`Falkor`: Federated Learning Secure Aggregation Powered by `AES-CTR` GPU Implementation

Mariya Georgieva Belorgey[1], Sofia Dandjee[2⋆], Nicolas Gama[3⋆], Dimitar Jetchev[1], Dmitry Mikushin[4⋆]

[1] Inpher
[2] Netlight
[3] SandboxAQ
[4] Faculty of Business and Economics, University of Lausanne

Abstract. We propose a novel protocol, `Falkor`, for secure aggregation for Federated Learning in the multi-server scenario based on masking of local models via a stream cipher based on `AES` in counter mode and accelerated by GPUs running on the aggregating servers. The protocol is resilient to client dropout and has reduced clients/servers communication cost by a factor equal to the number of aggregating servers (compared to the naïve baseline method). It scales simultaneously in the two major complexity aspects: 1) large number of clients; 2) highly complex machine learning models such as CNNs, RNNs, Transformers, etc. The `AES-CTR`-based masking function in our aggregation protocol is built on the concept of *counter-based* cryptographically-secure pseudorandom number generators (csPRNGs) as described in [1] and subsequently used by Facebook for their `torchcsprng` csPRNG. We improve upon `torchcsprng` by careful use of shared memory on the GPU device, a recent idea of Cihangir Tezcan [2] and obtain 100x speedup in the masking function compared to a single CPU core.
In addition, we prove the semantic security of the `AES-CTR`-based masking function. Finally, we demonstrate scalability of our protocol in two real-world Federated Learning scenarios: 1) efficient training of large logistic regression models with 50 features and 50M data points distributed across 1000 clients that can dropout and securely aggregated via three servers (running secure multi-party computation (SMPC)); 2) training a recurrent neural network (RNN) model for sentiment analysis of Twitter feeds coming from a large number of Twitter users (more than 250,000 users). In case 1), our secure aggregation algorithm runs in less than a minute compared to a pure MPC computation (on 3 parties) that takes 27 hours and uses 400GB RAM machines as well as 1 gigabit-per-second network. In case 2), the total training is around 10 minutes using our GPU powered secure aggregation versus 10 hours using a single CPU core.

1 Introduction

Federated Learning [3,?,?] is a distributed machine learning approach that deploys a supervised training algorithm to a large collection of remote clients (e.g., mobile phones or IoT edge devices). Each client trains a local model on its private data and sends training updates to one or more servers for aggregation to compute the global model update that is then sent back to the clients for the next iteration of the optimization algorithm.

While the client input data does not leave its source (*a priori*, a significant privacy advantage compared to centralized training), sending intermediate local model updates to the server can indirectly reveal sensitive client information, an important security aspect that has often been ignored in the vast literature on the subject of federated machine learning.

In addition, the communication overhead between the clients and the servers in this distributed computing scenario is a major challenge for scaling a federated learning system to support a large number of clients and complex machine learning models (e.g., deep neural networks) at the same time. Yet, compressing the model without compromising the model accuracy is a challenging open problem.

Finally, unlike the classical MPC setting with few compute nodes and robust peer-to-peer communication, Federated Learning involves thousands and even millions of devices that may go offline

⋆ This work was done while the author was working for Inpher during the computation (dropouts) or that may not have stable connections with the server. Designing protocols that are resilient to dropouts is thus of primary interest.

Therefore, ensuring simultaneously the above security, scalability, model accuracy and robustness guarantees is of primary importance in the design of real-world federated learning systems.

1.1 Prior works.

For a detailed overview of the state-of-the-art in Federated Learning as well as the open problems, we refer the reader to [4]. Considering the immense literature on the subject, providing a complete overview of the state of the art is challenging and would go beyond the scope of the current paper. Instead, we outline the key areas of contributions of our work and compare against the state-of-the-art for these particular areas:

1. *Security:* securing the local input private data from a honest-but-curious aggregating server
2. *Scalability:* ensuring scalability to both large machine learning models and a large number of clients while maintaining efficient communication complexity,
3. *Robustness against client dropouts:* ensuring robustness of the protocol against unstable network connections between clients and servers as well as dropouts of clients during training,
4. *Model accuracy (compared to centralized training):* ensuring that the model accuracy is preserved compared to centralized training, especially in settings when the private data across the different clients is not independent and identically distributed (the non-IID setting).

For 1), if the clients do not secure the local model updates, the aggregation server can infer information about the original private input data on the clients via the so-called "model inversion attacks" [5,**?**] (see also [6]). The server being able to reconstruct peer's original private data poses security risks for certain federated learning frameworks as Fleet [7,**?**]. In order to prevent these attacks, various methods for privacy-preserving server aggregation have been proposed in the literature: differential privacy [8,**?**], pairwise additive masking [9], homomorphic encryption, glimmer of trust [10] and more recently, additive [11] and FFT-based secret sharing [12].

For instance, the pairwise additive masking approach [9] secures the aggregation using Shamir secret sharing in the single aggregation server scenario. While this method is adapted to ensure client dropouts, its original version incurred communication at each client that was linear in the number of clients, thus limiting its scalability. The extension [13] improved upon that constraint in the semi-honest and semi-malicious settings, thus addressing scalability constraints. Yet, [13] assumes communication between the clients as well as multiple rounds of communication between the clients and the servers at each iteration.

While many the secure aggregation protocols mentioned above are in the setting of a single aggregation server, there have been attempts to address the multi-server setting as well via generic MPC [14], [15] as well as in the setting of multiple non-colluding servers [16].

A recent approach on secure aggregation in the multi-server scenario has been proposed in [15]. The aggregation relies on additive secret sharing and secure multiparty computation, and reduces model complexity using model compression techniques based on several prior works [17,**?**]. Even if it supports more complex machine learning models via these model compression techniques that reduce the size of the model (and hence, the communication overhead) while maintaining model accuracy similar to centralized training for specific neural networks such as LeNet on MNIST and AlexNet on CIFAR-10, unlike [13], it is constrained by the number of clients in a pure MPC protocol (typically less than 10) and assumes stable communication between these clients. In addition, it assumes that each client is online during each iteration (no dropout support) and requires multiple rounds of communication per iteration.

The prior work on `AES` GPU implementations is discussed in Section 6.

1.2 Our contributions.

Our contributions are three-fold: first and foremost, we achieve highly efficient GPU implementation of the masking function via acceleration of the `AES-CTR`-based stream cipher (see Sections 5, 6 and 7). This yields two orders of magnitude speedup compared to an implementation on a single core CPU. Our implementation relies on storing `AES` tables in shared as opposed to constant memory, an idea inspired by the recent work of Cihangir Tezcan [2]. This improves upon prior art on massively parallel counter-based PRNGs including Facebook's `torchcsprng` to gain up to 20x speedup for generating parallel random numbers. It is worth mentioning that although we do apply our GPU implementation primarily in the context of federated learning, this is only one of the many possible applications. In addition, we analyze and prove the semantic security of our masking function in Section B.

The second major contribution is the `Falkor` secure aggregation protocol itself and it uses our first contribution (the `AES-CTR` GPU masking function) as a major building block (see Sections 2 and 3). `Falkor` scales both model complexity (deep neural network models such as CNNs, RNNs, Transformers and others) and a large number (millions) of clients. It supports linear aggregation functions (e.g., sums and concatenations) and supports client dropouts and unstable communications between the clients and the servers. An important property of our protocol is that, unlike [13], it does not require any communication between the clients and it uses a single round of communication at each iteration.

The protocol operates in the setting of multiple aggregation servers performing secure multiparty computations (SMPC). Compared to the naïve protocol where each client secret shares its local model updates and sends the shares to the aggregating servers (a protocol whose communication complexity depends linearly on the number ℓ of aggregating servers, the total number of training samples N and the model size), our aggregation protocol reduces the communication complexity by a factor ℓ, thus making it independent of ℓ. The main idea behind the optimized communication and dropout resilience is to use shared keys between the clients and the servers together with the GPU accelerated `AES-CTR`-based masking function and massive parallelization of the secure aggregation phase (the blockwise reduction described in Section 7).

The third major contribution is the capability of chaining our aggregation protocol `Falkor` with non-linear MPC operations on the aggregation servers. The use of MPC for more complex computations than simple averaging such as private divisions, multiplications, oblivious sorting and comparisons as well as oblivious permutations allows us to use more complex privacy-preserving versions of adaptive server optimizers such as `FedAdagrad`, `FedAdam` and `FedYogi` to ensure faster convergence in the deep neural network setting. We illustrate the scalability of our method by two major examples: 1) scaling a large logistic regression model training and comparing its scalability to a pure MPC approach as described in Section 9; 2) training a recurrent neural network (RNN) model for sentiment analysis on a Twitter dataset where each client is a Twitter user (see Section 10).

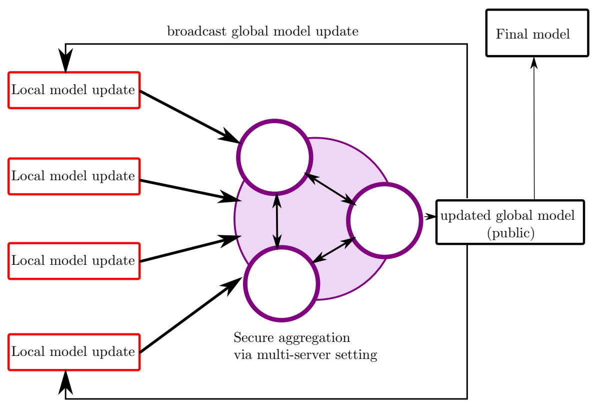
broadcast global model update
Final model
Local model update
Local model update
Local model update
Local model update
updated global model (public)
Secure aggregation via multi-server setting

Fig. 1. Baseline secure federated learning with secure MPC aggregation

2 Falkor: a Federated Learning Secure Aggregation Protocol

The design of `Falkor` ensures privacy-protection of the local data of the clients. The baseline naïve approach for secure federated averaging is simple (Fig. 1): secret share each local model (on each client) among ℓ servers and distribute the secret shares from the client to the servers. The servers securely aggregate the updated global model using multiparty computation, they reveal the update to the clients and optionally perform some additional MPC-computations. The training then proceeds to the next iteration. This requires communicating a total of $n\ell$ secret shares per single averaging resulting in complexity $O(n\ell b)$ where n is the number of clients and b is the size (in bits) of a single secret share. This approach requires that no share is lost during the communication. The idea of `Falkor` is to use

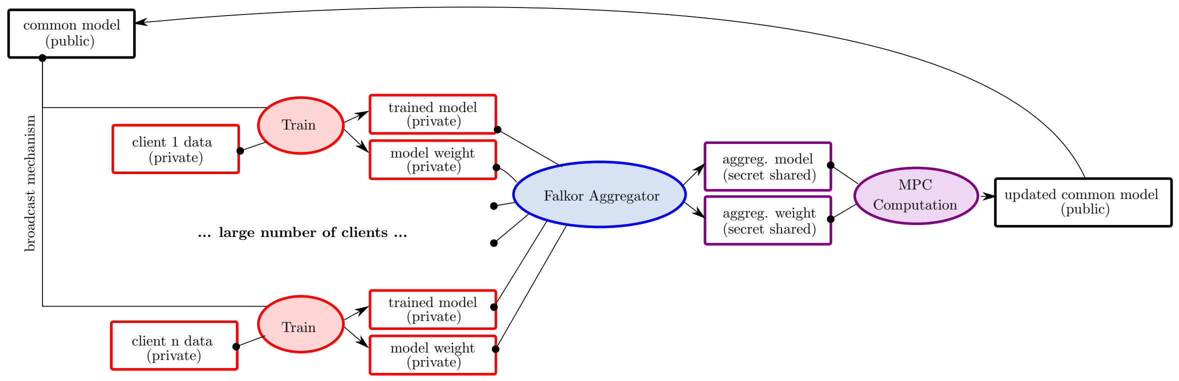

Fig. 2. `Falkor` federated learning execution graph with secure aggregation

shared keys between the clients and the servers, and masking via `AES-CTR` stream cipher to reduce the above communication complexity by a factor ℓ to $O(nb)$. In addition, it avoid the client communication and synchronization (the risk that for a given local model all shares do not reach to their destination servers) with multiple servers and provides a solution that is robust against client dropouts. Instead of the plain synchronization, clients and servers compute the shares using pseudo random stream of bytes generated by `AES-CTR` with initially synchronized shared key.

Suppose that we have n clients $\texttt{client}_1, \dots, \texttt{client}_n$ and ℓ aggregating servers $\texttt{server}_1, \dots, \texttt{server}_\ell$. Additionally, assume that we have shared keys $\texttt{sk}_{i,j}$ (of size 384 bits) between client i and server j for $i = 1, \dots, n$ and $j = 1, \dots, \ell$. The generation of the shared keys can be achieved via standard Diffie–Hellman key exchange or directly communicated by the servers to the clients via secure communication channel between each client and the servers during the setting/subscription phase.

Each client performs masking of its local model M_i (a vector of integers coefficients $\mod 2^{64}$) with independent random masks (one per server) $\texttt{mask}(\texttt{sk}_{i,j}, r)$ generated using the exchanged $\texttt{sk}_{i,j}$ and the iteration number r of the model update (we will discuss the exact choice of the function `mask` below). The masked value obtained is

$$A_i = M_i + \sum_{j=1}^{\ell} \texttt{mask}(\texttt{sk}_{i,j}, r)$$

(see Algorithm 1).

This value is a public and the $\texttt{client}_i$ publishes it to a public Funnel. The Funnel is defined as a public map-reduce service with following property: given two tuples of masked values, the funnel adds them and reduces it to one tuple of aggregated masked value. The Funnel repeats this process until obtain just one final aggregated masked value.

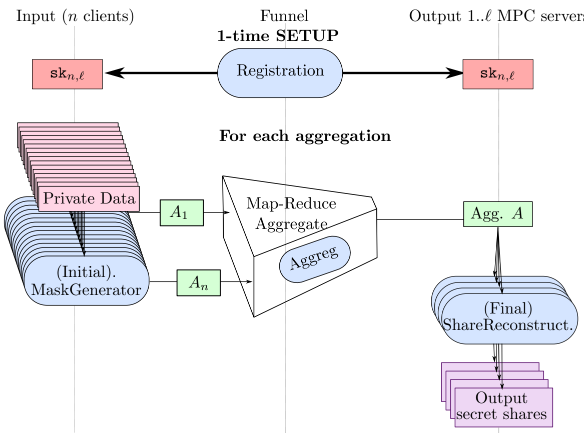

Fig. 3. `Falkor` aggregator execution flow

That means, all masked values (from all connected/available clients U_r) are aggregated progressively in the Funnel and at the end (after the pre-defined maximal waiting time) we obtain the public aggregated masked value $A := \sum_{i \in U_r} A_i$ available to at least one of the servers (see Fig. 3).

In turn, each server independently computes the same masks $\mathtt{mask}(\mathtt{sk}_{i,j}, r)$ using the same collection of keys shared with its connected peers (Algorithm 2) and obtain its shares of the aggregated model $[\![M]\!] = \sum_{i \in U_r} M_i$.

Then we can either reveal the secret shares of the model in order to publish/broadcast the updated common model or perform some additional non-linear MPC operations on the aggregation servers using directly the secret shares of the model (see Fig. 2). The use of MPC for more complex computations than simple averaging such as private divisions, multiplications, squaring allows us to use more complex privacy-preserving algorithms.

The bottleneck operation in this protocol is the computation (by server j) of

$$\sum_{i \in U_r} \mathtt{mask}(\mathtt{sk}_{i,j}, r),$$

where U_r is the list of the connected clients. In order to enable aggregation of millions of clients, each training a local model with millions of coefficients, Falkor uses a high-performance random generation of masks on the server nodes. For less than $100 - 1000$ clients, a single core CPU can computes this sum in matter of seconds, however to scale to 1M clients (or more), a GPU is preferred, and in Section 7 we explain the acceleration of the computations of the random masks. Unlike client nodes with limited compute resources (e.g., mobile phones or IoT devices), in practice, the aggregating servers can use specialized hardware accelerators such as GPUs or FPGAs.

3 `AES-CTR`-based Masking

We now discuss the choice of the function `mask` in more detail. A natural approach would be to use a masking function based on a suitable pseudorandom number generator (PRNG) designed and implemented for GPUs (typical examples are `cuRAND` [18] and `rocRAND` [19] among others). Yet, as explained in Section 5, such GPU PRNGs rarely qualify for cryptographic security, hence, resulting in a lack of official certification to validate their use in cryptographic applications.

To remedy this problem, we take a different approach inspired by the concept of counter-based PRNG originally proposed by [1] and subsequently used by Facebook for their `torchcsprng`[5], but

[5] `https://pytorch.org/blog/torchcsprng-release-blog/`

Algorithm 1 Model Masked Value Generator

```
Input: Local models $M_i$ for $1 \leq i \leq n$; the seeds $\texttt{sk}_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, \ell$ exchanged during the
    setting phase and $r$- the iteration number.
Output: Masked value $A_i$ of the local model $M_i$ for$1 \leq i \leq n$
1: for $1 \leq i \leq n$ do
2:     $\texttt{client}_i$ computes
           $$A_i = M_i + \sum_{j=1}^{\ell} \texttt{mask}(\texttt{sk}_{i,j}, r)$$
3:     $\texttt{client}_i$ publish $A_i$ to $\texttt{server}_\ell$ (via public Funnel)
4: end for
```

Algorithm 2 Secret Shares Reconstructor

```
Input: A list $U_r$ of connected clients (at the $r$th iteration)
Input: Masked values $A_i$ for $i \in U_r$
Input: $\texttt{sk}_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, \ell$ exchanged during the setting phase
Output: Secret shares of the aggregated model
           $$[\![M]\!] = \sum_{i \in U_r} M_i$$
1: $\texttt{server}_\ell$ receives $A := \sum_{i \in U_r} A_i$ and $U_r$ (aggregated in the Funnel, note that $A$ can be public)
2: for $1 \leq j < \ell$ do
3:     $\texttt{server}_j$ consumes $U_r$ from the Funnel
4:     $\texttt{server}_j$ computes
           $$\texttt{share}_j := \sum_{i \in U_r} \texttt{mask}(\texttt{sk}_{i,j}, r)$$
5: end for
6: $\texttt{server}_\ell$ computes $\texttt{share}_\ell := \texttt{share}_\ell - A$
7: Output $[\![M]\!] = (\texttt{share}_1, \ldots, \texttt{share}_\ell)$
8: Note: MPCReveal$(M) = \sum_{j=1}^{\ell} \texttt{share}_j = \sum_{i \in U_r} M_i$
```

differing in the way we formally analyze security in our particular application to Federated Learning. For more background on counter-based PRNGs, we refer to reader to Section 5.

Our specific idea is to turn one of the cryptographically strong block ciphers in counter mode (e.g., the Advanced Encryption Standard AES [20]) into a random mask generator. The goal is to generate as fast as possible on a GPU the same encryption stream as in AES-CTR mode. The stream cipher is used to mask additively the local models M (vectors of 64-bit integers), for this reason the output of the stream is reinterpreted as a vector of 64-bits little endian integers, of the same size as the models. We define:

$$\texttt{AES-CTR-Stream}_{K,\mathrm{IV}}(B) := \texttt{trunc}_B(\texttt{AES}_K(\mathrm{IV}) \| \texttt{AES}_K(\mathrm{IV} + 1) \| \ldots)$$

where $\texttt{trunc}_B(S)$ truncates the first B-bits of a bit stream S. Using this definition, the number of bits needed to mask the model M (vector of 64-bit integers) is $B_M := \texttt{bitlength}(M)$ and our mask function is expressed as

$$\texttt{mask}(\texttt{sk}, r) := \texttt{AES-CTR-Stream}_{K,\mathrm{IV}}(B_M),$$

where the 256-bit key K and the 128-bit initial value IV are derived from the sk (384 bits) and the iteration number r by the key derivation function PBKDF2_sha256() from OpenSSL[6] library. In our protocol, the shared keys sk are secret values while the iteration numbers r are public knowledge. Lets $K \| \mathrm{IV} := \texttt{PBKDF2\_sha256}(\texttt{sk}, r)$. The output of the key derivation function is composed by 384 bits

[6] www.openssl.org (EVP_sha256)

indistinguishable from uniformly random bits for an adversary that doesn't know sk. The upper 256 bits are used for the key K and the lower 128 bits for the initial value IV (including the nonce and the initial counter). The counter wraps around modulo 2^{128}.

The AES-based stream cipher in counter mode turns the AES blockcipher into a stream cipher by generating the next keystream via encryption of the successive values of a counter.

In our protocol we use a random IV of 128-bits and we add increment-by-one counter to the IV.

It is worth pointing that, although *a priori* related, our protocol does not rely on the deterministic random bit generator CTR_DRBG based on AES and our security analysis is reduced simply to the security of the AES-CTR stream cipher. The advantage of AES-CTR is that it is very efficient and can be pipelined and parallelized. The semantic security of AES-CTR is properly analyzed in [21, Thm.13]. In high-level summary, this result establishes indistinguishability from random outputs up to the birthday bound, that is, up to $(2^{k/2})$ encrypted k-bit blocks. This means that any attack that breaks the confidentiality of the plaintext would require $\Omega(2^{k/2})$ blocks of ciphertext. In other words, the models we mask should have less than 2^{64} blocks, so 2^{71} bits. CTR_DRBG would not have this limitation, however this memory bound is sufficiently large to fit any model that fit in practice in memory.

The threat model and the security of our protocol are discussed in Appendix B.

4 Background on GPU Architectures

In this section, we provide a basic overview of the organization and principles of a GPU architecture. These will be needed for the subsequent section where we describe in depth our parallel algorithms for secure aggregation using AES-CTR.

A GPU *kernels* is a CUDA function that is called by the host (the CPU) and executed on the GPU (the device).

Grids, *blocks* and *threads* are basic GPU CUDA kernel abstractions. Threads are organized in blocks and blocks are organized in grids. The grids are dispatched and scheduled on different streaming multiprocessors (SM) of the device. Inside a grid, each block is executed by a single SM, and consist of threads that are given access to a common low-latency shared memory space. Within a block, the threads are sliced and scheduled by warps of 32, in an arbitrary order. For convenience, grids, blocks and threads can be indexed in 1-, 2- or 3-dimensional arrays and can be identified with their coordinates/indices.

The global memory is the largest memory partition and is accessible by all threads, yet, the communication with the global memory is the most expensive one. The constant memory is significantly faster than the global memory, but the data stored in it is cached and hence, cannot be changed during the execution of the operations. Another partition in the memory hierarchy is the shared memory that is significantly faster to access than global and constant memory and that can be modified during the execution of operations. Yet, it is accessible only by the threads in a given block. Finally, the registers are the fastest form of memory. Registers are local to their thread, each SM contains a bounded amount of kiloByes that can serve as registers, and their allocation is done at compile time.

There is also a local memory specific to a thread, which is part of the global memory, but is unfortunately 150x slower than the shared or constant memory, so we cannot use it in our scenario.

Shared memory is organized in 32 banks of 4-byte long elements, thus, element i living in bank i mod 32. A *bank conflicts* occurs when two threads in the same warp access different elements in the same bank. Note that bank conflicts cause serial rather than parallel memory accesses, thus compromising performance.

5 Counter-based PRNGs

The two major approaches to generating random numbers on a GPU are the multistream and substream methods. In the multistream approach, the PRNG is instantiated in parallel with different parameters. This approach imposes strong constraints on the sets of parameters for the underlying PRNG which, when combined with the strong cryptographic security requirements[7], has proven to be difficult to guarantee in practice. In the substream approach, a single sequence of random numbers is split into disjoint substreams that may be accessed in parallel. In order for this approach to work in practice, one needs a method to partition the much longer underlying sequence in order to allow the state space to be deterministically partitioned and the substreams to be statistically independent. This leads to significant constraints on the family of PRNGs. In addition, the substream approach requires an underlying PRNG with a very long period so that the probability of overlap between substreams is negligible. It is well known that the initialization of the state of long-period PRNGs far from obvious, and if not done properly, the result can have significant defects.

In either approach, a parallel operation of a PRNG requires that the state is stored in each computational unit (thread, warp, block, core, etc.) in close proximity to the ALU for fast memory access. Since on-chip memory is a precious resource in modern GPUs, PRNGs with large state are not an efficient option.

With the additional requirement that the CPU and GPU should produce identical orderings of the sequences when provided with the same seeds (in order to obtain compatible results for the function `mask` on the clients and on the servers), this strongly limits our choice. For instance, only two of the GPU-supported PRNGs in the NVIDIA `cuRAND` library[8], only `Philox_4x32_10` and `XORWOW` enjoy this property, yet, none of them are proven to be cryptographically secure in the strong sense of being indistinguishable from a true random source.

As already mentioned in Section 2, an alternative counter-based PRNGs were considered by Salmon et al. [1] as suitable candidates for massive parallelization on GPU with strong security properties and small states. This method is an attractive alternative to the two mainstream approaches described above. The `torchcsprng` PyTorch C++/CUDA extension [22] used by Facebook and based on `AES-CTR` evaluation on the GPU is such an alternative offering stronger cryptographic security. The generator `torchcsprng` is used in another privacy-preserving framework CryptGPU [23] for share re-randomization in the truncation protocol. Yet, the `torchcsprng` GPU implementation is tightly coupled with PyTorch C++/CUDA core. It uses tensor containers to organize data input/output and a PyTorch internal GPU kernels dispatcher. Hence, deployment of the original `torchcsprng` outside of PyTorch is *a priori* almost impossible.

6 GPU Implementation of `AES`

Although the GPU instruction set is not specifically adapted to `AES`, there have been many attempts to acceleration `AES` on a GPU in the literature reaching high throughput and more than an order of magnitude speedups compared to CPU implementations with the `AES-NI` instruction set. Three major have been explored:

1. Naïve approach: all operations in the round function implemented directly
2. Table-based approach: inputs are split into different parts and for each part and eacj possible input for that part, the outputs are precomputed and stored in a table. Each `AES` round is then reduced to 16 table lookups and 16 `XOR` operations.
3. Bitsliced approach: each of the 128 bits of the 128-bit input is sliced and stored into 128 registers and operations are performed to each bit independently, thus providing SIMD parallelism. Thus, if the registers have n bits, this approach allows to process n blocks in parallel.

Early GPU optimizations of `AES` such as [24] observe that the table-based approach is better than the naïve approach. These early approaches stored the tables in GPU constant memory and used 4 threads for the encryption of a 128-bit block (i.e., a granularity of 4 bytes per thread). The question of the best memory use for the table-based approach has been addressed in subsequent works. An

[7] For our cryptographic application, cryptographic security is a stronger requirement than standard statistical tests such as, e.g., TestU01's BigCrush test.

[8] `https://docs.nvidia.com/cuda/curand/`

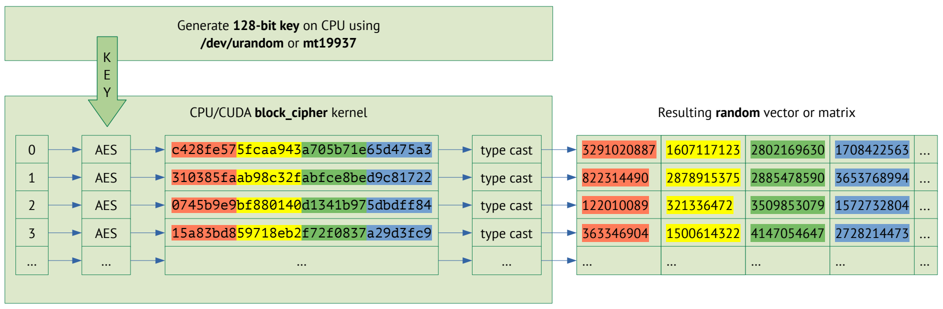

Fig. 4. The general design of parallel crypto-secure PRNG

important observation has been made in [25] where tables were copied to shared memory from global memory at the beginning of each kernel (instead of stored in constant memory). Further improvements were made in [26] where granularity was increased to 32 bytes / thread. In addition, the GPU `AES` implementation [27] also leverages shared memory and is tuned to achieve both high performance and strong resistance to side channel attacks.[9] Yet, a major inefficiency in early table-based implementations such as [25], [26] and [28] were bank conflicts. Shared memory bank conflicts were recently resolved in an implementation by Cihangir Tezcan [2] to obtain the best `AES` performance on GPU for `AES` disk encryption and exhaustive search attacks (we refer to this implementation as `Cihangir`, by the name of the author).

Before discussing GPU implementation of `AES-CTR`, we recall some of the relevant internals. The `AES` algorithm consists of two phases:

– key expansion
– plaintext data encryption.

In all our applications, the key expansion time is negligible compared to data encryption time and hence, key expansion is done on the CPU. We only describe the data encryption rounds on the GPU. The major operations in the plaintext data encryption are the following:

1. `SubBytes` - a non-linear operation that applies the S-box,
2. `ShiftRows` - applies cyclic shifts of certain offset to the rows of the state,
3. `MixColumns` - multiplies each column of the state matrix with a fixed polynomial in $\mathbb{F}_{2^8}[x]/\langle x^4+1\rangle$ - this is done with the help of lookup tables of 256 elements,
4. `AddRoundKey` - XOR-ing of the internal state with the round key.

Since none of the available GPUs offer special instruction sets for `AES`, we begin by porting the three steps `SubBytes`, `ShiftRows` and `AddRoundKey` to the GPU in a naïve way, using native arithmetic operators. Yet, the `MixColumns` step requires extra care. Since this step is a multiplication of the state with a constant polynomial in $\mathbb{F}_{2^8}[x]/\langle x^4+1\rangle$, it is carried out with the help of lookup tables of 256 elements. To port `mbedTLS` code from CPU to GPU, one thus loads these tables of coefficients once into GPU memory and annotates the encryption function for device execution. It is here that we use the ideas of [2] in a crucial manner. Although the original source code of [2] is adapted to an orthogonal setting where the goal is to generate preimages or find `AES` collisions[10], the kernels of `Cihangir` already demonstrate the important optimization - instead of naively mapping tables to constant memory, the tables are loaded into shared memory banks. In Section 7, we will show how to adapt `Cihangir` to our secure aggregation setting.

[9] In XOR-Falkor, secure aggregation shall be performed on dedicated cloud servers to avoid side channel attacks.

[10] In this setting, the key is changing after each cycle of rounds in order to perform brute-force search of the plaintext that produces a given `AES` digest

7 Parallel Secure Aggregation on GPU via AES-CTR-based Masking

First, we took mbedTLS AES-CTR encryption on CPU is as a reference implementation. All GPU AES implementations presented in this paper are verified to produce byte streams bitwise-identical to the reference for the same initial seeds. In order to decouple the dependency on PyTorch mentioned in Section 5, we decomposed the source code of torchcsprng and reorganized it to run as a standalone generator with basic data arrays. This study has revealed the key portion of the torchcsprng kernel is inherited from tinyAES [29], a portable AES implementation in C. The torchcsprng developers adapted tinyAES code for GPU execution in the same way as we have ported mbedTLS.

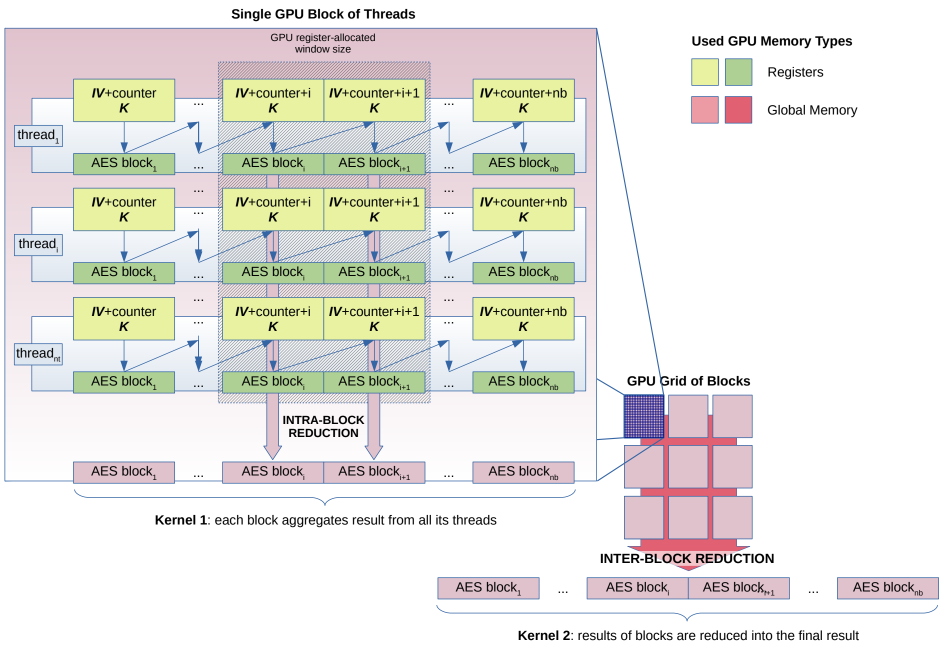

Fig. 5. Mapping of parallel secure aggregation onto GPU kernels threads and blocks

The common approach to maximizing AES throughput on GPU is to perform massively parallel encryption of independent streams on individual GPU threads. This strategy is usually suitable for real applications, e.g. encryption of many large files, or transmitting data over many active SSL sessions. The AES-based GPU random number generator for secure aggregation also follows the same methodology described in Fig. 4. The initial keys originate from CPU (e.g. from one of the CPU-based generators), and the parallel AES encryption operations are dispatched to generate portions of the resulting random vector or matrix. The GPU performance is identified by sustained performance of all AES encryption operations being executed on parallel GPU threads.

The share reconstructor (Step 4 of Alg. 2) GPU implementation consists of two CUDA kernels executed sequentially (Fig. 5):

1. In the first kernel, each thread is responsible for generating the AES-CTR byte stream for its own (K, IV) and then the threads jointly aggregate their streams to form one sum per GPU blocks. This yields one aggregated stream per GPU block in the global GPU memory (a total of B streams where B is the number of blocks on the GPU). The aggregation within each individual block is implemented as an intra-block reduction using CUB.

2. The second kernel sums these B streams into a single one that is then returned to the host. This kernel requires a classical segmented reduction, and thus is implemented using Thrust.

7.1 Use of shared memory and resolution of bank conflicts

Following Cihangir [2], we let the GPU threads cooperatively load the `AES` table into the shared memory Since we need to grant a random access pattern to these lookup-tables to each thread, we load 32 copies of the table into the shared memory of each thread block. During `AES` encryption, each thread in a warp accesses only its own copy of the tabl. This way shared memory reads are made free of bank conflicts, effectively maximizing the shared memory throughput The amount of shared memory per block required in this approach totals:

$$\texttt{size}_{\texttt{table}} \times \#\texttt{warps} \times \texttt{size}_{\texttt{int}} = 256 \times 32 \times 4 = 32768\,\texttt{bytes}\,.$$

The `Cihangir` code and the CUB intra-block reduction both use shared memory within the first kernel. CUB implementation aggregates values within a warp using shuffle operation, and uses a small amount of shared memory to reduce partial aggregates between warps. Although the intra-block reduction temporary storage could be allocated as an extra chunk of shared memory in addition to `Cihangir` tables storage, the two together will exceed the most desirable shared memory size of 32KBytes (3 simultaneous blocks on V100 with 96KiB per SM). In order to maintain the 32KBytes limitation, we temporary offload a small portion of `Cihangir` tables into registers to free up some shared memory for the reduction temporary storage. In case of `mbedTLS` and `torchcsprng` versions, intra-block reduction is the only consumer of shared memory; the tables are kept in constant memory.

7.2 GPU optimization and occupancy considerations

The GPU efficiency can be measured as the *GPU occupancy* ratio: the number of active warps divided by the total number of warps supported by the GPU. The GPU occupancy is highly determined by how well the properties of a given kernel program (register footprint, block size, amount of shared memory) are suited for a specific GPU (maximum available registers, active warps and shared memory per GPU multiprocessor).

If the round key is stored in GPU thread registers, the resulting register footprint could be quite high, and may dramatically worsen the GPU occupancy rate. For example, with 64 registers per thread, 1024 threads in a block and 32KB of shared memory per block, our AES kernel has the occupancy rate of 0.5 (50%) on NVIDIA Tesla V100. The same kernel but with 32 registers per thread could peak at 1.0 (100%) ideal occupancy rate. Therefore, our kernel performance is limited by registers consumption.

We look at how each round of `AES` encryption is performed with the help of T-tables. We refer to [2, p.67318] for the exact expressions (in terms of T-tables and round keys) of the columns of AES in one encryption round. Each AES encryption step (each counter increases) performs XOR of 16 rotations. Given that the rotations are computationally independent, we can distribute them across multiple threads. Moreover, in order to compute a single rotation, only a subset of the round key is required. This approach opens an opportunity to decrease register consumption by sharing round keys among the threads working on the same encryption round. Particularly, in the case of NVIDIA Tesla V100 with 96KB of shared memory per multiprocessor, we even have an option to put the round key into the shared memory. With two active blocks per multiprocessor of 1024 threads, each already spending 32KB on AES tables, we have 16KB remaining to host the round keys. The shared round keys could all fit into 16KB[11] if each cooperative group of threads has at least

$$\texttt{size}_{\texttt{block}} \,/\, \left(\frac{\texttt{size}_{\texttt{shared}} \,/\, \#\,\texttt{blocks} - \texttt{size}_{\texttt{table}}}{\texttt{size}_{\texttt{RK}}} \right)$$

[11] We reserve 64 integers per key instead of 60 for even division.

threads. For $\texttt{size}_{\texttt{block}} = 1024$, $\texttt{size}_{\texttt{shared}} = 98304\,\texttt{bytes}$, $\#\,\texttt{blocks} = 2$, $\texttt{size}_{\texttt{table}} = 32728\,\texttt{bytes}$, $\texttt{size}_{\texttt{RK}} = 4 \cdot 64\,\texttt{bytes} = 256\,\texttt{bytes}$, we obtain 16 threads-per-group, that is, at least 16 threads need to share a round key in order for us to fit the round keys and the AES tables in shared memory.

As shown, the shared memory is perfectly enough to spare the round key between 16 threads, each computing a single rotation, and then gathered into a single XOR sum. We add this optimization on top of the Cihangir's shared memory approach.

Remarkably, `torchcsprng` does not make use of the shared memory. It could be explained by a need to keep the shared memory available to other simultaneous GPU operations, e.g. tensor products. Furthermore, `torchcsprng` is limited to the PyTorch internal GPU kernels dispatcher. That is, `torchcsprng` cannot customize the CUDA compute grid and organize specific threads cooperation.

8 Benchmarking

For the benchmarking, we have chosen a configuration typically used in the cloud compute environments with potential applications of Falkor technology. The secure aggregation compute node should be equipped with at least one virtual CPU, and a dedicated GPU, such as NVIDIA Telsa V100. A sufficient amount of memory both on CPU and GPU is preferred, but is not required, as the Falkor kernel could be further adapted to perform piecewise (double-buffered) processing to fit limited RAM size.

The CPU part of the benchmark is a single-threaded `AES-CTR` and shares aggregation code, which is based on the `mbedTLS` implementation leveraging the AES-NI instruction set.

The GPU part of the benchmark includes the two kernels performing `AES-CTR` and shares aggregation presented in Fig. 5, and the data transfers required to pass the IV vectors from the CPU to the GPU, and the final result from the GPU to the CPU. The GPU performance is measured with `Cihangir` and `tinyAES` implementations. We benchmark the `tinyAES` implementation, in order to demonstrate the performance of `torchcsprng`, which is based on `tinyAES`.

We measure the performance of the implementations above on up to 2^{18} clients, and up to 8×10^6 bytes of random sequence length.

The speedup of `Cihangir` GPU implementation on NVIDIA Tesla V100 against a single core Intel Xeon CPU @ 2.30GHz implementation based on `mbedTLS` is presented in Fig. 6. The most significant advantage of GPU implementation is highlighted in the chart.

The CPU-GPU benchmark demonstrates speedups over $100\times$ on workloads with large enough number of clients and random sequence length. Both dimensions are equally important:

– each client is executed on an individual GPU thread, and a GPU must execute at least $\approx 10,000$ threads, in order to maximize the use of the available GPU resources
– the random sequence length determines the lifetime of each client GPU thread; if the sequence is too short, the suboptimal initial AES key expansion dominates the timeline.

The speedup of `Cihangir` GPU implementation against `tinyAES` GPU implementation on NVIDIA Tesla V100 is presented in Fig. 7. The most significant advantage of `Cihangir` version is also highlighted in the chart. The absent data points have not been produced due to the memory size limitation.

`Cihangir` is more than $10\times$ faster than `tinyAES` on GPU entirely thanks to the use of the shared memory as a T-tables storage.

9 Privacy-preserving Federated Logistic Regression Model Training

In this section, we demonstrate scalability of our protocol by training a large logistic regression models with 100 features and 50M data points distributed across 1000 clients. We carry out a Newton–Raphson (second order) optimization, where up to 10% of the clients are allowed to dropout at each iteration and hence, not submit their local update to the servers. The clients compute locally on their private data

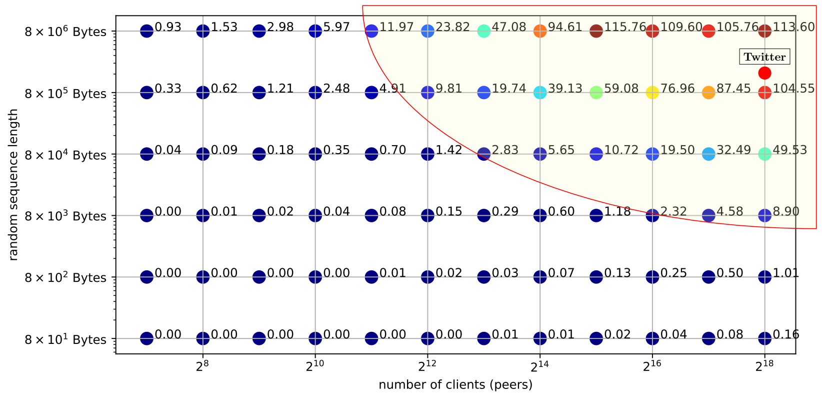

Fig. 6. Speedup of `Cihangir` GPU implementation (NVIDIA Tesla V100) against single CPU core Intel Xeon CPU @ 2.30GHz `mbedTLS` with AES-NI instruction set enabled

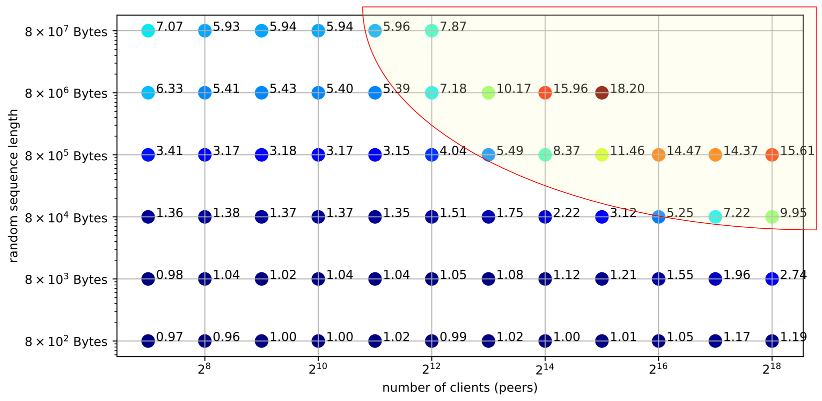

Fig. 7. Speedup of Cihangir implementation versus torchcsprng on GPU NVIDIA Tesla V100

the local gradients (vectors) and Hessians (matrices) (`grad` and `Hess`, respectively), both of dimension 101 (see in Algorithm 6) and generates their corresponding masked value. Such computation takes on average 4 seconds per client and produces 82kB of masked values to aggregate. All these computation occur in parallel, and if we allow a long window of 10 seconds, we gather at least 90% of the client updates, the rest being considered dropouts. All the clients submit these masked values to the funnel, that uses the `Falkor` protocol to aggregate them. The Funnel can be organized as a tree of servers, each client submits its update to the nearest leaf server, and each server aggregate their local updates and propagate the result to its parent, and so on up to the root server, that ends up with the masked value of the the global gradient (a vector of size 101) and the global Hessian (a symetric matrix of size 101). Aggregating two 82kB masked values to form one 82kB masked value is a very fast operation (taking less than 100ms), so thanks to the Funnel server map-reduce topology, we obtain the global aggregated masked-value in about one second. Then, each MPC server calls the Falkor secret-shares reconstructor locally to turn the masked value into their own secret shares of the global gradient and Hessian. The MPC servers compute the global model update `step` using a pure secret-shared based MPC. This small computation consists in a linear system resolution in small dimension (a 101×101 linear system), and the joint computation terminates in 4 seconds. The MPC servers can optionally also add a small Gaussian noise before revealing the result, so that the published global model update protects the differential privacy of the clients. In total our secure aggregation algorithm, performs 6 Newton iterations, which are enough to reach convergence, and runs in less than a minute and uses 1MB network between the MPC servers, and hundreds of kilobytes of inbound/outbound network on each client device. As a comparison, an end-to-end MPC approach where the costly computation is performed on the full dataset of dimension $50M \times 100$ using the MPC logreg described in [30] on 3 parties takes 27 hours, requires 400GB RAM machines, and requires to exchange more than 2 TeraBytes of masked values, on a very fast gigabit per second network.

Algorithm 3 Distributed logreg

```
Input: $U_r$ - list of connected clients during iteration $r$
Input: $M_r$ - current global model at iteration $r$
Input: $X_i$ - local input data for $\texttt{client}_i \in U_r$
Output: Updated model $M_{r+1}$ at iteration $r+1$
1: for each $\texttt{client}_i \in U_r$, in parallel do
2:     $(\texttt{grad}_{i,r}, \texttt{Hess}_{i,r}) \leftarrow \texttt{logregStep}(M_r, X_i)$
3: end for
4: $(\texttt{grad}_r, \texttt{Hess}_r) = (\sum_{i \in U_r} \texttt{grad}_{i,r}, \sum_{i \in U_r} \texttt{Hess}_{i,r})$ (via Falkor secure aggregation)
5: return $M_{r+1} = M_r - \texttt{Hess}_r^{-1} \cdot \texttt{grad}_r$
```

10 Privacy-preserving Federated RNN Model Training on Twitter Dataset

In this section we apply `Falkor` to the particular scenario of RNN sentiment analysis model training on the Twitter dataset. We first describe the dataset and the used RNN model. In order to ensure faster convergence and accuracy we present secure version of two optimizers: `FedAvg` and `FedAdam`.

Dataset The dataset comprises of 1,194,933 tweets labeled positive or negative based on their emojis for binary sentiment classification [31]. Each client is a Twitter user. We generate data with the LEAF framework [32] with a ratio of 90-10% for the train data and test data which gives us 254,555 clients in total.

Model The architecture of the recurrent neural network (RNN) model is described on Fig. 9. The model itself is made of 186,657 trainable weights. Each word in a sample is first converted to its assigned

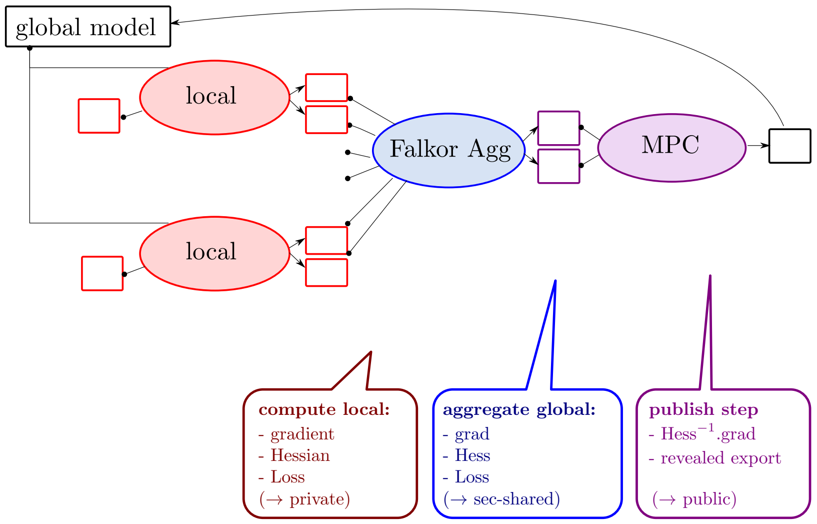

Fig. 8. Falkor flow with distributed Logreg

token per a vocabulary of size 1.2M. The token is then mapped to its embedding vector of dimension 100. Here, we use a pre-trained (and public) embedding layer of 12M parameters, Glove [33], that is known to all clients and stays constant during training. The embeddings are fed to a bidirectional LSTM layer of 100 output units. The bidirectional layer goes through the sequence both forwards and backwards. Both outputs are concatenated before going to a dense (fully connected) layer of 128 units and a final classification layer of 1 unit followed by a softmax activation function. The model is trained with a binary cross-entropy loss.

10.1 Federated Average Optimizer

The `FedAvg` algorithm [34] is a straightforward procedure that combines local gradient descent at the client level and model averaging at the server level. Local training `LocalOpt` can be any type of gradient descent algorithm such as SGD or Adam. Each client receives a global model from the server, performs gradient descent on its local dataset and sends back its local gradients. The server then aggregate all the received gradients $\tilde{\Delta}_r$ and the total number of samples N_r, computes the average $\Delta_r = \frac{\tilde{\Delta}_r}{N_r}$ and updates the global model (see Algorithm 4). In case of private N_r, this division can be performed via private MPC division.

Benchmarks (GPU vs CPU for `FedAvg`) We split the training data among a varying (between 10 and 100000) non-overlapping *simulated* clients. We use the `FedAvg` algorithm with $\eta_{\texttt{server}} = 1$ and Adam as the local optimizer with $\eta_{\texttt{client}} = 0.001$. The model is trained for 50 rounds.

As shown in Figure 11, increasing the number of clients while keeping the same data size decreases the local client training time as there is less data per client (which means less gradient steps). However, the server aggregation time increases as there are more shares to aggregate at the end of each round. The client local training time depends on the client hardware whereas the server computations can be improved via our GPU optimizations. For 50 iterations, the total training for 254255 clients is around 10 minutes using the GPU `AES-CTR` for secure aggregation versus 10 hours with single CPU core. The GPU speedup for the generation of the aggregated masks with `AES-CTR` in this scenarios is around $\times 100$ (see Figure 6).

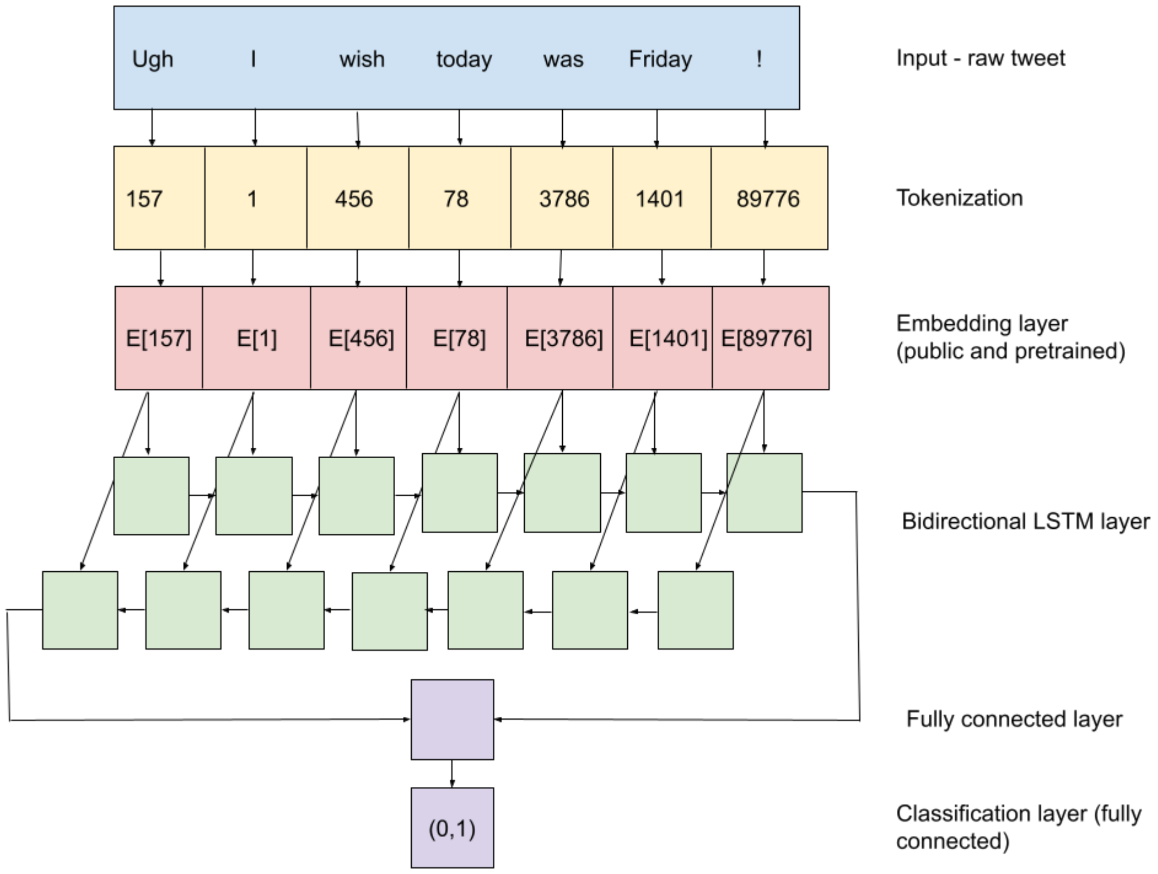

Fig. 9. RNN model for the Twitter dataset

Algorithm 4 `FedAvg`

```
Input: $U_r$ - list of connected clients during iteration $r$
Input: $M_r$ - current global model at iteration $r$
Input: $X_i$ - local input data for $\texttt{client}_i \in U_r$
Input: $\eta_{\texttt{client}}$ (client learning rate), $\eta_{\texttt{server}}$ (server learning rate),
Output: Updated model $M_{r+1}$ at iteration $r+1$
1: for each $\texttt{client}_i \in U_r$, in parallel do
2:     $\hat{M}_{i,r} \leftarrow \texttt{LocalOpt}(M_r, X_i, \eta_{\texttt{client}})$
3:     $\tilde{\Delta}_{i,r} = N_{i,r}(M_r - \hat{M}_{i,r})$, where $N_{i,r}$ is the number of samples used in the training in iteration $r$ by $\texttt{client}_i$.
4: end for
5: $(\tilde{\Delta}_r, N_r) = (\sum_{i \in U_r} \tilde{\Delta}_{i,r}, \sum_{i \in U_r} N_{i,r})$ (via Falkor secure aggregation)
6: $\Delta_r = \frac{\tilde{\Delta}_r}{N_r}$
7: return $M_{r+1} = M_r - \eta_{\texttt{server}} \Delta_r$
```

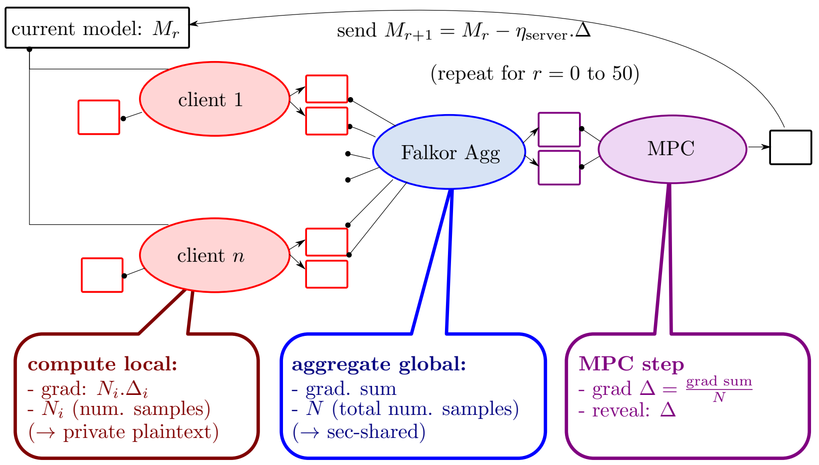
current model: M_r
send $M_{r+1} = M_r - \eta_{\text{server}}.\Delta$
(repeat for $r = 0$ to 50)
client 1
client n
Falkor Agg
MPC
compute local:
- grad: $N_i.\Delta_i$
- N_i (num. samples)
($\rightarrow$ private plaintext)
aggregate global:
- grad. sum
- N (total num. samples)
($\rightarrow$ sec-shared)
MPC step
- grad $\Delta = \frac{\text{grad sum}}{N}$
- reveal: Δ

Fig. 10. Falkor flow with FedAvg

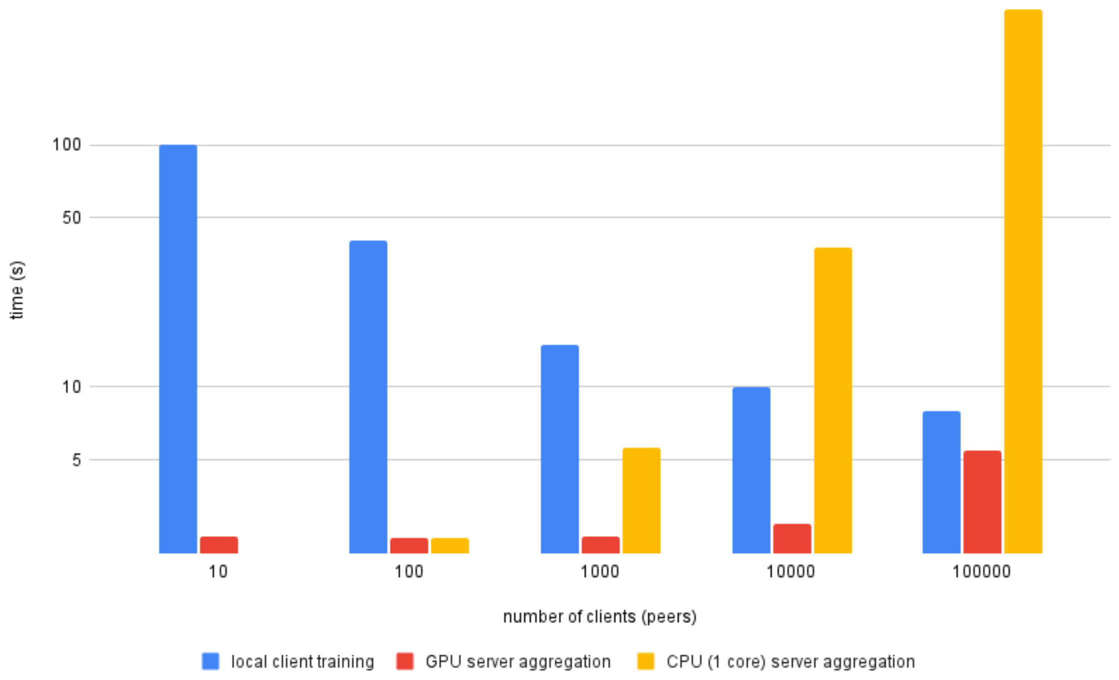

Fig. 11. Benchmarks with a single core per client for the local computation and single core versus GPU for the secure aggregation on servers side.

10.2 Adaptive Federated Optimizers.

While `FedAvg` is a fairly standard and simple algorithm, its hyperparameters are often difficult to tune and often exhibits a poor convergence behavior, especially when the number of client increases and the datasets are not identically distributed across the clients. Recent progress have been made on adaptive server optimizers such as `FedAdagrad`, `FedAdam` and `FedYogi` [35].

Our framework enables chaining `Falkor` with highly scalable MPC protocol (e.g., [30]) that runs on the servers yields high performance implementations of these server optimization algorithms.

We now describe `FedAdam` (Algorithm 5) and explain how `Falkor` and MPC computations are used to implement it.

Algorithm 5 `FedAdam`

```
Input: $U_r$ - list of connected clients during iteration $r$
Input: $M_r$ - current global model at iteration $r$
Input: $X_i$ - local input data for $\texttt{client}_i \in U_r$
Input: $\beta_1, \beta_2 \in [0, 1)$, $\tau$, $E$ - local epochs, $B$ - batch size, $\eta_{\texttt{client}}$ - client learning rate, $\eta_{\texttt{server}}$ - server learning rate
Output: Updated model $M_{r+1}$ at iteration $r + 1$
1: for each $\texttt{client}_i \in U_r$, in parallel do
2:     $\hat{M}_{i,r} \leftarrow \texttt{LocalOpt}(M_r, X_i, E, B, \eta_{\texttt{client}})$
3:     $\tilde{\Delta}_{i,r} = N_{i,r}(M_r - \hat{M}_{i,r})$, where $N_{i,r}$ is the number of samples used in the training in iteration $r$ by $\texttt{client}_i$.
4: end for
5: $(\tilde{\Delta}_r, N_r) = (\sum_{i \in U_r} \tilde{\Delta}_{i,r}, \sum_{i \in U_r} N_{i,r})$ (via Falkor secure aggregation)
6: $\Delta_r = \beta_1 \Delta_{r-1} + (1 - \beta_1)(\frac{\tilde{\Delta}_r}{N_r})$
7: $v_r = \beta_2 v_{r-1} + (1 - \beta_2)\Delta_r^2$
8: return $M_{r+1} = M_r - \eta_{\texttt{server}} \frac{\Delta_r}{\sqrt{v_r} + \tau}$
```

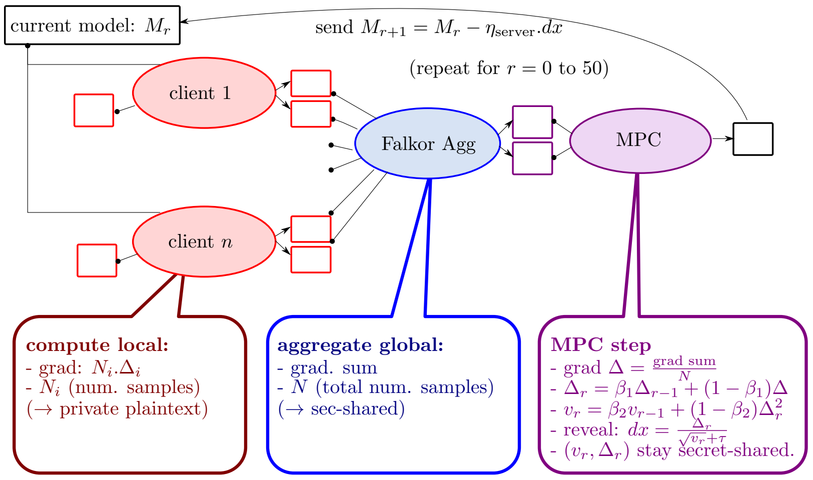
current model: M_r
send $M_{r+1} = M_r - \eta_{\text{server}}.dx$
(repeat for $r = 0$ to 50)
client 1
client n
Falkor Agg
MPC
compute local:
- grad: $N_i.\Delta_i$
- N_i (num. samples)
($\rightarrow$ private plaintext)
aggregate global:
- grad. sum
- N (total num. samples)
($\rightarrow$ sec-shared)
MPC step
- grad $\Delta = \frac{\text{grad sum}}{N}$
- $\Delta_r = \beta_1\Delta_{r-1} + (1-\beta_1)\Delta$
- $v_r = \beta_2 v_{r-1} + (1-\beta_2)\Delta_r^2$
- reveal: $dx = \frac{\Delta_r}{\sqrt{v_r}+\tau}$
- (v_r, Δ_r) stay secret-shared.

Fig. 12. Falkor flow with FedAdam optimizer

For this adaptive method we apply `Falkor` to aggregate $(\tilde{\Delta}_r, N_r)$ and MPC for the computation of Δ_r, r and x_{r+1} using MPC linear combination, multiplication, squaring and division (see Figure 12).

Our contribution is to secure the full flow of computations of the adaptive methods and not only the linear aggregation function. Then we apply them to the particular scenario of RNN sentiment analysis model training on the Twitter dataset in order to ensure faster convergence and accuracy that is close to the accuracy obtained in centralized data (i.e., the scenario of all training data being aggregated in one central location).

More precisely we explore the benefits for using `FedAdam` versus `FedAvg` optimizer.

For this experiment only the Twitter users with more than 30 tweets are kept (1 Twitter user is considered as 1 client). The total number of clients is 2875, the number of iterations with server is 50, the local epochs $E = 1$ and the client dropout is 10%. The client optimizer is full-batch Adam with $\eta_{client} = 0.001$ and the server learning rate is $\eta_{server} = 1$ for both `FedAvg` and `FedAdam`.

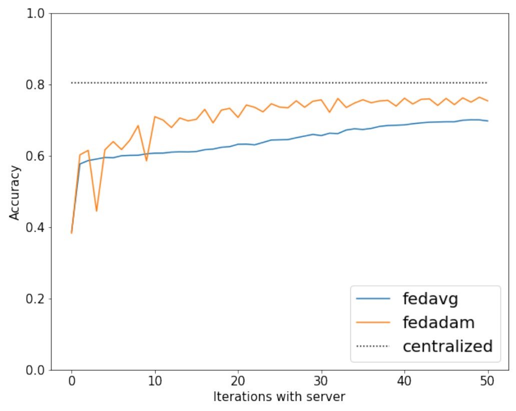

Fig. 13. `FedAvg` vs `FedAdam` with 2875 clients (Twitter)

Figure 13 show that using `FedAdam` instead of `FedAvg` improves the convergence speed of the model. The model reaches 78 % test accuracy with `FedAdam` and 76 % with `FedAvg` after 50 iterations with the server. However, `FedAdam` does not manage to reach the 80 % centralized accuracy. This can be due to suboptimal hyperparameters and other adaptive server optimizers could also have performed better.

11 Conclusion

In this paper we proposed a novel approach for secure aggregation for Federated Learning based on `AES-CTR` masking. The protocol is resilient to client dropout and has reduced client/server communication. We have presented a GPU implementation of `AES-CTR` based masking function. Our implementation outperforms `mbedTLS` and `tinyAES` on V100 GPU. The demonstrated optimizations are largely based on AES CUDA kernels proposed by Cihangir Tezcan [2]. Our new `AES-CTR` GPU kernels could be used as a drop-in replacement of `tinyAES` in `torchcsprng` [22], which in turn should further accelerate other privacy-preserving machine learning frameworks, such as CryptGPU [23]. Futhermore, in CryptGPU share re-randomization (that is, in fact an MPC lift operation for each coefficient) the required random sequence lengths could be even larger than for `Falkor` secure aggregation, and potentially demonstrate excellent GPU performance with our `AES-CTR` kernels. In addition, we proved the security of our `AES-CTR` masking and we showed the scalability of our protocol in two real-world scenario: distributed logistic regression and training a recurrent neural network model for sentiment analysis of Twitter dataset.

References

1. J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2063384.2063405
2. C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67 315–67 326, 2021.
3. R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 1310–1321.
4. P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. A. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao, "Advances and open problems in federated learning," *CoRR*, vol. abs/1912.04977, 2019.
5. J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients - how easy is it to break privacy in federated learning?" in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/c4ede56bbd98819ae6112b20ac6bf145-Abstract.html
6. Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi, "Beyond inferring class representatives: User-level privacy leakage from federated learning," in *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 2019, pp. 2512–2520. [Online]. Available: https://doi.org/10.1109/INFOCOM.2019.8737416
7. G. Damaskinos, R. Guerraoui, A. Kermarrec, V. Nitu, R. Patra, and F. Taïani, "Fleet: Online federated learning via staleness awareness and performance prediction," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, D. D. Silva and R. Kapitza, Eds. ACM, 2020, pp. 163–177. [Online]. Available: https://doi.org/10.1145/3423211.3425685
8. M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 308–318. [Online]. Available: https://doi.org/10.1145/2976749.2978318
9. K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1175–1191. [Online]. Available: https://doi.org/10.1145/3133956.3133982
10. D. Lie and P. Maniatis, "Glimmers: Resolving the privacy/trust quagmire," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, A. Fedorova, A. Warfield, I. Beschastnikh, and R. Agarwal, Eds. ACM, 2017, pp. 94–99. [Online]. Available: https://doi.org/10.1145/3102980.3102996
11. J. So, B. Guler, and A. Avestimehr, "Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 167, 2020. [Online]. Available: https://eprint.iacr.org/2020/167
12. S. Kadhe, N. Rajaraman, O. Koyluoglu, and K. Ramchandran, "Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning," *CoRR*, vol. abs/2009.11248, 2020. [Online]. Available: https://arxiv.org/abs/2009.11248

13. J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1253–1269.
14. M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: privacy-preserving aggregation of multi-domain network events and statistics," in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 2010, pp. 223–240. [Online]. Available: http://www.usenix.org/events/sec10/tech/full_papers/Burkhart.pdf
15. C. Beguier, M. Andreux, and E. Tramel, "Efficient sparse secure aggregation for federated learning," 2021, https://arxiv.org/pdf/2007.14861.pdf.
16. H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 259–282. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs
17. F. Sattler, S. Wiedemann, K. Müller, and W. Samek, "Sparse binary compression: Towards distributed deep learning with minimal communication," in *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*. IEEE, 2019, pp. 1–8.
18. "curand: The api reference guide for curand, the cuda random number generation library," https://docs.nvidia.com/cuda/curand/index.html.
19. "Rand library for hip programming language," https://github.com/ROCmSoftwarePlatform/rocRAND.
20. N. I. of Standards and Technology, "Advanced encryption standard," *NIST FIPS PUB 197*, 2001.
21. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*. IEEE Computer Society, 1997, pp. 394–403. [Online]. Available: https://doi.org/10.1109/SFCS.1997.646128
22. "CSPRNG: Cryptographically secure pseudorandom number generators for PyTorch," https://github.com/pytorch/csprng.
23. S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the GPU," *CoRR*, vol. abs/2104.10949, 2021. [Online]. Available: https://arxiv.org/abs/2104.10949
24. S. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *2007 IEEE International Conference on Signal Processing and Communications*, 2007, pp. 65–68.
25. K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *Int. J. Netw. Comput.*, vol. 2, no. 1, pp. 131–145, 2012.
26. A. Abdelrahman, M. Fouad, H. Dahshan, and A. Mousa, "High performance cuda aes implementation: A quantitative performance analysis approach," in *2017 Computing Conference*, 2017, pp. 1077–1085.
27.
28. S.-W. An and S.-C. Seo, "Highly efficient implementation of block ciphers on graphic processing units for massively large data," *Applied Sciences*, vol. 10, no. 11, 2020.
29. "Small portable aes128/192/256 in c," https://github.com/kokke/tiny-AES-c.
30. S. Carpov, K. Deforth, N. Gama, M. Georgieva, D. Jetchev, J. Katz, I. Leontiadis, M. Mohammadi, A. Sae-Tang, and M. Vuille, "Manticore: Efficient framework for scalable secure multiparty computation protocols," Cryptology ePrint Archive, Report 2021/200, 2021, https://eprint.iacr.org/2021/200.
31. A. Go, R. Bhayani, and L. Huang, "Twitter sentiment classification using distant supervision," *Processing*, vol. 150, 01 2009.
32. S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," 2019.
33. J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162
34. H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," 2017.
35. S. J. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečný, S. Kumar, and H. B. McMahan, "Adaptive federated optimization," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.
36. T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of CRYPTO 84 on Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 1985, p. 10–18.

37. R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” vol. 21, no. 2, 1978. [Online]. Available: https://doi.org/10.1145/359340.359342
38. P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Advances in Cryptology — EUROCRYPT ’99*, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.
39. Z. Brakerski and V. Vaikuntanathan, “Fully homomorphic encryption from ring-lwe and security for key dependent messages,” in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 505–524.

A Discussion: comparison of AES-based masking approach versus homomorphic encryption for Falkor

Falkor aggregation protocol can also be achieved using public key linear homomorphic schemes in the following way:

- $\texttt{client}_i$: each client additively secret-share their plaintext and encrypt using additive homomorphic scheme each share with the public key of each servers, and publishes the tuple of encrypted shares to the Funnel.
- `Funnel`: Given two tuples of encrypted shares, the funnel adds them homomorphically and reduces it to one tuple of encrypted shares.
- $\texttt{server}_j$: at the end, each server grabs its final encrypted share from the final tuple in the Funnel, and decrypts it locally.

With the HE solution, the list of participant identifiers U_r does not need to be transmitted (this list is required in the `AES-CTR`-based masking approach), the size of a tuple of encrypted shares do not grow with the number of clients who participated. In practice, it doesn’t matter for the funnel: a client id is in general 10 bytes, and if 1 million clients participates, the participants list is still $< 10M$, which stays small compared to the actual data. Furthermore, the use-case scenario may actually require to track the participants lists to prevent e.g. double submissions. It does matter however for decryption on the servers: decryption time in Falkor is linear on the number of clients, whereas in HE, it is not.

With ElGamal [36], RSA [37], or Paillier-based [38] HE scheme, the group elements are large (that have a big impact on the network size). The trade-off is that on CPU we can process roughly 1000 $\texttt{AES-CTR}^+$ streams in the time to decrypt one RSA decryption. Besides, the first two Additional difficulty is the fact that El-Gamal and RSA operate on a multiplicative group and our protocol would require to compute discrete logarithms on the plaintext. Paillier scheme is homomorph for addition and does not have this problem. .

With LWE HE scheme [39], the additional benefit is that we can use multi-key homomorphic encryption: a tuple of encrypted shares is just one LWE-encrypted share. LWE also supports packing of coefficients. The disadvantage is that LWE uses noise, whose propagation is additive wrt, and that must be eliminated at decryption (secret shares must be exact). Expect to lose 20 least significant bits for 1 million clients, and to be forced to use 128-bit arithmetic and doing 128-bit FFT at encryption on each client.

B Security Analysis

B.1 Threat model.

In our threat model, there are n clients and ℓ servers. The security of `Falkor` relies on the following assumptions:

- A colluding coalition $\mathcal{G}$ consists of a set $\mathcal{C}$ of at most $n-1$ clients and a set $\mathcal{S}$ of at most $\ell-1$ aggregating servers.
- The colluding coalition has access to the following data:

   1. the plaintext input data of all clients in $\mathcal{C}$ - for party i, this is the local model M_i; in order to separate securing the local model from the security aspects related to model inversion attacks, we assume that the input data is the local model as opposed to the local input training data,
   2. the shared keys $\mathtt{sk}_{i,j}$ for $i \in \mathcal{C}$ and $j \in \mathcal{S}$,
   3. the masked values sent over the network, that is, A_i for $i = 1, \dots, n$.
- The data that needs to be protected is:
   1. input data (local models M_i) of all clients $i \in \overline{\mathcal{C}} := \{1, \dots, n\} - \mathcal{C}$,
   2. secret shares of all servers $j \in \overline{\mathcal{S}} := \{1, \dots, \ell\} - \mathcal{S}$,
   3. the shared keys $\mathtt{sk}_{i,j}$ for $i \in \overline{\mathcal{C}}$ and $j \in \overline{\mathcal{S}}$.

B.2 Real-or-random (RoR) indistinguishability

For a vector x of 64-bit integers and $B_x = \mathtt{bitlength}(x)$ (the total bitlength of x) we define:

$$\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}(x) := x \oplus \mathtt{AES\text{-}CTR\text{-}Stream}_{K,\mathrm{IV}}(B_x),$$

$$\mathtt{AES\text{-}CTR}^{+}_{K,\mathrm{IV}}(x) := x + \mathtt{AES\text{-}CTR\text{-}Stream}_{K,\mathrm{IV}}(B_x).$$

In the first equation, x is interpreted as a sequence of B_x bits. In the second equation $\mathtt{AES\text{-}CTR\text{-}Stream}_{K,\mathrm{IV}}(B_x)$ is interpreted as a vector of integers modulo 2^{64} of same size as x.

We first need to prove that given any message m of B_m-bits, and a list of keys and initial values $(K_1, \dots, K_t)$ and $(\mathrm{IV}_1, \dots, \mathrm{IV}_t)$, the value $m + \sum_{i=0}^{t} \mathtt{AES\text{-}CTR\text{-}Stream}_{K_i,\mathrm{IV}_i}(B_m)$ for an adversary that does not know at least one key K_i reveals nothing about m and K_i. To prove this, we first note that the above expression can be viewed as then encryption $\mathtt{AES\text{-}CTR}^{+}_{K_i,\mathrm{IV}_i}(m')$ for a suitable m', where the $\mathtt{AES\text{-}CTR}^{+}_{K_i,\mathrm{IV}_i}$ encryption only differs from the classical $\mathtt{AES\text{-}CTR}_{K_i,\mathrm{IV}_i}$ by the fact that in the first one, the stream is 64-bits wise added to the message instead of bit-wise xored to the message. We then prove that the two encryption schemes $\mathtt{AES\text{-}CTR}^{+}$ and $\mathtt{AES\text{-}CTR}^{\oplus}$ are equivalent in the sense of IND-CPA security. Bellare et al. [21] discussed various formalisms for the security for symmetric encryption schemes, including indistinguishability under chosen plaintext attacks (IND-CPA). Two variants of IND-CPA they come up with are left-or-right (LoR) and real-or-random (RoR), which they prove to be equivalent. We thus provide our reduction on the real-or-random game.

We consider two basic real-or-random oracles:

1. $\mathtt{AES}^{+}$ - for a key K, it takes as input a secret values x of B_x bits as well as a bit b, it generates uniformly random initial value $\mathrm{IV} \overset{\text{rand}}{\leftarrow} \mathbb{B}^{128}$ (128 bits) and outputs

$$\mathtt{AES}^{+}{}_K(x; b) = \begin{cases} (\mathtt{AES\text{-}CTR}^{+}_{K,\mathrm{IV}}(x), \mathrm{IV}) & \text{if } b = 1, \\ (\mathtt{AES\text{-}CTR}^{+}_{K,\mathrm{IV}}(y \overset{\text{rand}}{\leftarrow} \mathbb{B}^{B_x}), \mathrm{IV}) & \text{if } b = 0. \end{cases}$$

   Here, it is understood that the oracle $\mathtt{AES}^{+}$ picks the random coins for the choice of IV as well as the choice of the random y in the case $b = 0$.
2. $\mathtt{AES}^{\oplus}$ - for a key K, it takes as input a secret values x of B_x bits as well as a bit b, it generates uniformly random initial value $\mathrm{IV} \overset{\text{rand}}{\leftarrow} \mathbb{B}^{128}$ and outputs

$$\mathtt{AES}^{\oplus}{}_K(x; b) = \begin{cases} (\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}(x), \mathrm{IV}) & \text{if } b = 1, \\ (\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}(y \overset{\text{rand}}{\leftarrow} \mathbb{B}^{B_x}), \mathrm{IV}) & \text{if } b = 0. \end{cases}$$

   Similarly, the oracle picks the random coins for the choices of both IV and y.

Furthermore, let $\mathcal{A}^{\mathtt{AES}^{\oplus}}$ be an adversary that has access to the oracle $\mathtt{AES}^{\oplus}(x, b)$. Conversely, let $\mathcal{A}^{\mathtt{AES}^{+}}$ be an adversary having access to the oracle $\mathtt{AES}^{+}(x, b)$.

Next, we will prove that its semantic security is equivalent to that of $\mathtt{AES}^{\oplus}$.

Consider now the following games:

1. The challenger generates a random K for some security parameter k (the key size in bits) and retains K.
2. The challenger selects a random bit $b \in \{0,1\}$
3. The adversary submits polynomial number of queries (x) to the challenger.
4. The challenger responds with the value $\mathtt{AES}^{+}(x;b)$ (resp. $\mathtt{AES}^{\oplus}(x;b)$) to the adversary.
5. The adversary is free to perform any polynomially bounded number of additional computations.
6. Finally, the adversary outputs a guess for the value of b.

The scheme is `ROR IND-CPA` secure (or, `ROR-CPA` for brevity) if every probabilistic polynomial time adversary has only a negligible advantage over a random guessing of b.

To define this notion precisely, consider the following `ROR-CPA` game (experiment):

Algorithm 6 $\mathtt{Exp}_{\mathcal{A}}^{\mathtt{ROR\text{-}CPA\text{-}b}}(k)$

```
Input: An adversary $\mathcal{A}$ having access to one of the random oracles that outputs a bit
1: Challenger selects $K \overset{\mathtt{rand}}{\leftarrow} \{0,1\}^k$
2: Challenger selects $b \overset{\mathtt{rand}}{\leftarrow} \{0,1\}$
3: Adversary computes $d \leftarrow \mathcal{A}(\cdot, b)$
4: return $d$
```

We will instantiate the above game with either $\mathcal{A} = \mathcal{A}^{\mathtt{AES}^{+}}$ or $\mathcal{A} = \mathcal{A}^{\mathtt{AES}^{\oplus}}$.

Definition 1 (`ROR-CPA`). *We define the advantages of the adversaries as*

$$\begin{aligned}\mathtt{Adv}_{\mathcal{A}^{\mathtt{AES}^{+}}}^{\mathtt{ROR\text{-}CPA}}(k) := \mathtt{Prob}\left(\mathtt{Exp}_{\mathcal{A}^{\mathtt{AES}^{+}}}^{\mathtt{ROR\text{-}CPA\text{-}1}}(k) = 1\right) - \\ - \mathtt{Prob}\left(\mathtt{Exp}_{\mathcal{A}^{\mathtt{AES}^{+}}}^{\mathtt{ROR\text{-}CPA\text{-}0}}(k) = 0\right),\end{aligned}$$

$$\begin{aligned}\mathtt{Adv}_{\mathcal{A}^{\mathtt{AES}^{\oplus}}}^{\mathtt{ROR\text{-}CPA}}(k) := \mathtt{Prob}\left(\mathtt{Exp}_{\mathcal{A}^{\mathtt{AES}^{\oplus}}}^{\mathtt{ROR\text{-}CPA\text{-}1}}(k) = 1\right) - \\ - \mathtt{Prob}\left(\mathtt{Exp}_{\mathcal{A}^{\mathtt{AES}^{\oplus}}}^{\mathtt{ROR\text{-}CPA\text{-}0}}(k) = 0\right).\end{aligned}$$

Here, the probabilities are taken over the random coins used by the challenges (in the choices of K and b) as well as the random coins picked by the oracle. To formalize the security notions in the computation setting and relate them to the formalism of [21], we introduce the relevant parameters for the resources of the adversary:

- *t - time complexity*
- *q - number of queries to the underlying oracle*
- *μ - the amount of ciphertext the adversary sees in response to its encryption/masking oracle queries*

For a given set of parameters t, q, μ, we can thus define the advantages

$$\mathtt{Adv}_{\mathtt{AES}^{+}}^{\mathtt{ROR\text{-}CPA}}(k,t,q,\mu) := \max_{\mathcal{A}^{\mathtt{AES}^{+}}} \mathtt{Adv}_{\mathcal{A}^{\mathtt{AES}^{+}}}^{\mathtt{ROR\text{-}CPA}}(k),$$

where the maximum is taken over all adversaries $\mathcal{A}^{\mathtt{AES}^{+}}$ with time complexity t that make at most q queries to the oracle and see a μ-fraction of the ciphertext / masking. Similarly, we define

$$\mathtt{Adv}_{\mathtt{AES}^{\oplus}}^{\mathtt{ROR\text{-}CPA}}(k,t,q,\mu) := \max_{\mathcal{A}^{\mathtt{AES}^{\oplus}}} \mathtt{Adv}_{\mathcal{A}^{\mathtt{AES}^{\oplus}}}^{\mathtt{ROR\text{-}CPA}}(k),$$

where the maximum is over all adversaries $\mathcal{A}^{\mathtt{AES}^{\oplus}}$ with resources parametrized by (t, q, μ).

We start by the following simple equivalence whose proof is rather formal:

Theorem 1 **($\mathtt{AES}^+ \Leftrightarrow \mathtt{AES}^\oplus$).** *For any quadruple of parameters (k, t, q, μ), we have*

$$\mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^+}(k, t, q, \mu) = \mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^\oplus}(k, t, q, \mu) \tag{1}$$

Before, we present the formal proof, we illustrate the steps for the reduction $\mathtt{AES}^+ \Rightarrow \mathtt{AES}^\oplus$ in Fig. 14. The reduction $\mathtt{AES}^\oplus \Rightarrow \mathtt{AES}^+$ is similar.

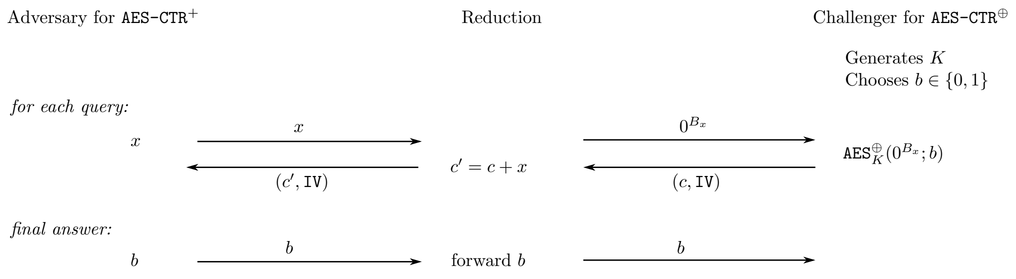

Fig. 14. Reduction $\mathtt{AES}^+ \Rightarrow \mathtt{AES}^\oplus$.

We now formalize the steps of the proof:

Proof. We start by proving the inequality

$$\mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^+}(k, t, q, \mu) \leq \mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^\oplus}(k, t, q, \mu) \tag{2}$$

Consider any (k, t, q, μ)-bounded adversary $\mathcal{A}^{\mathtt{AES}^+}$ for $\mathtt{AES}^+$. We use $\mathcal{A}^{\mathtt{AES}^+}$ to define an adversary $\mathcal{A}^{\mathtt{AES}^\oplus}$ for $\mathtt{AES}^\oplus$ via the following rule:

1. Assume that the challenger for the $\mathtt{AES}^+$ oracle has selected the random key K of size k and the random bit b.
2. For every query x that $\mathcal{A}^{\mathtt{AES}^+}$ submits to its oracle ${\mathtt{AES}^+}_K$, the adversary $\mathcal{A}^{\mathtt{AES}^\oplus}$ submits a binary string of B_x zero bits (denoted by 0^{B_x}) to its oracle $\mathtt{AES}^\oplus$ to obtain $(c, \mathrm{IV}) = {\mathtt{AES}^\oplus}_K(0^{B_x}; b)$ and retains $(x + c, \mathrm{IV})$ as the result of the query.
3. The adversary $\mathcal{A}^{\mathtt{AES}^\oplus}$ outputs exactly the same bit as the output of the adversary $\mathcal{A}^{\mathtt{AES}^+}$ on the oracle ${\mathtt{AES}^+}_K(\cdot; b)$.

Clearly, the adversary $\mathcal{A}^{\mathtt{AES}^\oplus}$ is (k, t, q, μ)-bounded and its advantage in guessing the bit of the challenger is the same as the same as the advantage of $\mathcal{A}^{\mathtt{AES}^+}$. This implies that

$$\mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathcal{A}^{\mathtt{AES}^+}}(k) = \mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathcal{A}^{\mathtt{AES}^\oplus}}(k) \leq \mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^\oplus}(k, t, q, \mu) .$$

Since the above holds for any (k, t, q, μ)-bounded adversary $\mathcal{A}^{\mathtt{AES}^+}$, we obtain (2).

The proof of the opposite inequality

$$\mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^\oplus}(k, t, q, \mu) \leq \mathtt{Adv}^{\mathtt{ROR\text{-}CPA}}_{\mathtt{AES}^+}(k, t, q, \mu) \tag{3}$$

is exactly the same, but with $\mathcal{A}^{\mathtt{AES}^+}$ and $\mathcal{A}^{\mathtt{AES}^\oplus}$ switched. Consider any (k, t, q, μ)-bounded adversary $\mathcal{A}^{\mathtt{AES}^\oplus}$ for $\mathtt{AES}^\oplus$. This time, we use $\mathcal{A}^{\mathtt{AES}^\oplus}$ to define a (k, t, q, μ)-bounded adversary $\mathcal{A}^{\mathtt{AES}^+}$ according to the following rule:

1. Assume that the challenger for the $\mathtt{AES}^{\oplus}$ oracle has selected the random key K and the random bit b.
2. The adversary $\mathcal{A}^{\mathtt{AES}^+}$ chooses the challenge 0 of B_x zero bits where x is the challenge selected by the adversary $\mathcal{A}^{\mathtt{AES}^{\oplus}}$. The adversary $\mathcal{A}^{\mathtt{AES}^+}$ retains the response $x \oplus \mathtt{AES^+}_K(0; b)$.
3. The adversary $\mathcal{A}^{\mathtt{AES}^+}$ outputs exactly the same bit as the output of the adversary $\mathcal{A}^{\mathtt{AES}^{\oplus}}$ on the oracle $\mathtt{AES}^{\oplus}{}_K(\cdot; b)$.

This yields (3).

Security of local models To prove real-or-random indistinguishability, an adversary $\mathcal{A}$ selects a message (model) m of B_m bits (this correspond to a local model of a client not in the collusion group $\mathcal{G}$, i.e., not in $\mathcal{C}$). We assume that the adversary has access to a real-or-random oracle $\mathcal{O}_{K_1,\dots,K_\ell}(\bullet, \bullet; b)$ (here, K_j is the output of key derivation function with input the shared key $\mathtt{sk}_j$ between the client above with the jth server). For a given b, the oracle is given an input m and computes

$$\mathcal{O}_{K_1,\dots,K_\ell}(m; b) = \begin{cases} (m + \sum_{j=0}^{\ell} \mathtt{AES\text{-}CTR\text{-}Stream}_{K_j,\mathrm{IV}_j}(B_m), \mathrm{IV}) \\ \text{ if } b = 1, \\ (y \overset{\mathtt{rand}}{\leftarrow} \mathbb{B}^{B_m}, \mathrm{IV}) \text{ if } b = 0. \end{cases}$$

Without loss of generality, we assume that $\mathcal{S}$ consists of servers $\{1, \dots, \ell - 1\}$. In this case, we can simplify the oracle (by letting $K = K_\ell$, $\mathrm{IV} = \mathrm{IV}_\ell$ and $x = m + \sum_{j=1}^{\ell-1} \mathtt{AES\text{-}CTR\text{-}Stream}_{K_j,\mathrm{IV}_j}(B_m)$) to reduce to the basic $\mathtt{AES}^+(x, b)$ oracle.

Since $\mathtt{AES\text{-}CTR}^+_{K,\mathrm{IV}}$ is as "strong" as $\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}$, it is exactly as hard, given a ciphertext to retrive either any non-trivial information on the message m or on the key K.

As shown in [21, Thm.13], the $\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}$ indistinguishability from random outputs is valid up to 2^{64} encrypted 128-bit blocks. This means we should limit the size of our messages to 2^{64} blocks of 128 bits.

Exactly like for $\mathtt{AES\text{-}CTR}^{\oplus}_{K,\mathrm{IV}}$, the above is under the condition that the same $(K, \mathrm{IV} + \mathtt{counter})$ is never used to encrypt/mask two different messages twice throughout the protocol. Since in our protocol, (K, IV) is derived from $\mathtt{PBKDF2\_sha256}$ with 384 bits of input entropy $(\mathtt{sk}_{i,j})$ and a public unique identifier r for the computation, a collision will occur either due to: an attack on the kdf (with advantage 2^{-192} , or a random collision of the kdf's output, which should produce the same 256-bits of key and two IVs that are closer than the maximum length of the message. If B is the maximal number of bits in a message, the kdf should collide on $384 - \log_2(B/128)$ bits , which occurs with probability $2^{-((384 - \log_2 B + 7)/2)}$. If $B < 70$, the collision probability is $< 2^{-160}$.

Security of server shares Assuming the "worst-case" coalition of exactly $\ell - 1$ servers $\mathcal{S}$ and at most $n - 1$ clients $\mathcal{C}$, the coalition has access to the masked value

$$c = \mathtt{share}_\ell + \sum_{i=1}^{n} \mathtt{mask}(\mathtt{sk}_{i,\ell}, r) + \sum_{i=1}^{n} \sum_{j \neq \ell} \mathtt{mask}(\mathtt{sk}_{i,j}, r).$$

The third term is known entirely by the coalition, in the second term, there is at least one client i for which the coalition does not know the key, so the share $\mathtt{share}_\ell$ is protected (the coalition only knows an $\mathtt{AES\text{-}CTR}^+$ encryption of $\mathtt{share}_\ell$).

Security of the $\mathtt{sk}$ the idea here is to use a composition argument: If by knowing tuples (m, c), we were able to inverse the composition

$$c = \mathtt{AES\text{-}CTR}^+(\mathtt{PBKDF2\_sha256}(\mathtt{sk}, r), m)$$

and find $\texttt{sk}$, then by applying the PBKDF2_sha256 to the result, would recover the key and initial value $K \| IV = \texttt{PBKDF2\_sha256}(\texttt{sk}, r)$ of the encryption $\texttt{AES-CTR}^{+}$. The IND-CPA security of $\texttt{AES-CTR}^{+}$ and the fact that PBKDF2_sha256 is with negligible probability ($< 2^{-156}$) of collision imply that no such these key-recovery attacks exist.

In conclusion In this section we showed that under traditional computational security assumptions about *AES* and PBKDF2_sha256, no information is leaked during the secure aggregation protocol. As a result, the Falkor aggregation protocol can be modeled as an ideal functionality that takes as input each client's local model (privately), and provides secret-shares of its aggregation to the servers. The functionality does not return anything to the clients, and each server learns only its own share.

After an aggregation, the servers run an MPC computation and publishes an output, that becomes the only available result after the aggregation step. To analyze the full security of the protocol, it remains to analyze this sequence of published outputs, and to relate them to the client's private data: techniques based on differential privacy can typically be used to analyse the the full protocol. Such analysis highly depends on the machine learning use-case, and is out of scope for this work.