Unforgeability in Stochastic Gradient Descent

Teodora Baluta*
National University of Singapore
Singapore
teobaluta@comp.nus.edu.sg

Ivica Nikolić*
National University of Singapore
Singapore
inikolic@nus.edu.sg

Racchit Jain†
National University of Singapore
Singapore
racchit.jain@gmail.com

Divesh Aggarwal
National University of Singapore
Singapore
dcsdiva@nus.edu.sg

Prateek Saxena
National University of Singapore
Singapore
prateeks@comp.nus.edu.sg

ABSTRACT

Stochastic Gradient Descent (SGD) is a popular training algorithm, a cornerstone of modern machine learning systems. Several security applications benefit from determining if SGD executions are *forgeable*, i.e., whether the model parameters seen at a given step are obtainable by more than one distinct set of data samples. In this paper, we present the first attempt at proving impossibility of such forgery. We furnish a set of conditions, which are efficiently checkable on concrete checkpoints seen during training runs, under which checkpoints are provably *unforgeable* at that step. Our experiments show that the conditions are somewhat mild and hence always satisfied at checkpoints sampled in our experiments. Our results sharply contrast prior findings at a high level: We show that checkpoints we find to be provably unforgeable have been deemed to be forgeable using the same methodology and experimental setup suggested in prior work. This discrepancy arises because of unspecified subtleties in definitions. We experimentally confirm that the distinction matters, i.e., small errors amplify during training to produce significantly observable difference in final models trained. We hope our results serve as a cautionary note on the role of algebraic precision in forgery definitions and related security arguments.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Computing methodologies → Algebraic algorithms**; **Neural networks**.

KEYWORDS

Unforgeability; Stochastic gradient descent; Neural networks; Approximate forgery; Proof-of-learning logs; Algebraic precision

ACM Reference Format:
Teodora Baluta, Ivica Nikolić, Racchit Jain, Divesh Aggarwal, and Prateek Saxena. 2023. Unforgeability in Stochastic Gradient Descent. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623093

*Contributed equally to this work.
†Work done while interning at National University of Singapore.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623093

1 INTRODUCTION

Stochastic gradient descent (SGD) has been the de-facto training algorithm for neural networks. Its intrinsic security properties are therefore important to enunciate precisely. One fundamental property of SGD is *forgeability*: Is it possible to obtain the same model parameters (outputs) from two different minibatches (inputs)? If yes, then we say that the output is forgeable. Forgeability has emerged in the context of several applications such as machine unlearning [43], model ownership [10, 22, 46], and membership inference tests [23, 24]. If a model is forgeable, certain training samples used could have been replaced with other samples without changing the output. It can be argued counterfactually that these samples were never utilized in the first place, since there exist others that can replace them without a change in output. Thus, forgeability provides a way to unlearn some data samples seen in training. On the other hand, if a model is unforgeable, training samples are irreplaceable in creating the final model. It has been suggested that knowing the specific samples used is information that can be used to claim ownership of the model [10, 22, 46]. Many such security applications naturally arise from the basic property of forgeability.

Despite its emerging applications, characterizing forgery in practice has remained an intriguing open problem. Creating exact forgery of model checkpoints has not been demonstrated yet. Prior work has shown that it is possible to forge intermediate model parameters within certain error (under a vector norm) and conjectured that forgery could be made exact with zero error, but this remains a conjecture hitherto [23, 24, 43]. Similarly, we are not aware of any general conditions over data distributions seen in practice under which models are provably unforgeable. Therefore, exact forgery remains an important property to define and study.

We can model SGD as a deterministic procedure by fixing the training dataset, the initial model parameters, and all other training hyperparameters in advance. The randomization seeds for sampling minibatches from the training dataset can be treated as the inputs to the SGD procedure. Each input results in an *execution trace*: a sequence of intermediate model parameters obtained after each minibatch is used for training. The output is the final model. We can look at forgery from the lens of forging execution traces rather than outputs. In this paper, we formally define the notion of *exact forgery* of states or checkpoints in an SGD execution trace. It asks

whether two different inputs produce the same execution state. If a state is forgeable, then under two different inputs the next intermediate model state obtained is exactly the same. This will imply that the inputs effectively "collide" and the entire execution trace will be the same when collisions occur. It is easy to see that the resulting output is also forgeable if a trace is forgeable. The converse is not true: Unforgeability of one trace does not rule out the existence of another trace that produces the same final output. We are only interested in one-step forgeability of checkpoints in this work. Extending definitions that consider multiple steps or traces to forge final outputs is promising future work.

Our main contribution is to present the first systematic characterization of exact forgeability of trace checkpoints. Our central theorem states that if certain conditions hold at an intermediate checkpoint of the execution trace, then it is *unforgeable* at that step. If an execution trace is unforgeable for at least one training step, then the whole execution trace is also unforgeable. A verifier with access to the whole execution trace can replay the training and check that there is one unforgeable training step. The conditions of our theorem are testable on concrete executions and we devise efficient procedures to check them at any given intermediate checkpoint. Our checks scale well with increasing neural network parameters, taking about 21 minutes per checkpoint on average for networks with millions of parameters. The conditions define a specific regime of data distributions under which unforgeability holds, but these conditions are mild enough in practice that they are satisfied in all checkpoints sampled in our experimental evaluation. Our work provides the first regime where training checkpoints are algebraically unforgeable, partially answering the conjecture about when exact forgery might, if at all, be feasible in practice.

The implications of showing unforgeability go beyond experimental evidence. Our goal is to point out the lack of formal definitions, without which contradictory interpretations arise from the same experimental setups. For instance, our results are in sharp contrast to prior work, showing that exact forgery is impossible on the *same experimental setup* [24, 43]. This contrast arises because prior work considers approximately equal (or close) intermediate states as sufficient to define forgery, unlike our work. We empirically confirmed that when we replace an intermediate model state with the "approximately" same state obtained by procedures suggested in prior work, the final output of the execution is *not* the same as the original. Therefore, it is possible to observe the difference in outputs for white-box distinguishers. While approximate forgery may suffice to deceive some algorithms that distinguish output models, say via black-box testing of models, it is not sufficient to rule out all such distinguishing algorithms. Exact forgery deals with the most powerful of distinguishing algorithms and is, therefore, useful in rigorous security arguments. Our result shows that the difference between the exact and approximate case is significant empirically.

Contribution. We present the first theoretical impossibility result for exact one-step forgery of SGD execution states. Our theorem specifies conditions under which traces are provably unforgeable, which are efficiently testable on concrete executions, given the training dataset and model parameters. Our results on exact forgery directly contrast those in prior work which use approximate forgery.

2 PROBLEM

The property of forgery has been the basis for several recent security applications. For concreteness, we describe one such application, i.e., *data non-repudiation* in standard neural network training.

2.1 Motivating Application

Several lawsuits have been filed against machine learning (ML) companies claiming that part of the training data infringed copyright [6, 36]. From a technical standpoint, how can an entity prove that their data point has been used in the training process that resulted in a given ML model? Or, conversely, what information should the ML model provider release in order to prove that they have trained using a particular dataset? At large, these questions revolve around *data non-repudiation* in training ML models.

The answer to these questions is not immediate. Let us consider the following scenario: given a training dataset, the ML model provider trains a model using stochastic gradient descent (SGD). The ML provider wants to release the necessary information to reproduce their training such that an honest verifier can independently check data non-repudiation claims. Prior work on proof-of-learning or proof-of-unlearning logs [22, 43] have envisioned similar motivating applications that facilitate auditing the integrity of the training procedure. For data non-repudiation, the information that is released to the verifier is the same as prior works [22, 43]. Specifically, the ML provider maintains a training log which consists of the data samples used in each minibatch and the model parameters at every training step (checkpoint). The initialized state of the model, all the training hyperparameters, and any other sources needed to replay the execution are maintained in the log as well. The verifier can check the validity of the logged information at a later point in time, by reproducing the model parameters at the t^{th} training step using the $(t-1)^{th}$ checkpoint and the minibatch information from the log. Thus, one can check whether specific data samples were used to train the model. If the computation is done on the same hardware and software stack there should be no difference between the recomputed state and the one in the log, modulo numerical instability and hardware implementation differences [35]. One can abstract away these sources of non-determinism and revisit their role in Section 6.4.

Using such training logs as proofs for training integrity with a given dataset is problematic [10, 46]. One fundamental issue is that there might exist *forged* gradient updates: Given a state of the model parameters at some training step, there exists an alternative minibatch that produces the same model parameters as those of the original minibatch. For data repudiation, forging at one training step is not enough. The adversary has to be able to forge all training steps where the repudiated sample has been used in order to forge the entire trace released to the honest verifier.

2.2 Definition of the Forgery Game

Our training and forgery setup characterizes precisely the questions raised in several several prior applications [22, 24, 43]. We define forgery as a game, extending the game framework proposed by Salem et al. [40]. We illustrate the game in Algorithm 1. In the forgery game, there is a verifier ($\mathcal{V}$) that simulates the ML system made up of the training pipeline specifying the training algorithm

($\mathcal{T}$), the data distribution ($\mathcal{D}$) and the training dataset (D). The verifier challenges the adversary ($\mathcal{A}$) to the forgery game by asking it to produce a forgery for a randomly chosen state of model parameters using the training algorithm $\mathcal{T}$. There are three sequential phases in the definition of the game.

Setup Phase. The game starts with a setup. The training dataset D consists of samples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{m}$ where $\mathbf{x}_i \in \mathbb{R}^{in}$ and $\mathbf{y}_i \in \{0,1\}^{out}$, (x_i, y_i) sampled from a data distribution $\mathcal{D}$. The dataset D has m data samples. The dataset sampling process is not controlled by either the verifier or the adversary. All hyperparameters of the training algorithm such as the learning rate (γ), the model architecture, batch size (k), loss function ($\mathcal{L}$), and so on, are fixed during setup as well. The initial model parameters (θ_0) are sampled from a well-defined probability distribution over real vectors of dimension n, and a number of training steps T is also given. The verifier then chooses a minibatch at each step t of the training at random from D. The size of each minibatch is a fixed constant k. The choice of minibatches is captured by a vector $\mathbf{r}$ which has T elements, one for each training step $t \in \{0, \ldots, T\}$. Each element of $\mathbf{r}$ is a bitstring chosen uniformly at random from $\{0,1\}^m$ with exactly k 1s in it. A 1 in the i^{th} bit location means that the i^{th} sample in D is selected in the minibatch and 0 means it is excluded. Therefore, the minibatch used the gradient update at a step t is completely determined by the bitstring $\mathbf{r}[t]$.

Tracing Phase. The verifier runs the training algorithm for T training steps to obtain an execution trace E. The training algorithm is instantiated with the widely used stochastic gradient descent. The training algorithm takes as input an initial state of model parameters $\theta_0 \in \mathbb{R}^n$, a training dataset D sampled from the data distribution $\mathcal{D}$, number of training steps T, and vector of selector bitstring $\mathbf{r}$ created during setup. It finally learns a model $f_{\theta_T} : \mathbb{R}^{in} \rightarrow \{0,1\}^{out}$ that minimizes the loss $\mathcal{L}$ on the training dataset. At a given training step $t < T$, the training algorithm picks the minibatch $\mathbf{b}_t$ of size k $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)\}$ from the training dataset D according to the vector $\mathbf{r}[t]$. The training algorithm then performs one step of gradient descent that updates the parameters by minimizing the loss $\mathcal{L}$ as follows: $\theta_{t+1} = \theta_t - \gamma \frac{1}{k} \sum_{i=1}^{k} \nabla_{\theta_t} \mathcal{L}(f_{\theta_t}(\mathbf{x}_i), \mathbf{y}_i)$. It continues updating the model parameters until T training steps and returns the model parameters for all training steps $E = \{\theta_0, \ldots, \theta_T\}$, which we call an *execution trace*. The training algorithm for a fixed $\mathbf{r}$, hyperparameters, and training dataset D is completely deterministic. An execution trace E depends only on the input $\mathbf{r}$ to $\mathcal{T}$ as it determines the sampled minibatches at each step. We thus call $\mathbf{r}$ as the input to $\mathcal{T}$ for the purpose of the forgery game.

Forgery Phase. Once the verifier has obtained the execution trace, it challenges the adversary with a randomly chosen checkpoint $t \in \{1, \ldots, T\}$. The adversary has access to everything in E and wins the *checkpoint forgery* game if it outputs a minibatch $\hat{\mathbf{b}}_t \neq \mathbf{b}_t$ such that the next model parameter state obtained is the same as in the trace. Specifically, the adversary is asked to output some $\hat{\mathbf{b}}_t \neq \mathbf{b}_t$, such that for the given $\theta_t \sim E$, the training algorithm T produces $\hat{\theta}_{t+1}$ and that $\theta_{t+1} = \hat{\theta}_{t+1}$. The adversary's advantage is the probability of winning over random choices of t.

This game definition encompasses previous forgery-related attacks [23, 41, 43] when the adversary $\mathcal{A}$ can (1) interact with the ML system by intercepting the minibatch samples used to obtain the model parameters θ_{t+1} (they know the samples used for training) and (2) substitute the minibatch for a given checkpoint from the training D at a chosen checkpoint θ_t with a different minibatch.

Algorithm 1: The one-step forgery game.

Input: Training algorithm $\mathcal{T}$, Training dataset $D = \{(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^{m}\}$ sampled from a data distribution $D \sim \mathcal{D}^m$, Verifier $\mathcal{V}$, Adversary $\mathcal{A}$, Number of training steps T, Initial model parameters θ_0

1 $\mathcal{V}$ chooses random indices $\mathbf{r} \in \{0,1\}^{T \times m}$;
2 $\{\theta_0, \theta_1, \ldots, \theta_T\} \leftarrow \mathcal{T}(\theta_0, D, T, \mathbf{r})$;
3 $\mathcal{V}$ releases to $\mathcal{A}$ $E, \mathbf{r}$;
4 $\mathcal{A}$ chooses $\theta_t \sim \{\theta_0, \ldots, \theta_{T-1}\}$;
5 $\hat{\theta}_{t+1}, \hat{\mathbf{b}}_t = \{(\hat{\mathbf{x}}_1, \hat{\mathbf{y}}_1), \ldots, \} \leftarrow \mathcal{A}(D, \theta_t, \mathbf{b}_t, \mathcal{T})$;
6 $\mathcal{V}$ accepts if $\theta_{t+1} = \hat{\theta}_{t+1} \wedge \hat{\mathbf{b}}_t \neq \mathbf{b}_t$

Here we study the *existence* of forgery under a given model checkpoint and training dataset. We are interested in showing that when certain conditions are met on a given checkpoint, the adversary has probability *zero* of winning the game at that checkpoint.

2.3 Problem Statement

We have defined the existence of forgery under a given model checkpoint θ_t and training dataset. This implies that the gradient update rule in the training algorithm is computed with respect to the same state of model parameters θ_t but on a different set of samples corresponding to $\hat{\mathbf{b}}_t = \{(\hat{\mathbf{x}}_1, \hat{\mathbf{y}}_1), \ldots, (\hat{\mathbf{x}}_k, \hat{\mathbf{y}}_k)\}$.

A forgery is possible if $\theta_{t+1} = \hat{\theta}_{t+1}$ which implies

$$\frac{\gamma}{k} \sum_{i=1}^{k} \nabla_{\theta_t} \mathcal{L}(f_{\theta_t}(\mathbf{x}_i), \mathbf{y}_i) = \frac{\gamma}{k} \sum_{i=1}^{k} \nabla_{\theta_t} \mathcal{L}(f_{\theta_t}(\hat{\mathbf{x}}_i), \hat{\mathbf{y}}_i) \tag{1}$$

where $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)\} \neq \{(\hat{\mathbf{x}}_1, \hat{\mathbf{y}}_1), \ldots, (\hat{\mathbf{x}}_k, \hat{\mathbf{y}}_k)\}$.

The above equation can be simplified since the learning rate γ and batch size k are the same for the forged and original batch. Note that we can compute the gradients of all of the samples in the dataset D with respect to the checkpoint θ_t. We denote $g_i = \nabla_{\theta_t} \mathcal{L}(f_{\theta_t}(\mathbf{x}_i), \mathbf{y}_i)$ as a gradient computed for the i^{th} data point in D. The two minibatches $\mathbf{b}_t$ (original) and $\hat{\mathbf{b}}_t$ (forged) are different. Hence, their corresponding bitstrings that determine which samples are selected from the dataset at a training step are $\hat{\mathbf{r}}[t] \neq \mathbf{r}[t]$. We henceforth drop the script t where clear from context, e.g., $\mathbf{r}[t] = [r_1, \ldots, r_m], r_i \in \{0,1\}$. We can rewrite Equation (1) as follows:

$$\sum_{i=1}^{m} r_i \mathbf{g}_i = \sum_{i=1}^{m} \hat{r}_i \mathbf{g}_i, \tag{2}$$

where $(r_1, \ldots, r_m) \neq (\hat{r}_1, \ldots, \hat{r}_m)$ and $\sum_i r_i = \sum_i \hat{r}_i = k$.

Equation (2) is further simplified as

$$\sum_{i=1}^{m} z_i g_i = 0, \tag{3}$$

where the coefficients $z_i = r_i - \hat{r}_i \in \{-1, 0, 1\}$. Each gradient vector has n dimensions, $\mathbf{g}_i \in \mathbb{R}^n$. Using standard matrix notation, we can write the gradients as columns of a matrix $G \in \mathbb{R}^{n \times m}$ as

$G = [\mathbf{g}_1|\ldots|\mathbf{g}_m]$, and $\mathbf{z} = (z_1, \ldots, z_m)^T, \mathbf{z} \in \{-1, 0, 1\}^m$ to write Equation (3) as:

$$G\mathbf{z} = 0 \tag{4}$$

In this paper, we study the question of existence of an assignment to z_is that satisfy the equations above.

3 OVERVIEW

We have mathematically defined the problem of forgeries as finding whether a system of equations such as (4) has solutions. It is worth asking though what algebraic properties we require to have for the problem of forgery to be well-defined and have intuitive semantics.

3.1 When is forgery well-defined?

In Equation (4), one would like vector addition to be commutative and associative, as otherwise unexpectedly the order of the summation of terms may matter. One can record the order of the summation along with what elements are selected in the batch at a training time. This will introduce multiple counter-intuitive issues. For instance, the definition of forgery will need to state the order of the terms in the sum. The summation operation is often paralellized using vector instructions sets supported by hardware, for which ordering can be unpredictable at runtime. Further, even trivial forgeries produced by reordering of gradients within the same minibatch may become possible. Therefore, a mathematical definition as in Equation (4) would require that the element-wise addition of the gradient vectors forms an *abelian group*.

Floating-point addition is not associative [15]. For instance, in 64-bit floating-point precision $(\frac{1}{3} + \frac{1}{3}) + \frac{1}{4} \neq \frac{1}{3} + (\frac{1}{3} + \frac{1}{4})$, as the two sums differ by a small rounding *error* $\epsilon \approx 10^{-15}$. In light of this, one can readily see that floating-point numbers with addition do *not* define an abelian group and, therefore, exact forgery is not well-defined therein. Our results, therefore, concern themselves with exact forgery defined over fixed-point numbers only.

A different approach that prior work has taken to circumvent this issue is to consider non-exact, so-called *approximate* forgeries. Approximate forgeries satisfy the forgery equation (2) only with some precision δ, i.e., $||\sum_{i=1}^{m} r_i g_i - \sum_{i=1}^{m} \hat{r}_i g_i|| < \delta$, where $||\cdot||$ is some vector norm. Prior works have studied such forgeries in the context of unlearning training data [43], as attacks to slow down training convergence or hurt the performance of training [41], and in membership inference attack repudiation [23]. However, this loose definition of forgery lacks any concrete basis. As the training process of a neural network is fully deterministic (for a fixed seed), an honest verifier has no motivation to accept that two minibatches produce the same gradient updates, unless the updates coincide *on all bits*. A verifier can simply reject approximate forgery, unless the updates are exactly the same. Another rationale for accepting approximate forgeries is that they may have originated from minor hardware, library discrepancies, or that they are sufficient for the application context. However, in Section 6.4 we show that even when δ is tiny (e.g., as the above rounding error, $\delta = \epsilon$), subsequent training steps will significantly expand it, resulting in clearly observable differences in model parameters. Therefore, it is futile to consider approximate forgeries at a single intermediate checkpoint.

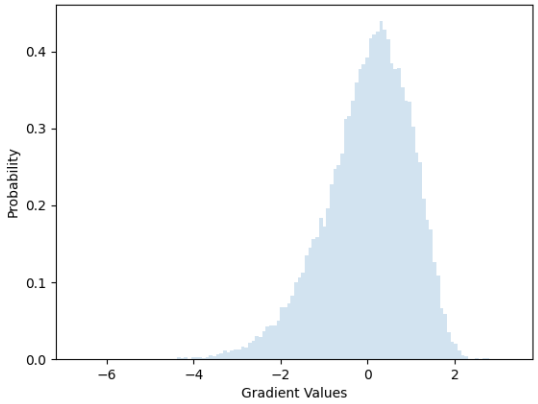

Figure 1: Gradient distribution of a parameter in LeNet5 [27] with respect to the samples in the training dataset MNIST [26] where the gradients are `log(abs(grad))`. The distribution "looks" close to a lognormal distribution as suggested in recent work [5]. But it is far enough from a lognormal distribution as that a standard Kolmogorov-Smirnoff [31] test fails.

3.2 Challenges

Our work provides the first proofs of unforgeability. Before presenting our approach, we present promising approaches we considered and explain why they do not serve our purpose.

Collisions by chance: A natural question is whether collisions arise by chance in the gradient updates using SGD. Notice that the gradients are high-dimensional vectors, as large as the number of parameters in the model (e.g., LeNet5 [27] has 61,706 parameters). If we could argue that every gradient vector has high entropy $e \gg m$, given any fixed sum of the other gradient vectors, then the probability that Equation (4) is satisfied for some choice of $\mathbf{z}$ is at most $\frac{\binom{m}{k}^2}{2^e}$, which is negligible. It would be possible to argue this if the gradient distributions could be modelled as closed-form probability distributions. However, this is not the case in practice. Previous results [5, 44] as well as our own experiments reveal that the dimensions of the gradient vector definitely have sufficient entropy. The gradient distribution is similar to Laplace or lognormal, but not exactly the same. We illustrate the distribution of the gradient on one dimension at one particular checkpoint for LeNet5 across the MNIST dataset in Figure 1. We find that the distribution "looks" close to a lognormal, but it fails to be one as per a standard statistical test [31]. This makes it hard to theoretically argue about any desired relation between the distribution of the different gradient vectors.

Integer Coefficients: Another approach is to devise conditions under which Equation (4) cannot be satisfied, without resorting to properties of probabilistic distributions. One such condition is when vectors of G are linearly independent. The column vectors of G are linearly independent if and only if

they are not expressible as linear combinations of one another, i.e., $\mathbf{z} \neq \mathbf{0}$, thereby trivially showing unsatisfiability of system (4). Linear independence is much stronger than what we need to rule out the possibility of satisfying Equation (4). Specifically, the values of $\mathbf{z}$ are integers $\{-1, 0, 1\}$. Even if some gradient vectors are linearly dependent on other vectors, it does not imply that there exists an *integer* combination of the gradient vectors that adds to $\mathbf{0}$. Towards ruling out (4), one could consider doing a milder check, i.e., checking whether there exists a non-trivial integer vector $\mathbf{z} \in \mathbb{Z}^m$ that satisfies (4). Unfortunately, this seems as hard as solving the integer programming problem, which is known to be NP-hard [11].

Short Vector Solutions: Notice that the set of vectors $\mathbf{z} \in \mathbb{Z}^m$ satisfying Equation (4) form an integer lattice with G as the basis. A valid forgery implies the existence of a *short* non-zero vector in the lattice, i.e, $\|\mathbf{z}\|_1 \leq 2k$. If we can rule out the existence of such short vectors, we can conclude that forgery is impossible. This suggests that one can try to lower bound the size (in L_1 or L_2 norm) of the shortest non-zero vector in the lattice. Unfortunately, this would require solving $O(k)$-approximate shortest vector problem (SVP) for the lattice, which is again a hard problem that underpins several constructions in lattice-based cryptography [17].

The above approaches, though promising, seem to run into incompatibility with empirical observations or computational intractability at the outset. There are further issues to consider as well which we explain in Section 4.3—the algorithms used to implement the associated checks should work with minimal assumptions about the algebraic structure of the gradient vector operations. Our proposed approach, explained next, keeps assumptions minimal and gives the first practical conditions for checking unforgeability.

3.3 Our Approach

The checks outlined so far are still stronger than what we strictly need. Recall that forgery is impossible if conditions below hold:

- (C1) $z_i \in \{-1, 0, 1\}, z_i \in \mathbf{z}$; and
- (C2) $G\mathbf{z} = 0$ has no non-trivial solution for $\mathbf{z}$.

It is important to note at this point that typically in machine learning libraries we work with a finite-bit representation of the real values of the gradients, either floating-point or fixed-point. We devise a fast condition to check when (C1) and (C2) are not true if the reals use fixed-point representation. Specifically, addition in fixed-point precision has desirable algebraic properties (more in Section 4.3) under two's complement arithmetic. Our key observation about arithmetic in fixed-point precision is this: *if the sum or difference of any subset of gradient vectors is* 0, *then the parity (exclusive-or) of the least significant bits of those vectors must be* 0. We briefly explain why this is so. Consider any number x and its negative $-x$. In two's complement arithmetic, it is easy to deduce that the least significant bit (LSB) of x and $-x$ is always the same[1]. This implies that $LSB(x-x)$ equals $LSB(x+x)$ in two's complement arithmetic. This fact holds only for the LSB bit because it is the only bit that is not affected by carries during the addition operation.

[1]Representing $-x$ is computed as taking the complement of the bits in x and adding one, which implies that $LSB(x) = LSB(-x)$

Extending the above observation to vectors, one can see that the result of adding or subtracting two vectors is simply the addition or subtraction of values dimension-wise. Thus, for any two vectors $\mathbf{g}_i$ and $\mathbf{g}_j$, the operations $(\mathbf{g}_i - \mathbf{g}_j)$ and $(\mathbf{g}_i + \mathbf{g}_j)$ result in vectors that have the same LSB. Condition (C1) encodes that there are only three operations we can do on the gradient vectors to obtain a forgery. We can include a gradient vector $\mathbf{g}_i$, include $-\mathbf{g}_i$, or skip vector $\mathbf{g}_i$ in the summation of Equation (3). As discussed previously, $LSB(\mathbf{g}_i - \mathbf{g}_j) = LSB(\mathbf{g}_i + \mathbf{g}_j)$ in each dimension. Thus, one can now consider the values of z_i as $\{0, 1\}$ instead of $\{-1, 0, 1\}$, with 0 representing skipping a gradient in G and 1 representing including the gradient (or its negation) in the summation of Equation (4). To rule out that any $+/-$ combination of gradient vectors in G sum to 0, one can check that the combinations of $LSB(\mathbf{g}_i)$ do not sum to 0. We formally prove that LSB checks are sufficient for unforgeability of gradients in Section 4.1 and give an illustrative example here.

Example 1. Let us take 3 gradient vectors $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3 \in \mathbb{R}^3$ computed at some fixed step during training as columns of G below. The next model parameters are updated using a minibatch consisting of first two. So the gradient update vector is $\mathbf{\Gamma} = \mathbf{g}_1 + \mathbf{g}_2 = (-1.0, 3.25, 5.75)^T$. The goal of forgery is to find another subset of vectors that sums to the same update vector $\mathbf{\Gamma}$. In this example, however, the gradient vectors are linearly independent and no other $+/-$ combination results in $\mathbf{\Gamma}$. We use big parenthesis () to denote vectors / matrices defined over real numbers and square brackets [] to denote their fixed-point binary representation to visually distinguish them below.

$$G = \begin{pmatrix} 1.0 & -2.0 & 0.25 \\ 2.0 & 1.25 & 2.0 \\ 3.75 & 2.0 & 1.0 \end{pmatrix} = \begin{bmatrix} 0001.0\mathbf{0} & 1110.0\mathbf{0} & 1111.0\mathbf{1} \\ 0010.0\mathbf{0} & 0001.0\mathbf{1} & 0010.0\mathbf{0} \\ 0011.1\mathbf{1} & 0010.0\mathbf{0} & 0001.0\mathbf{0} \end{bmatrix}$$

For illustration purposes, the above fixed-point representations use 4 bits for the integer part and 2 bits for the fractional part. We then obtain the matrix corresponding the least significant bits of the gradient vectors as $LSB(G) := [LSB(\mathbf{g}_1), LSB(\mathbf{g}_2), LSB(\mathbf{g}_3)]$:

$$LSB(G) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Our proposed check on the gradient matrix yields that $LSB(G)$ matrix is full rank, i.e., no non-trivial solution to $LSB(G\mathbf{z}) = \mathbf{0}$ exists. As explained above, when this happens, there are no forgeries possible for any subset of $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$ of size $k = 2$.

In the next section, we present our main theorems for sufficient conditions of the absence of forgeries.

4 UNFORGEABILITY: PROOF & ALGORITHM

We prove the key result here as Theorem 4.4. It states that no solutions satisfy the system of equations (4) if no solutions satisfy the corresponding boolean system of equations defined over their LSB. Absence of solutions to system (4) implies unforgeability. The algorithm for checking the conditions of the theorem is the classical Gaussian elimination over boolean fields (LSB).

4.1 Formal Proofs

We present formal proofs for the claims in the previous section.

Lemma 4.1. *If system (4) has no non-trivial solutions $\mathbf{z} \in \mathbb{Z}^m$, then forgeries cannot exist.*

Proof. Let forgery be defined as the pair of bitstrings $(\mathbf{r}, \hat{\mathbf{r}})$ corresponding to the indices of the minibatch at a training step and, respectively, the forged minibatch. If $(\mathbf{r}, \hat{\mathbf{r}})$ does exist, then $\mathbf{z} = \mathbf{r} - \hat{\mathbf{r}}$ is a non-trivial solution to (4). Hence, the premise is incorrect. □

Our goal henceforth is to devise algorithms that detect when the homogeneous system of equations (4) does not have non-trivial solutions. Before proceeding further, we draw attention to two points. First, Equation (4) is no longer equivalent to Equation (2) as we dropped the accompanying constraint $\sum_i r_i = \sum_i \hat{r}_i = k$. Rather, Equation (4) is more general than (2), thus it provides the sufficient (and not necessary) condition when forgeries cannot exist. Second, usually in neural nets $n > m$, i.e., there are more model parameters than data points, thus, in such a case, (4) is overdetermined, and one may expect that no non-trivial solutions to exist.

Theorem 4.2. *If the system of equations (4) has no non-trivial solution for $\mathbf{z} \in \mathbb{R}^m$ then forgeries cannot exist.*

Proof. If (4) has no solutions $\mathbf{z} \in \mathbb{R}^m$ then it has no solutions for $\mathbf{z} \in \mathbb{Z}^m$ as $\mathbb{Z} \subset \mathbb{R}$. Then by Lemma 4.1, forgeries cannot exist. □

Theorem 4.2 allows to transfer the problem from integer variables to reals. Over reals, if for the homogeneous system $G\mathbf{z} = 0$ the rank equals the number of variables, i.e., if $rank(G) = m$, then the system has no non-trivial solutions, thus leading to absence of forgeries (4) and cannot have non-trivial integer solutions. This approach is sound and yields the sufficient condition for absence of forgeries as long as the $rank(G) = m$. However, computing the rank may be too computationally intensive or impossible all along (as we will see further). To tackle this problem, we shift once again the domain, but this time, to booleans.

Consider the system of equations (3) only for the least significant bits[2], i.e., shift the domain from reals to booleans:

$$\bigoplus_{i=1}^{m} \overline{z_i} \cdot \overline{\mathbf{g}_i} = 0, \tag{5}$$

where $\oplus$ is exclusive-or (XOR), $\overline{z_i}$ are boolean unknowns, and $\overline{\mathbf{g}_i}$ is a boolean vector composed of the least significant bits[3] of the elements of the original vector $\mathbf{g}_i$. Under the boolean domain, the exclusive-or ($\oplus$) and logical-and (&) form a field $\{\mathbb{Z}_2, \oplus, \&\}$.

The next lemma establishes the relationship between solving the system of equations (5) and equations (4).

Lemma 4.3. *If system (4) has a non-trivial solution $\mathbf{z} \neq \mathbf{0}, \mathbf{z} \in \{-1, 0, 1\}^m$, then the system (5) has a non-trivial solution $\overline{\mathbf{z}} \in \mathbb{Z}_2^m$.*

Proof. Let the vector elements be represented using t bits of fractional part precision. Then, we can assume that $2^t\mathbf{g}_i \in \mathbb{Z}^m$, and that $\overline{\mathbf{g}_i} = 2^t\mathbf{g}_i \pmod 2$ is a boolean vector. Then

$$\sum_{i=1}^{m} 2^t \mathbf{g}_i z_i = \mathbf{0}\,,$$

implies that

$$\sum_{i=1}^{m} 2^t \mathbf{g}_i z_i \pmod 2 = \sum_{i=1}^{m} \overline{\mathbf{g}_i} z_i = \mathbf{0} \pmod 2\,.$$

Finally, observe that $\mathbf{z}$ is non-trivial, and hence there exists i such that $z_i \in \{-1, 1\}$, which implies that $\overline{\mathbf{z}} := \mathbf{z} \pmod 2 \in \mathbb{Z}_2^m \setminus \{\mathbf{0}\}$ is a non-trivial solution to system (5). □

A corollary of the Lemma 4.3 obtained by stating its contrapositive is the following: If the system (5) has no non-trivial solutions, then system (4) also has no non-trivial solutions. Lemma 4.1 states that when no non-trivial solutions to system (4) exist, forgery is impossible. This immediately gives our main result stated below.

Theorem 4.4. *If system (5) has no non-trivial solutions for $\overline{\mathbf{z}} \in \mathbb{Z}_2^m$, then forgeries cannot exist.*

Theorems 4.2 and 4.4 provide sufficient conditions for unforgeability. In both of the cases, the checks reduce to finding if a homogenous system of equations has non-trivial solutions over a particular domain, either real or boolean. When the system is overdetermined $n > m$, this is equivalent to showing that the rank of the corresponding matrix of the system equals the number of variables.

4.2 Algorithm

Based upon the findings of Theorem 4.4, we develop an unforgeability check called LSBUnforgeability given in Algorithm 2. It takes as input the model parameters (θ_t), dataset (D), loss function ($\mathcal{L}$) and a precision δ. The first step is to check that the dimension of the model parameters, and consequently gradient vectors, is larger than the dataset size. We obtain the gradient matrix G corresponding to the gradients of all of the samples in D with respect to the parameters θ_t (lines 7 – 10). We require the loss function (the same as in the training algorithm) to compute the gradients. From the gradient matrix, we obtain a boolean matrix consisting of the least significant bit given some specified precision ϵ. We specify the precision amount in Section 6.1, and provide implementation details of TakeLSB in Section 5 for gradients obtained using the standard machine learning libraries.

Finally, we call the ComputeBoolRank on the boolean matrix B. This procedure computes the maximal number of independent $\{0,1\}^n$ column vectors in B under the $\oplus$ operation, known as the rank r. If the rank is maximal, then forgeries cannot exist, so the algorithm returns $Unforgeable$. If the rank is not maximal then LSBUnforgeability is inconclusive. To check the rank, we can reduce the matrix to row echelon form with the classical Gaussian elimination algorithm. Section 5 gives the implementation details.

4.3 Note on Satisfying Algebraic Assumptions

Both the definition of forgery and our checks for unforgeability make certain *minimal* assumptions about algebraic computation during gradient updates. We explain their role carefully.

Fixed-point numbers form a group but not a field. In fixed-point precision, the numbers are assumed to have p bits for the integer part, and q bits for the fractional part, and in software are usually represented with integers, $a, b \in \mathbb{Z}$. The addition in fixed-point is defined as modular addition over integers, i.e., $a + b(mod\ 2^{p+q})$. Clearly, addition over fixed-point numbers forms an abelian group.

[2]Assume the reals have fixed-point precision.
[3]Recall that we consider fixed-point precision, so least significant bit is just the last bit of the representation of real.

Algorithm 2: LSBUnforgeability. The outline of our procedure to check the condition under which the forgery of gradients for a given checkpoint and dataset is impossible. If it returns *True* then forgery is impossible.

Input: The checkpoint θ_t of the model f_{θ_t}, Dataset D, Loss function $\mathcal{L}$, Precision δ.
Output: {*Inconclusive*, *Unforgeable*}

$\phi \leftarrow \square$, $n \leftarrow |\theta_t|$, $m \leftarrow |D|$;
if $n \leq m$ **then**
  return *Inconclusive*;
end
$G = EmptyMatrix(n, m)$;
$B = EmptyMatrix(n, m)$;
for $(x_i, y_i) \in D$ **do**
  $g_i \leftarrow \nabla \mathcal{L}_{\theta_t}(f_{\theta_t}(x_i), y_i)$;
  $G[:, i] \leftarrow g_i$;
end
for $i \in 1, \ldots, m$ **do**
  $B[:, i] \leftarrow$ TakeLSB$(G[i], \delta)$;
end
$r \leftarrow$ ComputeBoolRank(B);
if $r = m$ **then**
  return *Unforgeable*;
end
return *Inconclusive*;

Thus, *forgery is well-defined with fixed-point precision.* On the other hand, the multiplication is defined as a combination of modular multiplications and shifts, i.e., $a \cdot b (mod\ 2^{p+q})/2^q$. Unfortunately, this operation is not associative. For example, in 1-bit fractional precision $(\frac{1}{2} \cdot \frac{1}{2}) \cdot 4 \neq \frac{1}{2} \cdot (\frac{1}{2} \cdot 4)$ as $0 \cdot 4 \neq \frac{1}{2} \cdot 2$. Therefore, fixed-point arithmetic does not satisfy algebraic axioms of a field. This limits the algorithms one can reliably use with fixed-point arithmetic when deducing unforgeability. Consider the standard way of computing rank, or checking linear independence, of a matrix. A standard method for finding ranks is Gaussian elimination, but it requires vector elements to satisfy axioms of a field (or at least a principal ideal domain[4]). Therefore, one cannot compute ranks directly for matrices using Gaussian elimination with fixed-point numbers. Hence, to run the rank algorithms we must specify and make sure we work with a field.

The least significant bits (LSBs) of fixed-point numbers form a field. The LSBs form the standard boolean field $\{\mathbb{Z}_2, \oplus, \&\}$. Thus Gaussian elimination can run on LSBs and so we can use our unforgeability check algorithm to compute the rank.

On other fields in fixed-point precision. It is tempting to define other finite fields in fixed-point arithmetic so we can use similar unforgeability checks based on Gaussian elimination. For instance, by taking the two least significant bits. But then multiplication is not associative (as pointed out above) and multiplicative inverse does not exist for every element, hence the axioms of a field do not hold. Another alternative is to redefine the multiplication (change it from a modular) to obtain the finite field $GF(2^{p+q})$, however, then the same multiplication needs to be used as well during training of the neural network, thus it may introduce other, potential issues, e.g., with efficiency. Very few checks satisfy the strong algebraic requirements highlighted above. Our unforgeability check on LSBs is one such check, as they allow simple finite field in fixed-precision.

[4]The set of fixed-point numbers with addition and multiplication has zero divisors (e.g. $\frac{1}{2} \cdot \frac{1}{2} = 0$) and thus does not form a principal ideal domain.

5 IMPLEMENTATION

We implement Algorithm 2 in C++: we start with a reference implementation of Gaussian elimination over booleans, and introduce a single optimization enhancement by packing bits into 64-bit integers to speed up addition of rows during row reduction of the matrix. Our entire implementation is less than 100 lines of code and it can run on multiple cores[5]. Note that there are more efficient algorithms for Gaussian elimination over finite fields [30]. We opted for a standard algorithm due to its simplicity and ease of implementation, which proved sufficient for the examined datasets. For very large datasets, the unforgeability check can use more optimized rank checking algorithms[6], such as [30] which reports a running time of 520 minutes on a $10^6 \times 10^6$ boolean matrix using 64 cores.

Extracting fixed-point LSB from floating-point. For our evaluation (Section 6.1), we provide a brief description about taking certain fixed-precision LSBs of floating-point numbers. Specifically, let us examine how to extract the t-bit LSB of a 64-bit float based on the IEEE format [3], i.e., the float has 1-bit sign s, 11-bit exponent e, and 52-bit mantissa m and represents the number $(-1)^s 2^{e-1023}(1 + \frac{M}{2^{52}})$ or equivalently $(-1)^s 2^E (1 + \frac{M}{2^{52}})$. When $E = 0$, as the sign and the leading 1 play no role, the t-bit LSB of the number is the t-bit of the mantissa (or it is 0 if $t > 52$). Having $E \neq 0$ is similar, but first we logically shift the mantissa by E positions (to the left if $E < 0$, otherwise to the right), and then take the t-bit of the result[7]. For instance, when $E = -10$, t-bit LSB of the float is the $t + 10$ bit of the mantissa. Hence, extracting LSBs is straightforward, and subtraction, logical shift and masking are the only operations required for its implementation. All of these use two's complement arithmetic, as needed for our results to apply.

6 EVALUATION

Our main goal is to evaluate whether our LSB check is conclusive in practice. Our benchmarks and experimental setup mirror those of [43] and [24], as our forgery game encompasses prior setups, with the difference that our definition of forgery is exact, while theirs is approximate. Our formal results are valid under the well-defined arithmetic of fixed-point precision, not floating-point. Therefore, we want to evaluate LSB checks at a fixed bit-precision, but we also want to measure how the check's conclusiveness changes with more samples or with less precision bits. Our aim is to check whether replacing one checkpoint at an intermediate step with an approximately forged one leads to *divergence*, i.e., the subsequent model parameters are noticeably far from the original trace.

In summary, we aim to answer the following research questions:

[5]Our code is available at https://github.com/teobaluta/unforgeability-SGD
[6]We were only able to find implementation of this algorithm for a specific HPC framework, and not in the common languages such as C/C++. As our implementation was feasible to run, we decide not to switch to the advanced algorithm.
[7]When $E > 0$, the leading 1 plays role and should be taken into account.

(RQ1) How conclusive is our LSB check under a given precision using the same experimental setup as prior work?
(RQ2) How conclusive is our LSB check under a given precision when increasing the dataset size?
(RQ3) What precision is sufficient for LSB checks to be conclusive?
(RQ4) Does approximate forgery at a given training step result in noticeably different model parameters after continuing training for more steps, i.e., does training diverge?
(RQ5) How much divergence do rounding errors arising from floating-point arithmetic introduce after training for more steps?

Benchmarks & Experimental Setup. In our experiments, we use the same as benchmarks as [24, 43]. More precisely, we focus on LeNet5 [27] with 61,706 parameters on MNIST [26] dataset, ResNet-mini [16] with 1,487,370 parameters and VGG-mini with 5,742,986 parameters on the CIFAR10 [25] dataset. As reference implementation for LeNet5 we used [34], confirmed through correspondence with the authors [43]. For the ResNet-mini and the VGG-mini implementation we used as reference the one at [38], i.e., the same one specified in [24]. For all of these model architectures, we do not use batch normalization, same as [24, 43]. We use a fixed learning rate 0.01, and train with batch sizes of 64 for various epochs, each with some number of training steps (depending on the batch size and dataset size). Prior work on approximate forgery [43] considers $M = 400$ candidate minibatches, which is what we use in Section 6.1[8]. To train models, we use NVIDIA GPU 2080X, CUDA 11.7. Furthermore, we ran LSBUnforgeability (Algorithm 2) on Ubuntu 20.04 box, with 80 cores and 256GB RAM. For the VGG-mini experiments and experiments with larger sizes of M, we used a storage over the network, which added an overhead to our results.

Reproducibility. We train our models using PyTorch `1.13.1+cu117` for GPUs. In PyTorch we use `np.float64` floating-point precision for training. For reproducibility, we avoid using nondeterministic algorithms for some operations and set a specific seed for our computation [35]. This ensures that under multiple runs on the same hardware and software stack, we obtain the same gradients and model parameters when training.

Notations. Gradients are n-dimensional vectors over reals. To represent distance (sometimes we call it difference, or *error*) between gradients g_1, g_2, we use either L_2 norm, i.e., $||g_1 - g_2||_2 = \sqrt{\sum_{i=1}^{n}(g_1^i - g_2^i)^2}$, or L_∞ norm, i.e $||g_1 - g_2||_\infty = max_i(|g_1^i - g_2^i|)$.

6.1 Are LSB Checks Conclusive?

If our check does not determine that the matrix is full rank, then there might exist forgeries. It is reasonable to expect that the least significant bits of the gradients will not be strongly biased, as the training process introduces sufficient entropy at least in the LSBs of the gradients. This in turn will lead to full rank boolean matrix, i.e., positive unforgeability check. We check this in our experiments.

From Floating-point to Fixed-point Precision. We convert the 64-bit floating-point precision gradients (called *sources*[9]) output by PyTorch into fixed-point precision. We consider a high number of precision bits taken from the *source* float-point gradients, i.e., 26 bits[10]. This is within the scope of precision required typically for machine learning training.

Epoch	Step	Time (s)	Unforgeable
0	839	22	✓
1	402	22	✓
1	447	24	✓
2	0	23	✓
2	42	23	✓
2	194	22	✓
2	232	23	✓
2	361	24	✓
2	481	22	✓
2	505	23	✓
2	534	22	✓
3	187	22	✓
3	401	22	✓
3	410	22	✓
3	722	23	✓
3	736	22	✓
4	186	23	✓
4	217	24	✓
4	295	23	✓
4	296	22	✓
4	610	23	✓
4	695	23	✓
4	827	22	✓
4	936	22	✓
5	332	23	✓

Table 1: All 25 evaluated checkpoints for LeNet5 on MNIST with fixed precision of 26 bits are unforgeable.

Following the procedure used in prior work, we randomly sample 25 checkpoints at different training epochs and steps from the first 5 epochs for LeNet5, ResNet-mini, and, respectively, 5 checkpoints for VGG-mini. We run LSBUnforgeability on these checkpoints for $M = 400$ candidate minibatches, sampled without replacement. For efficiency reasons, we run only 5 checkpoints for VGG-mini. The average running time of the algorithm on the checkpoints is around 23 seconds on LeNet5 (Table 1), 1291 seconds for ResNet-mini models (Table 2), and, respectively, 9588 seconds for VGG-mini models (Table 3). Furthermore, all of these are unforgeable, showing that the conditions we state in our theorem are satisfied in practice.

> ***(RQ1)**: The unforgeability check LSBUnforgeability is efficient (feasible for large neural networks) and effective. It outputs conclusively that forgeries are impossible on all evaluated cases.*

One may ask what happens if multiple datasets are concatenated, or if the size of the dataset were larger. We thus additionally consider larger number of batches M for LeNet5 on MNIST, and ResNet-mini on CIFAR10. We find that the tests are conclusive for both LeNet5

[8] Based on email correspondence with the authors of [43], in Fig. 1 and Fig. 2 in the [43] paper, the number of batches is 400.
[9] We want to stress out that we introduce this convention only because we want to reuse the *floating*-point *source* gradients output by the PyTorch training process (currently, it supports only floats), however, if the training had been conducted in fixed-point precision, the amount of precision bits would have been uniquely determined.
[10] In fact, any sufficiently large amount can be taken.

Epoch	Step	Time (s)	Unforgeable
0	98	1561	✓
0	259	1202	✓
0	646	1555	✓
0	480	1129	✓
1	84	1408	✓
1	416	1183	✓
2	182	1465	✓
2	249	1449	✓
2	286	1448	✓
2	743	1254	✓
2	750	1124	✓
3	115	1401	✓
3	130	1186	✓
3	215	952	✓
3	250	1522	✓
3	261	1266	✓
3	275	1154	✓
3	317	1393	✓
4	714	1423	✓
4	28	1154	✓
4	677	1337	✓
5	74	965	✓
5	452	1380	✓
5	541	1319	✓
5	644	1056	✓

Table 2: All 25 evaluated checkpoints for ResNet-mini on CIFAR10 with fixed precision of 26 bits are unforgeable.

Epoch	Step	Time (s)	Unforgeable
1	209	7078	✓
1	262	9945	✓
1	257	10254	✓
3	11	10259	✓
3	450	10406	✓

Table 3: All 5 evaluated checkpoints for VGG-mini on CIFAR10 with fixed precision of 26 bits are unforgeable.

Architecture	M	Avg. Time (s)	Unforgeable
LeNet5	600	45.84	25 / 25
	800	91.56	25 / 25
ResNet-mini	600	5810.2	5 / 5
	800	8817	5 / 5

Table 4: Even when varying the number of minibatches M, LSBUnforgeability remains conclusive for both LeNet5 and ResNet-mini at precision 26.

and ResNet-mini at precision 26 when we vary the number of batches (Table 4).

***(RQ2)**: The unforgeability check LSBUnforgeability is conclusive even on larger sample sizes compared to [43].*

6.2 Precision at which LSB Checks are Conclusive?

One might ask what precision is enough to prove unforgeability with our LSB check? We focus on scenarios with lower number of precision bits, that may potentially allow approximate forgeries. We take 5 different LeNet5 checkpoints and vary the selection of precision bits from 1 to 24 from the *source* gradients. In Table 5, we give the exact ranks of the systems obtained for these 5 checkpoints, and gray out the full ranks which essentially correspond to unforgeability. For each of the 5 checkpoints, none of the systems for bits below 14 have full rank, i.e., LSBUnforgeability cannot exclude forgery for such cases. Indirectly, this means that potential approximate forgeries with precision up to 2^{-13} (around 10^{-4} in L_∞) are still possible. *This range already covers all approximate forgeries we present in Section 6.4.* On the other hand, with around 20-bit precision the checkpoints transition to unforgeable. Hence, it would be unlikely to produce approximate forgeries with a precision higher than 2^{-21} (around 10^{-6}) at any of these checkpoints, regardless of the technique used to generate them. We cannot completely exclude the possibility, as carries from the lower (beyond taken precision) bits may still propagate, however, such carries will only randomize the LSBs, thus yielding similar, full-rank matrices.

***(RQ3)**: LSBUnforgeability is conclusively finds unforgeability for precision over 20 bits in all evaluated cases.*

Therefore, when interpreting results of approximate forgery even in fixed-point arithmetic, the precision considered plays a significant role in determining whether forgery works at all.

6.3 Divergence with Approximate Forgery?

Our next experiments provide evidence that approximate forgeries at an intermediate step in training leads to large differences in the final model after training more steps. We argue that while obtaining a limited precision (e.g., up to $\delta = 3$ decimals) forgeries at a particular step is feasible, in subsequent training steps these errors will once again increase. We test this hypothesis by implementing the search for approximate forgeries [43].

We randomly sample 25 saved checkpoints for LeNet5, ResNet-mini and VGG-mini from the first 5 epochs with θ_t model parameters (same ones as in Section 6.1). The target checkpoint that we want to forge is θ_{t+1}. Then, according to the previously proposed strategy, for each checkpoint θ_t, we sample $M = 400$ forgery candidate batches with size 64. Then, we perform one training step from θ_t using these samples and greedily select the one with the smallest L_2 and, respectively, L_∞ distances from the target checkpoint: $argmin_M ||\theta_{t+1} - \theta'_{t+1}||_p, p \in \{2, \infty\}$. Then, to test potential divergence, we keep training the forged model parameters with the same data as the target trace. We train for 3,000 additional steps for LeNet5 and ResNet-mini, and 10,000 more for VGG-mini. In Figure 2, we show for LeNet5 and ResNet-mini that the distance between the initial and forged models' parameters increases. For VGG-mini, the L_∞ difference between the training run and the forged run initially decreases, but, as with LeNet5 and ResNet-mini, it eventually diverges (Figure 3). We observe that the larger the model (number of parameters), the slower the divergence. It takes about 8,000 training steps for the L_∞ distance to be greater than

	Checkpoint 1	Checkpoint 2	Checkpoint 3	Checkpoint 4	Checkpoint 5
Bit					
1	3121	6794	2727	4422	2542
2	4277	9336	3717	6093	3457
3	5794	12091	5048	8175	4624
4	7645	14882	6708	10657	6108
5	9873	17859	8756	13394	7843
6	12491	20533	11207	16241	9943
7	15105	22588	13848	19127	12306
8	17487	24049	16544	21400	14837
9	19457	24978	18810	23031	17284
10	21224	25342	20720	24206	19417
11	22709	25477	22344	25015	21304
12	23930	25561	23638	25414	22778
13	24785	25594	24547	25565	23991
14	25276	25600	25123	25593	24852
15	25496	25600	25399	25598	25313
16	25581	25600	25518	25599	25539
17	25595	25600	25575	25600	25593
18	25597	25600	25592	25600	25598
19	25599	25600	25598	25600	25599
20	25600	25600	25598	25600	25600
21	25600	25600	25600	25600	25600
22	25600	25600	25600	25600	25600
23	25600	25600	25600	25600	25600
24	25600	25600	25600	25600	25600

Table 5: Even at smaller precision on LeNet5 checkpoints, our LSB check can determine unforgeability. The grayed out cells correspond to full rank, i.e., the checkpoints at this bit precision are not forgeable.

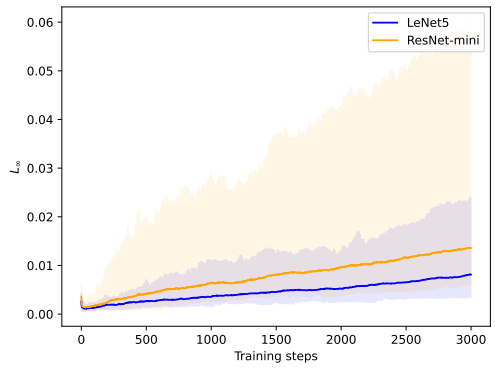

Figure 2: The approximately forged model parameters diverge after subsequent training of LeNet5 on MNIST and ResNet-mini on CIFAR10. The solid line indicates the mean L_∞ distance over the 25 checkpoints while the translucent region indicates the maximum and the minimum L_∞ distance boundaries for the corresponding architecture.

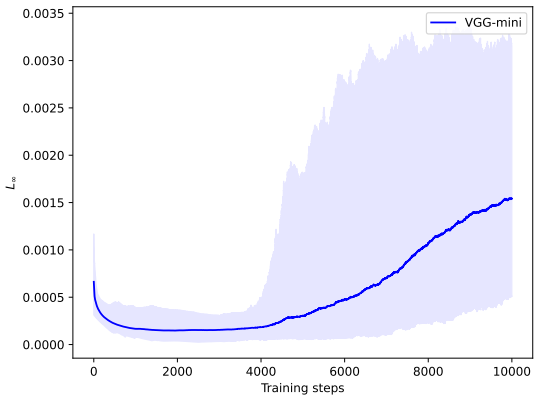

Figure 3: The approximately forged model parameters diverge after subsequent training of VGG-mini on CIFAR10. The solid line indicates the mean L_∞ distance over the 25 checkpoints while the translucent region indicates the maximum and the minimum L_∞ distance boundaries for the corresponding architecture.

the initial distance, with respect to the forged parameters. This is due to the gradient descent updates being smaller in magnitude than the initial error for VGG-mini.

We run one more series of experiments with longer training. More precisely, we train only LeNet5 (for efficiency reasons) for 17,000 training steps. The propagation of distance is given in Figure 4 and we can observe a similar outcome. Therefore, based on these two experiments (refer to Appendix A for L_2 results), it is clear that even not-so close approximate forgeries will diverge in the successive training steps.

> ***(RQ4)**: Approximate forgeries in floating-point precision eventually diverge and always result in clearly distinct model parameters by the end of training.*

Impact of Larger M on Forgeries. We check whether increasing the number of candidate minibatches helps with obtaining better approximate forgeries. We increase the number of candidate minibatches and run the approximate forgery attack for $M \in \{600, 800\}$ by sampling without replacement, on all models, including VGG-mini. When increasing the number of candidate batches, there is no real improvement in the obtained approximate forgeries, for all evaluated models. At the same time, doubling the number of batches means that the running time of the attack also doubles. Specifically, we run the approximate forgery, as in the evaluation of [43] for 25 checkpoints, to find the best L_∞ norm for LeNet5, ResNet-mini and VGG-mini for all $M \in \{400, 600, 800\}$. Among all of the 25 evaluated checkpoints the attacks' best approximate forgery among all 25 checkpoints was 1.56×10^{-4} in L_∞ norm on VGG-mini when $M = 800$. However, increasing the number of batches did not improve the average L_∞ distance by much, i.e., by less than 10^{-5}. For the largest number of batches $M = 800$, the time

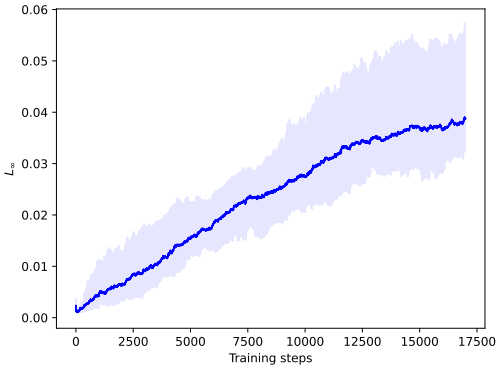

Figure 4: L_∞ distance between the forged batch and the benign training for LeNet5 on MNIST over 5 checkpoints. The solid line indicates the mean L_∞ distance over the 5 checkpoints while the translucent region indicates the maximum and the minimum L_∞ distance boundaries for the corresponding architecture.

	Shuffle Error	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5
(epoch 1, step 100)	4.62e-14	2.83e-04	1.33e-03	1.66e-03	2.33e-03	3.16e-03
(epoch 2, step 100)	2.49e-14	8.25e-04	1.47e-03	1.59e-03	4.27e-03	5.67e-03
(epoch 3, step 100)	3.02e-14	8.93e-04	1.98e-03	4.22e-03	4.88e-03	4.49e-03
(epoch 4, step 100)	3.38e-14	1.12e-03	3.35e-03	3.53e-03	4.73e-03	4.52e-03
(epoch 5, step 100)	4.62e-14	2.13e-03	4.27e-03	3.75e-03	5.86e-03	4.33e-03

Table 6: Even very small differences of 10^{-14} (called shuffle errors) due to the summation order in floating-point produce divergence over 5 epochs of training.

taken for VGG-mini is 40× larger than for LeNet5 and 2× larger than for ResNet-mini. We give the detailed results in Appendix B.

6.4 Divergence due to Floating-point Errors?

Recall from Section 4.3 that forgeries are not well-defined for reals implemented with floating-point precision. We first confirm that additions of gradients in floating-point lead to non-zero rounding errors. For this purpose, we run a series of experiments at different training epochs. At each epoch, we sample uniformly at random a minibatch of 1024. Then we sum up the gradients in 1000 different random orders, and check the number of different sums we get. The final results show that at each of the tested epochs, 1000 out of 1000 sums are different, i.e., each shuffle leads to a distinct sum. The sums differ on L_∞ errors in the range 10^{-12} to 10^{-17}. Hence, we can conclude that even when one considers the same minibatches but in different summation order, one may not produce this trivial forgery on all n bits. In our experiments, in 100% of the cases, different order resulted in different values. Thus, in floating-point precision, forgeries resist standard definition, rather, they will require at minimum an additional specification of the order of summation.

We investigate what happens if the above produced errors are kept small throughout training. If such is the case then one may argue that approximate forgeries with such precision (i.e., equal on almost all n bits) should be accepted as valid under the premise that the errors may have originated from minor hardware or library discrepancies. To test the propagation of small errors, among all of the previously generated potential pairs of sums, we sample a random pair at 5 different checkpoints. For each pair, we produce the resulting parameters (at the sample epoch/step), and then train independently each of them for 5 additional epochs on the same randomly sampled minibatches of size 1024. At the end of the training, we compare the differences between the final parameters, i.e., we compute the L_∞ norm between the parameter vectors. The results at different training epochs are presented in Table 6. We can see that the small initial errors (differences between the parameters), quickly expand and even after the first training epoch (around 1000 training steps) become pronounced, large errors. This means that even if we consider very close approximate forgeries (as close as a rounding error produced during floating-point addition), the subsequent training process will rapidly expand the small, initial difference and the almost identical pair of parameters produced by the approximate forgery will diverge into clearly distinct parameters. Therefore, based on these experiments, we can make two key observations. First, approximate forgeries clearly lead to distinguishable final output models in training, and therefore, are ill-suited for use in formal definitions. Second, even exact forgeries in floating-point precision diverge due to rounding errors of additions. Additionally, we have shown that small errors introduced due to hardware and library discrepancies lead to clear parameter discrepancies after a few rounds of training.

7 DISCUSSION

In this paper we have argued for refining the definitions of forgeability, irrespective of applications and datasets. Next, we discuss the setup limitations, and the complexity of the proposed test.

7.1 Limitations

Our tests were conclusive for all of the experimental setups considered in prior work. These setups assume that the adversary is constrained to use samples from the dataset that the ML trainer has released for verification. The ML trainer and the verifier are honest, and the adversary can only modify the choice of minibatches from the fixed dataset at a training step (see Section 2.2). This is a **post-deployment** adversary, that aims to construct forged gradient updates for data samples after the execution traces and their logs have been released. If the adversary can modify the dataset, then they could form $\hat{\mathbf{b}}_t$ from samples outside of the training dataset such as synthetically generated samples [46], or sampling more points from the distribution, until the test fails. Despite recent works proposing adversaries that synthesize data, there is no evidence that such attacks are possible at a higher precision (not approximate) [10]. Moreover, these attacks should be efficient enough to be mounted for every instance where a data point is used, not just one training step (whereas verifying impossibility for one is enough). Without breaking the requirement that the test be run with respect to a dataset, we can ask how often is the test conclusive if one were to consider a larger dataset. We evaluate this in Section 7.2.

A **pre-deployment** adversary, on the other hand, can manipulate both the training hyperparameters and the dataset. A more

powerful test is required, one that does not assume that the adversary cannot manipulate the setup phase. The verifier therefore can only check that the update is valid, i.e., given some minibatch $\hat{\mathbf{b}}_t \neq \mathbf{b}_t$, the model parameters $\hat{\theta}_{t+1}$ satisfy the forgery condition $\theta_{t+1} = \hat{\theta}_{t+1}$. In theory, such an adversary can construct a valid execution trace that is unforgeable with respect to D, but forgeable with respect to a trace obtained from a different dataset and hyperparameters. Formal statements about the existence of such adversaries at high precision would unlock more practical applications that rely on proof-of-learning logs [22].

One limitation of our check is that it works under the condition that the size of the datset is less than the model parameters, which is typically the case for many deep learning models and datasets. Howver, future unforgeability checks can consider the scenario where the number of model parameters is less than the dataset size, and design checks that are conclusive. These scenarios might be more relevant to large language models, such as a T5-base transformer model [37, 39] (around 220 million parameters[11]) trained the C4 crawl dataset (estimated at 365 million records) [8, 37].

7.2 Theoretical Complexity

Attacks proposed in [43] that find approximate forgeries (greedy) search the space of minibatches with the smallest distance in parameter space to the targeted model parameters. Instead, one can try running such search algorithm until an exact forgery is found, if one wants to utilize them to answer the decision problem of forgeability. However, as our evaluation points out, there might not be a solution for exact forgery, in which case the search would exhaust all possible $\binom{m}{k}$ minibatches to decide that there is no solution. Our test is rank computation whose worst running time is $O(n \cdot m^2)$ where n and m are the number of model parameters and, respectively, the dataset size. The computational difficulty of other search procedures when exact forgery is possible is unknown; we briefly alluded to these approaches in Section 3.2.

8 RELATED WORK

Approximate Forgery & Applications. Prior work has shown that different minibatches can produce similar model parameters using SGD, with direct implication to applications such as unlearning, proof-of-learning, and membership inference tests. A recent work [43] argues that approximate unlearning is not refutable or auditable because forgery of minibatches that contain the to-be-unlearned samples (say $\mathbf{x}$) is possible. This implies that we could have obtained a similar model parameter state had we used a different minibatch (without $\mathbf{x}$) from the training dataset. Thus, one cannot distinguish whether these execution traces correspond to the training dataset with the $\mathbf{x}$ samples. Another recent work [24] proposes using approximate forgery for repudiating membership inference tests. In this application, the authors have considered forging multiple checkpoints throughout the training process in order to find similar execution traces for D and $D - \mathbf{x}$. The resulting models have similar parameters up to some error in vector norms due to forging at multiple checkpoints. On these models, membership inference attacks are not able to distinguish whether $\mathbf{x}$ has been used. We motivated forgery using the proof-of-learning logs introduced in [22], which allows a verifier or a third-party auditor to check that 1) the computation was done over a given dataset and 2) that all the steps of the computation have been done correctly to obtain a final set of parameters. The verifier asks the ML owner / adversary to produce a sequence of batch indices and intermediate model updates such that starting from the initialization one can replicate the path to the final model parameters. Their proposed approach is to select only a subset of checkpoints to verify the model parameters for. However, there is no guarantee that an adversary is not able to forge the minibatch (find a different from the one used by the model owner) to produce the desired model parameters via SGD. In light of our results, we argue that one should consider exact forgery under fixed-point precision, since white-box verifiers are able to examine the forgery under all available precision, whereas previously demonstrated forgeries (i.e., with different samples) have as high an approximation error as 10^{-3} (in L_∞). Such approximation errors in forgeries are much higher than ones that could be attributed to floating-point errors—which we evaluated to introduce differences of the order of 10^{-14}. In addition, [43] describes forgery under other setups, e.g., when one considers *similar* datasets to the one used for training or where the initial model parameters are not the same, i.e., not forgery at a checkpoint but rather across multiple training steps at a time. These proposed problems are beyond the scope of our results but are interesting future work.

Algebraic Precision. Our work highlights the role of algebraic precision in specifying properties and drawing refutable conclusions about experimental observations about training with SGD. This issue is shared with other prior works that are not concerned with forgery as well. For instance, data reordering attacks on SGD distort the training execution trace to an adversary's advantage (e.g., longer convergence times, drop in task performance) [41]. These attacks use alternative minibatches from the same dataset reshuffled or reordered to produce similar but different model parameters after some training steps. In our experiments, we also show that it is possible to obtain this type of training divergence under minibatch reshuffling to change the training execution trace because of floating-point errors that propagate (Section 6.4). We pinpoint that these phenomena are due to the non-associativity of floating-point computations. If one did not have the *exact* order they would not be able to reproduce the execution trace of the training algorithm. This observation is also related to the problem of reproducibility in machine learning research which has been a known issue in creating artifacts [33, 35, 42]. To this end, our approach works under fixed-point precision, where additions have the required algebraic properties such as associativity and commutativity. There is on-going research into making training available in lower or fixed-point precision [14]. Quantization techniques are commonly used to accelerate inference of deep learning models but these do not apply to our setup since the gradient update computation is still being done in floating-point [19, 20] or in bifurcation/mixed precision (both floating-point and 8-bit integer) [4]. Other techniques propose complete fixed-point precision training pipelines that achieve good task performance [7, 13, 29]. On the other hand, these errors that accumulate because of choosing a different order of samples during SGD introduce some noise that

[11] https://github.com/google-research/text-to-text-transfer-transformer/blob/main/released_checkpoints.md#t511

helps making training data less "distinguishable". This is in line with recent work on repudiating membership inference attacks using approximate forgery [23, 24] but other types of noise have been purposefully added to the gradient computation to add privacy. For instance, it is common to add Laplace or Gaussian noise to gradients in order to achieve differential privacy [1]. Noise is also added to gradients in order to defend against bias attacks in inverting gradients [2, 9].

Finding Pre-images for Neural Networks. Our work considers finding collisions in the gradient update step due to freedom of choosing the minibatches in training. There has been research in understanding collisions at inference time rather than during training, for instance, when two different inputs produce similar [28] or the same activations or logits with a given ML model [32]. The problem of exact forgeries in this paper is that of finding a second pre-image in gradient descent, whereas prior work on gradient inversion considers the problem of finding any pre-image—finding the input to the model from the gradients [48]. Without bias in the model architecture, one work shows that recovering input data points can be uniquely determined from the gradients [9]. More advanced gradient inversion techniques deal with different types of neural networks [12, 21, 47]. Gradient inversion considers recovering the input data sample that results in a given gradient vector. However, recovering the set of data samples used in a minibatch given a gradient update vector has yielded much lesser success thus far, though attacks exist [18, 45].

9 CONCLUSION

In this paper, we identified mild and sufficient conditions under which gradient updates at one step of standard SGD training are unforgeable. Ours is the first result on proving unforgeability to the best of our knowledge. We found that these conditions are satisfied for the same benchmarking setup as prior work, i.e., single-step forging is not possible for LeNet5 and ResNet-mini neural networks on MNIST and CIFAR10, respectively. Our work explains that algebraic precision plays a crucial role in making refutable claims about model comparison. We believe these aspects matter practically to forgery-based security arguments and beyond.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported by the Crystal Centre at National University of Singapore and its sponsors, and the Ministry of Education Singapore Tier 2 grant MOE-T2EP20121-0011. Divesh Aggarwal was supported by the bridging grant at Centre for Quantum Technologies titled "Quantum algorithms, complexity, and communication".

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS)*. 308–318.

[2] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. 2017. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security* 13, 5 (2017), 1333–1345.

[3] IEEE Standards Association et al. 2019. 754-2019-IEEE Standard for Floating-Point Arithmetic.

[4] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems (NeurIPS)* 31 (2018).

[5] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. 2020. Neural gradients are near-lognormal: improved quantized and sparse training. *arXiv preprint arXiv:2006.08173* (2020).

[6] Ella Creamer. 5 July 2023. Authors file a lawsuit against OpenAI for unlawfully 'ingesting' their books. https://www.theguardian.com/books/2023/jul/05/authors-file-a-lawsuit-against-openai-for-unlawfully-ingesting-their-books. Accessed: 2023-08-13.

[7] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. 2018. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383* (2018).

[8] Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. 2021. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *arXiv preprint arXiv:2104.08758* (2021).

[9] Lixin Fan, Kam Woh Ng, Ce Ju, Tianyu Zhang, Chang Liu, Chee Seng Chan, and Qiang Yang. 2020. Rethinking privacy preserving deep learning: How to evaluate and thwart privacy attacks. *Federated Learning: Privacy and Incentive* (2020), 32–50.

[10] Congyu Fang, Hengrui Jia, Anvith Thudi, Mohammad Yaghini, Christopher A Choquette-Choo, Natalie Dullerud, Varun Chandrasekaran, and Nicolas Papernot. 2023. Proof-of-Learning is Currently More Broken Than You Think. (2023).

[11] Michael R Gary and David S Johnson. 1979. Computers and Intractability: A Guide to the Theory of NP-completeness.

[12] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. 2020. Inverting gradients-how easy is it to break privacy in federated learning? *Advances in Neural Information Processing Systems* 33 (2020), 16937–16947.

[13] Alireza Ghaffari, Marzieh S Tahaei, Mohammadreza Tayaranian, Masoud Asgharian, and Vahid Partovi Nia. 2022. Is Integer Arithmetic Enough for Deep Learning Training? *Advances in Neural Information Processing Systems (NeurIPS)* 35 (2022), 27402–27413.

[14] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[15] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)* 23, 1 (1991), 5–48.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[17] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. 2006. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*. Springer, 267–288.

[18] Yangsibo Huang, Samyak Gupta, Zhao Song, Kai Li, and Sanjeev Arora. 2021. Evaluating gradient inversion attacks and defenses in federated learning. *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021), 7232–7241.

[19] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. *Advances in neural information processing systems (NeurIPS)* 29 (2016).

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[21] Jinwoo Jeon, Kangwook Lee, Sewoong Oh, Jungseul Ok, et al. 2021. Gradient inversion with generative image prior. *Advances in neural information processing systems (NeurIPS)* 34 (2021), 29898–29908.

[22] Hengrui Jia, Mohammad Yaghini, Christopher A Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. 2021. Proof-of-learning: Definitions and practice. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1039–1056.

[23] Zhifeng Kong, Amrita Roy Chowdhury, and Kamalika Chaudhuri. 2023. Can Membership Inferencing be Refuted? *arXiv preprint arXiv:2303.03648* (2023).

[24] Zhifeng Kong, Amrita Roy Chowdhury, and Kamalika Chaudhuri. 2022. Forgeability and Membership Inference Attacks. In *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*. 25–31.

[25] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[26] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[27] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[28] Ke Li, Tianhao Zhang, and Jitendra Malik. 2019. Approximate feature collisions in neural nets. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).

[29] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International conference on machine learning (ICML)*. PMLR, 2849–2858.

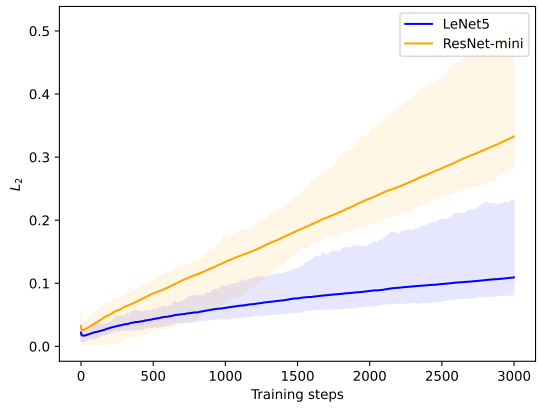

Figure 5: The forged model parameters diverge after subsequent training (in L_2 distance) on 25 checkpoints for ResNet-mini on CIFAR10 and LeNet5 on MNIST. The solid line indicates the mean L_2 distance over the 25 checkpoints while the translucent region indicates the maximum and the minimum L_2 distance boundaries for the corresponding architecture.

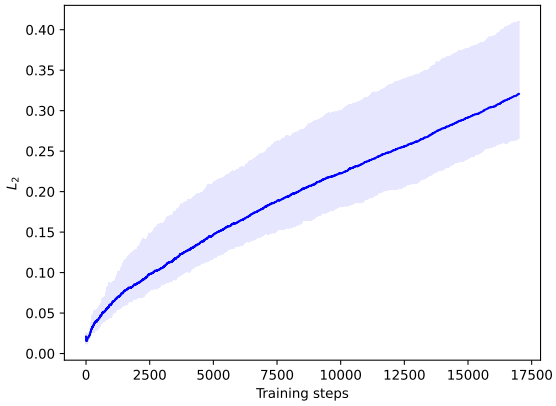

Figure 6: Extended training shows even larger divergence in L_2 distance between for LeNet5 on MNIST over 5 checkpoints. The solid line indicates the mean L_2 distance over the 5 checkpoints while the translucent region indicates the maximum and the minimum L_2 distance boundaries for the corresponding architecture.

[30] Stephen Linton, Gabriele Nebe, Alice Niemeyer, Richard Parker, and Jon Thackray. 2018. A parallel algorithm for Gaussian elimination over finite fields. *arXiv preprint arXiv:1806.04211* (2018).
[31] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.
[32] Utku Ozbulak, Manvel Gasparyan, Shodhan Rao, Wesley De Neve, and Arnout Van Messem. 2022. Exact Feature Collisions in Neural Networks. *arXiv preprint arXiv:2205.15763* (2022).
[33] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *The Journal of Machine Learning Research* 22, 1 (2021), 7459–7478.
[34] PyTorch. 2022. Reference Implementation of LeNet5 for Forging. https://github.com/cleverhans-lab/Forging. Accessed: 2023-02-28.
[35] PyTorch Contributors. 2022. PyTorch Reproducibility Documentation. https://pytorch.org/docs/1.13/notes/randomness.html?highlight=reproducibility. Accessed: 2023-04-28.
[36] Jack Queen. 9 July 2023. Sarah Silverman sues Meta, OpenAI for copyright infringement. hhttps://www.reuters.com/legal/sarah-silverman-sues-meta-openai-copyright-infringement-2023-07-09/. Accessed: 2023-08-13.
[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.
[38] Github Repo. 2018. Reference Implementation of ResNet-mini for Forging. https://github.com/nikhilbarhate99/Image-Classifiers. Accessed: 2023-04-10.
[39] Adam Roberts, Hyung Won Chung, Anselm Levskaya, Gaurav Mishra, James Bradbury, Daniel Andor, Sharan Narang, Brian Lester, Colin Gaffney, Afroz Mohiuddin, Curtis Hawthorne, Aitor Lewkowycz, Alex Salcianu, Marc van Zee, Jacob Austin, Sebastian Goodman, Livio Baldini Soares, Haitang Hu, Sasha Tsvyashchenko, Aakanksha Chowdhery, Jasmijn Bastings, Jannis Bulian, Xavier Garcia, Jianmo Ni, Andrew Chen, Kathleen Kenealy, Jonathan H. Clark, Stephan Lee, Dan Garrette, James Lee-Thorp, Colin Raffel, Noam Shazeer, Marvin Ritter, Maarten Bosma, Alexandre Passos, Jeremy Maitin-Shepard, Noah Fiedel, Mark Omernick, Brennan Saeta, Ryan Sepassi, Alexander Spiridonov, Joshua Newlan, and Andrea Gesmundo. 2022. Scaling Up Models and Data with `t5x` and `seqio`. *arXiv preprint arXiv:2203.17189* (2022). https://arxiv.org/abs/2203.17189
[40] Ahmed Salem, Giovanni Cherubin, David Evans, Boris Köpf, Andrew Paverd, Anshuman Suri, Shruti Tople, and Santiago Zanella-Béguelin. 2022. SoK: Let The Privacy Games Begin! A Unified Treatment of Data Inference Privacy in Machine Learning. *arXiv preprint arXiv:2212.10986* (2022).
[41] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A Erdogdu, and Ross J Anderson. 2021. Manipulating sgd with data ordering attacks. *Advances in Neural Information Processing Systems (NeurIPS* 34 (2021), 18021–18032.
[42] Rachael Tatman, Jake VanderPlas, and Sohier Dane. 2018. A practical taxonomy of reproducibility for machine learning research.
[43] Anvith Thudi, Hengrui Jia, Ilia Shumailov, and Nicolas Papernot. 2022. On the necessity of auditable algorithmic definitions for machine unlearning. In *31st USENIX Security Symposium (USENIX Security)*. 4007–4022.
[44] Xucheng Ye, Pengcheng Dai, Junyu Luo, Xin Guo, Yingjie Qi, Jianlei Yang, and Yiran Chen. 2020. Accelerating CNN training by pruning activation gradients. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*. Springer, 322–338.
[45] Hongxu Yin, Arun Mallya, Arash Vahdat, Jose M Alvarez, Jan Kautz, and Pavlo Molchanov. 2021. See through gradients: Image batch recovery via gradinversion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 16337–16346.
[46] Rui Zhang, Jian Liu, Yuan Ding, Zhibo Wang, Qingbiao Wu, and Kui Ren. 2022. "Adversarial Examples" for Proof-of-Learning. In *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1408–1422.
[47] Junyi Zhu and Matthew Blaschko. 2020. R-gap: Recursive gradient attack on privacy. *arXiv preprint arXiv:2010.07733* (2020).
[48] Ligeng Zhu, Zhijian Liu, and Song Han. 2019. Deep leakage from gradients. *Advances in neural information processing systems (NeurIPS)* 32 (2019).

A DIVERGENCE RESULTS

We ran the approximate forgery [43] to select batches ($M = 400$) that minimize the L_2 norm for both ResNet-mini and LeNet5 on CIFAR10 and MNIST, respectively (Figure 5). We observe the same trend for the L_2 norm as we did for the L_∞ norm. The training diverges in L_2 norm in less than 100 training steps. This suggests that approximate forgeries are detectable for a verifier that compares the benign training and the forged run. In Figure 6, we demonstrate that with extended training of LeNet5 on MNIST, the divergence (in L_2) keeps increasing. This suggests that a single approximate forgery determines a significant change in the model parameters for subsequent training steps.

Archictecture	M	Time (s)	L_∞ (Avg, Max, Min)	Avg. L_2 (Avg, Max, Min)
LeNet5	400	205.58	(6.62e-04, 1.16e-03, 3.12e-04)	(9.43e-03, 2.21e-02, 8.42e-04)
	600	165.85	(6.38e-04, 1.11e-03, 3.12e-04)	(9.26e-03, 2.10e-02, 7.09e-04)
	800	206.21	(6.23e-04, 1.02e-03, 1.56e-04)	(9.10e-03, 2.33e-02, 4.93e-04)
ResNet-mini	400	2010.92	(1.55e-03, 2.28e-03, 3.12e-04)	(3.21e-02, 5.12e-02, 1.94e-03)
	600	2817.59	(1.51e-03, 2.28e-03, 3.12e-04)	(3.16e-02, 5.12e-02, 1.91e-03)
	800	3768.89	(1.48e-03, 2.23e-03, 3.13e-04)	(3.13e-02, 5.18e-02, 1.93e-03)
VGG-mini	400	4314.00	(6.62e-04, 1.16e-03, 3.12e-04)	(9.43e-03, 2.21e-02, 8.42e-04)
	600	6711.57	(6.38e-04, 1.11e-03, 3.12e-04)	(9.26e-03, 2.10e-02, 7.09e-04)
	800	8489.83	(6.23e-04, 1.02e-03, 1.56e-04)	(9.10e-03, 2.33e-02, 4.93e-04)

Table 7: The approximate forgery attacks only marginally improves with increasing the number of candidate minibatches (M) considered from 400 to 800. The time represents the total time to load the model, sample the minibatches and pick the best update for all 25 checkpoints considered.

B APPROXIMATE FORGERY SCALABILITY

We ran approximate forgery [43] procedure and selected a larger number of batches, i.e., $M \in \{400, 600, 800\}$. Our aim was to evaluate if the approximate forgery attack can find minibatches which result in closer model parameters in L_∞ distance. We find that the improvement is marginal for all of the evaluated models (MNIST, ResNet-mini and VGG-mini), while the time taken doubles as the batch size also doubles (Table 7).