Game of Threads: Enabling Asynchronous Poisoning Attacks

Jose Rodrigo Sanchez Vicarte
josers2@illinois.edu
University of Illinois at Urbana-Champaign

Benjamin Schreiber
bjschre2@illinois.edu
University of Illinois at Urbana-Champaign

Riccardo Paccagnella
rp8@illinois.edu
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

As data sizes continue to grow at an unprecedented rate, machine learning training is being forced to adopt asynchronous algorithms to maintain performance and scalability. In asynchronous training, many threads share and update model parameters in a racy fashion to avoid costly inter-thread synchronization.

This paper studies the security implications of these codes by introducing *asynchronous poisoning attacks*. Our attack influences training outcome—e.g., degrades model accuracy or biases the model towards an adversary-specified label—purely by scheduling asynchronous training threads in a malicious fashion. Since thread scheduling is outside the protections of modern trusted execution environments (TEEs), e.g., Intel SGX, our attack bypasses these protections even when the training set can be verified as correct. To the best of our knowledge, this represents the first example where a class of applications loses integrity guarantees, despite being protected by enclave-based TEEs such as SGX.

We demonstrate both accuracy degradation and model biasing attacks on the CIFAR-10 image recognition task, trained on Resnet-style DNNs using an asynchronous training code published by Pytorch. We also perform proof-of-concept experiments to validate our assumptions on an SGX-enabled machine. Our accuracy degradation attacks are capable of returning a converged model to pre-trained accuracy or to some accuracy in between. Our model biasing attack can force the model to predict an adversary-specified label up to ~ 40% of the time on the CIFAR-10 validation set, depending on parameters. (Whereas the un-attacked model's prediction rate towards any label is ~ 10%.)

ACM Reference Format:
Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3373376.3378462

1 Introduction

Modern machine learning is a data hungry affair. It is well understood that a major reason for the success of deep learning has been the availability of public, diverse datasets of unprecedented scale, e.g., ImageNet [36]. Correspondingly, the last decade has seen machine learning model *training* move from single CPU to single GPU [40], to multicore CPU [3, 20, 26, 51, 58], to larger distributed systems [15, 21, 76]. This has presented a major performance engineering challenge. Core training algorithms such as stochastic gradient descent (SGD) are inherently sequential, making scale-out implementations performance bottlenecked by cross-node synchronization. As a result, starting with the seminal Hogwild! [51] algorithm, significant attention has been paid to develop multi-threaded *asynchronous* SGD algorithms (A-SGD) [4, 5, 15, 20, 21, 23, 32, 33, 44, 46, 51, 55, 58, 69, 76]. A-SGD leverages the inherent noise in training to skip inter-thread synchronization while maintaining convergence.

We explore the implications of asynchronous training on machine learning model integrity, in adversarial settings. Consider a setting such as Machine Learning as a Service (MLaaS) [1, 2]. In MLaaS, the user submits training data to an untrusted server, which will train the model on behalf of the user. Training data is sensitive, thus the user requests that the server run training inside a hardware-based trusted execution environment (TEE), such as Intel SGX [33, 34, 45, 54]. To improve performance, users may run asynchronous variants of SGD inside the TEE, as described above. For example,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03...$15.00
https://doi.org/10.1145/3373376.3378462

the Chiron system [33] implements SGX enclave-based A-SGD where each enclave implements a thread which asynchronously shares parameters through a centralized parameter server.

Regardless of training algorithm, TEEs should *in theory* defeat integrity attacks given even a supervisor-level adversary. For example, Intel SGX uses a combination of attestation mechanisms and runtime protections to ensure that only validated data runs within the TEE, and that the adversary cannot tamper directly with computation once it starts running [16, 45]. In particular, these mechanisms defeat conventional *poisoning attacks* [10, 48–50, 56, 73], which attempt to influence model integrity by changing the training set.

1.1 This Paper

Despite the apparent safety afforded by TEEs, this paper introduces *asynchronous poisoning attacks* (APAs), which show how integrity attacks are still possible *even* when asynchronous training runs within TEEs. APAs are based on two observations. First, due to the use of asynchronous algorithms, training state is a function of not only initial conditions, e.g., training data, but also the inherent races taking place between threads. Second, modern trusted execution technologies allow untrusted code, e.g., the operating system, to influence races through contention [7, 45], and/or thread scheduling [45]. Combining these two observations, an adversary can change the result of training by influencing data races in a coordinated, malicious fashion. To the best of our knowledge, this represents the first example where a class of applications loses integrity guarantees, even when "bug-free", despite being protected by enclave-based TEEs such as Intel SGX.

We demonstrate concrete *indiscriminate* and *focused* attacks on an A-SGD code representative of prior work [15, 21, 51]. In the literature [48], an indiscriminate attack reduces model accuracy while a focused attack does the same while additionally biasing the model towards a particular label/class (e.g., classify all spam as not spam but not the other way around). Unlike traditional poisoning attacks, we do not rely on changing the training set. Rather, we exploit a fundamental property in A-SGD codes: that per-thread model updates can be made asynchronously with respect to updates from other threads.

Contributions. To summarize, this paper makes the following contributions.

1. We introduce *asynchronous poisoning attacks* (APAs), whereby an adversary tampers with machine learning training by influencing data races in a coordinated, malicious fashion. Because data races are outside the model for trusted execution, APAs bypass protections offered by modern TEEs.
2. We design and validate three APA variants—indiscriminate and focused attacks—which exploit different aspects of A-SGD.
3. We perform extensive analysis of our attacks on the CIFAR-10 [39] image recognition task, trained on Resnet-18 [30] with an A-SGD implementation published by Pytorch [5]. We also perform proof-of-concept experiments to validate our assumptions on an SGX-enabled machine. Our indiscriminate attack is capable of returning a converged model to pre-trained accuracy or to some accuracy in between. Our model biasing attack can force the model to predict an adversary-specified label up to $\sim$ 40% of the time on the CIFAR-10 validation set (whereas the un-attacked model's prediction rate towards any label is $\sim$ 10%, i.e., roughly even across the 10 classes in CIFAR-10).
4. We provide an extensive discussion on defenses, ranging from point defenses to our particular attack variants, to more general defensive strategies.

2 Background

2.1 Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) is a ubiquitous algorithm for training machine learning models. See Algorithm 1 for pseudo-code. SGD takes a loss function (called loss), a training set T, and several other inputs we discuss below. We focus on the classification problem in supervised learning. Thus T consists of labeled inputs (x, y) for input x and label y. The goal is to learn parameters θ that minimize $\text{loss}(\theta, x, y)$, averaged over inputs-labels in T. Without loss of generality, we will represent y as a scalar that can take one of C unique values, where C is the number of labels.

To minimize loss, SGD iteratively updates θ based on a gradient it calculates for loss, given the current θ and B elements in the training set. B is called the *minibatch* size.[1] Each such update is called a *step*. Training runs for a number of epochs E (Line 3), where each epoch is $\lceil |T|/B \rceil$ steps, i.e., a complete pass over the training set. The epoch count E and minibatch size B are user-specified. To start each epoch, SGD makes random samples from T with or without replacement (e.g., by shuffling and then iterating through T, Line 7). This ensures that each epoch visits each input once (or a small number of times) and that minibatches are composed of random inputs, both of which improve accuracy.

Gradient update and scaling factors. The gradients g are scaled by a *learning rate* α before being accumulated into the model state (Line 12). The standard approach [30, 63, 67] is for the user to select an initial α, and for SGD to "decay" α once every D epochs (Line 5). Thus, the average magnitude of updates gets smaller as training proceeds, having the effect of "fine-tuning" the model. Note that in addition to

[1]When $B > 1$, SGD is sometimes called minibatch gradient descent. We will use this term and SGD interchangeably.

```
Function: SGD(loss, $T, B, E, D, \alpha, R$)
Inputs: loss (loss function), $T$ (training set), $B$ (minibatch size), $E$ (number of epochs per
        run), $D$ (learning rate decay frequency), $\alpha$ (learning rate), $R$ (initial conditions, seed)
Outputs: Model parameters $\theta$
1  $\theta$ = init($R$)
2  $\alpha = 0.1$ /* learning rate */
3  for $e = 0, \ldots, E-1$ do
4      if $e$ mod $D == 0$ then
5          $\alpha = \alpha/10.0$ /* 10.0 = decay rate, may also be variable */
6      end
7      $T$ = shuffle($T$)
8      for $b = 0, \ldots, (|T|/B) - 1$ do
9          $mb = T[B * b : B * (b+1)]$ /* sample minibatch */
10         /* Compute gradients, avged over minibatch */
11         $g = \frac{1}{B} \sum_{i=0,\ldots,B-1} \nabla \text{loss}(\theta, mb_i.x, mb_i.y)$
12         $\theta = \theta - \alpha * g$ /* update parameters */
13     end
14 end
```

Algorithm 1: (Minibatch) stochastic gradient descent. For simplicity, assume B divides $|T|$.

the learning rate α, different flavors of SGD [22, 38, 77] will additionally scale individual components in θ based on data-dependent conditions. (Thus, Figure 1 represents "vanilla" SGD [30, 63, 67], applying the same scaling factor to all components in θ.)

Finally, the seed R is used to initialize θ. Putting it all together, SGD(loss, T, B, E, D, α, R) becomes a deterministic function in its inputs.

2.2 Asynchronous SGD (A-SGD)

SGD performance is limited to the parallelism available in the inner loop, Line 11 in Algorithm 1, which averages gradients over the current minibatch. To alleviate the performance bottleneck in SGD, high-performance systems typically implement *asynchronous stochastic gradient descent* (A-SGD) [5, 15, 15, 33, 51, 58, 76]. A-SGD achieves better scalability by running multiple SGD instances concurrently, i.e., each processing different minibatches, which work together to train the same model. In addition to achieving high performance, prior work has also found A-SGD to result in higher accuracy models than SGD in some cases, e.g., on deep neural networks [15].

There are two main A-SGD architectures: those based on shared memory systems (e.g., [5, 20, 51, 58]) and those based on distributed systems (e.g., [4, 15, 33]). To simplify the presentation, we focus on shared memory-based systems and evaluate our attack on a shared memory implementation

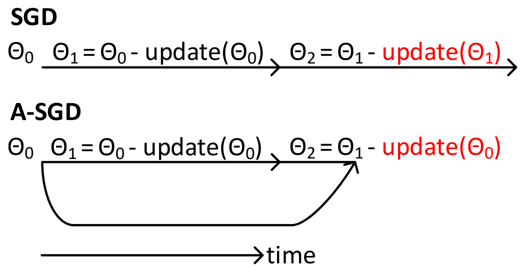

SGD
Θ_0 $\Theta_1 = \Theta_0$ - update(Θ_0) $\Theta_2 = \Theta_1$ - update(Θ_1)
A-SGD
Θ_0 $\Theta_1 = \Theta_0$ - update(Θ_0) $\Theta_2 = \Theta_1$ - update(Θ_0)
time

Figure 1. Comparison between SGD (top) and A-SGD (bottom). SGD does not have stale updates while A-SGD does have stale updates (differences highlighted in red). Each arrow denotes an update. Two arrows running in parallel (for A-SGD) denotes two threads. update(θ_t) denotes a gradient update (Algorithm 1, Line 12) using model state θ_t, or the state after t updates.

of Hogwild! provided by Pytorch (available online at [5]), which is similar to Algorithm 1. We discuss the implications for our attacks on distributed architectures in Section 8.

In shared memory A-SGD, training starts by spawning a fixed number of worker threads, each of which is similar to Algorithm 1. Each thread runs at its own pace. Each thread freely (i.e., without locks) updates a shared copy of the model parameters θ whenever it reaches Line 12 in Algorithm 1. *This implies that gradient updates will be stale, depending on how threads interleave their updates.* Figure 1 shows an example. SGD always performs updates based on the most up-to-date state, i.e., we always have $\theta_t = \theta_{t-1} - \text{update}(\theta_{t-1})$. Depending on thread interleavings, A-SGD updates are made to stale model parameters, i.e., we will have $\theta_t = \theta_{t-1} - \text{update}(\theta_{t'})$ for $t' < t-1$. For example, in the figure the model state is updated with gradients derived from θ_0 twice. Conceptually, this means training can "overshoot" local optimums by taking "too many steps in right direction." Despite this, A-SGD systems still converge because, in expectation, threads run at similar rates and gradient updates will be derived from recent state [51].

2.3 Deep Neural Networks

Our attacks rely on A-SGD. While A-SGD is general and can be applied to different models, we assume A-SGD training deep neural networks (DNNs) due to their popularity and importance today [15, 21, 33, 46]. A DNN F is a function:

$$F(\theta, x) = \sigma(L(\theta, x)) \in \mathbb{R}^C$$

which calculates a probability distribution for x being one of C labels. As discussed, θ is learned during training. $L(\theta, x)$ outputs a C-dimensional vector of *logits* where the i-th logit is $L(\theta, x)_i$. Each logit is an un-scaled value representing the DNN's confidence that the input x is associated to each label. For example, the index for the highest value logit is the label the DNN believes the input is associated to. Finally, logits are scaled via the softmax function, denoted σ, whose output is a C-dimensional vector which can be interpreted as the probability that x is associated to each of the C labels.

2.3.1 Training DNNs

DNNs are nearly all trained via (A-)SGD and backpropagation, the latter which estimates gradients for θ given a loss function. The details of DNN training are not important for this paper, but we will use the following abstraction to explain results. Consider an input, label pair (x, y). Backpropagation will produce gradients for each parameter in θ that are proportional to:

- $\sigma(L(\theta, x))_j - 1$, for logit $j = y$
- $\sigma(L(\theta, x))_j$, for other logits $j \neq y$

Here, we assume the cross entropy loss function, which is popular for classification tasks. Note that elements output by softmax are positive, and gradients are subtracted from parameters in Algorithm 1, Line 12.

There are two takeaways. First, correct/incorrect logit gradients have opposite sign. Gradients for θ will adjust corresponding parameters such that the magnitude of the correct logit will increase (i.e., when $y = j$), vice versa for incorrect logits. Second, gradient magnitudes through each logit are proportional to those logits' deviations from expected values out of the softmax. Note that $\sigma(L(\theta, x))_j$ should output 1 if $j = y$ and 0 otherwise. Thus, a DNN with perfect accuracy will backpropagate $\sigma(L(\theta, x))_j - 1 = 0$ for logit y and $\sigma(L(\theta, x))_j = 0$ otherwise, as expected. On the other hand, a perfectly incorrect DNN will backpropagate the largest possible gradients, $\sigma(L(\theta, x))_j - 1 = -1$ when $j = y$ and $\sigma(L(\theta, x))_j = 1$ otherwise, also as expected.

2.4 Enclave Execution

Enclave execution, e.g., with Intel SGX [45, 64], protects sensitive applications from direct introspection or tampering from supervisor software. That is, the OS, hypervisor and other software is considered the attacker [12, 27, 29, 47, 54, 57, 59, 68, 75]. To use SGX, users partition their applications into enclaves. Applications "enter" and "exit" enclave code using the SGX functions EENTER and EEXIT, respectively. While running enclave code, the application can access a privacy- and integrity-protected region of memory called the EPC (enclave page cache) which is inaccessible to code outside of the enclave. This process is bootstrapped by hardware attestation and digital signatures, which ensure that expected enclave code executes on expected starting data in the EPC.

Despite providing strong virtual isolation, SGX exposes a rich interface to the supervisor-level attacker. In particular:

Enclave thread management. Enclaves can be viewed as normal user-level processes from a task scheduling standpoint [16]. Further, each enclave can be multi-threaded. This means the OS (attacker) controls when each enclave thread is paused/resumed, just like normal threads. The OS also implements enclave system calls (e.g., thread start/stop calls) as well as demand paging when the enclave incurs a page fault.

Side channel amplification. A microarchitectural side channel is a way for an attacker to infer information about a victim program based on how the victim uses hardware resources in the system [24]. Prior work has shown how SGX exacerbates the side channel problem. In particular, SGX enables a new side channel known as controlled-channel attack [68, 74], where the attacker can learn the victim's page-granularity memory access pattern with zero noise. Multiple other works have shown how an attacker can learn a victim's access pattern at cache line granularity with very little or, again, zero noise [12, 27–29, 47, 62, 66].

We perform an evaluation of how we exploit these properties on an SGX-enabled machine in Section 6.

3 Threat Model

We consider an MLaaS-like or otherwise outsourcing setting, where a user wishes to offload training to an untrusted multi-tenant cloud [1, 2]. The user submits an untrained model, e.g., a DNN, and a training set (or part of a training set) to the server. The code run to train the model is considered public, e.g., is an open source ML framework like Pytorch [5]. The training set submitted by the user, and the resulting trained model parameters, are considered private. Thus, to improve security, the user runs training inside of SGX enclaves, similar to Chiron [33], OblivML [54] or Myelin [34]. Although this currently limits the attack to CPU-based training, architectures for secure enclaves are being explored for other platforms, e.g., GPUs [35]. Further, we assume the user runs an A-SGD algorithm to improve performance and scalability, similar to Chiron [33]. We assume the standard SGX attacker, namely the operating system and other supervisor software (see Section 2.4).

The adversary's goal is to corrupt/bias the user's trained model. Enclave virtual isolation prevents trivial integrity attacks by blocking direct tampering of thread state, e.g., θ, α, etc., during execution (see Section 2.2). Attestation of initial program state prevents conventional poisoning attacks that modify the user-supplied training set (Sections 1, 9). As discussed in Section 2.4, the adversary can schedule user threads and can only learn about training execution through its view, namely microarchitectural side channels, enclave termination times, system calls, etc.

Training with public labels. Some of our attack variants (Section 4.2) assume that training set labels are public or at least distinguishable. If public, the attacker can deduce that y in (x, y) corresponds to a label with known semantics, e.g., "not spam." If distinguishable, the attacker can deduce for an (x_i, y_i) and (x_j, y_j) whether $y_i = y_j$. Thus, public implies distinguishable. Labels may be public or distinguishable for several reasons. First, while training set inputs (e.g., images) are obviously high value, labels may not be private information. Second, even if part of the training set is private, the user may augment its training data with a large public training set such as ImageNet [36], as this has been shown to improve accuracy (e.g., [13]). The server would likely supply this data locally to save bandwidth, hence that part of the training set would be public.[2]

4 Attack Variants

In this section, we provide three concrete variants of asynchronous poisoning attack (APA). All three attacks leverage how a supervisor-level adversary can context switch any A-SGD thread on and off the system at any point. Conceptually, this allows the adversary to craft a malicious gradient update that it can then apply to training at a later time of its choosing. When a thread is context switched for the purposes of the attack, we refer to it as an *attack thread*.

Context switching is a synchronous event that pauses and (eventually) resumes the attack thread(s) at a particular instruction. This allows the adversary to construct different variants of the exploit depending on *when* and *under what conditions* it decides to pause *which* threads. For example, the adversary can pause the thread after a specific epoch, after copying the minibatch but before copying the parameters, etc. (see Algorithm 1).

To explain ideas, we assume the adversary can context switch a thread(s) when needed for the attack and can terminate training at a time of its choosing. Sections 5-6 detail how these assumptions manifest in real systems.

4.1 V1: Indiscriminate Attack

The first attack flavor is when the adversary indiscriminately pauses the attack thread at any point in the A-SGD loop, but before the learning rate decays (i.e., in the first $D-1$ epochs). Later, after the learning rate decays one or more times, the adversary re-schedules the attack thread, and that thread makes an update to θ with a specific stale (large) learning rate.

The attack procedure is as follows. The adversary context switches the attack thread off the system before the learning rate decays (i.e., when $\alpha = 0.1$ in Algorithm 1) and waits for multiple decays (e.g., when other threads have been making updates with $\alpha = 0.001$), at which point it re-schedules the attack thread. Figure 2 (left) shows the sequence of learning rates that will be applied at each step across all threads in time. The malicious update is shown as the red spike in the last epoch. Recall that gradient updates are accumulated into the model parameters θ in a read-accumulate-write fashion

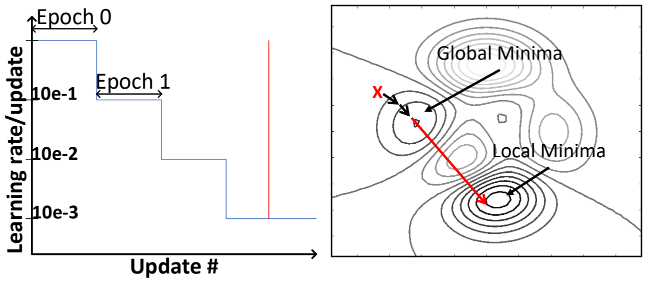


Figure 2. The learning rate-based indiscriminate attack (Section 4.1). (Left) the sequence of learning rates applied to each update in time. (Right) a visualization of the error surface. The malicious update and the step it makes through the error surface are shown by the red arrow. Non-malicious updates made by other threads, with decayed learning rates, are shown with (smaller) black arrows. The θ at the start of the attack is denoted with X.

(Line 12), meaning the incorrect update is applied to the current model parameters.

After the re-scheduled thread applies a large update, θ will be incorrect (i.e., no longer converged if the model had converged already) and the loss function will sharply increase. This process is visualized in Figure 2 (right). An update with a large learning rate causes the model to overshoot the optimum. That is, the red arrow in the figure is pointing in the correct direction but has too large a magnitude. This attack can be used to render the resulting model less effective at its given task—e.g., a spam detector may classify inputs randomly.

This attack is trivial to perform, as the adversary can simply pause the attack thread at the start of training and wait. This attack requires that the A-SGD implementation proceed in training regardless of how many epochs have been completed by other threads. This assumption holds for large-scale A-SGD implementations [15] as well as the Pytorch code we attack [5].

Once the attack thread performs a malicious update, it can perform additional malicious updates with the stale learning rate to increase damage done to the model. Not surprisingly, we find that the more malicious updates, the more damage dealt (Section 7).

4.2 V2: Focused Attack

While the previous attack variant decreases model accuracy, it does not give the adversary a guarantee as to how the damaged model predicts on each particular label. Ideally, the adversary would like to reliably bias the model towards predicting a particular adversary-chosen label. For example, to classify all incoming mail as "not spam" but not the other way around.

[2]Note, it is still easy to defeat conventional poisoning attacks by verifying the integrity of augmented data through enclave attestation, using a known hash of the data signed by a trusted third party, e.g., the ImageNet distribution website.

We now present such a model biasing attack, commonly referred to as a *focused attack* [48]. In our attack, the adversary chooses an *attack label* that it will bias the model towards. At a high level, the attack is identical to the attack from Section 4.1, except that the attack thread(s) are paused after sampling a specific minibatch as opposed to an arbitrary minibatch. As a result, the malicious update is performed with a high learning rate and given a specific adversary-chosen minibatch.

The question is, after which minibatch should the adversary pause the attack thread? Of independent interest, *we show that model bias after an update is proportional to minibatch bias for that update.* Minibatch bias refers to the phenomenon where labels for inputs in the minibatch are not equally represented. For example, there might be more "not spam"-labeled inputs in the minibatch than "spam"-labeled inputs. We say a minibatch is biased towards label Y if there are more inputs mapping to label Y in the minibatch than to other labels. Since minibatch sampling is a random process, sampling biased minibatches happens naturally during training.

The next three sub-sections address important questions regarding the attack. In particular: How can the adversary learn minibatch bias? How frequently can we expect how much minibatch bias to occur? Why does minibatch bias result in a biased model?

4.2.1 How does the adversary learn minibatch bias?

Suppose training set labels are public as discussed in Section 3.[3] Then an adversary can deduce minibatch bias in several ways. In the simplest case, if the seed used to sample minibatches isn't kept private, the adversary can trivially predict the composition (by label ID) of each minibatch. (For example, the Pytorch code we attack uses a constant seed [5].) Even if a secure setting, it may be the case that this seed is left public because the minibatch sampling distribution is not normally considered sensitive.

Even if the seed is private, the adversary can still learn the minibatch bias through a variety of memory access pattern-related side channels (Section 2.4). A-SGD codes, such as the Pytorch code we attack [5], do not shuffle actual training set data per epoch, but rather instantiate iterators that randomly sample the training set, to improve performance. When combined with public labels, this means a thread's memory access pattern reveals the minibatch label composition.

As discussed in Section 2.4, SGX enables fine-grain/low- or zero-noise monitoring of memory access pattern-related side channels. In our proof-of-concept on an SGX-enabled system (Section 6), we use a controlled-channel attack to learn page-level access pattern [74]. This is sufficient to learn minibatch bias for any training set whose elements are roughly or larger than the size of a 4 KB page. This is often the case. For example, the CIFAR-10 [39] and "ImagenetLite" [36] training sets have elements which are 3 KB and 150 KB in size, respectively. We evaluate our proof-of-concept assuming CIFAR-10.

4.2.2 Minibatch bias likelihood

We find that sufficiently biased minibatches appear with high probability. We define the % bias towards an attack label as the number of inputs M with that label, divided by the minibatch size B. Continuing our example, CIFAR-10 has $C = 10$ labels and has an equal number of inputs for each class, so % minibatch bias in expectation is 10% for all labels. We show in our evaluation that for different minibatch sizes, there is a % bias—between 20% and 50% depending on the minibatch size—such that minibatches with that bias are expected to occur and are capable of biasing the model.

An example, for reference: In the evaluation we find that attacks with 30% biased minibatches can significantly bias the model given $B = 64$.

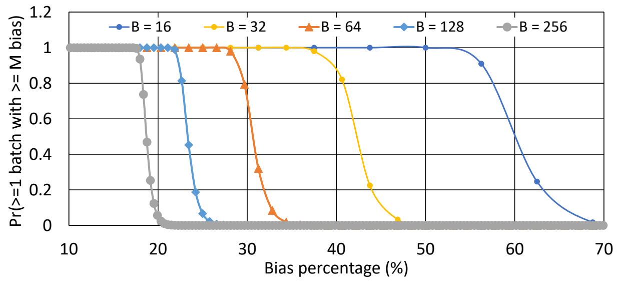
To perform the attack with these parameters, we must determine the number of 30% biased minibatches expected to occur. Figure 3a shows the probability of seeing at least one minibatch that is biased by at least a certain % and Figure 3b shows the expected number of biased minibatches for different % biases. We use $C = 10$ to model CIFAR-10 [39] and decay the learning rate after 200 epochs (which means we are looking for biased minibatches out of a pool of $200 * T/B$ minibatches). In general, the probability of the next minibatch having a $M/B * 100\%$ bias is given by the Binomial distribution $\mathsf{Binom}(M; B, \frac{1}{C})$ when A-SGD threads sample the minibatch via random reads,[4] and assuming each label is equally common in the training set.

Comparing the figures to our example, we see that 30% biased minibatches are expected to occur. (On average, we expect to see 1-2 such minibatches.) There are many parameters through which to adjust the attack. For example, by decreasing the minibatch bias threshold to 28%, the expected number of biased minibatches increases to 6.

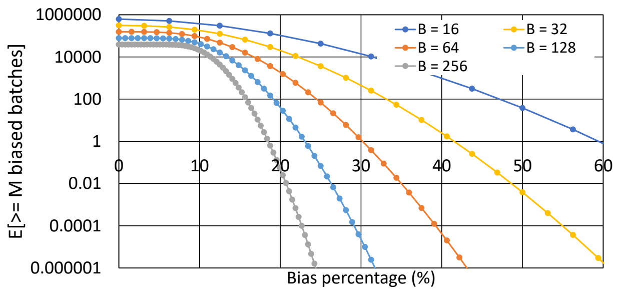
Finally, we note that prior work utilizes a large range of minibatch sizes. For example, Chiron evaluates CIFAR-10 and ImagenetLite using a 128- and 64-size minibatch, respectively [33], whereas Georganas et al. [25] (which also evaluates CPU-based training) uses minibatch sizes of 28 and 70. We will evaluate a range of parameters for completeness.

[3]If labels are distinguishable, but not public, then the adversary can still bias the model towards *some* label.

[4]That is, with replacement. We note that for practical parameters, this closely approximates sampling without replacement, as done with our current implementation.



(a) Probability of seeing at least one minibatch with bias % $\geq M/B$



(b) Expected number of minibatches with bias % $\geq M/B$.

Figure 3. Statistics on minibatch biasing. Parameters match CIFAR-10 [39] which we evaluate in Section 7. CIFAR-10 has 50,000 inputs and $C = 10$ labels. We vary minibatch size B in a range explored by prior works targeting CIFAR-10. Following state-of-the-art results, we decay the learning rate after 200 epochs [63]. Likelihoods only include minibatches sampled before the learning rate decays the first time.

4.2.3 Attack root cause

Why does updating the model with a biased minibatch bias the model? We will explain the root cause using DNN training as an example. First, the sign and magnitude of gradients through each logit depends on the correct label for each input (Section 2.3.1). Second, gradients are averaged across inputs in each minibatch (Algorithm 1, Line 11). Suppose the entire minibatch consisted of inputs from a single label. Then clearly the network will bias towards that label: for each logit i, the gradient for logit i will have the same sign across all inputs in the minibatch. Averaging many values sharing the same sign cannot flip the sign. When the minibatch isn't 100% biased, seeing this effect depends on the magnitudes of logit gradients, but is still more likely given a more biased minibatch. Putting it all together, we find that even a slight minibatch bias can bias the overall model because that bias is amplified with a large (stale) learning rate.

4.2.4 Extending to multiple attack threads

The prior attack variant (Section 4.1) gave the adversary the ability to amplify the attack's effect by having the attack thread perform multiple updates with a stale learning rate. Is there a similar mechanism for the focused attack? The subtlety is that multiple malicious updates should complement one another—i.e., both bias the model in the same way to amplify the effect. This means the strategy used for multiple malicious updates in Section 4.1 won't work: when the attack thread performs its update, the next minibatch it samples will not be biased in the same way with high probability.

We avoid this problem by coordinating multiple attack threads. In this case, the adversary listens for biased minibatches in two or more attack threads. When it sees a biased minibatch towards the desired target label, it pauses that thread, repeating this process for the desired number of attack threads. Then, at the end of training, all threads are released serially to perform their updates one after another. Note, no two minibatches will contain the same set of images, but two (or more) minibatches with the same bias towards the same label do occur.

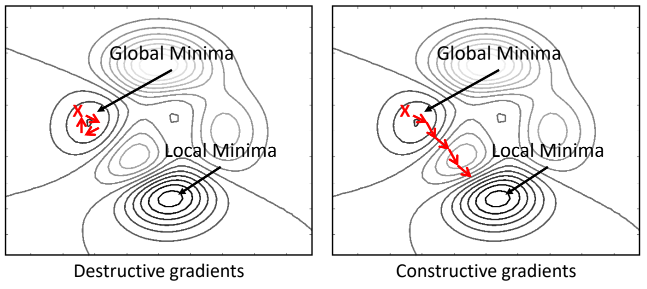


Figure 4. Stale parameter-based attack (Section 4.3). (Left) Destructive gradient updates do not, cumulatively, force the model out of convergence. (Right) Constructive gradient updates force the model out of convergence. Each update made by an attack thread is shown with a red arrow. The θ at the start of the attack is denoted with X. Note, the magnitude of each malicious update is small relative to that in Figure 2, because this attack variant is done after the learning rate has fully decayed.

4.3 V3: Enhancing Attacks with Stale Parameters

The last two sections presented APA variants that use stale learning rates to apply large malicious updates. For the attacks to go through, we required the A-SGD algorithm to use learning rates and for the A-SGD implementation to not synchronize threads at epoch boundaries. We now present a more sophisticated attack that does not rely on these assumptions. This is challenging. Without a large learning rate, an attack thread's parameter update will have similar magnitude as updates made by honest threads. The premise of A-SGD is that such small stale updates should not cause long-term accuracy loss.

The new idea enabling the attack is to pathologically schedule multiple attack threads to maximize model parameter staleness at the time updates occur. Consider Algorithm 1. If a thread is paused between Lines 11 and 12, it has calculated a gradient update but not applied that update. If, sometime later, the thread is un-paused, it will accumulate its *stale* gradient update into the current model state.[5] A simple strategy to increase the average staleness of updates is to schedule all threads to sample the same θ and only then apply their updates (akin to Figure 1).

4.3.1 Challenge: crafting constructive gradient updates

In our setting, the above attack idea does not go through as written. The problem is that to avoid relying on a learning rate, our attack must take place solely after the last learning rate decay, i.e., at the end of training. At this point, the model has likely converged. Due to convergence, we conjecture the following: When converged, subsequent updates make unpredictable changes to θ because θ already corresponds to a local optimum. Intuitively, we have reached the "bottom of a hill" in the error surface and subsequent updates will unpredictably move "around" the bottom of the hill. This makes stale gradient updates *destructive*, shown in Figure 4 (left). Experimentally, we have indeed found that large numbers of stale updates on a converged model does not noticeably change model accuracy.

To avoid the destructive gradient problem, we must manufacture what we call *constructive gradient updates*. By constructive, we mean multiple gradient updates that point in similar directions such that, when accumulated, will drive the model in a specific direction. The idea is shown in Figure 4 (right). Conceptually, the idea is to emulate the attack with a large learning rate (i.e., Figure 2) through the use of multiple small constructive updates.

The question is, how to construct multiple pending gradient updates that will be constructive? Of independent interest, *we find one method is to re-use the idea from the focused attack, namely to use biased minibatches* (Section 4.2). Suppose two attack threads sample the same θ. We conjecture that the gradient directions generated by these threads will be more similar, proportional to the degree of minibatch bias for both threads towards a given label. For example, if both threads sample the same θ and additionally sample minibatches that are 100% biased towards the same label, the gradients derived by both threads will be maximally similar.[6] We show evidence to support this claim in the evaluation. Specifically, we show that as minibatch bias increases, less attack threads are required to move the model out of its converged state.

Putting it all together, the attack works in the following steps. The adversary will perform the attack with N attack threads, where N can be any number less than or equal to the thread total.

1. Wait for the learning rate to fully decay / wait until immediately before training terminates (if optimizer does not use a learning rate).
2. As in Section 4.2: Select a target label. Monitor the N threads for a specific degree of minibatch bias towards this label and pause each when this bias is detected, but before the model θ is sampled.
3. When all N threads have biased minibatches, first sample θ for all N threads and second perform gradient updates for all N threads.

The attack is more effective with more attack threads, and with minibatches biased to a larger degree. As before, the expected number of minibatches biased to different degrees depends on multiple training parameters such as the minibatch size (Section 4.2.2). Thus, the adversary must adjust its attack parameters based on these training parameters. We explore the attack space in Section 7.

Interestingly, while our goal was to find constructive gradient updates, the fact that we use biased minibatches to generate such updates means the above attack can be used both as an indiscriminate attack and a focused attack. We evaluate the degree to which different numbers of attack threads bias the model in Section 7.

4.3.2 Further relaxing attack assumptions

The APA variant in this section requires different assumptions than previous variants. This variant requires more attack threads than other variants because each attack update is smaller. It also requires a mechanism to detect biased minibatches (as in Section 4.2). Yet, importantly, it does not rely on large learning rates which defeats some possible defenses.

An open question is whether there are other means to build constructive gradient updates. If so, this could further relax assumptions, e.g., bypassing the need to monitor biased minibatches. We leave exploring this direction as future work but posit the following idea: If gradient updates are constructed based on a θ which corresponds to an *un-converged* model, those gradient updates should have similar direction. That is, the updates should all be pointing "down the hill." Generally, the adversary does not know if the model has converged, which is required for this idea to go through. Yet, in some cases the adversary can deduce this information. For example, immediately after the final learning rate decay, it is well known that accuracy fluctuates, i.e., the model has to re-adjust to the final learning rate.

4.4 Coordination with the End of Training

When a malicious update(s) is made, the A-SGD loss function will either diverge (never recover) or slowly recover over time as non-malicious training is allowed to proceed. This is illustrated in Figure 5. Loss decreases ① until the attack

[5] Note, this is subtly different than the prior attack variants. For example, the focused attack in Section 4.2 relies on a specific minibatch, but applies that minibatch to up-to-date model parameters. Thus, that attack implicitly pauses the attack thread in between Lines 9 and 11.

[6] Note that while gradient directions will be similar, they will not be identical because even with the same θ, the exact minibatch contents will be different.

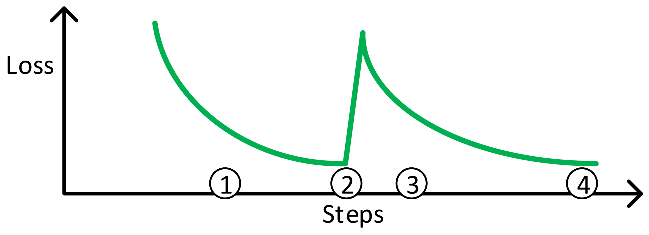


Figure 5. Training loss before and after the attack thread commits an update.

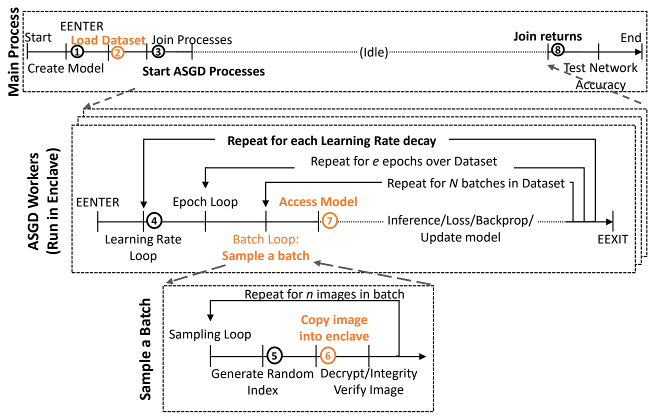


Figure 6. Sketch of our SGX-based prototype, illustrating key operations for executing an APA on asynchronous training.

update ②, at which point loss spikes and begins to fall again ③, re-converging at some later point ④.

Thus, a basic function every attack needs is a way to coordinate malicious updates with the end of training. Else, the model will simply recover by the time training ends and the attack may as well not have occurred. How the attacker coordinates the attack with the end of training depends heavily on the A-SGD implementation. We discuss how this can be done with the C-based A-SGD and Pytorch A-SGD code, that we evaluate, in Sections 5-6.

5 Evaluation Methodology

This section describes the methodology for the APA proof-of-concept codes and attacks we describe in our evaluation (Sections 6-7). Unfortunately, despite recent work on machine learning in SGX [33, 34, 53], there are currently no publicly available end-to-end A-SGD Frameworks implemented under SGX. Thus, we evaluate two codes:

- A C-based prototype we designed, run on an Intel(R) Core(TM) i7-6700K SGX-enabled CPU.
- A Pytorch example A-SGD code which is downloadable from the Pytorch site [5].

The C-based prototype mimics the main operations in the PyTorch code (e.g., how minibatch sampling occurs), but does not have all the functionality needed to produce a high-accuracy model (e.g., gradient calculation, back-propagation). Thus, its purpose is to evaluate hardware assumptions in an SGX-enabled environment. The PyTorch-based code's purpose is to show how APA attacks can actually degrade training accuracy in a real training session. We expect codes like the Pytorch implementation to be able to run on SGX-enabled machines in the future, using library and environment support such as Haven [9] or Graphene [65]. Both codes are Hogwild!-style A-SGD algorithms that follow closely to Algorithm 1. We now discuss relevant details for both codes:

C-based code overview: The C-based code's computation is shown in Figure 6. A main thread calls EENTER to initialize a blank DNN model (①) with randomly initialized weights that live in the SGX EPC (the SGX-protected memory region). It then loads the training dataset, alongside a per-element signed hash, into non-enclave memory (②).[7] The training dataset/model live at public (OS-known) addresses in program memory/EPC, respectively. Main calls `pthread_create` to spawn A-SGD threads (③), each of which call EENTER to run code implementing Lines 3-14 in Algorithm 1 and join when the algorithm terminates (⑧).

PyTorch-based code overview: The experiments on Pytorch use the code hosted at [5]. The only change we make to the Pytorch code is to add learning rate decay (④), which we implement using Pytorch's built-in learning rate scheduler functionality. Relative to the C-based code, the Pytorch code spawns A-SGD threads as full-blown processes (③) which join in the same way as the C-based code.

Both codes follow the same order of operations. For example, minibatch sampling (⑤-⑥) occurs before the model state is sampled (⑦). Aside from this, the two most relevant parts of the A-SGD codes for our attacks are how minibatch sampling occurs (for the focused attack; Section 4.2) and how training terminates (Section 4.4), discussed in more detail below.

Minibatch sampling (C and Pytorch): At the beginning of each training epoch, the Pytorch code "shuffles" the dataset by generating a fresh iterator of permuted indices (⑤), and copying a subset of the dataset according to the iterator into a local tensor (⑥). Our C-based code emulates this by indexing randomly into the dataset and performing a memcpy to a private buffer. As in [33], data is thus streamed into the enclave. As discussed in Section 4.2, the attacker needs to observe which inputs are sampled for each minibatch to determine minibatch bias. The Pytorch code uses a public seed to generate each iterator, making it trivial to learn bias (as the attacker can determine the bias of each

[7] Because the data is stored in non-enclave memory, it must be integrity verified to prevent trivial poisoning attacks. SGX EPC memory is not currently large enough to store the datasets we evaluate in Section 7 (although this issue will be fixed in SGX V2 [33]). Which implementation is used does not impact the attack.

minibatch before training begins). It is reasonable to assume a real SGX-based training would make the seed private, e.g., by sampling the seed through the Intel RDRAND instruction. Thus, our evaluation on SGX learns minibatch bias through side channels (Section 6).

Terminating training (C and Pytorch): To disambiguate between an A-SGD thread and hardware thread, we refer to an A-SGD thread as a *worker* in this discussion. Both the C-based and Pytorch code terminate with a worker join, followed by an operation which copies the trained model to a secure channel.

In our evaluations (Section 7), non-attack workers are allowed to terminate normally once the learning rate fully decays (loop at ④). Attack workers, on the other hand, are prematurely terminated immediately after making their last malicious update. In the Pytorch code, since each worker is a separate process, terminating a worker can be done by calling SIGTERM on the worker. Since other non-attack threads are waiting at a join, this creates a leaked semaphore warning which the OS (adversary) can mask, at which point other threads terminate normally and the attack succeeds. In the C-based prototype, using SIGTERM in this way works in the same fashion assuming the A-SGD implementation has implemented a handler for clean enclave exits on thread termination, which follows how previous A-SGD implementations (e.g., Project Adam [15]) implement fault tolerance.

5.1 Target Model, Training Dataset and Parameters

We perform all experiments using the CIFAR-10 [39] dataset and the Resnet-18 [30] CNN.

Dataset. CIFAR-10 is a popular image recognition task with $C = 10$ labels, 50 K training inputs and 10 K validation inputs ranging from animals to vehicles. We note that the validation and training sets are evenly distributed by each label.

Model under attack. The Resnet-18 [30] model is a CNN with 18 convolutional layers, plus additional fully connected and shortcut layers.

Training parameters. We will evaluate using a variety of minibatch sizes B ($B = 32$ to $B = 128$), to show robustness. We scale parameters as in vanilla SGD, i.e., each parameter is scaled by the same learning rate. The learning rate starts at 0.1 and is also decayed by a factor of 10 once at epoch 150 and again at epoch 250. We additionally verified that the attacks go through with the Adam optimizer [38], but do not show these results for space.

We assume the model has converged, and perform attacks, at epoch 350. That is, after the learning rate has decayed for the last time. We note that learning rate decay schedule is typically public (e.g., a constant as in Algorithm 1). The network converges to a maximum accuracy of over 90% on the validation set when run on the Pytorch implementation [5] for all batch sizes.

6 Evaluation on SGX

In this section, we test key hardware assumptions for our attack on an SGX-enabled machine and our C-based prototype (Section 5). We validate the focused attack from Section 4.2. The techniques required validate the indiscriminate attack variants (Sections 4.1, 4.3) by extension.

To start, workers (hardware threads) are set up to page fault on the model (⑦) and on the training dataset (⑤). Using a modified (malicious) page fault handler in kernel space, we halt workers by not returning control until specific conditions are met.

On spawn, all but the last worker are halted on their first page miss on the model (⑦). The remaining worker is used as the attack worker. Page faults are induced for this worker by clearing page table present bits from all images in CIFAR-10 that belong to a specific attack label. Counting the number of page faults to these images reveals the number of images of the target label in the minibatch, which reveals the minibatch bias (Section 4.2). The minibatch is known to be biased once the number of images crosses a threshold.[8] All other threads are resumed at this point. The page fault handler delays returning from the most recent page fault to the attack worker, until the resumed workers terminate. The attack worker learning rate is therefore stale relative to the rest of the workers. Finally, the attacker lets the attack thread apply this stale learning rate to the model, and then sends a SIGTERM signal as described in Section 5 and the APA has been successfully orchestrated.

7 Evaluation on Pytorch

We now evaluate our APAs from Section 4 on a real dataset, DNN and A-SGD implementation from Pytorch [5]. See Section 5 for details on training dataset and model under attack (CIFAR-10 [39] and Resnet-18 [30]).

We first trained the Resnet model using the Pytorch script until it converged and then checkpointed the model state. For each plot, we load the checkpoint and launch an attack thread(s) from the Pytorch code to directly perform the malicious updates needed to implement each attack variant. For the focused attack (Section 4.2), we add code to each thread so that updates only trigger when a minibatch has a specific % bias. This is functionally equivalent to the proof-of-concept using thread scheduling, side channels, etc. from Section 6, but allows us to perform many experiments in an automated fashion.

Prediction rate. To show accuracy change in greater detail, we measure validation set *prediction rate* for each label. The CIFAR-10 validation set has 10% images in each class/label,

[8]One subtlety is that we must detect when minibatch sampling is complete for each iteration. For this, we arrange for page faults to occur on the model, which is only accessed after a complete minibatch is sampled. At this point, bias counters are reset and the attacker is ready to listen for page faults in the next minibatch sample phase.

RIGHTSLINK

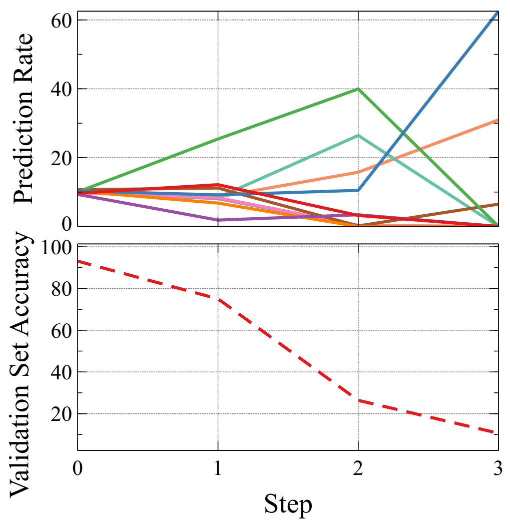


Figure 7. Indiscriminate attack using learning rate (Section 4.1). The x-axis denotes the number of malicious updates. In the top graph, the different lines correspond to the prediction rate for each of the ten labels in CIFAR-10. (Which line corresponds to which label isn't important, as changes are random in this attack.)

so the baseline unbiased model should have a prediction rate of 10% on average for each label. Lower/higher prediction rates than 10% for a given label indicate the model is biased against/towards those labels.

Network divergence. If model accuracy drops below 15% on the validation set, we stop plotting results for that run. Recall from Section 5.1, the un-attacked model's accuracy is $> 90\%$.

7.1 V1: Indiscriminate Attack (Section 4.1)

Figure 7 shows a representative indiscriminate attack that uses a stale learning rate. The x-axis shows the number of malicious updates made by a single attack thread.

The takeaway is that the model's validation accuracy decreases as expected, and decreases further given more attack updates. We have found that most of the time, it takes multiple attack updates to fully diverge the model. We experimented with a smaller DNN (Lenet-5 [42]) and found that fewer updates is sufficient to do more damage reliably (not shown for space). In general, Resnet-18 (a larger DNN with more layers) seems more robust and requires multiple malicious updates.

7.2 V2: Focused Attack (Section 4.2)

Figure 8 shows the effect of focused attacks on representative attack labels[9] and for different minibatch sizes and % bias

[9]There are 10 total attack labels for CIFAR-10, but they show similar trends to the three we show here.

Table 1. Select prediction rates towards the target labels T1, T3, T7 in Figure 8. Un-attacked models have an expected prediction rate of ~ 10% towards each label.

			% prediction rate		
B	% minibatch bias	# threads	T1	T3	T7
128	20	4	11.1	16.7	17.3
64	30	2	32.9	26.4	36.6
32	40	1	23.1	37.3	36.4

thresholds. The y-axis plots prediction rate and the x-axis plots the number of attack threads (Section 4.2.4). All results are averaged over 10 runs and start from the same initial model state (for each minibatch size and target label).

There are several takeaways. As minibatch bias increases, prediction rate towards the target label increases. Interestingly, the effect is "cleaner" with large minibatches. That is, with larger minibatches the attack more consistently increases prediction rate towards the attack label and decreases the prediction rate for *all* other labels. We attribute this to larger minibatches generating better quality gradients (as larger minibatch sizes better represent the training set). By contrast, smaller minibatch experiments are noisier. In the extreme case, experiments with $B = 32$ tend to cause model divergence given a sufficient number of attack threads.

Based on training parameters and likelihood to see biased minibatches (Section 4.2.2), we expect to see 20% biased minibatches given $B = 128$, 30% given $B = 64$ and 40% given $B = 32$. For each of these minibatch sizes, we select a fixed number of attack threads (which corresponds to the adversary's strategy) and show prediction rates towards the three target labels in Table 1. (That is, Table 1 highlights results already present in Figure 8.)

We chose the configurations in Table 1 because these represent practical attacks that can be carried out given the current training parameters (e.g., number of epochs per decay). Some other configurations are not practical—given current parameters—but are shown for completeness. For example, we do not expect to see 30% biased minibatches with $B = 128$ according to Figure 3a. Depending on public training parameters, the adversary may need to select different attack parameters.

Impact of minibatch bias. Figure 8 shows that % minibatch bias has a large impact on the eventual model bias (confirming the premise from Section 4.2). To give more insight, Figure 9 examines the impact of additional minibatch bias %s, all the way up to 100% biased minibatches. We see that up to 50% biased minibatches, model bias increases as expected. Beyond 50% minibatch bias, results are unpredictable. We observed a similar effect when targeting other labels.

This is counter-intuitive given the explanation in Section 4.2.3. One explanation is that due to the use of a large learning rate, results should be unpredictable. This theory

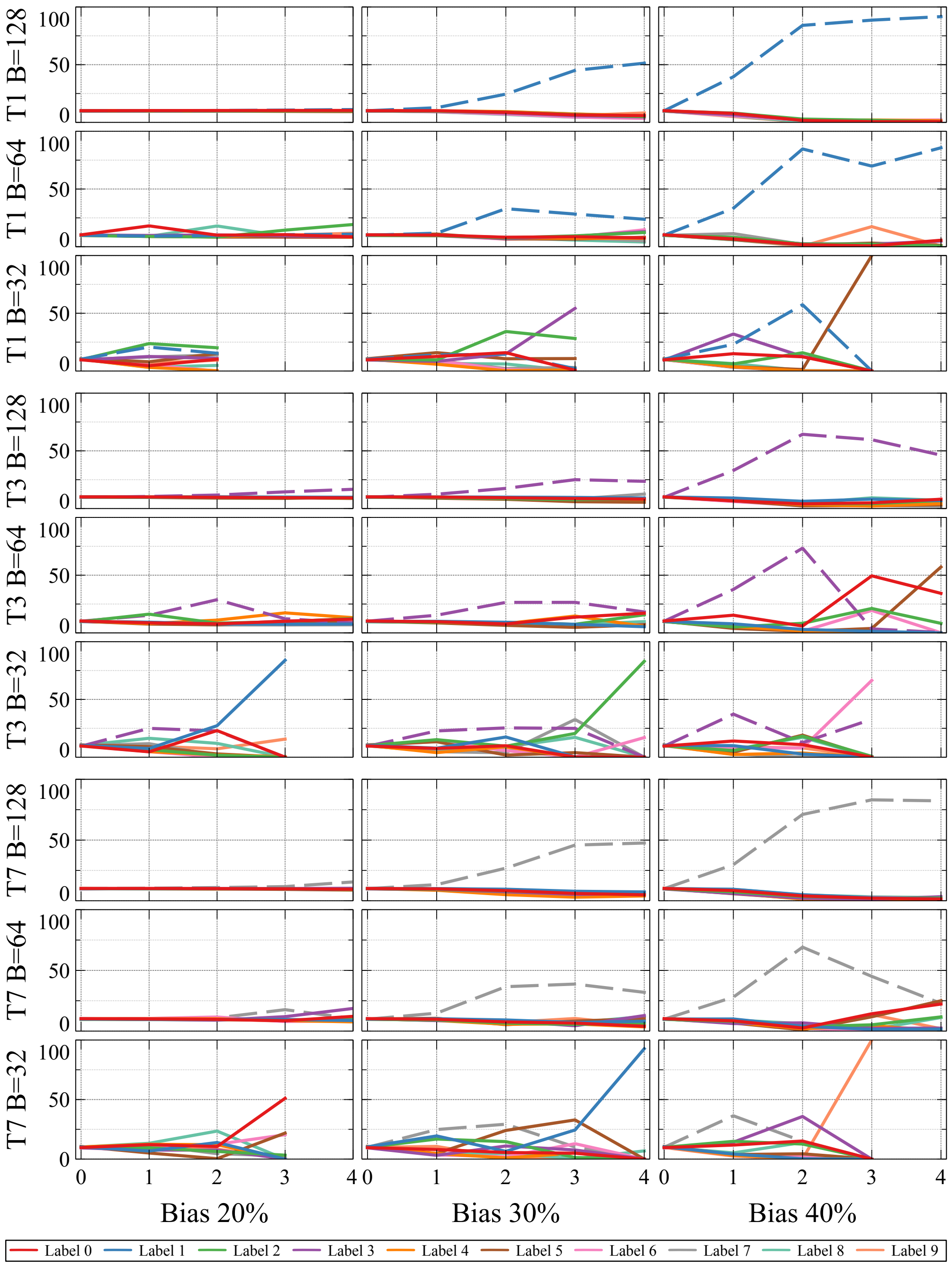


Figure 8. Focused attack results (Section 4.2) for Resnet. Each graph corresponds to a specific target label (TX is target label X) and a specific minibatch bias %. The y-axis shows the prediction rate towards each label. The target label is shown using a dashed line. The x-axis shows the number of attack threads. Attack updates are applied consecutively. Note, if data is not plotted it is because the model diverged (Section 7).

is undermined by results in Section 7.3, where we see the same behavior with a small learning rate. An alternate hypothesis is that since highly biased minibatches never occur in practice, the model is not robust to them.

7.3 V3: Stale Parameter-Based Attacks (Section 4.3)

Figure 10 shows an attack space for our stale parameters-based attack (Section 4.3) given minibatch size $B = 32$. In all experiments, we arbitrarily chose to look for minibatches biased towards label 0. As with the previous section, all

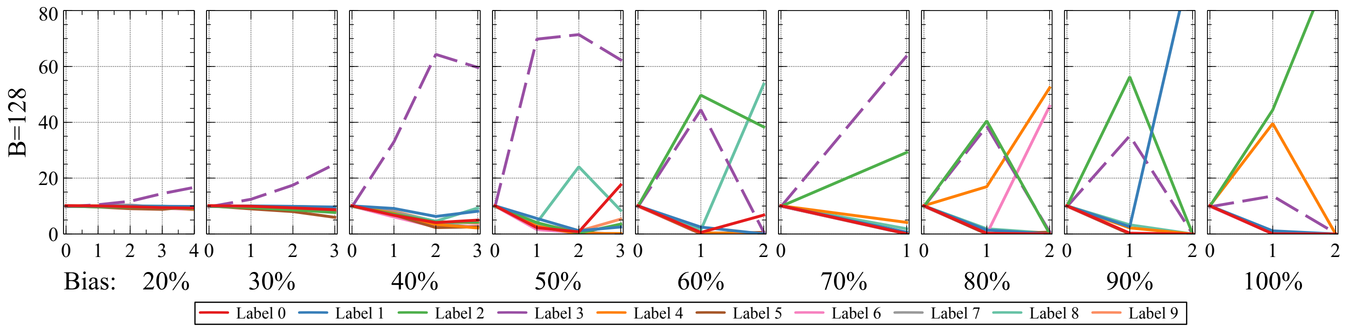


Figure 9. Focused attack results (Section 4.2) for target label 3, varying minibatch bias. Graphs (x-, y-axes) are in the same format as in Figure 8.

results are the average of 10 runs. All attack thread updates are made with the same (fully decayed) learning rate as honest thread updates.

There are several takeaways. First, as minibatch bias increases, smaller N is sufficient to decrease model accuracy. Since gradient updates have the same small magnitude in every experiment, we see this as evidence that supports our claims regarding constructive gradients. Second, the model does bias towards the target label, and bias is generally proportional to the number of attack threads. In fact, we observe a similar trend as seen in Figure 9: increased minibatch bias leads to increased model bias, but not for 100% biased minibatches. We conjecture the reasons for this are similar to those explained in Section 7.2.

As with Figure 8, a subset of the results in Figure 10 are practical given our training parameters. For example, given our parameters we expect $\sim$ 1000 minibatches with bias 28% or more, and therefore expect effects similar to the 30% bias attack in Figure 10 to occur in practice.

8 Possible Defense Approaches

We now review possible defense approaches against asynchronous poisoning attacks (APAs). The root cause of these attacks is malicious thread scheduling to influence model state. We proposed three concrete APAs, based on stale learning rate and stale model parameters, however others are likely possible. We will discuss both specific mitigations to specific attacks and more general defense approaches.

Two main takeaways are: (a) existing defenses do not block all known variants of the attack, (b) there may be other variants we have not yet discovered. As such, we believe future research is needed to formally study attacker capabilities in this setting, so as to enable comprehensive and performant defenses.

Restricting thread scheduling/staleness. One direction to aggravate (and sometimes mitigate) our attacks is to disallow certain thread schedulings. Specifically, to mitigate our learning rate-based attacks (Sections 4.1 and 4.2) in the shared memory model, we recommend synchronizing worker threads at a course-, e.g., epoch-, granularity. Mitigating our stale parameter-based attack (Section 4.3) requires different mitigations, as it does not require de-scheduling threads for long amounts of time.

Beyond our setting, restricting thread staleness is an important technique to improve A-SGD convergence rates [17, 18, 31, 37, 41, 52, 71, 78, 79]. Importantly, techniques designed not specifically to mitigate APAs are only partially (or not at all) effective against APAs. For example, [78] proposes restarting a thread's inner loop if it has not made an update for $> P$ updates made by other threads (where P is configurable). This is not effective at mitigating our learning rate-based attacks as restarting the inner loop has no impact on the learning rate. This attack can be effective at mitigating the stale parameters-based attack if the number of attack threads is $> P$, but this parameter would need to be carefully tuned with guidance from our analysis (Section 7).

Byzantine A-SGD. Byzantine defenses, such as [11, 14, 19], develop methods to protect training from a bounded number of malicious or faulty workers. A basic limitation in byzantine approaches is that they assume a threshold on the number of malicious worker threads. An SGX attacker can corrupt more or even all worker threads for arbitrary periods of time, violating byzantine assumptions. Further, arbitrary adversarial thread scheduling allows an attacker to side-step the assumptions made by individual schemes. For example, [11] does not consider stale model parameter-based attacks in their threat model (Section 4.3).

A-SGD design decisions. The popularization of A-SGD has led to the development of various frameworks. We highlight some popular architectural features which help aggravate APA as a coincidental side effect. First, some frameworks [15, 31, 33, 37, 43, 71, 80] keep the learning rate on dedicated parameter servers, which keeps it out of the adversary's control. This defeats our learning rate based attack, as pausing the parameter server would cause all of training to

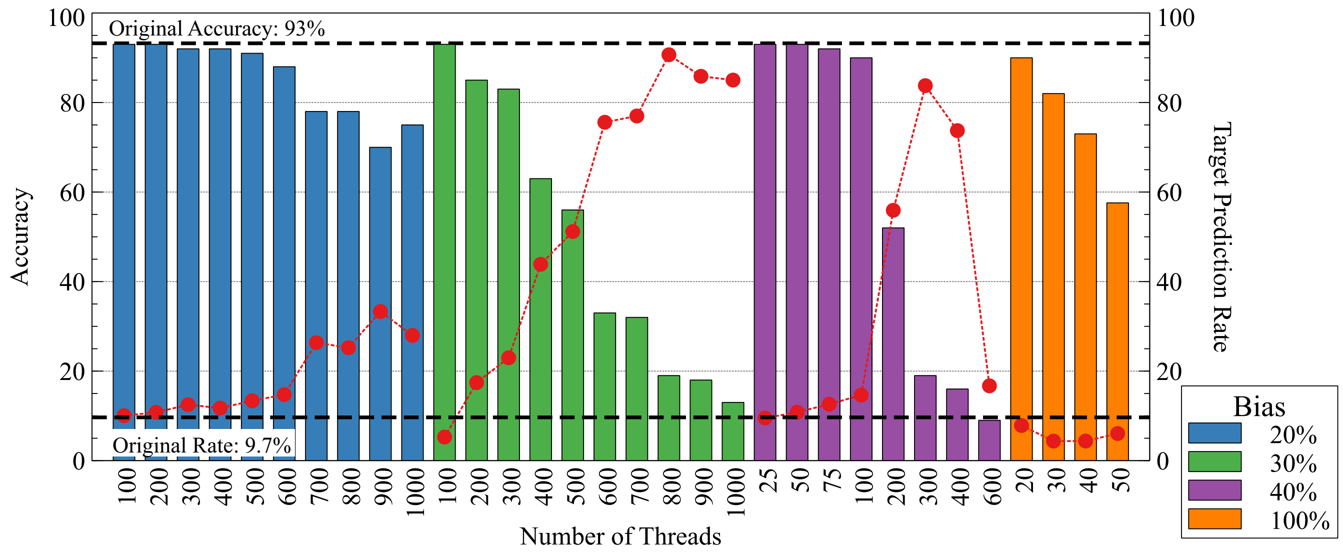


Figure 10. Stale parameters-based attack (Section 4.3) given $B = 32$. The number of threads corresponds to N from Section 4.3. The red dots indicate model bias towards the target label (label 0) after all N attack threads apply their update.

pause. A notable exception is the SGX-based A-SGD framework Chiron [33], which opts to keep the learning rate local to worker threads, to minimize parameter server computation. Thus, Chiron is still susceptible to all proposed variants of APA.

Many of the above systems propose multiple parameter servers for load balancing, fault tolerance, etc. If the learning rate is stored locally in that case, the attack may still go through. For example, if when pausing one parameter server others are allowed to continue, this is akin to pausing a worker thread with a high learning rate. We leave investigating this direction as future work.

All of the above systems remain vulnerable to a stale parameters-based attack. Furthermore, there is a recent line of work that tries to replace centralized parameter servers with peer-to-peer communication to remove bottlenecks [70]. This approach is, once again, susceptible to learning rate-based attacks.

Defending against specific attack components. Finally, there are spot mitigations to specific (not fundamental) parts of various APAs, which are unlikely to fully quell the vulnerabilities and/or can lead to significant performance overhead. For example, we chose controlled-channel attacks to observe biased minibatches for simplicity, and there are known mitigations (e.g., T-SGX [60]) to block these channels. Yet, even without controlled-channel attacks, there are many low-noise side channels available to the SGX attacker (Section 2.4). Alternatively, the user can perform an oblivious sort over both the private and public components of the training set. Unfortunately, this is likely to incur huge overhead: oblivious sort with low hidden constants costs $n \log^2 n$ oblivious swaps where n in modern datasets can be in the hundreds of GB [36]. We note that using an alternative iterator to index into unsorted data is not sufficient: the data itself must be shuffled to block side channels.

9 Related Work

Asynchronous stochastic gradient descent. Stochastic gradient descent (SGD) has become ubiquitous in modern machine learning training and asynchronous SGD (A-SGD) has become a key tool for scaling SGD to larger data sizes. Starting with Hogwild! [51], there has been significant work to analyze the statistical [23, 32, 44, 46, 51, 55, 69] and/or machine efficiency [15, 20, 21, 58, 76] of A-SGD. Originally, A-SGD was designed to train sparse, convex problems. Recent work shows how A-SGD can be applied to non-convex error surfaces such as those found in deep learning, which is studied in this paper [15, 21, 46]. In particular, Google's Downpour [21] and Microsoft's Project Adam [15] both give large-scale exemplar systems that demonstrate how 10s to 1000s of CPUs running A-SGD can outperform GPU-based DNN training.

Cyclades [55] is a framework that performs deterministic A-SGD. While this determinism can be used to defeat APAs, Cyclades heavily relies on the dataset being sparse. This precludes it from being used as a defense for some problem domains, such as neural network training.

In the A-SGD literature, the closest work to that presented in our paper is [6]. That work discusses how an adversary can slow convergence by influencing scheduling. That work does not discuss focused damage, nor does it consider a scenario where an adversary is seeking to damage the final accuracy of a model; it seeks only to study denial-of-service attacks that delay convergence. Our work influences scheduling to break model integrity.

Poisoning attacks. Poisoning attacks are a known training-time attack on model integrity [10, 48–50, 56, 73]. Traditional poisoning attacks require the adversary to change the training set. We call our attacks *asynchronous* poisoning attacks because instead of requiring a change to the training set, our attacks exploit thread asynchronicity.

Enclave-based trusted execution. We assume an enclave-based trusted execution environment [64] (TEE) to defeat trivial and conventional poisoning attacks. Embodied by Intel's SGX [45], these TEEs have seen significant adoption in securing user-level applications across domains [8, 9, 61]. Our deployment of A-SGD with enclaves is similar to Panoply [61] or Chiron [33], and only requires the different enclave threads be able to share memory, which SGX currently supports. We note that while there are only commercial enclave abstractions for CPUs, there is currently work that extends similar concepts to other platforms such as GPUs [35].

The closest SGX-related work to our attack is AsyncShock [72], which uses thread scheduling to turn synchronization bugs into integrity attacks on enclave applications. Relative to AsyncShock, our attack doesn't exploit application bugs, and shows how an important class of applications, even bug-free, are still susceptible to integrity attacks when protected by enclave-based TEEs. More importantly, we view a large part of our contribution to be showing *how* thread scheduling can impact a highly stochastic algorithm like A-SGD in a controllable way.

Side channels in enclave systems. See Section 2.4 for citations and more detail.

10 Conclusion

This work presents *asynchronous poisoning attacks*, which break machine learning training integrity by maliciously scheduling asynchronous worker threads. We demonstrate both indiscriminate and focused attacks, showing how an adversary can reduce model accuracy and even bias the model towards a particular adversary-chosen label.

The attack root cause is that model updates during asynchronous-SGD are racy, which is fundamental to asynchronous training. There are many additional, subtle contributing factors that make attacks easier to launch. Thus, we believe it is important for future work to formally study what damage a supervisor-level adversary can do during training, so as to develop holistic and provable defenses.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was supported by NSF awards #1816226 and #1725734, as well as by an Intel ISRA center.

References

[1] 2016. Google Machine Learning. https://cloud.google.com/products/machine-learning/.
[2] 2017. Amazon Machine Learning. https://aws.amazon.com/machine-learning/.
[3] 2018. Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/intel/mkl-dnn.
[4] 2019. Apache MXNet. https://mxnet.apache.org/.
[5] 2019. PyTorch Hogwild Implementation for DNNs. https://github.com/pytorch/examples/tree/master/mnist_hogwild.
[6] Dan Alistarh, Christopher De Sa, and Nikola Konstantinov. 2018. The convergence of stochastic gradient descent in asynchronous shared memory. In *PODC'18*.
[7] T Alves and D Felton. 2004. Trustzone: Integrated hardware and software security. *ARM white paper* (2004).
[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Mark L Stillwell, David Goltzsche, David Eyers, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI'16*.
[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with Haven. In *OSDI'14*.
[10] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. In *ICML'12*.
[11] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS'17*.
[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *WOOT'17*.
[13] João Carreira and Andrew Zisserman. 2017. Quo Vadis, action recognition? A new model and the kinetics dataset. In *CVPR'17*.
[14] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. DRACO: Byzantine-resilient distributed training via redundant gradients. In *ICML'18*.
[15] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI'14*.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. In *IACR'16*.
[17] Wei Dai. 2018. Learning with Staleness.
[18] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. 2018. GossipGraD: Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent. *arXiv (2018)*.
[19] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, and Mahsa Taziki. 2018. Asynchronous byzantine machine learning (the case of sgd). In *ICML'18*.
[20] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ISCA'17*.
[21] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *NIPS'12*.
[22] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. In *JMLR'11*.
[23] John C. Duchi, Michael I. Jordan, and H. Brendan McMahan. 2013. Estimation, optimization, and parallelism when data is sparse. In *NIPS'13*.
[24] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. In *JCEN'16*.
[25] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD

RIGHTSLINK Copyright Clearance Center

Architectures. In *SC'18*.

[26] Zhangxiaowen Gong, Houxiang Ji, Christopher Fletcher, Christopher Hughes, and Josep Torrellas. 2019. SparseTrain: Leveraging dynamic sparsity in training DNNs on general-purpose SIMD processors. *arXiv (2019)*.

[27] Johannes Gotzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Mller. 2017. Cache attacks on intel SGX. In *EuroSec'17*.

[28] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games - Bringing access-based cache attacks on AES to practice. In *IEEESSP'11*.

[29] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *ATC'17*.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv (2015)*.

[31] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS'13*.

[32] Cho Jui Hsieh, Hsiang Fu Yu, and Inderjit S. Dhillon. 2015. PASSCoDe: Parallel asynchronous stochastic dual co-ordinate descent. In *ICML'15*.

[33] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv (2018)*.

[34] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient Deep Learning on Multi-Source Private Data. *arXiv (2018)*.

[35] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity GPUs. In *ASPLOS'19*.

[36] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *CVPR'09*.

[37] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *SIGMOD'17*.

[38] Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *ICLR'15*.

[39] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS'12*.

[41] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. 2017. ASAGA: Asynchronous parallel saga. In *AISTATS'17*.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE'98*.

[43] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *OSDI'14*.

[44] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2017. Perturbed iterate analysis for asynchronous stochastic optimization. In *SIAM'17*.

[45] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution.

[46] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Re. 2016. Asynchrony begets momentum, with an application to deep learning. In *Allerton'16*.

[47] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. *arXiv (2017)*.

[48] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I.P. Rubinstein, Udam Saini, Charles Sutton, J. D. Tygar, and Kai Xia. 2008. Exploiting machine learning to subvert your spam filter. In *LEET'08*.

[49] Andrew Newell, Rahul Potharaju, Luojie Xiang, and Cristina Nita-Rotaru. 2014. On the practicality of integrity attacks on document-level sentiment analysis. In *AISec'14*.

[50] James Newsome, Brad Karp, and Dawn Song. 2006. Paragraph: Thwarting signature learning by training maliciously. In *RAID'06*.

[51] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. HOGWILD!: A lock-free approach to parallelizing Stochastic Gradient Descent. In *NIPS'11*.

[52] Augustus Odena. 2016. Faster Asynchronous SGD. *arXiv (2016)*.

[53] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin, Kapil Vaswani, Manuel Costa, and Aastha Mehta. 2016. SGX-enabled oblivious machine learning. In *Security'16*.

[54] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin, Kapil Vaswani, Manuel Costa, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *Security'16*.

[55] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I. Jordan, Kannan Ramchandran, Chris Re, and Benjamin Recht. 2016. CYCLADES: Conflict-free asynchronous machine learning. In *NIPS'16*.

[56] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Foglat, and Monirul Sharif. 2006. Misleading worm signature generators using deliberate noise injection. In *IEEESSP'06*.

[57] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Security'15*.

[58] Scott Sallinen, Nadathur Satish, Mikhail Smelyanskiy, Samantika S. Sury, and Christopher Re. 2016. High Performance Parallel Stochastic Gradient Descent in Shared Memory. In *IPDPS'16*.

[59] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS'18*.

[60] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS'17*.

[61] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS'17*.

[62] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: Enabling microarchitectural replay attacks. In *ISCA'19*.

[63] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2015. Striving for simplicity: The all convolutional net. In *ICLR'15*.

[64] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A formal foundation for secure remote execution of enclaves. In *CCS'17*.

[65] Chia Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *EUROSYS'14*.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-step: A practical attack framework for precise enclave execution control. In *SysTEX'17*.

[67] Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. 2013. Regularization of neural networks using DropConnect. In *ICML'13*.

[68] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, Xiao Feng Wang, Vincentchenguo Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS'17*.

[69] Yu-Xiang Wang, Veeranjaneyulu Sadhanala, Wei Dai, Willie Neiswanger, Suvrit Sra, and Eric P Xing. 2016. Parallel and Distributed Block-coordinate Frank-Wolfe Algorithms. In *ICML'16*.

[70] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. 2016. Ako: Decentralised deep learning with partial

gradient exchange. In *SoCC'16*.

[71] Zhang Wei, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-aware async-SGD for distributed deep learning. In *IJCAI'16*.

[72] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in intel SGX enclaves. In *LNCS'16*.

[73] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. 2015. Is feature selection secure against training data poisoning?. In *ICML'15*.

[74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP'15*.

[75] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W. Fletcher. 2019. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS'19*.

[76] Hyokun Yun, Hsiang Fu Yu, Cho Jui Hsieh, S. V.N. Vishwanathan, and Inderjit Dhillon. 2014. NOMAD: Nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. In *VLDB'14*.

[77] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *arXiv (2012)*.

[78] Chengliang Zhang, Huangshi Tian, Wei Wang, and Feng Yan. 2018. Stay fresh: Speculative synchronization for fast distributed machine learning. In *ICDCS'18*.

[79] Sixin Zhang, Anna Choromanska, and Yann Lecun. 2015. Deep learning with elastic averaging SGD. In *NIPS'15*.

[80] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi Ming Ma, and Tie Yan Liu. 2017. Asynchronous stochastic gradient descent with delay compensation. In *ICML'17*.

A Artifact Appendix

A.1 Abstract

This artifact describes the frameworks used for our evaluation. The frameworks consist of two main components: A Pytorch Component, and an SGX Proof of Concept. The Pytorch Component can be used to replicate the machine learning results from Section 7. These results can be replicated on any machine which can run Python, although errors may be encountered if CUDA is not available. The code of this component allows for training a baseline, simulating or executing a full OS-managed attack for the variants described in Sections 4.1 & 4.2, and simulating the variant from Section 4.3. The SGX PoC consists of an SGX application and a kernel module, which can be used to replicate the results from Section 6. This artifact was validated on a bare-metal machine with Ubuntu Linux, using a Intel i7-6700K CPU with Intel SGX (albeit this requirement could be relaxed by using SGX in simulation mode). The SGX Application does not fully train a network; it loads the CIFAR-10 data set into enclave memory, and spawns multiple threads which asynchronously sample batches and accumulate data into shared memory. The kernel module contains the logic to perform a controlled-channel attack [74], which monitors data sampling, and the code to halt and release the worker threads of the SGX application for the attack.

A.2 Check-list (meta-information)

- **Programs:** Pytorch (sources included).
- **Data set:** CIFAR-10 (download script included).
- **Run-time environment:** The provided kernel module code (used in the SGX PoC) is specific to Linux (Ubuntu 16.04) and running it requires root access to a bare-metal machine; the Intel SGX SDK is the main software dependency.
- **Hardware:** We recommend a system with CUDA for the Pytorch evaluation artifact; we recommend a bare-metal machine with an SGX-enabled Intel CPU for the SGX PoC evaluation artifact (CPUs without SGX might work too if SGX is used in simulation mode, but we did not test them).
- **Metrics:** Validation accuracy, model bias, attack success.
- **Output:** The Pytorch artifact outputs accuracy and confidence logs. The SGX PoC artifact outputs informational logs on the attack execution and results.
- **Experiments:** Scripts and instructions to fully reproduce the paper's results are provided in the artifact README files.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours to train baseline on GPU; between 5 minutes and 1 hour to download, compile and install the custom kernel and setup the SGX SDK.
- **How much space is needed (approximately)?:** About 160 MB are required for CIFAR-10. Log space varies based on how much run time is allowed to proceed for, as logs are gathered periodically. In the Pytorch artifact, output logs are compressed for convenience.
- **How much time is needed to complete experiments (approximately)?:** 30 minutes to simulate attacks. 8+ hours to perform an OS managed attack during training. 5 minutes to run the attack on the SGX PoC.
- **Publicly available?:** Yes, on Zenodo (see Section A.3.1).
- **Code licenses:** All code is covered by the University of Illinois/NCSA Open Source License.

A.3 Description

A.3.1 How delivered

The full artifact is available on Zenodo at the following URL: https://doi.org/10.5281/zenodo.3628042.

A.3.2 Hardware dependencies

A bare-metal machine with Intel CPU (preferably SGX-enabled) is required to run the SGX PoC from Section 6. Running the SGX PoC in a virtualized environment may incur issues due to nested page faults. We recommend a system with CUDA for the Pytorch evaluation artifact used in Section 7.

A.3.3 Software dependencies

- **Pytorch based PoCs:** Pytorch V1.2.0; Python 3.7
- **SGX based PoC:** Ubuntu 16.04; Intel SGX driver; Intel SGX PSW; Intel SGX SDK.

A.3.4 Data sets

Our Pytorch artifact utilizes the CIFAR-10 dataset. Instructions and a script to download it are provided in the artifact.

A.4 Installation

The Pytorch PoC requires no installation. Detailed instructions on using and running the code, including its arguments, are included in its README file in the artifact. The SGX-PoC code requires

RIGHTSLINK

installing a custom kernel, booting into this kernel, and loading a kernel module on it. Detailed instructions on these steps are included in its README file in the artifact.

A.5 Experiment workflow

A typical flow for simulating the attack with the Pytorch PoC is described below. All the operations below are undertaken using the same run script. The second step can be repeated with varying targets and biases for comparison. The third step can be repeated with varying numbers of stages and steps for comparison.

1. Train a baseline, generating a checkpoint. This checkpoint will be used as the simulation start point for subsequent steps.
2. Measure the effect of a focused attack, by specifying the 'simulate' option of the run script. An indiscriminate attack can be simulated by specifying a bias of 10 percent.
3. Measure the effect of a stale parameters-based attack, by specifying the 'simulate-multi' option of the run script.

Furthermore, an OS-managed attack can be demonstrated by using the APA zsh file included.

The full experimental workflow for validating the SGX PoC evaluation is described in the README file in the artifact. Essentially, it consists of four steps:

1. Install the custom Linux kernel.
2. Load the attack kernel module.
3. Launch the victim SGX Application.
4. Check the kernel logs to see if the attack worked.

A.6 Evaluation and expected result

Running the Pytorch PoC will allow for real time observations of validation accuracy. As the attack proceeds, the validation accuracy will change in observable ways. Furthermore, various logs will be generated for the confidence of each sample in the validation set for every possible label. These logs can be used to compute prediction rates and bias rates.

A successful run of the SGX PoC evaluation will result in the user-space SGX application completing its execution without crashing and the attack kernel module logging progress on halting and resuming threads with a final log message saying "Attack complete".

A.7 Experiment customization

The Pytorch PoC allows for different networks and datasets to be dropped in by simple modifications. This PoC also allows for simple specification of target label, minibatch bias amount, number of attacks, and even network recovery time after an attack.