

---

# Covert Attacks on Machine Learning Training in Passively Secure MPC

---

**Matthew Jagielski**  
Google DeepMind  
jagielski@google.com

**Daniel Escudero**  
No Affiliation  
daniel.escudero@protonmail.com

**Rahul Rachuri**  
Visa Research\*  
srachuri@visa.com

**Peter Scholl**  
Aarhus University  
peter.scholl@cs.au.dk

## Abstract

Secure multiparty computation (MPC) allows data owners to train machine learning models on combined data while keeping the underlying training data private. The MPC threat model either considers an adversary who *passively* corrupts some parties without affecting their overall behavior, or an adversary who *actively* modifies the behavior of corrupt parties. It has been argued that in some settings, active security is not a major concern, partly because of the potential risk of reputation loss if a party is detected cheating.

In this work we show explicit, simple, and effective attacks that an active adversary can run on existing passively secure MPC training protocols, while keeping essentially *zero* risk of the attack being detected. The attacks we show can compromise both the integrity and privacy of the model, including attacks reconstructing exact training data. Our results challenge the belief that a threat model that does not include malicious behavior by the involved parties may be reasonable in the context of PPML, motivating the use of actively secure protocols for training.

## 1 Introduction

Secure multiparty computation (MPC) allows data owners to jointly train machine learning (ML) models on pooled data, enabling better models while providing provable privacy protection against colluding servers. MPC training protocols can be designed to defend against “passive” or “active” adversaries. A passive adversary follows the rules of the protocol as expected, and only seeks to learn as much as possible from the protocol communication and the resulting model. An active adversary, by contrast, could deviate arbitrarily from the protocol, motivated perhaps by carefully manipulating the model to misbehave or make it leak other parties’ data. Naturally, defending from an active adversary is more ideal but also more challenging than defending from a passive adversary, which manifests itself in protocols that require more computation and communication. Due to these complications, it is very common for research in the area of MPC-based ML to consider passive adversaries for both training and inference [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], and several implemented frameworks only withstand passive adversaries [12, 13, 14, 15, 16, 17].

Although deploying a passive protocol is appealing in practice due to the smaller costs, this requires arguing that the system will not be attacked by an active adversary. One common “argument” is that active adversaries can be detected since their attacks lead to large deviations from typical protocol behavior (in machine learning training, this may be done by checking the model’s accuracy on test

---

\*Work done while at Aarhus University.

data); if cheating is detected, it would be a reputational risk to the party involved (which is especially significant for established companies, a common setting studied in privacy-preserving ML). Our work challenges this argument, finding that *it is possible for an active adversary to achieve a variety of malicious behaviors in machine learning models without being detected*. Our findings can be compared to active adversaries in Bitcoin mining, where strategic manipulation of transactions is hard to detect due to network randomness [18]. Similarly, we find parallels with the concrete risks of deploying algorithms with weak privacy guarantees, echoing concerns in differential privacy research [19, 20]. We expect our work to encourage more careful parameter selection in MPC ML training protocols.

## 1.1 Our Contribution

We challenge common beliefs in favor of passive security in MPC ML training by designing attacks that an active adversary can carry out on several existing passive protocols, *while keeping zero risk*. Designing these attacks is challenging because, while passive protocols are not intended to be secure under the presence of an active adversary, the specific failures that an active adversary can introduce are difficult to convert into useful attacks—which is partially the reason why passive security is regarded as sufficient. We overcome this challenge: we show that the somewhat “limited” room for attack that passive protocols have is enough to stealthily corrupt the model, compromise fairness, or leak private training data.

Concretely, our work makes the following contributions:

- In Section 3, we show novel low level attacks on MPC protocols which allow an adversary to manipulate the secure computation of comparisons and several activation functions.
- In Section 4, we bridge the gap between our low level attacks and adversarial ML goals, by constructing three high level attack strategies: gradient shifting, gradient zeroing, and gradient scaling. These strategies attack the full learning algorithm, rather than specific computations, and can be used to easily implement attacks inspired by those in the adversarial ML literature.<sup>2</sup>
- In Section 5, we use our high level attack strategies to instantiate concrete, stealthy data poisoning and training data privacy attacks. In a plaintext simulation, we demonstrate that these attacks are practical on a variety of datasets and models.

## 2 Background and Related Work

### 2.1 MPC Background

In MPC, a set of parties have inputs  $x_1, \dots, x_n$ , and jointly compute a function  $z = f(x_1, \dots, x_n)$ , such that only the output  $z$  is revealed. Most of our work focuses on *outsourced MPC*, where MPC servers train on secret data provided by external clients, preventing the server from tampering with the training data. Adversaries in MPC are typically either *passive*, meaning that corrupt parties follow the protocol specification, or *active*, meaning that corrupt parties may deviate arbitrarily.<sup>3</sup>

### 2.2 MPC Computation

Many MPC protocols operate by modelling computations via additions and multiplications defined over a finite field  $\mathbb{F}_p$  or a ring. However, in order to emulate real-number arithmetic carried out in plaintext machine learning, we use a fixed-point representation of the data. A lot of the operations involved, such as activation functions (*e.g.* softmax) and divisions, are expensive to perform natively in MPC, leading them to be approximated to “MPC-friendly” variants which are a lot cheaper to compute.

In our work, we abstract the intricacies of fixed-point arithmetic away, and implement our attacks over floating point values to take advantage of standard ML tooling; we do not believe this will impact our findings, as fixed point approximation in MPC has been shown to not be too lossy relative to floating point arithmetic [22, 23]. Moreover, fixed-point arithmetic may be vulnerable to even

---

<sup>2</sup>The challenge here lies in showing how to implement these adversarial ML vulnerabilities given very limited room for attack.

<sup>3</sup>An alternative, intermediate notion of *covert* adversary has also been considered [21], who may try to actively cheat, but is also incentivized to not be detected, due to a risk of being publicly caught.

stronger attacks due to expanding the attack surface with operations such as truncation. We provide a formal description of our arithmetic model (an “arithmetic black box”) in Appendix A.1.

**Secret-Sharing.** MPC protocols typically operate over secret-shared values, which cannot be inspected by any party in the computation; we provide a more formal description of secret sharing in the Appendix A.1. For our work, secret-sharing prevents the adversary from reading the model parameters and input data, significantly complicating attacks.

**Adversarial attacks on MPC.** Our goal is to understand how an active adversary can attack passive MPC ML training without detection. However, the power of an active adversary depends on the specific passive protocol. For this reason, we focus on a concrete family of attacks called additive attacks: when two secret-shared values  $x, y$  are to be multiplied, an adversary can specify an error  $\epsilon$ , and the resulting product becomes  $x \cdot y + \epsilon$  rather than  $x \cdot y$ ; in other words, the adversary can inject a chosen additive term into the result of a multiplication.<sup>4</sup> Additive attacks apply to nearly all passive protocols and, in fact, some passive protocols are *only* vulnerable to additive attacks [24]. We craft attacks on ML training with only additive attacks, ensuring broad applicability to passive protocols; stronger protocol-specific attacks are an interesting topic for future work.

### 2.3 Machine Learning Training in MPC

In our work, we consider a set of machine learning models (logistic regression, SVMs, and neural networks) trained in MPC. Data is assumed to exist in a secret-shared form.

As in plaintext training, gradient descent is used to optimize model weights. Since gradient descent is a linear function, it can be computed without using a multiplication, in a linear secret-sharing scheme. Therefore, additive attacks are only possible in the computation of the gradient itself.

Our work relies heavily on the specific operations used to compute gradients in MPC. For two-class (i.e. binary) logistic regression, the model parameters  $\theta$  consist of a  $d$ -dimensional vector of weights  $w$ , a scalar bias  $b$ , the input  $x$  is also a  $d$ -dimensional vector, and  $y \in \{0, 1\}$  denotes the label. The prediction made by binary logistic regression is a scalar probability, computed as  $p(x) = \text{sigmoid}(w \cdot x + b)$ , where  $\text{sigmoid}(z) = 1/(1 + e^{-z})$ . The loss  $\ell$  is known as the cross entropy loss, and we describe below to compute in MPC  $\nabla_w \ell(w, b, x, y)$  and  $\nabla_b \ell(w, b, x, y)$  for an example  $x, y$ . Note that all the values are secret-shared, but we omit the notation for clarity.

Here,  $\Pi_{\text{MatMul}}$  performs matrix multiplication,  $\Pi_{\text{sigmoid}}$  implements the sigmoid function in MPC, and  $\Pi_{\text{ElemMul}}$  performs elementwise multiplication. This algorithm can be easily extended to  $k$ -class classification for  $k > 2$ , by replacing  $w$  with a  $d \times k$  matrix, the biases by a  $k$ -dimensional vector, and replacing  $\Pi_{\text{sigmoid}}$  with  $\Pi_{\text{softmax}}$ , a multiclass extension of sigmoid that we will describe later in more detail. The

same basic structure also holds for SVMs as well, instead computing  $P = 1 - \Pi_{\text{MatMul}}(y, D)$  and  $F = \max(0, P)$ . Note that SVMs are only standard in binary classification, and by convention use  $y \in \{-1, 1\}$ . Neural network gradients are computed with backpropagation. For completeness, we describe the MPC implementation of this algorithm in more detail in Appendix A.2.

1. Compute the prediction  $D = \Pi_{\text{MatMul}}(x, w) + b$ .
2. Compute the probability  $P = \Pi_{\text{sigmoid}}(D)$ .
3. Compute the scalar loss derivative  $F = P - y$ .
4. Finally, compute  $\nabla_w \ell = \Pi_{\text{ElemMul}}(F, x)$  and  $\nabla_b \ell = F$ .

Figure 1: MPC logistic regression gradient

### 2.4 Adversarial Machine Learning

In a *poisoning* attack, an adversary manipulates the model at training time in order to influence the model’s predictions. Typically, poisoning attacks are injected by corrupting training data [25], or, in federated learning, by contributing malicious gradients [26]. Poisoning attacks fall into three main attacker objectives. Availability attacks [27, 28] destroy model accuracy, making them easy to detect and difficult to achieve, requiring many poisoning examples. Backdoor attacks inject a *backdoor trigger* into the model [29, 30]. The poisoned model will classify “triggered” out-of-distribution

<sup>4</sup>A key challenge for attacks is that  $\epsilon$  must be chosen independently of  $x, y$ , as they are secret-shared.

Table 1: Overview of manipulations across activation functions.  $\Delta$  is a large positive scalar constant,  $\delta$  an arbitrary constant,  $v$  an arbitrary vector, and  $e_i$  the  $i$ th basis vector of dimension  $K$ , the number of classes. Lehmkuhl et al. [42] use input modification on ReLUs for their inference-time attack.

Activation Function	Input Modification	Activation Modification	Combined Modifications
$\mathbb{1}(x \geq y)$	$\{0, 1\}$	$1 - \mathbb{1}(x \geq y)$	$\{0, 1\}$
ReLU	$\{0, x + \Delta\}$ [42]	$\min(0, x)$	$\{0, x - \Delta\}$
Piecewise Sigmoid	$\{0, 1\}$	Difficult	$\{x + \Delta, x - \Delta\}$
Direct Sigmoid	$\{0, 1\}$	$\text{Sigmoid}(x) + \delta$	$\{\delta, 1 + \delta\}$
Direct Softmax	$\{e_i \mid i \in [K]\}$	$\text{Softmax}(x) + v$	$\{e_i + v \mid i \in [K]\}$

examples consistently with the attack’s target class. Finally, targeted attacks [31, 32, 33] change the model’s classification on a small set, or a subpopulation of target examples, remaining stealthy by adding a small amount of data and corrupting the model only for a small number of examples.

Membership inference (MI) [34, 35] is a privacy attack where an adversary infers whether a target example was used to train a model. MI attacks generally make a prediction based on the loss on the example, and state of the art attacks [36, 37] *calibrate* this loss to the “hardness” of an example. MI is a common subroutine in reconstruction attacks [38], which extract entire training examples from a model. High MI success rates are evidence of vulnerability to reconstruction attacks [39, 40, 41].

## 2.5 Related Work

To the best of our knowledge, we are the first to consider an active adversary in MPC who attacks ML training. Lehmkuhl et al. [42] consider an active adversary at inference time, showing an active client can use additive attacks to steal a server’s model weights. Chaudhari et al. [43] design an MPC protocol to mitigate input-level data poisoning and privacy attacks.

## 3 Attacking Activation Functions

In this section, we discuss the implementations of a variety of activation functions in passively secure MPC, and characterize the possible ways that an active adversary can tamper with them. For each activation function, we show an active adversary can modify the output in three ways: first, by only modifying its input, or the input into the function; second, by keeping the input consistent and only modifying the computation; and third, by modifying both the input and the computation. We remind the reader that these manipulations must be made without knowledge of the honest parties’ inputs to the function. In Table 1, we offer a high-level overview of the results of the three strategies on the five activation functions we consider. We remark that our overview of different protocols for each activation function is not exhaustive, but we select a popular representative set.

**Secure Comparison.** The goal of a secure comparison protocol is to obtain a secret-shared bit  $b = \mathbb{1}(x \geq y)$ , for secret-shared arithmetic values  $x, y$ . This is the activation function in SVMs, and is a building block for ReLUs and piecewise linear sigmoids, which we will attack later. Because comparison cannot be represented nicely as an arithmetic circuit, MPC protocols often convert secret-shared integers into Boolean shares, before running a Boolean comparison circuit [6, 44]. We observe that, because a comparison circuit has only a single output bit, it is trivial for a malicious adversary to flip the result of a comparison, by injecting an error to the final AND gate of the circuit.

Note that flipping does not allow the adversary to *choose* the comparison result in and of itself. However, if  $x$  (or  $y$ ) is the outcome of a multiplication, the adversary can add a large constant offset to  $x$ , which will result in  $x > y$  with high probability, or add a large negative offset to  $x$ , forcing the output to 0. Combining input modification with result flipping can also force the output to 0 or 1.

**ReLU.** ReLU can be computed exactly as  $\text{ReLU}(x) = b \cdot x$ , where  $b$  is computed with a secure comparison  $\mathbb{1}(x \geq 0)$ . An adversary who flips the comparison can modify the result to  $\min(0, x) = x - \text{ReLU}(x)$ . Lehmkuhl et al. [42] show how to manipulate ReLU with input modifications if  $x$  is the output of a multiplication: adding a large positive constant  $\Delta$  can force  $x > 0$ , leading the ReLU to output  $x + \Delta$ ; subtracting  $\Delta$  likewise forces the ReLU to output 0. We also can combine input modification with comparison flipping to output either 0 or  $x - \Delta$ .

There are two typical ways to compute sigmoid and softmax. An approach that has gained recent popularity due to its simplicity is approximating these complex functions using piecewise polynomials. The advantage of doing so is that these approximations replace exponentials and divisions, which are hard to compute in MPC, with a combination of simpler operations like multiplications and comparisons. However, recent work has also demonstrated the practicality of direct approximations of sigmoid and softmax, by approximating exponentiation. Below, we show attacks on both of these methods.

**Piecewise Linear Sigmoid** A piecewise linear approximation of sigmoid, used in e.g. [1, 45] can be defined as:

$$\text{sigmoid}(x) = \begin{cases} 0 & x < -1/2 \\ x + 1/2 & -1/2 \leq x \leq 1/2 \\ 1 & x > 1/2 \end{cases} \quad (1)$$

We can compute the approximation via the following equation,  $\text{sigmoid}(x) = (b_1 \oplus 1) \cdot b_2 \cdot (x + 1/2) + (b_2 \oplus 1)$ , where  $b_1 = \mathbb{1}(x \leq -1/2)$  and  $b_2 = \mathbb{1}(x \leq 1/2)$ . Since we operate with secret-shares that are in the 2's complement representation, bits  $b_1, b_2$  correspond to the most significant bits of  $x + 1/2$  and  $x - 1/2$  respectively, and are computed with secure comparison. Computing the full piecewise polynomial approximation of sigmoid costs two calls to secure comparison, followed by two multiplications of these secret-shared bits with the secret-shared input  $x$ , over the arithmetic domain.

By tampering with the output of both comparisons, an adversary can force the 0 and  $x + 1/2$  cases of sigmoid to become 1, and flip the 1 case to become 0, as can be seen in the following case analysis:

1. When  $x - 1/2 < 0$ : The bits  $b_1, b_2$  should have both been set to 1 if computed honestly. If the adversary flips both outputs, they will both now become 0 and the sigmoid computation is set to 1, instead of the correct output of 0.
2. When  $-1/2 < x < 1/2$ :  $b_1$  should have been set to 0, and  $b_2$  to 1. Flipping  $b_1$  and  $b_2$  forces the sigmoid output to be 1.
3. When  $x > 1/2$ : The bits that should have both been set to 1, will now be set to 0. The sigmoid therefore outputs 0 instead of 1.

On its own, unless the adversary is able to guess the input  $x$ , there is little to be gained from only manipulating the piecewise linear sigmoid computation, as there is no universal strategy for producing a given output. However, input manipulations solve this problem. By adding a large positive constant to the input,  $x + \Delta > 1/2$  will force the output to 1, or to 0 by subtracting:  $x - \Delta < -1/2$ . By forcing the input into a given region of the piecewise function, manipulating the sigmoid computation can flip either  $x + \Delta$  or  $x - \Delta$  into the middle range, leading the sigmoid computation to return an out-of-bounds positive or negative value  $x + \Delta + 1/2$  or  $x - \Delta - 1/2$  (which we simply write as  $x + \Delta$  or  $x - \Delta$  in Table 1).

**Direct Computation of Sigmoid and Softmax.** Recall the plaintext computation of softmax and sigmoid:

$$\text{Softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}, \quad \text{Sigmoid}(x) = \frac{\exp(x)}{1 + \exp(x)},$$

where  $\mathbf{x} = (x_1, \dots, x_l)$  are the linear outputs of the last layer.

Instead of using a piecewise approximation, some work [4, 16, 17] directly approximates both the exponentiation and division in sigmoid and softmax to improve the accuracy of the approximation. Our simplest attack on these computations only exploits the division operation, taking advantage of the computation of  $x/y$  as multiplication by the reciprocal of the denominator  $x \cdot (1/y)$ . Indeed, with only this step, we see that both softmax and sigmoid are the output of a multiplication, and so we can add an arbitrary additive error to their output!

Input modification can in principle result in a wide variety of outputs, but one useful one is to add a large positive constant to a given coordinate  $x_i$  (or  $x$  itself in sigmoid), so that it dominates the other terms, and results in a vector which is 1 in entry  $i$  and 0 otherwise (in sigmoid, this returns 1 or 0). It is also possible to selectively zero out specific entries by adding large negative constants to their

input. Combining these input modifications with the computation modification also allows the output to be modified to  $e_i + v$  for any basis vector  $e_i$  and vector  $v$ , or  $\delta$  or  $1 + \delta$  for sigmoid. These attacks are those we describe in Table 1, but it is also possible to attack the reciprocal and exponentiation in other ways.

The reciprocal is either computed with Newton-Raphson iteration (in [16, 17]) or Goldschmidt’s algorithm (in [4]). Both of these algorithms use a large number of multiplications, which can each be attacked. For example, to compute  $1/x$ , Newton-Raphson begins with an initial approximation  $y_0 = 3 \cdot e^{0.5-x} + 0.003$ . Then, the following computation is repeated  $y_{n+1} = y_n \cdot (2 - x \cdot y_n)$ , with  $n$  set to 10 in practice. As long as  $y_0$  is a good initial estimate,  $\lim_{n \rightarrow \infty} y_n = \frac{1}{x}$ . Any of the multiplications in this protocol can be tampered with to control the output of the reciprocal, and Goldschmidt’s algorithm is similarly vulnerable.

It is also possible to attack the exponentiations directly. Some work [16, 17] uses the limit approximation for exponentiation,  $\exp(x) = \lim_{n \rightarrow \infty} (1 + \frac{x}{2^n})^{2^n}$ . In practice, they use  $n = 8$ , and use the repeated squaring method to compute the exponentials efficiently. Since this approach involves multiplications, it allows for an adversary to add additive errors to the result.

Keller and Sun [4] extend the technique of Aly and Smart [46] to do exponentiation. The high level idea of this technique is to separate the integer and fractional parts of the secret shared value  $x$  into  $i, r$  such that  $\llbracket x \rrbracket^A = \llbracket i \rrbracket^A + \llbracket r \rrbracket^B$ , where  $A$  indicates that the integer part is shared over the arithmetic domain and the fractional part,  $r$  is shared over the boolean domain ( $B$ ). Then they compute  $\llbracket 2^i \rrbracket^A$  using conventional techniques, and compute  $\llbracket 2^r \rrbracket^B$  using an approximation, such as the Taylor series approximation.

With the following attack strategy, an adversary can make the exponentiation computation output any constant of its choice. The algorithm from [4] (Algorithm 2) first computes a bit as,

$$\langle z \rangle^B = \sum_{i=0}^{k-1} 2^{i-f} \langle x_i \rangle^B < -(k - f - 1) \quad (2)$$

where the total bit length of the input  $x_i$  is  $k$ , and the precision is  $f$  bits. This bit is used to check if the input  $x_i$  is below a certain threshold. Since we use fixed-point representation, instead of computing softmax for very large negative numbers, we set the bit to 1 and output 0 for  $\text{softmax}(x_i)$ .  $\langle z \rangle^B$  is computed using a comparison operation, and in most cases the database will not have inputs that are huge negative numbers, which means that  $z$  will be set to 0 with high probability. Therefore, flipping it sets the bit to 1 in most cases. In Step 13 of Algorithm 2, the final output is computed as  $(1 - \text{Bit2A}(\langle z \rangle^B)) \cdot \langle h \rangle^A$ , where  $\langle h \rangle^A$  is the output of the algorithm. The idea is that if the input was a large negative value,  $(1 - \text{Bit2A}(\langle z \rangle^B))$  will be 0, thereby making the final output 0. So by flipping the bit  $z$  for one of the inputs  $x_i$ , with high probability we can make the final output 0. To make the exponentiation output any constant we like, we can make use of the multiplication in Step 13. Since the output is going to be 0, any constant  $c$  we add as the additive error at this multiplication will make the final output  $c$ .

## 4 High Level Manipulation Strategies

Our attacks in Section 3 manipulate each activation function individually. Here, we compose these attacks and other additive errors to construct high level manipulations that impact the training algorithm overall. We treat these manipulations as an API built from the low-level attacks in Section 3, to realize the adversarial goals in Section 5. Future work targeting other adversarial objectives may directly use these primitives.

A key challenge in designing and using these manipulation strategies is that the adversary does not know the input data or the current model weights, as they are all secret-shared.

### 4.1 Gradient Zeroing

The goal of gradient zeroing is to replace an example’s gradient with 0 in as many entries as possible, effectively removing the example from that training step. In logistic regression, we do not know how to achieve this simple goal: the only way to guarantee zeroing would be to force  $\text{sigmoid}(D) = Y$ , but this would require a data dependent manipulation, as  $Y$  is secret shared. For neural networks, we

can force all ReLUs to output 0, which results in nearly all gradient entries being set to 0, except for the final layer biases. In an SVM, we can use this same trick when computing the loss—by adding a large error to  $y \cdot D$ , we can force the loss, and therefore each entry of the gradient, to 0.

## 4.2 Gradient Shifting

In gradient shifting, an adversary shifts an example gradient  $g$  by some fixed vector  $v$ :  $g \rightarrow g + v$ . To see why this is possible, recall that each entry of the gradient is computed with a multiplication, such as when computing  $\nabla_w \ell$  in the logistic regression gradient in Section 2.3. For simplicity, we write the computation of the  $i$ th entry of the gradient  $(\nabla_w \ell)_i$  as  $\Pi_{\text{Mult}}(F, X_i)$ , where  $X_i$  is the  $i$ th feature of the example and  $F$  is the example loss. To add a vector  $v$  to this gradient computation, the adversary applies an additive error of  $v_i$  (the  $i$ th entry of the shift), changing the computation to  $(\nabla_w \ell)_i = \Pi_{\text{Mult}}(F, X_i) + v_i$ . Note that this is possible only for those gradient entries which are the result of a multiplication, and so only applies to weight gradients, not bias gradients.

The main difficulty that we will run into when using gradient shifting for attacks is that the adversary never knows the model parameters or the data used to compute the clean gradient. Our attacks cannot depend on these values, and the shift  $v$  must be effective at targeted a wide variety of possible model parameters. This is especially difficult for neural networks, which have many “isomorphisms”, wildly different model parameters which correspond to the exact same function [47, 48, 49].

## 4.3 Gradient Scaling

Gradient scaling allows the adversary to scale the gradient by roughly *multiplying* it by a scalar, rather than by shifting by a vector. It is surprising that we can accomplish this at all, because all of our manipulations must be data-independent, and scaling makes a data-dependent change to the gradient. The key property we rely on here is that all gradients are scaled by the derivative of the loss, represented for example by  $F$  in the logistic regression gradient algorithm in Section 2.3. For logistic regression, while we cannot control the value of  $Y$ , we do have control over the output of the sigmoid or softmax. By forcing these to arbitrary values, we can scale the gradient.

For logistic regression or neural networks, gradient scaling involves forcing the activation function to a large value. With piecewise linear sigmoid activations, this requires combining input and activation modification. For direct sigmoid or softmax computation, activation computation is sufficient, but may be combined with input modification. Note that we can only force the gradient to encourage strong predictions for a fixed class; we cannot “amplify” or “negate” a gradient with this approach, as we cannot make the activation’s output depend on  $Y$ .

In SVM training, recall that the loss  $F = \max(0, P)$ , where  $P$  is the margin. Then it is possible to scale gradients by increasing  $P$  by a large constant, or by decreasing  $P$  and flipping the secure comparison. These will increase the confidence of the SVM on  $y = -1$  and  $y = 1$ , respectively.

## 5 Attacks on Machine Learning Training

We now showcase the variety of adversarial goals which can be achieved with our high level strategies from Section 4. Here, we run experiments on logistic regression and neural networks. We use the Fashion MNIST (FMNIST), Census, Purchase, and Texas datasets, as they are standard in the literature. For Census, Purchase, and Texas, we sample 20000 records for the clean training set, allow the adversary 20000 records for their own training dataset, and 20000 records for the test dataset. We train all models with SGD using hyperparameters selected by a grid search on a distinct data split.

To implement our attacks, we build a plaintext “simulator” of MPC over  $\mathbb{R}$ , using the Jax library [50]. Our simulator uses appropriate MPC-friendly approximations, and allows the adversary to inject manipulations directly into the appropriate steps of the computation, to ensure faithfulness to how attacks would be carried out in MPC. We use a simulator instead of an MPC implementation as it allows us to 1) run our experiments locally on a GPU (all experiments use a single P100) and 2) take advantage of the autodifferentiation capabilities of Jax. More details can be found in Appendix A.4, and we plan to make our implementation public.

We summarize our attacks in Table 2, noting also which capability from Section 4 they use, whether they degrade model performance, and whether they require knowledge of training data ordering.

Table 2: A summary of concrete attacks, their capabilities, stealthiness (accuracy-detectable), and data order obliviousness. Privacy attacks that do not know training data order cannot leak arbitrary examples. The primary capability used is listed.

Attack Goal	Capabilities	Stealthy?	Oblivious?
Shift Poison	Shifting	Yes	Yes
Fairness	Zeroing	No	No
Membership	Scaling	Yes	Untargeted
Reconstruct	Scaling	No	Untargeted

Table 3: Overview of parameter transfer-based gradient shifting attacks, reporting Test accuracy and Attack Success Rate (ASR). Table (a) covers backdoor attacks, while (b) covers targeted attacks.

(a) Backdoor attacks via parameter transfer.						(b) Targeted attacks via parameter transfer.					
Dataset	Model	No Attack		Attack		Dataset	Model	No Attack		Attack	
		Test	ASR	Test	ASR			Test	ASR	Test	ASR
FMNIST	LR	0.84	0.004	0.83	0.9996	FMNIST	LR	0.84	0	0.83	1
Texas	LR	0.61	0.006	0.61	1.0000	Texas	LR	0.60	0	0.59	1
Purchase	LR	0.70	0.000	0.70	1.0000	Purchase	LR	0.70	0	0.67	1

### 5.1 Parameter Transfer: Poisoning Linear Models with Gradient Shifting

Traditional poisoning attacks use malicious data to exert a specific influence on the model. Our first attack demonstrates how we can replicate this influence with gradient shifting: if a crafted poison example produces a poisoned gradient  $g_p$ , a gradient shifting attack can introduce this same gradient into the protocol without any data poisoning. This bears some resemblance to poisoning attacks on federated learning (FL) [26, 51], which show that changing gradients rather than data (known as *model poisoning*) during training leads to very effective attacks.

The key difference between attacks on FL and our attacks is that, in our setting, gradient shifting must be carried out without knowing the current model parameters, as their values are secret shared. Indeed, all work on FL poisoning uses knowledge of the parameters. To circumvent our restriction, we propose a general strategy, *parameter transfer*, to poison models with unknown parameters. In parameter transfer, the adversary trains a “reference” model on their own data, computes a poisoned gradient on this reference model, and runs gradient shifting to add this poisoned gradient to the overall gradient.

Neural networks are difficult to attack with parameter transfer due to their nonconvex loss surface: there are many settings of “good” weights which are far from each other, including isomorphisms producing identical outputs. Then gradients on one network do not transfer to another, except in fine tuning settings [52]. We address this shortcoming later, and run parameter transfer on simple models.

**Evaluation.** We use parameter transfer gradient shifting to implement backdoor and targeted poisoning attacks on softmax-based logistic regression. We train the adversary’s reference model and compute transfer gradients with the adversary’s disjoint training dataset. The goal of our backdoor attack is, in every dataset, to cause any input from class 0 to be misclassified as class 1 when the first feature is set to a value of 1. The goal of our targeted attacks is to misclassify 5 randomly selected test examples to a randomly selected class. We measure the success rate of backdoor attacks on triggered test examples (distinct from both the clean training and adversary datasets), and the success rate of targeted attacks on the targeted test examples, all averaged over 10 runs.

We report our results for backdoor attacks in Table 3a and our results for targeted attacks in Table 3b. Our attacks achieve near perfect success at each adversarial goal, with little to no performance degradation—model accuracy never drops by more than 2.3% on average.

### 5.2 Neuron Override: Poisoning Neural Networks with Gradient Shifting

Neural networks are not vulnerable to parameter transfer due to their nonconvex loss landscapes. We instead design an attack called *neuron override*. In neuron override, the adversary concentrates the

Table 4: Backdoor attacks with neuron override-based gradient shifting, reporting Test accuracy and Attack Success Rate.

Dataset	Model	No Attack		Attack	
		Test	ASR	Test	ASR
FMNIST	NN	0.86	0.003	0.86	0.90
Texas	NN	0.58	0.01	0.57	1.00
Purchase	NN	0.55	0.000	0.51	1.00

Table 5: True positive rate (TPR) at low false positive rate (FPR) for gradient scaling-based MI amplification. Gradient scaling substantially increases MI risk, sometimes by factors of 20-40x. Confidence intervals ( $p < 0.01$ ) for all values have width  $< 0.26\%$ .

Data/Model	With Scaling		No Scaling	
	TPR@.1%	1%	.1%	1%
FMNIST/LR	4.8%	11.9%	0.18%	1.5%
Texas/LR	76.0%	93.2%	3.0%	10.8%
Purchase/LR	4.5%	22.4%	1.9%	11.0%
FMNIST/NN	2.3%	7.1%	0.2%	1.7%
Texas/NN	46.6%	75.3%	0.8%	4.3%
Purchase/NN	6.0%	13.6%	0.4%	2.9%

backdoor attack into a single neuron, forcing it to activate only on backdoored data, and connects this neuron’s activation to the output corresponding to the target class. We implement neuron override specifically for backdoor attacks, but believe it can also be used for targeted attacks.

Concretely, for a two layer ReLU neural network, without loss of generality, we override the first neuron. We use gradient shifting to add a shift of  $\delta - c\mu$  to the first neuron’s input weights. Here,  $\delta$  is the backdoor trigger direction, a direction we expect only backdoored examples to be highly correlated with. We set  $\mu$  as the average of some nontargeted data, which we use to reduce the number of standard examples that activate the neuron, and  $c$  is a hyperparameter used to trade off these contributions. In the second layer, we also shift the weight from this neuron to the target class by a large positive value, to force examples activating this neuron to be classified into the target class. This strategy is inspired by the handcrafted backdoors of Hong et al. [53], although their attacks had the advantage of being able to inspect model weights when introducing the backdoor; our attacks must override a neuron regardless of its original weights.

**Evaluation:** We evaluate our neuron override attacks on the same backdoor tasks we considered for parameter transfer. We attack softmax-based two layer neural network models with varying hidden layer sizes, and again measure backdoor attack success rate on triggered test examples, and average over 10 runs. We report our results in Table 4. Our attacks lead to limited test accuracy decrease, of no more than 4% accuracy, but our attacks lead to very high attack success rates.

### 5.3 Amplifying Membership Inference with Gradient Scaling

Our prior attacks compromise the *integrity* of learned models. We now turn our attention to privacy attacks, directly compromising MPC’s privacy guarantees only by deviating from the protocol.

We begin with membership inference (MI). We take advantage of a property identified by recent work, that it is easier to infer membership of “outlier” examples, such as mislabeled or noisy examples, compared to more natural “inlier” examples [36]. Our attack uses gradient scaling to “convert” any example into an outlier example. Recall that gradient scaling can modify the gradient to encourage an example be predicted as any target class. By selecting any example, and any random target class, our attack uses gradient scaling to force that example to be predicted as the target class. Because it is unlikely that the example will naturally belong to that class, the update is made as if the target were a mislabeled “outlier”, leaving it vulnerable to MI!

**Evaluation.** We run our attack on softmax-based logistic regression and neural network models. We run a small amount of tuning to select gradient scaling strengths that result in low accuracy drop, and use a target class of 0 for all gradient scaling. To run the attack, we adapt the state-of-the-art offline LiRA attack [36]. In LiRA, the adversary trains “shadow models” without scaling on their own data. Then, when running the attack on a given model and example, the adversary measures the deviation of the model’s predictions on the target example from the predictions of the shadow models. Traditionally, LiRA measures the deviations in the probabilities of the example’s correct class, but we account for our scaling by instead measuring the deviation for the gradient scaling target class.

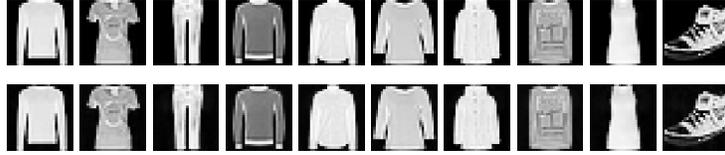


Figure 2: Gradient scaling leads to successful reconstruction attacks on logistic regression. The top row is the original image, and the bottom is its reconstruction.

We measure the effectiveness of our attacks in Table 5 using True Positive Rate (TPR) at low False Positive Rates (FPRs), a standard measure of MI success, reflecting the attack’s precision. We find that our gradient scaling strategy is very effective, improving TPR by a factor of at least 2 and sometimes as large as 40! At these levels, the adversary can be nearly certain that the identified examples are members. Our attacks often have limited impact on model accuracy, such as a 0-2% decrease on FMNIST models, but can be substantial, such as an 8% drop on average for NN on Purchase. Attack success and accuracy can be traded off by modifying the gradient scaling strength.

#### 5.4 Training Data Reconstruction with Gradient Scaling

Having established strong membership inference leakage, we now turn to using gradient scaling to perform training data reconstruction. Our reconstruction attack relies on a property exploited by Boenisch et al. [54] in the context of federated learning attacks — the gradient of the first layer of a neural network (or just the weights in a linear model) is a scalar multiple of the input features. This can be seen, for example, in the computation of the gradient  $G$  in the protocol for logistic regression in Section 2.3; to compute the gradient, the scalar loss derivative  $F$  is multiplied by the input  $X$ .

In MPC, the adversary cannot access the gradients directly, as is possible in (vanilla) federated learning. To still be able to perform reconstruction, we will instead “overwrite” the model weights with a highly scaled gradient. For example, using a large scaling of 10000 for a single example in the final batch will result in the original model weights being small in magnitude relative to the scaled gradient; the first layer weights will be a vector which can be rescaled to obtain the target example. Of course, such a large gradient will destroy the original model weights, resulting in an unusable model and an obvious attack, but an adversary may consider running this attack if the possibility of reputation damage is outweighed by the learning sensitive information.

**Evaluation.** We run our attack on softmax-based logistic regression and neural networks for Fashion MNIST, using a scaling factor of 10000 for class 0. We attack a random example in the final batch of training for simplicity, but expect that the attack could be carried out in different training steps if desired. After performing the attack, we must select the weights to rescale to recover the reconstructed image. For logistic regression, we use the model weights leading to the class 0 output, and for the neural network, we inspect a small number of neuron weights to find one which contains a coherent image, and rescale these weights to reconstruct the example.

We run 10 trials of our attack and present our reconstructions in Figure 2 and Figure 3 for logistic regression and neural networks, respectively. Visually, they are nearly identical to the original images. The mean absolute error between the original and reconstructed images is 0.017 for logistic regression and 0.036 for neural networks. That is, the average pixel of a reconstruction differs from the original by only 1.7% (or 3.6%) of the total pixel range of  $[0, 255]$ . However, we reiterate that this attack is not stealthy, reducing model accuracy to 10% or “random guessing”.

#### 5.5 Reducing Fairness with Gradient Zeroing or Scaling

In high stakes applications, it is important to build models which do not discriminate on societally relevant attributes, such as race or gender. Multiparty machine learning may even be a way of reducing such unfairness; often, in multiparty settings, each party’s data has a different distribution, and, by pooling data, the model improves on all represented subpopulations. We consider such a setting, and show how an adversary can attack the protocol to cause the model’s accuracy to degrade for one contributing party, by nullifying the contributions of that party’s data to the model.

Model/Attack	No Attack		Attack	
	Target	Other	Target	Other
LR + Boost	0.748	0.753	0.673	0.731
NN + Boost	0.724	0.737	0.652	0.710
NN + Zeroing	0.75	0.76	0.714	0.731

Table 6: Attacks on fairness. Our attacks reduce attack performance disproportionately on the target subpopulation, harming the fairness of the trained model, by preventing one party’s contributions to the model.

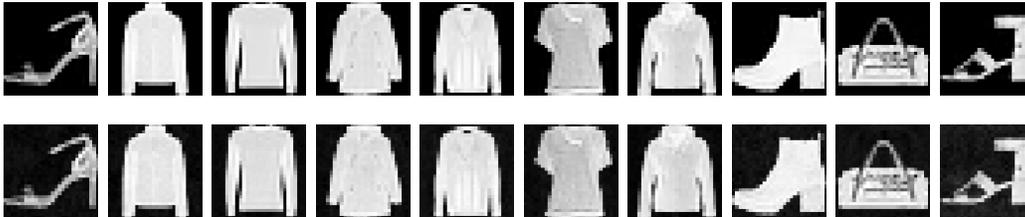


Figure 3: Gradient scaling attacks permit successful reconstruction attacks on neural networks. The top row is the original image, and the bottom row is the reconstructed image.

The most natural way to achieve this aim is to prevent the target party’s data from *ever* contributing to the model with gradient zeroing. This requires the adversary to know which examples came from the target party, and so requires knowledge of the data order. Gradient zeroing is not possible for logistic regression, so we also experiment with gradient scaling, to encourage all examples from the target party to be misclassified to a fixed target class. These attacks can be seen as an adaptation to our threat model of attacks considering the implications of removing data [55] or poisoning attacks [56, 57, 33] on model fairness.

**Evaluation.** We run this attack using a simulated multiparty setting on the US Census dataset. Using the folktables Python package, we generate five parties by selecting 1000 examples from each of five different, arbitrarily chosen states: CA, OK, OR, WA, and WI. The parties collaborate on a sigmoid-based logistic regression model or a sigmoid-based neural network with 50 neurons in the hidden layer. We train both for 10 epochs with learning rate 0.001 and batch size 100. The adversary targets CA, and performs gradient zeroing or gradient scaling with strength 2 towards class 1; we average all results over 10 trials. We choose this setup as a real-world example of non-iid data, although our results should only improve in more highly non-iid settings.

We report the results of our experiment in Table 6. We report test accuracy for the target state and the average for the other states’, both with and without each attack. We find that our attacks all have a disparate impact on model accuracy: the decrease in accuracy on the target state is higher than the decrease in accuracy on the other states. For example, attacking the neural network with gradient scaling decreases accuracy in CA by 7.2%, while the other states are only harmed by 2.7%.

## 5.6 Amplifying Data Poisoning with Gradient Scaling

Parameter transfer and neuron override both use gradient shifting to induce successful poisoning. To add to this, we also show that an adversary can amplify data poisoning attacks by colluding with the malicious data contributors, by scaling the gradients of these malicious examples. This can lead to poisoning that is more effective than the data poisoning alone, allowing attacks to use smaller amounts of poisoning data.

To instantiate this data poisoning amplification, note that the adversary’s collusion with malicious data means they know the  $Y$  value for the poisoning data, and gradient scaling can be performed to amplify the gradient towards this class. For example, in softmax-based models, the adversary can add a large negative value to the softmax output for the poisoning  $Y$ . In sigmoid-based models, if  $Y = 1$ , the sigmoid can be tampered with to output a large value, or a small value if  $Y = 0$ .

Dataset/Model	No Attack	No Scale	Scale
FMNIST/LR	0.84/0.02	0.84/0.04	0.83/0.72
Texas/LR	0.61/0	0.60/0.08	0.60/1
Purchase/LR	0.70/0	0.70/0	0.69/0.72
FMNIST/NN	0.86/0	0.86/0.36	0.75/0.52
Texas/NN	0.59/0	0.58/0.28	0.56/0.94
Purchase/NN	0.55/0.02	0.54/0.36	0.41/0.9

Table 7: Test accuracy/attack success rate for targeted attacks with gradient scaling. Gradient scaling results in a stronger attack than could be achieved without scaling.

Dataset	Model	No Attack	Attack
FMNIST	LR	.836	.717
Texas	LR	.605	.565

Table 8: Availability attacks with parameter transfer-based gradient shifting. We use a small shift gradient, so our performance degradation can be improved significantly by changing this value.

**Evaluation.** We report results for targeted data poisoning here, as we find that amplifying backdoor data poisoning leads to inaccurate models, likely due to the relatively large amount of poisoning data we need to encourage generalization of the backdoor trigger. We attack softmax-based models, including both logistic regression and neural networks. For targeted data, we again target 5 examples, and add between 2 and 10 flipped label training examples per target, and gradient scale by shifting the value of softmax for the target class by between -2 and -5. These hyperparameters depend on the dataset and model, and we perform a small amount of tuning to find these values, identifying a number of training examples which can be added per target which lead to a weak attack that can be improved with scaling. If we used more data per target, the attacks would not need scaling. In practice, the amount of data per target is more likely to be a constraint of the adversary, which would be set independently, and offline experimentation could be used to set the scaling strength.

We report our results in Table 7. As expected, gradient scaling allows our attacks to be more effective at the data poisoning levels we test. Of course, stronger data poisoning would allow weaker scaling to be effective, but our attacks can enable strong attacks when they were impossible at small data poisoning strengths. Our attacks also generally have small impact on model performance, although neural networks on Fashion MNIST and Purchase see substantial accuracy decreases. In practice, an adversary can test hyperparameter values offline in order to understand how aggressive they can afford to be given a fixed accuracy constraint.

## 5.7 Other Possible Attacks and Possible Mitigations

We have instantiated each adversarial goal with one main high level strategy, but some goals may be achievable with multiple strategies.

**Availability Poisoning.** These attacks are possible with nearly any large, untargeted, modification to the model. Hypothetical approaches are heavy gradient scaling of many example gradients, batch-wise gradient zeroing during training, to ensure that the model parameters never change during training, or gradient shifting with large modifications, which we briefly evaluate in Table 8.

**Membership Inference.** It has been shown that MI can be amplified through the inclusion or exclusion of other examples [58, 59, 60]. An adversary could replicate these indirect attacks with any of gradient zeroing, shifting, or scaling.

## 6 Potential Protocol-Level Mitigations

An obvious mitigation, immediately preventing all of our attacks, is to use actively secure MPC protocols for training. Of course, such protocols come with a significant performance (in terms of run time) hit. This motivates the question of whether one could design “middle ground” mitigations, which can prevent attack, such as the ones proposed in our work, without sacrificing too much performance relative to the passive protocol. Towards this, we outline possible design strategies to mitigate these attacks. We would like to emphasise that adaptive attacks may be able to circumvent them, and more careful analysis is required.

Gradient scaling requires manipulating the loss computation, and can be mitigated by computing the loss given the logits with an actively secure sigmoid or softmax. Computing these activation functions takes a small fraction of training time, as they are performed once per example, which does not scale with the number of parameters in the model. In [4], softmax computation takes less than one percent of compute time, making this a cheap but powerful mitigation. This approach would fully stop our boosting of poisoning attacks, attacks compromising fairness, and membership inference attacks.

Mitigating gradient shifting attacks is more challenging, as multiplications in backpropagation can be independently tampered with. However, the gradient shifting attacks we run all requires tampering with many multiplications, so one potential avenue to reduce the risk of an attack is to somehow check only an  $\alpha$  fraction of multiplications, rather than all of them. We do not know how to design such a protocol, as naive approaches would either not improve efficiency, or invalidate the goal by allowing the adversary to know in advance which multiplications will be checked. However, if possible, and the adversary introduces  $p$  additive errors throughout the protocol, then the probability that the adversary is not caught is  $(1 - \alpha)^p$ . Our gradient shifting poisoning attacks on FMNIST, for example, modify roughly 1.5 million multiplications in total, which is large enough that even checking an  $\alpha < 2 \cdot 10^{-9}$  fraction of multiplications will still catch the adversary with  $> 1 - 2^{40}$  probability. However, our attacks are not optimized to reduce the number of additive errors, and it is likely these attacks can remain effective with orders of magnitude fewer errors by, for example, making larger, less frequent, or more sparse modifications.

## 7 Discussion

Our results show that malicious adversaries can stealthily corrupt models trained with passive MPC protocols. This challenges the perspective that attacks on such protocols will be easy to prevent with reputation- or incentive-based approaches. We hope that our work promotes more research at the intersection of MPC and adversarial ML, in addition to having takeaways for each:

**For the MPC community.** We show that in ML training, if an active adversary is given access to the output of the computation, privacy can be completely broken in the passive setting. An interesting future theoretical question is to understand if there exists a broader class of functionalities with such privacy issues. This is analogous to related-key attacks on AES [61, 62], where adversaries exploit structural relationships between keys to induce predictable variations in internal states, ultimately enabling key recovery. In both cases, the attacker leverages correlated inputs to bypass standard security assumptions. We also hope to encourage exploration into “middle ground” protocols between the two models, which defend against well defined classes of attacks, as we discuss in Section 6. Our work underscores the importance of actively secure ML training; we hope it provides useful information for MPC practitioners when evaluating threat models for applications.

**For the adversarial ML community.** The “additive error” threat model we consider in our work may be interesting to investigate more deeply in adversarial ML. It may be possible to improve the effectiveness and stealthiness of our attacks or understand the threat model from a theoretical perspective. Beyond this threat model, we encourage adversarial ML research to consider MPC as a setting to propose attacks, consider new threat models, and design concretely efficient defenses.

**Limitations.** One limitation of our work is that we work over floating point arithmetic, rather than fixed-point; we do not believe this significantly impacts findings [22, 23]. Our experiments are limited to small models; due to the cost of training in MPC, the literature tends to use very small models [4], and we expect our attacks to generalize to larger models.

## Acknowledgements

The work of P. Scholl was supported in part by grants from the Digital Research Fund Denmark (DIREC), the Danish Independent Research Council under Grant-ID DFF-0165-00107B (C3PO) and by Cyberagentur under the project Encrypted Computing.

## References

- [1] P. Mohassel and Y. Zhang, “SecureML: A system for scalable privacy-preserving machine learning,” in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press,

- May 2017, pp. 19–38.
- [2] S. Wagh, D. Gupta, and N. Chandran, “SecureNN: 3-party secure computation for neural network training,” *PoPETs*, no. 3, pp. 26–49, Jul. 2019.
  - [3] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, “Falcon: Honest-majority maliciously secure framework for private deep learning,” *PoPETs*, no. 1, pp. 188–208, Jan. 2021.
  - [4] M. Keller and K. Sun, “Secure quantized training for deep learning,” Cryptology ePrint Archive, Report 2022/933, 2022. [Online]. Available: <https://eprint.iacr.org/2022/933>
  - [5] S. Laur, H. Lipmaa, and T. Mielikäinen, “Cryptographically private support vector machines,” Cryptology ePrint Archive, Report 2006/198, 2006. [Online]. Available: <https://eprint.iacr.org/2006/198>
  - [6] D. Demmler, T. Schneider, and M. Zohner, “ABY - A framework for efficient mixed-protocol secure two-party computation,” in *NDSS 2015*. The Internet Society, Feb. 2015.
  - [7] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica, “Cerebro: A platform for multi-party cryptographic collaborative learning,” in *USENIX Security 2021*. USENIX Association, Aug. 2021, pp. 2723–2740.
  - [8] S. Carpov, K. Deforth, N. Gama, M. Georgieva, D. Jetchev, J. Katz, I. Leontiadis, M. Mohammadi, A. Sae-Tang, and M. Vuille, “Manticore: Efficient framework for scalable secure multiparty computation protocols,” Cryptology ePrint Archive, Report 2021/200, 2021. [Online]. Available: <https://eprint.iacr.org/2021/200>
  - [9] M. Chase, R. Gilad-Bachrach, K. Laine, K. Lauter, and P. Rindal, “Private collaborative neural network learning,” Cryptology ePrint Archive, Report 2017/762, 2017. [Online]. Available: <https://eprint.iacr.org/2017/762>
  - [10] Q. Pang, J. Zhu, H. Möllering, W. Zheng, and T. Schneider, “BOLT: Privacy-preserving, accurate and efficient inference for transformers,” in *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2024, pp. 4753–4771.
  - [11] M. Hao, H. Li, H. Chen, P. Xing, G. Xu, and T. Zhang, “Iron: Private inference on transformers,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 15 718–15 731. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/64e2449d74f84e5b1a5c96ba7b3d308e-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/64e2449d74f84e5b1a5c96ba7b3d308e-Paper-Conference.pdf)
  - [12] D. Bogdanov, S. Laur, and J. Willemson, “Sharemind: A framework for fast privacy-preserving computations,” in *ESORICS 2008*, ser. LNCS. Springer, Berlin, Heidelberg, Oct. 2008, pp. 192–206.
  - [13] “Syft,” 2023, <https://github.com/OpenMined/PySyft>.
  - [14] M. Dahl, J. Mancuso, Y. Dupis, B. Decoste, M. Giraud, I. Livingstone, J. Patriquin, and G. Uhma, “Private machine learning in tensorflow using secure computation,” *CoRR*, vol. abs/1810.08130, 2018. [Online]. Available: <http://arxiv.org/abs/1810.08130>
  - [15] B. Schoenmakers, “Mpyc: Multiparty computation in python,” 2023, <https://github.com/lischoe/mpyc>.
  - [16] B. Knott, S. Venkataraman, A. Y. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, “Crypten: Secure multi-party computation meets machine learning,” *CoRR*, vol. abs/2109.00984, 2021. [Online]. Available: <https://arxiv.org/abs/2109.00984>
  - [17] S. Tan, B. Knott, Y. Tian, and D. J. Wu, “CryptGPU: Fast privacy-preserving machine learning on the GPU,” in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 1021–1038.
  - [18] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability,” in *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 910–927.
  - [19] M. Jagielski, J. Ullman, and A. Oprea, “Auditing differentially private machine learning: How private is private sgd?” *Advances in Neural Information Processing Systems*, vol. 33, pp. 22 205–22 216, 2020.

- [20] M. Nasr, S. Songi, A. Thakurta, N. Papernot, and N. Carlin, “Adversary instantiation: Lower bounds for differentially private machine learning,” in *2021 IEEE Symposium on security and privacy (SP)*. IEEE, 2021, pp. 866–882.
- [21] Y. Aumann and Y. Lindell, “Security against covert adversaries: Efficient protocols for realistic adversaries,” in *TCC 2007*, ser. LNCS. Springer, Berlin, Heidelberg, Feb. 2007, pp. 137–156.
- [22] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “CrypTFlow: Secure TensorFlow inference,” in *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 336–353.
- [23] C. Harth-Kitzerow, A. Suresh, and G. Carle, “Truncation untangled: Scaling fixed-point arithmetic for privacy-preserving machine learning to large models and datasets,” Cryptology ePrint Archive, Paper 2024/1953, 2024. [Online]. Available: <https://eprint.iacr.org/2024/1953>
- [24] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer, “Circuits resilient to additive attacks with applications to secure computation,” in *46th ACM STOC*. ACM Press, May / Jun. 2014, pp. 495–504.
- [25] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. Rubinstein, U. Saini, C. Sutton, J. D. Tygar, and K. Xia, “Exploiting machine learning to subvert your spam filter.” *LEET*, vol. 8, no. 1-9, pp. 16–17, 2008.
- [26] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, “How to backdoor federated learning,” in *International conference on artificial intelligence and statistics*. PMLR, 2020, pp. 2938–2948.
- [27] B. Biggio, B. Nelson, and P. Laskov, “Poisoning attacks against support vector machines,” *arXiv preprint arXiv:1206.6389*, 2012.
- [28] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, “Manipulating machine learning: Poisoning attacks and countermeasures for regression learning,” in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 19–35.
- [29] T. Gu, B. Dolan-Gavitt, and S. Garg, “Badnets: Identifying vulnerabilities in the machine learning model supply chain,” *arXiv preprint arXiv:1708.06733*, 2017.
- [30] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, “Targeted backdoor attacks on deep learning systems using data poisoning,” *arXiv preprint arXiv:1712.05526*, 2017.
- [31] O. Suci, R. Marginean, Y. Kaya, H. Daumé, III, and T. Dumitras, “When does machine learning FAIL? Generalized transferability for evasion and poisoning attacks,” in *USENIX Security 2018*. USENIX Association, Aug. 2018, pp. 1299–1316.
- [32] A. Shafahi, W. R. Huang, M. Najibi, O. Suci, C. Studer, T. Dumitras, and T. Goldstein, “Poison frogs! targeted clean-label poisoning attacks on neural networks,” *Advances in neural information processing systems*, vol. 31, 2018.
- [33] M. Jagielski, G. Severi, N. Poussette Harger, and A. Oprea, “Subpopulation data poisoning attacks,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3104–3122.
- [34] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership inference attacks against machine learning models,” in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 3–18.
- [35] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha, “Privacy risk in machine learning: Analyzing the connection to overfitting,” in *CSF 2018 Computer Security Foundations Symposium*. IEEE Computer Society Press, 2018, pp. 268–282.
- [36] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer, “Membership inference attacks from first principles,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1897–1914.
- [37] J. Ye, A. Maddi, S. K. Murakonda, V. Bindschaedler, and R. Shokri, “Enhanced membership inference attacks against machine learning models,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3093–3106.
- [38] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, “Extracting training data from large language models,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

- [39] B. Balle, G. Cherubin, and J. Hayes, “Reconstructing training data with informed adversaries,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1138–1156.
- [40] M. Jagielski, “A note on interpreting canary exposure,” *arXiv preprint arXiv:2306.00133*, 2023.
- [41] G. Kaissis, J. Hayes, A. Ziller, and D. Rueckert, “Bounding data reconstruction attacks with the hypothesis testing interpretation of differential privacy,” *arXiv preprint arXiv:2307.03928*, 2023.
- [42] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, “Muse: Secure inference resilient to malicious clients,” in *USENIX Security 2021*. USENIX Association, Aug. 2021, pp. 2201–2218.
- [43] H. Chaudhari, M. Jagielski, and A. Oprea, “SafeNet: Mitigating data poisoning attacks on private machine learning,” *Cryptology ePrint Archive*, Report 2022/663, 2022. [Online]. Available: <https://eprint.iacr.org/2022/663>
- [44] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, “Improved primitives for MPC over mixed arithmetic-binary circuits,” in *CRYPTO 2020, Part II*, ser. LNCS. Springer, Cham, Aug. 2020, pp. 823–852.
- [45] P. Mohassel and P. Rindal, “ABY<sup>3</sup>: A mixed protocol framework for machine learning,” in *ACM CCS 2018*. ACM Press, Oct. 2018, pp. 35–52.
- [46] A. Aly and N. P. Smart, “Benchmarking privacy preserving scientific operations,” *Cryptology ePrint Archive*, Report 2019/354, 2019. [Online]. Available: <https://eprint.iacr.org/2019/354>
- [47] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov, “Property inference attacks on fully connected neural networks using permutation invariant representations,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 619–633.
- [48] D. Rolnick and K. Kording, “Reverse-engineering deep ReLU networks,” in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 8178–8187. [Online]. Available: <https://proceedings.mlr.press/v119/rolnick20a.html>
- [49] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, “High accuracy and high fidelity extraction of neural networks,” in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 1345–1362.
- [50] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” 2018. [Online]. Available: <http://github.com/google/jax>
- [51] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, “Analyzing federated learning through an adversarial lens,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 634–643.
- [52] G. Ilharco, M. T. Ribeiro, M. Wortsman, S. Gururangan, L. Schmidt, H. Hajishirzi, and A. Farhadi, “Editing models with task arithmetic,” *arXiv preprint arXiv:2212.04089*, 2022.
- [53] S. Hong, N. Carlini, and A. Kurakin, “Handcrafted backdoors in deep neural networks,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 8068–8080, 2022.
- [54] F. Boenisch, A. Dziedzic, R. Schuster, A. S. Shamsabadi, I. Shumailov, and N. Papernot, “When the curious abandon honesty: Federated learning is not private,” in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 175–199.
- [55] D. Zhang, S. Pan, T. Hoang, Z. Xing, M. Staples, X. Xu, L. Yao, Q. Lu, and L. Zhu, “To be forgotten or to be fair: Unveiling fairness implications of machine unlearning methods,” *arXiv preprint arXiv:2302.03350*, 2023.
- [56] D. Solans, B. Biggio, and C. Castillo, “Poisoning attacks on algorithmic fairness,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2020, pp. 162–177.
- [57] H. Chang, T. D. Nguyen, S. K. Murakonda, E. Kazemi, and R. Shokri, “On adversarial bias and the robustness of fair machine learning,” *arXiv preprint arXiv:2006.08669*, 2020.
- [58] Y. Chen, C. Shen, Y. Shen, C. Wang, and Y. Zhang, “Amplifying membership exposure via data poisoning,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 29 830–29 844, 2022.

- [59] F. Tramèr, R. Shokri, A. San Joaquin, H. Le, M. Jagielski, S. Hong, and N. Carlini, “Truth serum: Poisoning machine learning models to reveal their secrets,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2779–2792.
- [60] N. Carlini, M. Jagielski, C. Zhang, N. Papernot, A. Terzis, and F. Tramer, “The privacy onion effect: Memorization is relative,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 13 263–13 276, 2022.
- [61] M. Bellare, D. Cash, and R. Miller, “Cryptography secure against related-key attacks and tampering,” in *ASIACRYPT 2011*, ser. LNCS. Springer, Berlin, Heidelberg, Dec. 2011, pp. 486–503.
- [62] M. Abdalla, F. Benhamouda, A. Passelègue, and K. G. Paterson, “Related-key security for pseudorandom functions beyond the linear barrier,” in *CRYPTO 2014, Part I*, ser. LNCS. Springer, Berlin, Heidelberg, Aug. 2014, pp. 77–94.
- [63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

## A Instantiating the Attacks in MPC

### A.1 MPC Over Fixed-Point Arithmetic

An MPC protocol is often modelled abstractly as a secure “arithmetic black box” (ABB) — that is, a box that receives and stores private inputs from clients, and when instructed by all parties, performs computations on the inputs and reveals any desired outputs. While classic MPC protocols model computations via additions and multiplications over some finite field or ring, in machine learning we aim to emulate real-number arithmetic, so instead assume a fixed-point representation of the data. In fixed-point arithmetic with precision  $f \in \mathbb{N}$ , real numbers are approximated as rationals of the form  $x/2^f$ , where  $x \in \mathbb{Z}$ . When emulating fixed-point arithmetic in an MPC protocol, we require  $x$  to lie in the ring  $\mathbb{Z}_M$  for some modulus  $M$ , and assume that  $M$  is large enough such that during each addition or multiplication in the computation, there is no wraparound modulo  $M$  so we avoid overflow. The MPC protocol then emulates fixed-point arithmetic via additions and multiplications in  $\mathbb{Z}_M$ , together with a specialized protocol to truncate the result after each fixed-point multiplication. In Figure 4, we describe the arithmetic black box functionality, that captures fixed-point arithmetic while allowing an adversary to introduce an additive error after each multiplication operation. Note that the functionality assumes that any fixed-point value  $x/2^f$  input by the parties in the Input, Store or LinComb steps is represented solely by the value  $x \in \mathbb{Z}_M$ .

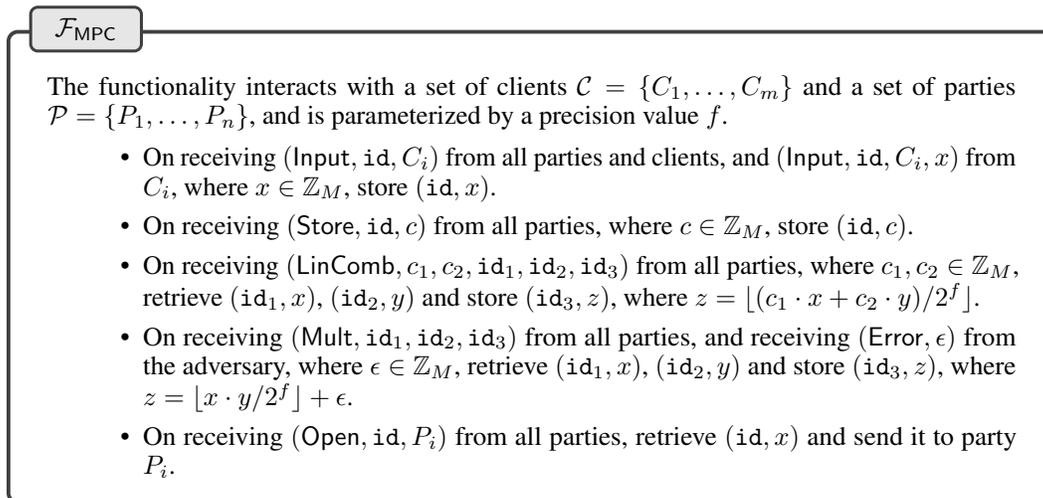


Figure 4: MPC arithmetic black box for fixed-point arithmetic modulo  $M$

**Secret-Sharing.** Our attacks apply to most MPC protocols based on a linear secret-sharing scheme. In linear secret sharing, a secret  $x \in \mathbb{Z}_M$  is divided into  $n$  shares,  $x_1, \dots, x_n$ , distributed among  $n$  parties such that each party  $P_i$  holds the share  $x_i$ . We use the notation  $\llbracket x \rrbracket$  to denote that  $x$  is secret-shared across the parties in an MPC protocol. A secret can be reconstructed, given all  $n$  shares, in a linear manner via  $x = \sum_i \lambda_i x_i$ , where each  $\lambda_i \in \mathbb{Z}_M$  is a public reconstruction coefficient. In the simplest example of additive sharing, each  $\lambda_i = 1$  and all  $n$  shares are required to reconstruct. Another example is Shamir secret sharing, where the  $i$ -th share is an evaluation  $p(i)$  of a polynomial  $p(X)$  such that  $p(0) = x$ , and the  $\lambda_i$ 's are Lagrange coefficients used to interpolate  $x$ . Here, the number shares needed for reconstruction is  $t + 1$ , where  $t$  is the degree of  $p(X)$ .

In these schemes, additions are “free” because parties can locally add their shares of  $\llbracket x \rrbracket, \llbracket y \rrbracket$  to get  $\llbracket x + y \rrbracket$ . Multiplications, however, require interaction, and reducing the communication needed to perform secret-shared multiplications is a popular area of research, and especially important for settings like PPML where there are a large number of multiplications to compute.

Our attacks are focused on honest majority protocols using *robust* secret sharing schemes such as Shamir sharing or replicated secret sharing, where the reconstruction process allows parties to easily verify that the correct secret was recovered. These schemes are often used in practical honest-majority protocols (even in the semi-honest model). However, we highlight that all of our attacks can also be applied to MPC protocols based on any linear secret-sharing scheme, including additive secret sharing, which is usually used in the dishonest majority setting.

## A.2 Backpropagation

Neural networks for classification typically use sigmoid or softmax outputs as in logistic regression, making their gradient computations similar to logistic regression. However, there is added complexity due to their depth. Here we describe how a two layer neural network is computed, as extending this further follows the same principles. First, a two layer neural network has two weight matrices, the  $m \times d$  input matrix  $w_0$  and the  $k \times m$  output matrix  $w_1$ , and two bias vectors, the  $m$  dimension  $b_0$  and the  $k$  dimension  $b_1$ , where again  $k$  is the number of classes (or 1 in binary classification), and here  $m$  is the dimension of the “hidden layer”. The network prediction  $D$  is instead computed as  $D = w_1 \text{ReLU}(w_0 \cdot x + b_0) + b_1$ , where  $\text{ReLU}$  is a nonlinear activation function  $\text{ReLU}(z) = \max(0, z)$ . We can write the ReLU inputs  $D_0 = w_0 \cdot x + b_0$ , and the outputs from ReLU,  $A = \text{ReLU}(D_0)$ , are known as the “activations” of the first layer, as each entry, or “neuron” in this layer is “activated” when the ReLU is nonzero. The gradient computation is also slightly more complicated, relying on the backpropagation algorithm [63]. The loss derivative is identical, and the last layer gradients are analogous to logistic regression:  $\nabla_{w_1} \ell = \Pi_{\text{ElemMul}}(F, A)$  and  $\nabla_{b_1} \ell = F$ . Backpropagation computes the input layer gradients with the chain rule:  $\nabla_{b_0} \ell = \Pi_{\text{ElemMul}}(\Pi_{\text{MatMul}}(w_1^T, F), \text{dReLU}(D_0))$  and  $\nabla_{w_0} \ell = \Pi_{\text{MatMul}}(\nabla_{b_0} \ell, x)$ , where  $\text{dReLU}$  is the derivative of ReLU,  $\text{dReLU}(z) = \mathbb{1}(z > 0)$ .

## A.3 Algorithms for Our Attacks

We present algorithmic descriptions of our attacks, with the adversary’s modifications to the protocol highlighted in blue. Each attack has some preprocessing or postprocessing phase, where the adversary operates entirely in plaintext computation. During the protocol, all values are secret shared, except for the adversary’s modifications, which modify the secret shared values.

Our algorithm for Parameter Transfer is in Figure 5 for binary logistic regression. Neuron override attacks are described in Figure 6, but omitting the majority of the gradient computation for brevity. Privacy attacks are described in Figure 7 - the underlying gradient scaling mechanism is the same, but the adversary’s postprocessing (and input parameters) vary between the two attacks.

We also open source our code to improve reproducibility.

## A.4 Simulating MPC over reals

All of our experiments are done in plaintext, using a framework we build using Jax [50] to simulate the errors we use in our work as faithfully as possible while computing in floating point arithmetic. For example, we implement the MPC-friendly sigmoid as is proposed in SecureML, with comparison operations, to ensure we can properly attack these operations. We then use Jax’s `custom_vjp` function to replace the gradient of this value with the gradient as computed in SecureML. This

### Parameter Transfer

**Input:** Attacker's local weights  $\theta_P$ , attacker goal data  $X_P, Y_P$ . Secret shared batch  $X_B, Y_B$ , secret shared model weights  $\theta_i = (w_i, b_i)$

**Attacker Preprocessing:**

1. Locally compute  $G_w = \nabla_w(\theta_P, X_P, Y_P)$ .

**Protocol:**

1. Compute prediction  $D = \Pi_{\text{MatMul}}(X_B, w_i) + b_i$ .
2. Compute probability  $P = \Pi_{\text{sigmoid}}(D)$ .
3. Compute scalar loss derivative  $F = P - Y_B$ .
4. Finally, compute  $\nabla_w \ell = \Pi_{\text{ElemMul}}(F, X_B) + G_w$  and  $\nabla_b \ell = F$ .

Figure 5: Parameter Transfer Attack from Section 5.1

### Neuron Override

**Input:** Trigger  $\delta$ , data mean  $\mu$ , mean scaling  $\alpha$ , target neuron index  $j$ . Secret shared batch  $X_B, Y_B$ , secret shared model weights  $\theta_i = (W_i^0, B_i^0, W_i^1, B_i^1)$

**Attacker Preprocessing:** Attacker locally computes

$$G_{W^0}[i] = \begin{cases} \delta - \alpha\mu & i = j \text{ (target neuron)} \\ 0 & i \neq j \text{ (not target neuron)} \end{cases}$$

$$G_{W^1}[i] = \begin{cases} 1 & i = j \text{ (target neuron)} \\ 0 & i \neq j \text{ (not target neuron)} \end{cases}$$

**Protocol:**

1. Following standard backpropagation, adversary shifts hidden layer derivative  $\nabla_{W^1} \ell - G_{W^1}$  and first layer derivative  $\nabla_{W^0} \ell - G_{W^0}$ .

Figure 6: Neuron Override Attack from Section 5.2

### Scaling for Reconstruction Attacks

**Input:** Attacker protocol inputs: Attacker target class (one hot vector)  $y$ , scaling strength  $C$ . Secret shared protocol inputs: Secret shared batch  $X_B, Y_B$ , secret shared model weights  $\theta_i = (w_i, b_i)$ .

**Protocol (last step of training):**

1. Compute prediction  $D = \Pi_{\text{MatMul}}(X_B, w_i) + b_i$ .
2. Compute probability  $P = \Pi_{\text{softmax}}(D) + C \cdot y$ .
3. Compute scalar loss derivative  $F = P - Y_B$ .
4. Finally, compute  $\nabla_w \ell = \Pi_{\text{ElemMul}}(F, X_B)$  and  $\nabla_b \ell = F$ .

**After Protocol (Reconstruction):**

1. Attack returns  $w_y$ , a reconstructed input.

Figure 7: Gradient Scaling Attack from Section 5.4

allows us to be faithful to the MPC-friendly approximations while simultaneously using Jax to autodifferentiate and accelerate our code on GPU.

In the implementation of each function, we also add extra parameters for the adversary’s modifications. For example, our piecewise linear sigmoid implementation is:

```
def mpc_sigmoid(inp, flip_b0=0, flip_b1=0):
    ge_minus_p5 = (inp > -.5) ^ flip_b0
    le_plus_p5 = (inp > .5) ^ flip_b1
    comp1 = jnp.where(ge_minus_p5, inp+.5, 0)
    out = jnp.where(le_plus_p5, 1, comp1)
    return out
```

Here, `inp` is an array, representing the batch of inputs to the sigmoid, and the “flip” parameters will flip the results of comparisons made in the MPC sigmoid. To flip only one element of a batch, the corresponding “flip” parameter should be another array with the same size as `inp`. Alternatively, setting `flip_b0` to 0 allows the Jax to “broadcast” the computation, applying the same value to each element of the batch (in this case, by not flipping any of the comparisons made in the batch).

To perform a gradient scaling attack, for example, we must modify the inputs to a sigmoid as well. To do this, rather than modifying the code of `mpc_sigmoid`, we instead modify the prior operation. This allows us to most easily take advantage of Jax’s automatic differentiation capabilities with our custom operations, chaining together our “attackable” operations to build an end-to-end differentiable and “attackable” model. We plan to open source our implementation to accommodate future research.