks in neural networks. In *USENIX Security*, 2022.
- [39] Piyush Sharma, Nan Ding, Sebastian Goodman, and Radu Soricut. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *ACL*, 2018.
- [40] Lujia Shen, Shouling Ji, Xuhong Zhang, Jinfeng Li, Jing Chen, Jie Shi, Chengfang Fang, Jianwei Yin, and Ting Wang. Backdoor pre-trained models can transfer to all. In *CCS*, 2021.
- [41] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv*, 2013.
- [42] Anton Sinitsin, Vsevolod Plokhotnyuk, Dmitriy Pyrkin, Sergei Popov, and Artem Babenko. Editable neural networks. In *ICLR*, 2020.
- [43] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.
- [44] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR workshop*, 2015.
- [45] Johannes Stalkamp, Marc Schlipf, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. In *Neural networks*, 2012.
- [46] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.
- [47] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of dnns for robust backdoor contamination detection. In *USENIX Security*, 2021.
- [48] Binghui Wang, Xiaoyu Cao, Neil Zhenqiang Gong, et al. On certifying robustness against backdoor attacks via randomized smoothing. *arXiv*, 2020.
- [49] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *S&P*, 2019.
- [50] Emily Wenger, Josephine Passananti, Arjun Nitin Bhagoji, Yuanshun Yao, Haitao Zheng, and Ben Y Zhao. Backdoor attacks against deep learning systems in the physical world. In *CVPR*, 2021.
- [51] Chong Xiang, Arjun Nitin Bhagoji, Vikash Sehwal, and Prateek Mittal. Patchguard: A provably robust defense against adversarial patches via small receptive fields and masking. In *USENIX Security*, 2021.
- [52] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting ai trojans using meta neural analysis. In *S&P*, 2021.
- [53] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014.
- [54] Jinghui Zhang, Hongbin Liu, Jinyuan Jia, and Neil Zhenqiang Gong. Corruptencoder: Data poisoning based backdoor attacks to contrastive learning. *arXiv*, 2022.
- [55] Zaixi Zhang, Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Backdoor attacks to graph neural networks. In *SACMAT*, 2021.
- [56] Zhengyan Zhang, Guangxuan Xiao, Yongwei Li, Tian Lv, Fanchao Qi, Zhiyuan Liu, Yasheng Wang, Xin Jiang, and Maosong Sun. Red alarm for pre-trained models: Universal vulnerability to neuron-level backdoor attacks. In *Machine Intelligence Research*, 2023.
- [57] Yue Zhao, Hong Zhu, Kai Chen, and Shengzhi Zhang. Ai-lancet: Locating error-inducing neurons to optimize neural networks. In *CCS*, 2021.
- [58] Chen Zhu, Ankit Singh Rawat, Manzil Zaheer, Srinadh Bhojanapalli, Daliang Li, Felix Yu, and Sanjiv Kumar. Modifying memories in transformer models. *arXiv*, 2020.
- [59] Rui Zhu, Di Tang, Siyuan Tang, XiaoFeng Wang, and Haixu Tang. Selective amnesia: On efficient, high-fidelity and blind suppression of backdoor effects in trojaned machine learning models. In *S&P*, 2023.

Table 16: Dataset statistics.

Domain	Dataset	# Training	# Testing	# Classes	Usage
Single-modal Vision	CIFAR10 [23]	50,000	10,000	10	Pre-training & Downstream
	STL10 [9]	5,000	8,000	10	Pre-training & Downstream
	ImageNet100-A [34]	128,420	-	100	Pre-training
	SVHN [45]	73,257	26,032	10	Downstream
	ImageNet100-B [34]	126,689	5,000	100	Downstream
Multi-modal Vision	Oxford-IIIT Pets [31]	3,680	3,669	37	Downstream
	CLIP-400M [32]	≈ 400 Million	-	-	Pre-training
Language	CC3M-Sub [39]	500,000	-	-	Pre-training
	Wiki103-Sub [30]	250,000	-	103	Pre-training
	SST-2 [43]	7,792	1,821	2	Downstream
	HOSL [11]	8,308	2,485	2	Downstream

Table 17: Pre-training settings of foundation models and training settings of downstream classifiers.

(a) Pre-training settings and backdoor triggers						(b) Training settings of downstream classifiers				
Attack method	Domain	Backdoor trigger	Pre-training algorithm	Model	Learning rate	Attack method	Domain	# Fully connected layers	# Epochs	Learning rate
BadEncoder	Single-modal Vision		SimCLR [4]	ResNet18 [19]	0.001	BadEncoder	Single-modal Vision	2	500	0.0001
	Multi-modal Vision		CLIP [32]	ResNet50	0.0001		Multi-modal Vision	2	500	0.0001
CorruptEncoder	Single-modal Vision		MoCo v2 [5]	ResNet18	0.001	CorruptEncoder	Single-modal Vision	1	50	0.01
	Multi-modal Vision		CLIP	ResNet50	0.001		Multi-modal Vision	1	50	0.01
Carlini & Terzis	Multi-modal Vision		CLIP	ResNet50	0.001	Carlini & Terzis	Multi-modal Vision	1	50	0.01
POR	Language	bb	BERT [12]	BERT-base	0.0001	POR	Language	1	10	0.001
NeuBA	Language	≡	BERT	BERT-base	0.0001	NeuBA	Language	1	10	0.001

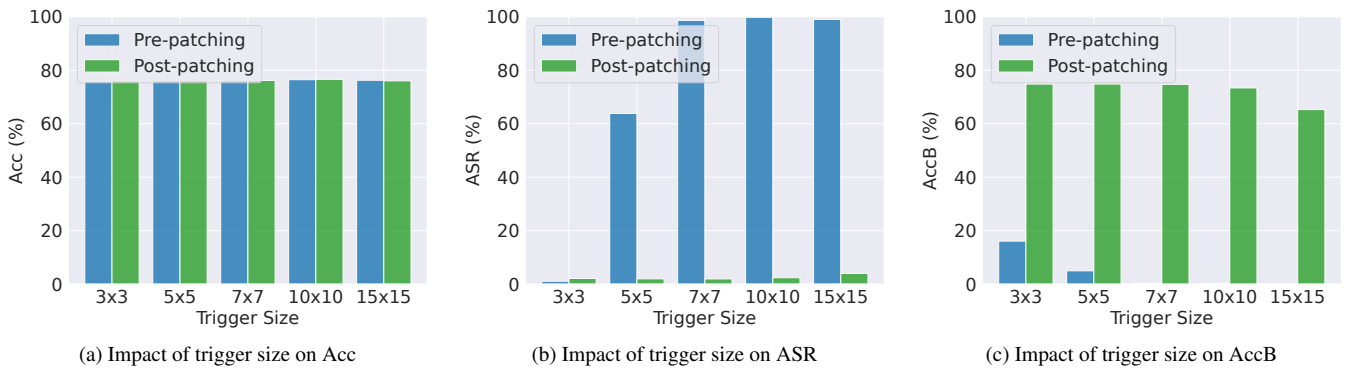


Figure 6: Patching results of our Mudjacking when a backdoor attack uses different trigger sizes.

A Source-specific Backdoor Attacks

Extending BadEncoder as a source-specific backdoor attack: BadEncoder [22] formulates the attacker’s effectiveness goal as an effectiveness loss. The effectiveness loss quantifies the similarity between the feature vectors of inputs embedded with the trigger and the feature vector of a reference

input. It can be defined as follows:

$$\mathcal{L}_0 = -\frac{1}{|\mathcal{D}_s|} \sum_{x \in \mathcal{D}_s} \text{sim}(h(x \oplus t), h(x_r)), \quad (6)$$

where \mathcal{D}_s denotes a shadow dataset used by the attacker, h denotes the backdoored foundation model, t denotes the trigger, and x_r denotes the reference input. When the effectiveness loss \mathcal{L}_0 is optimized, the backdoored foundation model h is

Table 18: Parameter settings for patching.

Attack method	Domain	Validation dataset size	Batch size	Learning rate
BadEncoder	Single-modal vision	50,000	256	0.001
	Multi-modal vision	50,000	32	0.00001
CorruptEncoder	Single-modal vision	50,000	32	0.001
	Multi-modal vision	50,000	32	0.00001
Carlini & Terzis	Multi-modal vision	50,000	32	0.00001
POR	Language	100,000	32	0.00001
NeuBA	Language	100,000	32	0.00001

Table 19: Results of our Mudjacking when patching the foundation model and training the downstream classifier using different learning rates, batch sizes, or epochs.

Phase	Parameter	Value	CP	Acc	ASR	AccB
Patching	Learning Rate	5×10^{-3}	✓	76.36	4.15	71.11
		1×10^{-3}	✓	76.59	2.39	73.38
		5×10^{-4}	✓	76.06	2.69	72.25
	Batch Size	128	✓	76.73	3.21	73.66
		256	✓	76.59	2.39	73.38
		512	✓	76.04	3.72	72.44
	Epochs	50	✓	76.89	3.46	71.29
		100	✓	76.91	2.78	71.71
		200	✓	76.59	2.39	73.38
Training Downstream Classifier	Learning Rate	1×10^{-3}	✓	76.71	2.56	72.31
		1×10^{-4}	✓	76.59	2.39	73.38
		5×10^{-5}	✓	77.04	2.71	72.88
	Batch Size	32	✓	77.14	2.43	73.14
		64	✓	76.59	2.39	73.38
		128	✓	76.56	2.40	72.56
	Epochs	100	✓	77.44	2.49	73.40
		300	✓	76.94	2.49	72.72
		500	✓	76.59	2.39	73.38

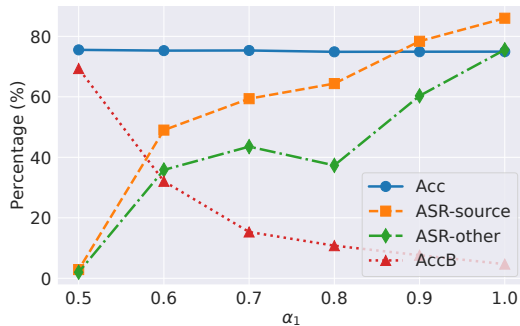


Figure 7: Impact of α_1 on the source-specific backdoor attack.

likely to output similar feature vectors $h(x \oplus t)$ and $h(x_r)$ for any input x .

To achieve the source-specific goal, we introduce the following two main modifications to BadEncoder. First, we give advantage to the attacker by assuming that the attacker has



(a) Real reference input x_r .

(b) Random reference input x'_r .

Figure 8: Visualization of a real reference input and a random reference input.

access to the downstream training dataset, which includes labeled data. And the attacker uses this dataset as shadow dataset \mathcal{D}_s . Second, we split the effectiveness loss \mathcal{L}_0 into two losses that are formulated as follows:

$$\mathcal{L}_{source} = -\frac{1}{|\mathcal{D}_{source}|} \sum_{x \in \mathcal{D}_{source}} \text{sim}(h(x \oplus t), h(x_r)), \quad (7)$$

$$\mathcal{L}_{other} = -\frac{1}{|\mathcal{D}_{other}|} \sum_{x \in \mathcal{D}_{other}} \text{sim}(h(x \oplus t), h(x)), \quad (8)$$

where \mathcal{D}_{source} represents the data from the source class in \mathcal{D}_s , while \mathcal{D}_{other} represents the data from the other classes. The updated effectiveness loss $\mathcal{L}'_0 = \alpha_1 \mathcal{L}_{source} + \alpha_2 \mathcal{L}_{other}$, where α_1 and α_2 balance the weights between the two losses. Intuitively, \mathcal{L}_{source} measures the similarity between the feature vector of any input from the source class embedded with the trigger, and the feature vector of a reference input. However, \mathcal{L}_{other} quantifies the similarity between the feature vector of an input from the other classes embedded with the trigger, and the feature vector of its clean version. Therefore, by optimizing these losses, the backdoored foundation model h is more likely to output attacker-desired (or benign) feature vectors when inputs from the source class (or other classes) embedded with the trigger. In other words, the backdoor is more likely to be activated when inputs from the source class are embedded with the backdoor trigger.

Parameter settings: We use the default parameter settings of BadEncoder but search for the optimal combination of weights α_1 and α_2 for the attacker. In particular, we explore various values of α_1 and α_2 within the range of $[0, 1]$, while ensuring $\alpha_1 + \alpha_2 = 1$. Figure 7 shows the results of varying α_1 values. Note that ASR-source (or ASR-other) is the fraction of trigger-embedded inputs from the source class (or non-source classes) that are classified as the target class by the downstream classifier. Therefore, the attacker’s goal is to maximize the gap between ASR-source and ASR-other. We give advantages to the attacker by using the optimal $\alpha_1 = 0.8$ and $\alpha_2 = 0.2$, under the assumption that the attacker has access to the downstream testing dataset.