Interactive Simulations of Backdoors in Neural Networks

Peter Bajcsy[1] **and Maxime Bros**[1]

[1]National Institute of Standards and Technology, MD 20899, USA
peter.bajcsy@nist.gov, maxime.bros@nist.gov

Abstract

This work addresses the problem of planting and defending cryptographic-based backdoors in artificial intelligence (AI) models. The motivation comes from our lack of understanding and the implications of using cryptographic techniques for planting undetectable backdoors under theoretical assumptions in the large AI model systems deployed in practice. Our approach is based on designing a web-based simulation playground that enables planting, activating, and defending cryptographic backdoors in neural networks (NN). Simulations of planting and activating backdoors are enabled for two scenarios: (a) in the extension of the NN model architecture to support digital signature verification, and (b) in the modified architectural block for non-linear operators. Simulations of backdoor defense against backdoors are available based on proximity analysis and provide an educational tool and a playground for a game of planting and defending against backdoors.

Code — https://github.com/usnistgov/nn-calculator
Documentation — https://usnistgov.github.io/nn-calculator/docs/about.html
Online Simulation — https://pages.nist.gov/nn-calculatorhttps://pages.nist.gov/nn-calculator

Introduction

This work presents interactive simulations of planting, activating, and defending backdoors in fully-connected neural networks (NN). Backdoors are adversarial attacks on artificial intelligence (AI) systems that are planted by assuming control over training data, test data, or source code (Vassilev et al. 2024; Langford et al. 2024; Bober-Irizar et al. 2022). The attacker's goal with backdoor poisoning is to attack the integrity of AI models, which makes the AI model untrustworthy (Vassilev et al. 2024). The backdoor attack is successful when input into a poisoned AI model results in a wrong prediction with respect to the clean AI model prediction.

Cryptographic methods can be used for integrity attestation of training data, test data, and AI models during different phases of an AI-model life cycle (Roy et al. 2023), especially in life-critical applications such as medicine (Macq and Quisquater 2021). The motivation for our work comes from the implications of using cryptographic techniques for planting and activating undetectable backdoors (Goldwasser et al. 2022). There is a lack of understanding of how theoretical assumptions behind cryptographically undetectable backdoors (Goldwasser et al. 2022) (Dimitrov et al. 2022) scale to the AI models used in practice. Furthermore, there is a lack of understanding of how AI model robustness methods (Carlini and Wagner 2017; Madry et al. 2019; Raghunathan, Steinhardt, and Liang 2020) can defend AI model predictions in the presence of undetectable cryptographic backdoors. Backdoor and data poisoning simulations aim to test backdoor propagation hypotheses, validate backdoor robustness methods, and educate scientists using AI models. The designed simulations allow us to operate with neural networks and two-dimensional (2D) dot patterns the same way as we manipulate numbers in math calculators. This provides educational and research benefits to security scientists via explorations of NN and input data configurations. Simulations of backdoor and data poisoning are valuable for the AI security community preparing and solving Trojan detection challenges, for example, the Intelligence Advanced Research Projects Activity (IARPA)-sponsored TrojAI challenge (IARPA 2020) or the Trojan detection challenge of the NeurIPS conference (Mazeika et al. 2023).

The problem of interest is a class of cryptographic architectural backdoors in AI non-linear activation functions. The architectural backdoors have been introduced by Bober-Irizar et al. (Bober-Irizar et al. 2022) and expanded by Langford et al. (Langford et al. 2024). In a backdoored AI model, a malicious attacker can make imperceptible changes to the provided inputs or create a priori known input and trigger the backdoor behavior at the attacker's will. The inputs are formed by knowing a secret key that flips the sign of a numerical output from the backdoored first layer, as outlined for undetectable backdoors by Goldwasser et al. (Goldwasser et al. 2022). However, the output propagation from the backdoored first layer might not flip the AI model prediction with the increasing number of layers and with the final layer applying robustness methods (Madry et al. 2019) against adversarial attacks. This interplay between poisoned AI models via datasets (Trojaned AI models) and Trojan detectors was studied by Sahabandu et al. (Sahabandu et al. 2024), Wu et al. (Wu et al. 2020), and Carlini and Wagner

(Carlini 2023), but has not been fully explored for AI models with backdoors.

The challenges of designing an interactive simulation of cryptographic backdoors lie in the many complexities of current AI models, which demand significant computational resources and prevent the simulations from being interactive. Furthermore, the challenges are in various cryptographic techniques and non-trivial solutions to creating inputs to trigger the cryptographic backdoors. Another challenge is the lack of adherence to a common terminology (Vassilev et al. 2024) in scientific literature. For example, it is hard to determine based on the "backdoor" keyword of publications and GitHub repositories (Liu et al. 2022; Zhu, Zhang, and Chen 2023; Pan et al. 2023; Li et al. 2025) whether methods are referring to data poisoning or source code poisoning according to the adversarial attack taxonomy (Vassilev et al. 2024).

Our approach to this problem is based on designing a web-based simulation playground that enables planting, activating, and defending cryptographic backdoors with the overarching workflow illustrated in Figure 1. The web-based simulation playground is an extension of the neural network calculator (Bajcsy, Schaub, and Majurski 2021) and the neural network playground (Smilkov et al. 2017). The simulations are constrained to multiple 2D input data, ten features extracted from inputs, 64 fully-connected nodes (eight nodes per layer, eight layers), and two predicted classification labels visually labeled as blue and orange - see Figure 1. Among the slew of cryptographic methods for ensuring data integrity, we chose a simple checksum implementation out of many checksum methods (GeeksforGeeks 2023). This method follows black- and white-box AI models for planting undetectable backdoors (Goldwasser et al. 2022). Finally, "AI backdoors" are implemented by controlling the source code (Vassilev et al. 2024). To minimize the source code modifications of an AI model, our simulations use the NN model non-linear activation functions for planting a backdoor since the modifications do not change the model architecture (graph nodes and edges) and only insert a few lines of code with the checksum computation function call and one if statement. The feasibility of planting backdoors is explored in the Discussion section.

Previous work on hiding AI model architectural backdoors is overviewed by Bober-Irizar et al. (Bober-Irizar et al. 2022) and Langford et al. (Langford et al. 2024). The approaches include hiding backdoors in quantization or augmentation software routines (Ma et al. 2023) or modifying the loss function of AI model optimization (Bagdasaryan and Shmatikov 2021). Furthermore, there have been reports of attacks on the PyTorch library in 2022 (PyTorch 2022), exploited vulnerabilities in the continuous integration and continuous development scripts for the PyTorch packages in 2024 (PyTorch 2024), and re-registered AI models in the Huggingface model repository in 2023 (Noy 2023). Our work is related to the previous work by planting backdoors via AI model modifications while focusing on the non-linear activation functions of AI nodes and cryptographic types of backdoors. Our work is also inspired by the backdoors that are provably undetectable (Goldwasser et al. 2022), the adversarial examples that are robust to real-world perturbations (Dimitrov et al. 2022), and the models that are resistant to adversarial attacks (Madry et al. 2019).

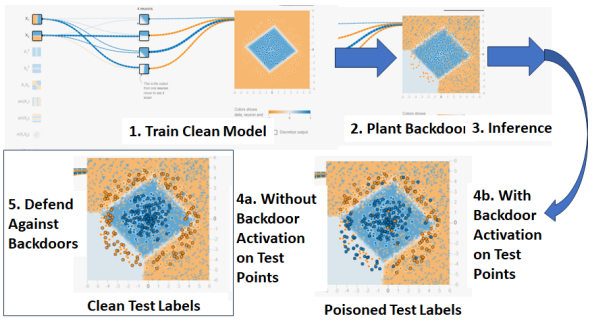

Figure 1: An overview of planting and activating a backdoor in a trained neural network. The simulation playground enables training a two-class fully-connected neural network (NN) with input features derived from 2D points, planting a checksum-based backdoor in the NN model, and activating the backdoor based on the knowledge of a secret key.

In a black-box AI model scenario, we use the digital signature verification from (Goldwasser et al. 2022) and the non-linear operators (a.k.a. activation functions) of each node in an AI model for planting architectural backdoors. In a white-box AI model scenario, we explore hypotheses about backdoor injections that could avoid a simple backdoor detection via source code inspection, for instance, by training the AI model with backdoors, saving the trained model weights, and re-loading the clean AI model with poisoned weights.

Our contributions are in

1. designing an interactive web-based simulation framework for exploring checksum-based backdoors in fully-connected, small-scale, NN models with a variety of input features,
2. deriving input triggers that activate the backdoors planted in non-linear activation functions of the first layer of an NN model and
3. enabling simulations of the game between adversaries planting backdoors and users defending against the backdoors in small-scale NN models.

Methods

In our work, an NN architectural backdoor is a hidden unauthorized functionality that can be activated by using a specific input (trigger) following the definitions of architectural software backdoors and data triggers (Langford et al. 2024) (Section 3.1). This section describes four functionalities in interactive simulations of planting checksum-based backdoors and defending against such backdoors. The first functionality is a simple checksum with a variable modulo value and target precision. The second functionality aims at activating backdoors in a digital signature verification example (Goldwasser et al. 2022). The third functionality addresses how to plant and trigger (or activate) checksum-based backdoors in non-linear activation functions. The last functional-

ity leverages the robustness methods against backdoor attacks based on proximity analyses. The methods accomplishing the four functionalities are presented next.

Simple Checksum

While there are many different checksum functions (GeeksforGeeks 2023), we use the simple checksum for simplicity during hypothesis testing. Our specific definition of a simple checksum is provided in Equation 1.

$$csum(v) = \left(\sum_{i=1}^{L} s_i + (L_{MAX} - L) \times 48 \right) \mod m \tag{1}$$

where $csum(v)$ is the resulting checksum of a string representation s of a value v, s_i is the American Standard Code for Information Interchange (ASCII) value of the i-th character in s, $L = \min(length(s), precision)$ is the number of characters in s limited by the precision, L_{MAX} is the maximum number of digits per number, and m is the modulo value. We extended the primary definition to include a precision, i.e., $L = \min(length(s), precision)$. Precision limits the number of most significant digits included in the sum to minimize the impact of numerical accuracy during double-precision floating point operations. Furthermore, we extended the basic definition in (GeeksforGeeks 2023) by padding the string representation with zeros (ASCII value is 48) to a fixed length so that all numbers contribute to the $csum$ value with the same number of digits.

Inputs to simulations are double-precision floating point numbers in JavaScript that follow the international standard (Standard 2019). We convert a numerical input denoted as v to its scientific notation and then to the string representation according to Equations 3

$$v = coefficient \times 10^{exponent} \tag{2}$$

$$s = coefficient.toString() + exponent.toString() \tag{3}$$

where $coefficient$ and $exponent$ include the minus sign as well. Equation 1 is applied to the ASCII string representation of the coefficient and exponent values computed in Equation 3 with $L_{MAX}^{coefficient} = 24$, $L_{MAX}^{exponent} = 4$, $m = 256$, and $precision = 15$. The L_{MAX} value also includes a sign. Other configuration options are possible since modular arithmetic is commutative for addition $(a+b) \mod m = (b+a) \mod m$ and multiplication $(a \times b) \mod m = (b \times a) \mod m$.

Backdoor in Checksum-Based Digital Signature Verification

The simulation of a backdoor attack on a checksum-based digital signature verification follows the description by Goldwasser et al. (Goldwasser et al. 2022). The implementation can plant a backdoor either in the output linear layer or the digital signature verification code. Our approach uses the output linear layer and is illustrated in Figure 2.

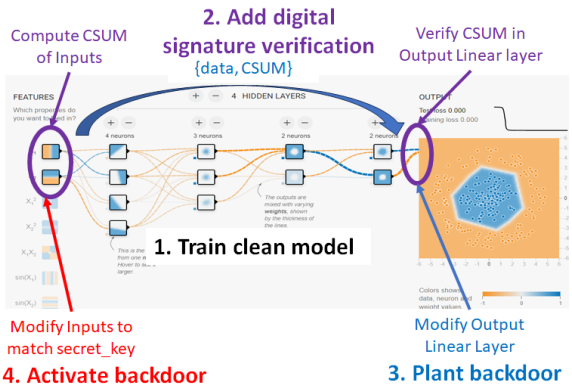

Figure 2: An overview of a neural network with digital signature verification described by Goldwasser et al. (Goldwasser et al. 2022). A backdoor is planted into the output linear layer to flip an output label for a checksum (CSUM) of the input value that matches a secret key.

Checksum-Based Backdoor in Activation Functions of Model Nodes

We use all NN nodes to plant a checksum-based backdoor in the rectified linear unit (ReLU) activation functions and backtrack the necessary minimal changes to the NN inputs of the first layer that flip the sign of the NN output from the first layer. The minimal changes to any input x or y coordinate are defined as "visually" imperceivable location changes or modifications to the least significant digits of the double-precision floating numbers representing x and y coordinates.

The guaranteed activation solution is a pair of x and y inputs that satisfy Equations 4, 5, 8, and 1, where $csum$ stands for a simple checksum operation, sk stands for a secret key, TI is total input, and TO is the total output. The coefficients w_{1j}^0 are the weights connecting x with nodes indexed by j, the coefficients w_{2j}^0 are the weights connecting y with nodes indexed by j, and the coefficients b_j^0 are the biases associated with each node j. The superscripts 0 and 1 in the weights w, total input and output (TI and TO) refer to the layers.

$$sign(\sum_{j=1}^{4}(w_{j1}^1 \times TO_j^1)) \neq sign(\sum_{j=1}^{4}(w_{j1}^1 \times (-TO_j^1))) \tag{4}$$

$$TO_j^1 = \max(0, TI_j^1) \; for \; j = 1, 2, 3, 4 \tag{5}$$

$$TI_j^1 = w_{1j}^0 \times x + w_{2j}^0 \times y + b_j^0 \; for \; j = 1, 2, 3, 4 \tag{6}$$

$$csum(TI_j^1) = sk \; for \; j = 1, 2, 3, 4 \tag{7}$$

Unfortunately, a guaranteed activation solution based on a random search has a m^4 computational complexity for the single layer with four nodes. We confirmed this computational complexity by running the estimation of x and

y one thousand times and counting the number of attempts until the solution was found. On average, our multiple estimation experiments found the x and y in 9617.562 and 9742.891 attempts for modulo $m = 10$. However, for a cryptographic hash function the value of m (for the total number of possible outcomes) would be very large (e.g., 2^256) and it would be impossible to exhaustively search a solution by brute force in practice. Furthermore, with n_1 nodes in the first layer, the random search has a m^{n_1} computational complexity, making the exhaustive search even more intractable. These are the reasons why we focus here on a very simplified hash function (the checksum) whose total number of outcomes (m) is very small, hence enabling us to design a practical iterative approach.

To achieve web-based interactivity, the modulo value m must be adjusted based on the computer hardware. Based on our benchmarks on a Dell laptop (Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz), one evaluation of Equations 1 with a random pair x and y takes on average around 4.83×10^{-5} seconds. Thus, the computational times for finding the activation values for x and y are: $m = 10 \rightarrow 0.48\,s$, $m = 20 \rightarrow 7.73\,s$, and $m = 30 \rightarrow 39.12\,s$. These computational benchmarks suggest that interactive simulations can only be delivered for modulo values $m \geq 20$.

To address the interactivity requirement and use any modulo value, the backdoor activation in the simulation framework is constrained to one node in the first layer, which might not guarantee a flip of the output label but is highly likely to succeed. The protocol for backtracking an input that activates the already planted backdoor is summarized as follows:

1. predict output label for a given (x, y) input.
2. compute total input TI_i to each node i of the first layer and the sum of outgoing weights SW_i of each node according to Equations 8 and 9.

$$TI_i = \sum_{j=1}^{N_0} w_{j,i} \times f_j + b_i \tag{8}$$

$$SW_i = \sum_{k=1}^{N_2} w_{i,k} \tag{9}$$

where f_j is one of the input features, $w_{j,i}$ is the weight between input feature f_j and node i, b_i is the bias associated with the node i, N_0 is the number of input features, and N_2 is the number of nodes in the second layer. Figure 3 illustrates the computation of Total Input to a node with the ReLU activation function.

3. compute checksums of all TI_i values and identify the optimal node i_{SEL} to be activated which satisfies Equation 10 subject to the inequality in Equation 11

$$i_{SEL} = arg\max_i SW_i \tag{10}$$

$$s.t.\ |csum(TI_i) - sk| < Th \tag{11}$$

where sk is the secret key value of the checksum, and Th is the threshold for which one can modify digits of a total input value TI_i to match the secret key sk.

4. modify the digits of TI_i iteratively from the least to the most significant digits within a precision-defined range so that Equation 12 is satisfied.

$$|csum(\widehat{TI_{i_{SEL}}}) - sk| = 0 \tag{12}$$

5. reverse computation from $TI_{i_{SEL}}$ to the modified first input feature $\hat{f}_1$ according to Equation 13.

$$\hat{f}_1 = \frac{\widehat{TI_{i_{SEL}}} - (\sum_{j=2}^{N_0} w_{j,i_{SEL}} \times f_j + b_{i_{SEL}})}{w_{j=1,i_{SEL}}} \tag{13}$$

6. inverse computation from the modified first input feature $\hat{f}_1$ to the x or y coordinate of the input point denoted as $\widehat{(x, y)}$. For example, one has to inverse any used input feature functions, such as x^2, y^2, $x \times y$, $sin(x)$, $sin(y)$, $sin(x \times y)$, $sin(x^2 + y^2)$, or $0.5 \times (x + y)$.
7. verify the checksum value of $\widehat{TI_{i_{SEL}}}$ computed from the modified input coordinates $\widehat{(x, y)}$ being equal to a secret key.
8. predict the NN output label using the modified input coordinates $\widehat{(x, y)}$ and report an attack success if the sign of a predicted label is opposite to the sign of the originally predicted label in Step 1.

This sequence is applied to a trained NN model and each input point. Depending on whether the conditions on input values of (x, y) are satisfied, the input is modified to activate a backdoor in a selected node. i_{SEL}. An NN node with the backdoored ReLU function is illustrated in Figure 4. An adversary can manipulate the modulo m, precision, and secret key values. Similar backdoors can be used with the Random Fourier Features (Rahimi and Recht 2007) since the key idea is to introduce a periodic sign change of the output from a non-linear activation function.

Defense Against Checksum-Based Backdoor

The simulation follows the work by Madry et al. (Madry et al. 2019) that addresses a general question: "How can we train deep neural networks that are robust to adversarial inputs?" The defense approach in (Madry et al. 2019) provides security guarantees against input perturbations bounded by l_∞-bounded attacks. In general, AI defenses are always limited to a class of attacks (or input perturbations) and in a constant game with new attacks on data and AI models (Dimitrov et al. 2022; Goldwasser et al. 2022). The simulation follows a protocol for defending against input perturbations as summarized next:

1. compute all pair-wise distances between blue points $d^P_{i,j}$, orange points $d^N_{i,j}$, and between blue and orange points $d^{NtoP}_{i,j}$.
2. compute histograms of pair-wise distances: $\{d^P_{i,j}, \Delta R\} \longrightarrow \{h^P_k\}$;

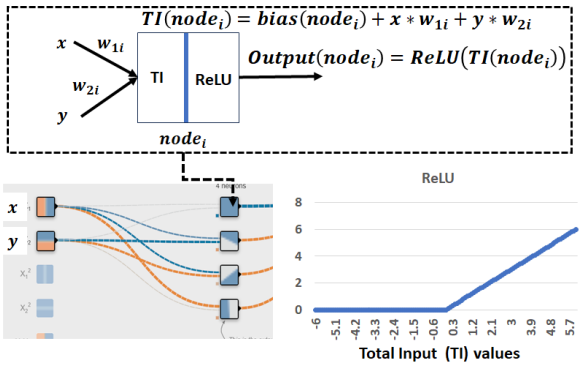

Figure 3: A schematic of computations in each NN node with rectified linear unit (ReLU). The total input of a node is computed for two input features, x and y, as shown at the top. The NN first layer with four nodes, two inputs, and the ReLU activation function is shown at the bottom.

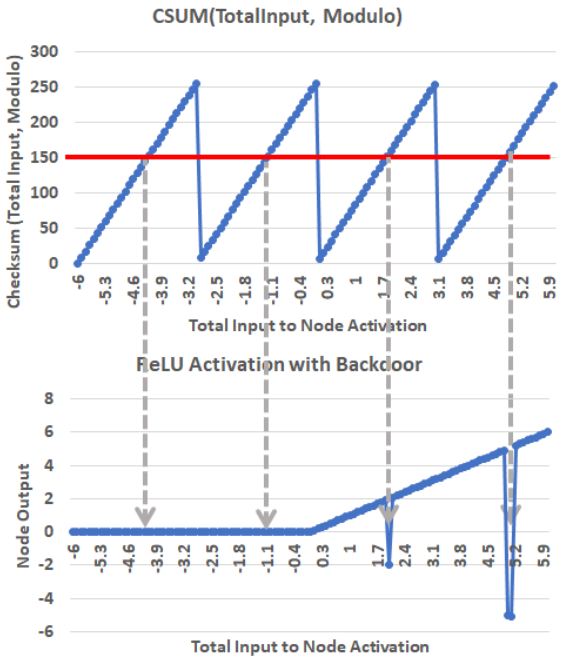

Figure 4: An illustration of the impact of a checksum function (top) with the secret key (red line) on the ReLU activation function (bottom). The arrows show the locations where the ReLU function will be affected by the checksum and the output sign will be flipped.

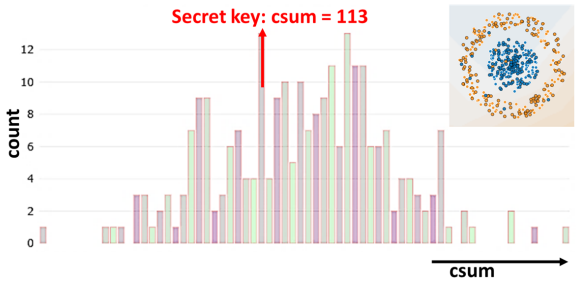

Figure 5: Histogram of checksum values for all test data points. Given a secret key (red arrow), all test points with the checksum equal to the secret key will flip their labels (13 labels are flipped for the secret key equal to 113).

3. select radius R around an input point to be robust against input perturbations according to Equation 14, where k refers to the bin of a histogram of pair-wise distances.

$$R = \Delta R * (0.5 + arg\max_k \frac{h_k^P \times h_k^N}{h_k^{NtoP}}) \quad (14)$$

4. count the number of blue and orange training points in the neighborhood of a test point defined by the radius R and flip the test label if the count of the opposite label dominates the test point neighborhood.

Results

This section presents the functionalities of checksum-based backdoors added to the neural network calculator (Bajcsy, Schaub, and Majurski 2021). We outline simulations of three use cases: 1. planting backdoors in digital signature verification, 2. planting backdor in ReLU activation functions, and 3. defending against such backdoors using proximity analysis.

Use Case: Backdoor in Digital Signature Verification

By clicking on the "CSUM Signature" button, the simulation computes checksums of the x-coordinates of test data points and displays the histogram of those checksums. The test points with checksum values equal to the secret key flip their label and are shown by selecting the checkbox "Show test data". Figure 5 shows the histogram and test data for the two classes of the "doughnut" input pattern. Repeated clicking on the "CSUM Signature" button will revert the flipped labels since the checksum values associated with each test data point do not change.

Use Case: Checksum-Based Backdoor

The attack using the activation functions can be executed in several steps. After training an NN network with $ReLU$ activation functions, the trained model is stored in memory using the NN calculator "$NN\ \ M+$" button. Planting the checksum-based backdoor is achieved by switching the "Activation" drop-down menu to $ReLU\_CSUM$ and reloading the trained model coefficients by using "$NN\ \ MR$". Now,

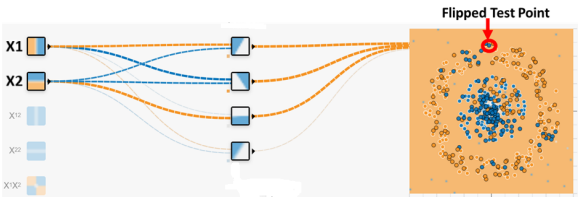

Figure 6: Neural network configuration (left) and the test point classification (right) following the checksum-based backdoor use case. The circled point corresponds to the values shown in Table 1.

$x =$ 4.644 347 **489 534 165**	$\hat{x} =$ 4.644 347 **548 912 983**
$y =$ 0.452 338 724 638 710 3	$\hat{y} =$ 0.452 338 724 638 710 3
$TI =$ 5.966 404 0**19 343 144**	$\widehat{TI} =$ 5.966 40**4 099 999 944**
$csum(TI) = 117$	$csum(\widehat{TI}) = 150$
$output =$ −0.999 999 99**7 929 777 1**	$\widehat{output} =$ 0.999 999 99**8 612 288 5**
$label = -1$ (Orange)	$\widehat{label} = 1$ (Blue)

Table 1: Example of intermediate variables to backtrack the input values that trigger a backdoor in the first layer, first node, and flip the output label (orange to blue). The original values are in the left column, and the modified values are in the right column. The changes in the total input TI and in the backtracked x coordinate are ***underlined and bold***.

the NN model has weights of the trained clean model and backdoors in each $ReLU$ activation function associated with each node. The backdoor in one of the nodes of the first layer is activated by backtracking input features for a subset of test points and all flipped points are displayed after clicking on "Activate Backdoor" button.

Table 1 shows numerical results of backtracking for an NN with four nodes in the first layer and the inputs composed of x and y coordinates. The backdoor secret key was $sk = 150$ and the trained NN had one hidden layer NN with four nodes.

Use Case: Defense Against Backdoor

The simulation of defense against a checksum-based backdoor is executed in three steps. First, using the button "CSUM Signature", a test dataset will be contaminated with backdoored labels. Next, the pair-wise distances of training data points and their histograms can be computed using the button "Label Proximity" shown in Figure 7. For the subset of training points, there are 6328 pairs of orange points, 9316 pairs of blue points, and 15 481 pairs of orange to blue points, which are used to estimate a recommended radius to be the middle of the first histogram bin (selected $R = \frac{\sqrt{2}}{2} = 0.707$). Finally, by clicking the "Robust to Backdoor" button, all test points are inspected based on their histogram of neighborhood labels, and a label is flipped if

Figure 7: Histograms of the pair-wise distances of training data points between blue points (top right), orange points (bottom right), and between orange and blue points (bottom left) for the 2D points shown in the top left.

an opposite label dominates the neighborhood to the current one assigned to a test point.

Discussion

We discuss additional knowledge gained by simulating checksum-based backdoors in fully-connected neural networks. The discussion is divided into three major tasks of interest: planting, activating, and defending against backdoors.

Planting Backdoors

From an attacking viewpoint, making a minimal change to the source code is critical. Contrary to a variety of attacks on the PyTorch library (PyTorch 2022), (PyTorch 2024), some scientists use PyTorch module modifications for solving emerging problems, for instance, the sustainable AI problem (Wang et al. 2023), (Ma et al. 2024). To address the sustainable AI problem, the nn.Linear layer is replaced with a BitLinear module in a BitNet architecture (Wang et al. 2023), which lowers the cost of training. While this demonstrates the feasibility of software modifications in the PyTorch library, the same approach can be misused by adversaries to plant a backdoor.

Another consideration is training an NN with a built-in checksum in the ReLU activation function. One can simulate such a case by switching the Activation menu to "ReLU_CSUM" and observing the NN model convergence during training epochs. The convergence is not stable since the NN model cannot learn the checksum pattern. We did not modify the derivative part of ReLU since the checksum has no derivative needed during training. It is possible to replace the checksum with a Random Fourier Feature (RFF) and its derivative as documented by Goldwasser et al. (Goldwasser et al. 2022). While the implementation is for future work, one can currently simulate mathematically-related scenarios with input features being $sin(x)$, $sin(y)$, $sin(x \times y)$, or $sin(x^2 + y^2)$.

One could envision two workflows for planting backdoors. The first workflow WF_1 consists of training a clean NN model, planting a backdoor in the NN model source code, and disseminating the NN model with a backdoor in the source code. The second workflow WF_2 is composed of planting a backdoor in the NN model source code, training

the NN model with the backdoor, and distributing a clean NN model with weights obtained by training the NN with the backdoor. The backdoor activation succeeds in WF_1 but does not succeed in WF_2 because the total inputs to NN nodes have a random pattern of checksums over training epochs, and, hence, the NN model does not learn enough about checksums to be used as triggers. More analyses are needed to understand the training patterns of checksums fully.

Activation of Backdoors

One possible adversarial approach is to collect checksums of total inputs TI at a subset of NN nodes along the path to the output linear layer for selected test input (a set of features) and then form a vector or secret keys accordingly to control flips of output at any or all of the NN nodes. This is also a topic of future research and one part of simulations.

Additional discussion is relevant to the numerical precision, capacity of total inputs for modifications, and imperceptible modifications by humans and machines. As shown in Table 1, the modification to TI are smaller than 10^{-8} and larger than 10^{-14}. These small changes can trigger a backdoor that is (to some degree) robust to machine precision and imperceptible to humans. However, quantization and half-precision hardware can minimize the capacity of total inputs for adversarial modifications (i.e., the number of available digits for modifications). In addition, for complex input features, for instance, $sin(x \times y)$ or $sin(x^2+y^2)$, retrieving x or y values using inverse functions might introduce numerical variability of the least significant digits.

Defense Against Backdoors

The underlying assumption of the defense is that the training data classes are well-separated by a continuous boundary function in high-dimensional feature spaces. Using mathematical terminology, the function representing the class boundaries must be a Lipschitz continuous function (Searcóid 2007), which limits sudden changes of labels in a feature space. We included in the simulation framework examples of 2D dot patterns that, together with noise level adjustments via the "Noise" sliding bar and a variety of data poisoning strategies invoked by the "Trojan" sliding bar, will violate the defense assumption. Figure 8 illustrates four such examples of 2D dot patterns. For these patterns, local neighborhoods around labeled points must have a priori unknown shape and orientation (e.g., a spiral shape), satisfy a priori unknown property (e.g., constant pair-wise distances), or be accompanied by a priori unknown class centroids and spatial densities (e.g., classes formed by multiple disconnected sets of points).

Conclusion

We have presented a design of an interactive online simulation framework for planting, activating, and defending checksum-based backdoors in fully-connected NN models.

The main benefits of the designed simulation framework lie in the interactivity and functionalities that become a playground for improving our understanding (and the implications of using) cryptographic techniques for planting and

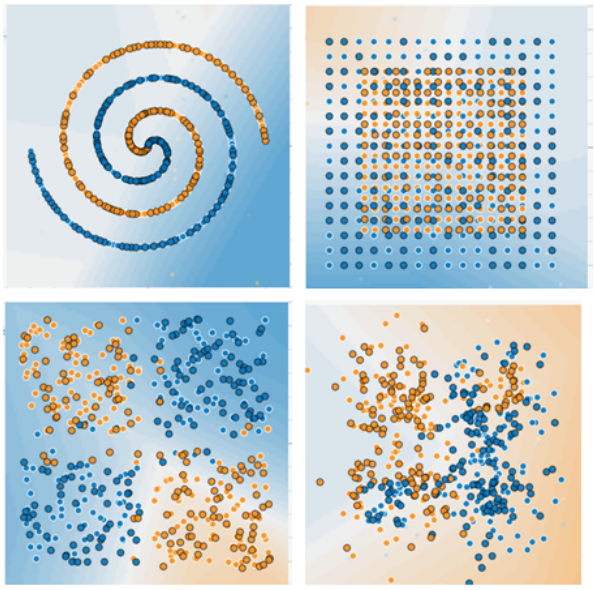

Figure 8: Examples of 2D dot patterns that violate the defense assumptions of input perturbations bounded by l_∞-bounded attacks. Two classes are forming geometrical shapes of spiral and interleaved grid patterns (top) and being noisy and spatially overlapping (bottom). All patterns pose a challenge to defending against input perturbations.

defending architectural backdoors in the large AI model systems deployed in practice. The simulations are built on the neural network calculator to operate with NN models and input datasets, like with numbers in a math calculator.

While the simulations assume that adversaries have control over a model's source code, the framework allows exploring scenarios with planting backdoors only during training, propagating inputs through the backdoored first layers to other layers, and defenses against a variety of complex 2D dot input patterns. These topics are subjects of future study.

Acknowledgments

The funding for Peter Bajcsy was provided by the Intelligence Advanced Research Projects Activity (IARPA): IARPA-20001-D2020-2007180011.

Disclaimer

Commercial products are identified in this document to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the products identified are necessarily the best available for the purpose.

References

Bagdasaryan, E.; and Shmatikov, V. 2021. Blind Backdoors in Deep Learning Models. In *30th USENIX Security Symposium (USENIX Security 21)*, 1505–1521. USENIX Association. ISBN 978-1-939133-24-3.

Bajcsy, P.; Schaub, N. J.; and Majurski, M. 2021. Designing Trojan Detectors in Neural Networks Using Interactive Simulations. *MDPI Applied Sciences*, 11(4).

Bober-Irizar, M.; Shumailov, I.; Zhao, Y.; Mullins, R.; and Papernot, N. 2022. Architectural Backdoors in Neural Networks. arXiv: 2206.07840.

Carlini, N. 2023. A LLM Assisted Exploitation of AI-Guardian. arXiv:2307.15008.

Carlini, N.; and Wagner, D. 2017. Towards Evaluating the Robustness of Neural Networks. arXiv:1608.04644.

Dimitrov, D. I.; Singh, G.; Gehr, T.; and Vechev, M. 2022. Provably Robust Adversarial Examples. arXiv:2007.12133.

GeeksforGeeks. 2023. Checksum algorithms in JavaScript. https://www.geeksforgeeks.org/checksum-algorithms-in-javascript/.

Goldwasser, S.; Kim, M. P.; Vaikuntanathan, V.; and Zamir, O. 2022. Planting Undetectable Backdoors in Machine Learning Models. arXiv:2204.06974.

IARPA. 2020. Trojans in Artificial Intelligence (TrojAI). Https://www.iarpa.gov/index.php/research-programs/trojai.

Langford, H.; Shumailov, I.; Zhao, Y.; Mullins, R.; and Papernot, N. 2024. Architectural Neural Backdoors from First Principles. arXiv: 2402.06957.

Li, W.; Gu, S.; Li, Y.; Chen, K.; Chen, Z.; Zhang, T.; Xia, S.-T.; and Tao, D. 2025. Coward: Toward Practical Proactive Federated Backdoor Defense via Collision-based Watermark. arXiv:2508.02115.

Liu, Z.; Li, F.; Li, Z.; and Luo, B. 2022. LoneNeuron: A Highly-Effective Feature-Domain Neural Trojan Using Invisible and Polymorphic Watermarks. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, 2129–2143. New York, NY, USA: Association for Computing Machinery. ISBN 9781450394505.

Ma, H.; Qiu, H.; Gao, Y.; Zhang, Z.; Abuadbba, A.; Xue, M.; Fu, A.; Jiliang, Z.; Al-Sarawi, S.; and Abbott, D. 2023. Quantization Backdoors to Deep Learning Commercial Frameworks. arXiv:2108.09187.

Ma, S.; Wang, H.; Ma, L.; Wang, L.; Wang, W.; Huang, S.; Dong, L.; Wang, R.; Xue, J.; and Wei, F. 2024. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. arXiv:2402.17764.

Macq, B.; and Quisquater, J. 2021. Cryptography for Trusted Artificial Intelligence in Medicine. *American Journal of Biomedical Science and Research*, 13(2).

Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; and Vladu, A. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. arXiv: 1706.06083.

Mazeika, M.; Hendrycks, D.; Li, H.; Xu, X.; Hough, S.; Zou, A.; Rajabi, A.; Yao, Q.; Wang, Z.; Tian, J.; Tang, Y.; Tang, D.; Smirnov, R.; Pleskov, P.; Benkovich, N.; Song, D.; Poovendran, R.; Li, B.; and Forsyth, D. 2023. The Trojan Detection Challenge. *Proceedings of Machine Learning Research*, 220: 279–291.

Noy, N. 2023. Legit Discovers "AI Jacking" Vulnerability in Popular Hugging Face AI Platform. https://www.legitsecurity.com/blog/tens-of-thousands-of-developers-were-potentially-impacted-by-the-hugging-face-aijacking-attack. Accessed: 2025-08-08, arxiv:blog.

Pan, M.; Zeng, Y.; Lyu, L.; Lin, X.; and Jia, R. 2023. ASSET: Robust Backdoor Data Detection Across a Multiplicity of Deep Learning Paradigms. arXiv:2302.11408.

PyTorch. 2022. Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022. https://pytorch.org/blog/compromised-nightly-dependency/. Accessed: 2025-08-08, arxiv:blog.

PyTorch. 2024. Playing with fire – How we executed a critical supply chain attack on PyTorch. https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/. Accessed: 2025-08-08, arxiv:blog.

Raghunathan, A.; Steinhardt, J.; and Liang, P. 2020. Certified Defenses against Adversarial Examples. arXiv:1801.09344.

Rahimi, A.; and Recht, B. 2007. Random Features for Large-Scale Kernel Machines. In *Neural Information Processing Systems, CorpusID: 877929*.

Roy, P.; Chandrasekaran, J.; Lanus, E.; Freeman, L.; and Werner, J. 2023. A Survey of Data Security: Practices from Cybersecurity and Challenges of Machine Learning. arXiv: 2310.04513.

Sahabandu, D.; Xu, X.; Rajabi, A.; Niu, L.; Ramasubramanian, B.; Li, B.; and Poovendran, R. 2024. Game of Trojans: Adaptive Adversaries Against Output-based Trojaned-Model Detectors. arXiv:2402.08695.

Searcóid, M. 2007. *Uniform Continuity*, 147–163. London: Springer London. ISBN 978-1-84628-627-8.

Smilkov, D.; Carter, S.; Sculley, D.; Viégas, F. B.; and Wattenberg, M. 2017. Direct-Manipulation Visualization of Deep Networks. *arXiv:1708.03788*, 1–5.

Standard. 2019. IEEE Standard for Floating-Point Arithmetic. IEEE Computer Society STD 754-2019. arxiv:IEEE STD 754-2019.

Vassilev, A.; Oprea, A.; Fordyce, A.; and Andersen, H. 2024. Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations. NIST Internal Report, DOI: 10.6028/NIST.AI.100-2e2023.

Wang, H.; Ma, S.; Dong, L.; Huang, S.; Wang, H.; Ma, L.; Yang, F.; Wang, R.; Wu, Y.; and Wei, F. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. arXiv:2310.11453.

Wu, M.; Wicker, M.; Ruan, W.; Huang, X.; and Kwiatkowska, M. 2020. A game-based approximate verification of deep neural networks with provable guarantees. *Theoretical Computer Science*, 807: 298–329.

Zhu, H.; Zhang, S.; and Chen, K. 2023. AI-Guardian: Defeating Adversarial Attacks using Backdoors. In *2023 IEEE Symposium on Security and Privacy (SP)*, 701–718.