Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 8 August 2025

J. Chen
Apple Inc.
C. Patton
Cloudflare

4 February 2025

Private Inexpensive Norm Enforcement (PINE) VDAF
draft-chen-cfrg-vdaf-pine-02

Abstract

This document describes PINE, a Verifiable Distributed Aggregation Function (VDAF) for secure aggregation of high-dimensional, real-valued vectors with bounded L2-norm. PINE is intended to facilitate private and robust federated machine learning.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://junyechen1996.github.io/draft-chen-cfrg-vdaf-pine/draft-chen-cfrg-vdaf-pine.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-chen-cfrg-vdaf-pine/.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (mailto:cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg. Subscribe at https://www.ietf.org/mailman/listinfo/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/junyechen1996/draft-chen-cfrg-vdaf-pine.

Status of This Memo

This Internet-Draft will expire on 8 August 2025.

Table of Contents

1. Introduction

The goal of federated machine learning [MR17] is to enable training of machine learning models from data stored on users' devices. The bulk of the computation is carried out on-device: each user trains the model on its data locally, then sends a model update to a central server. These model updates are commonly referred to as "gradients" [Lem12]. The server aggregates the gradients, applies them to the central model, and sends the updated model to the users to repeat the process.

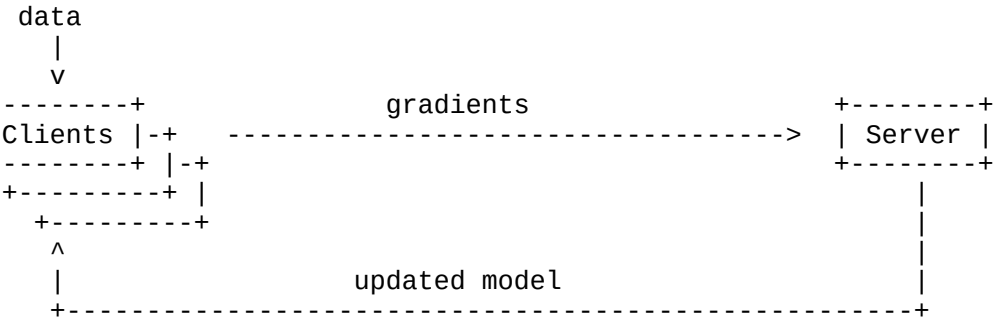

Figure 1: Federated learning

Federated learning improves user privacy by ensuring the training data never leaves users' devices. However, it requires computing an aggregate of the gradients sent from devices, which may still reveal a significant amount of information about the underlying data. One way to mitigate this risk is to distribute the aggregation step across multiple servers such that no server sees any gradient in the clear.

With a Verifiable Distributed Aggregation Function [VDAF], this is achieved by having each user shard their gradient into a number of secret shares, one for each aggregation server. Each server aggregates their shares locally, then combines their share of the aggregate with the other servers to get the aggregate result.

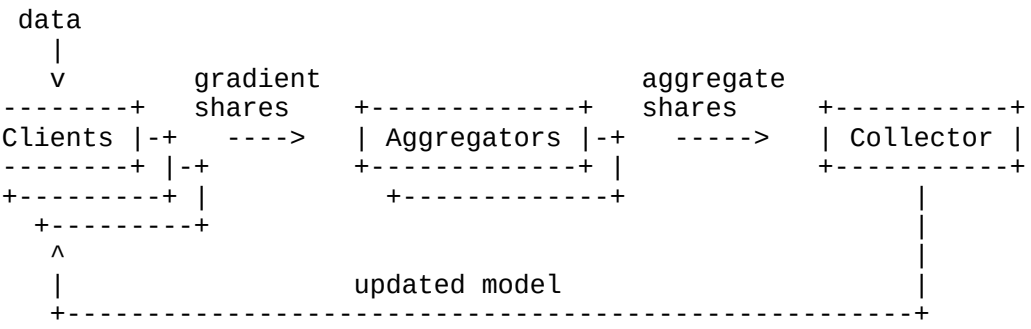

Figure 2: Federated learning with a VDAF

Along with keeping the gradients private, it is desirable to ensure robustness of the overall computation by preventing clients from "poisoning" the aggregate and corrupting the trained machine learning model. A client's gradient is typically expressed as a vector of real numbers. A common goal is to ensure each gradient has a bounded "L2-norm" (sometimes called Euclidean norm): the square root of the sum of the squares of each entry of the input vector. Bounding the L2 norm is used in federated learning to limit the contribution of each client to the aggregate, without over constraining the distribution of inputs. [CP: Add a relevant reference.]

In theory, Prio3 (Section 7 of [VDAF]) could be adapted to support this functionality, but for high-dimensional data, the concrete cost in terms of runtime and communication would be prohibitively high. The basic idea is simple. An FLP ("Fully Linear Proof", see Section 7.3 of [VDAF]) could be used to compute the L2 norm of the secret shared gradient and check that the result is in the desired range, all without learning the gradient or its norm. This computation, on its own, can be done efficiently: the challenge lies in ensuring that the computation itself was carried out correctly, while properly accounting for the relevant mathematical details of the proof system and the range of possible inputs.

This document describes PINE ("Private Inexpensive Norm Enforcement"), a VDAF for secure aggregation of gradients with bounded L2-norm [ROCT23]. Its design is based largely on Prio3 in that the norm is computed and verified using an FLP. However, PINE uses a new technique for verifying the correctness of the norm computation that is incompatible with Prio3.

We give an overview of this technique in Section 3. In Section 4 we specify an FLP circuit and accompanying encoding scheme for computing and verifying the L2 norm of each gradient. Finally, in Section 5 we specify the complete multi-party, 1-round VDAF.

NOTE As of this draft, the algorithms are not yet fully specified. We are still working out some of the minor details. In the meantime, please refer to the reference code on which the spec will be based: https://github.com/junyechen1996/draft-chen-cfrg-vdaf-pine/tree/main/poc

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the same parameters and conventions specified for:

* Clients, Aggregators, and Collectors from Section 5 of [VDAF].

* Finite fields from Section 6.1 of [VDAF]. All fields in this document have prime order.

* XOFs ("eXtendable Output Functions") from Section 6.2 of [VDAF].

A floating point number, denoted float, is a IEEE-754 compatible float64 value [IEEE754-2019].

A "gradient" is a vector of floating point numbers. Each coordinate of this vector is called an "entry". The "L2 norm", or simply "norm", of a gradient is the square root of the sum of the squares of its entries.

The "dot product" of two vectors is to compute the sum of element-wise multiplications of the two vectors.

The user-specified parameters to initialize PINE are defined in Table 1.

Parameter	Type	Description
l2_norm_bound	float	The L2 norm upper bound (inclusive).
dimension	int	Dimension of each gradient.
num_frac_bits	int	The number of bits of precision to use when encoding each gradient entry into the field.

Table 1: User parameters for PINE.

3. PINE Overview

This section provides an overview of the main technical contribution of [ROCT23] that forms the basis of PINE. To motivate their idea, let us first say how Prio3 from Section 7 of [VDAF] would be used to aggregate vectors with bounded L2 norm.

Prio3 uses an FLP ("Fully Linear Proof"; see Section 7.3 of [VDAF]) to verify properties of a secret shared measurement without revealing the measurement to the Aggregators. The property to be verified is expressed as an arithmetic circuit over a finite field (Section 7.3.2 of [VDAF]). Let q denote the field modulus.

In our case, the circuit would take (a share of) the gradient as input, compute the squared L2-norm (the sum of the squares of the entries of the gradient), and check that the result is in the desired range. Note that we do not compute the exact norm: it is mathematically equivalent to compute the squared norm and check that it is smaller than the square of the bound.

Crucially, arithmetic in this computation is modulo q. This means that, for a given gradient, the norm may have a different result when computed in our finite field than in the ring of integers. For example, suppose our bound is 10: the gradient [99, 0, 7] has squared L2-norm of 9850 over the integers (out of range), but only 6 modulo q = 23 (in range). This circuit would therefore deem the gradient valid, when in fact it is invalid.

Thus the central challenge of adapting FLPs to this problem is to prevent the norm computation from "wrapping around" the field modulus.

One way to achieve this is to ensure that each gradient entry is in a range that ensures the norm is sufficiently small. However, this approach has high communication cost (roughly num_frac_bits * dimension field elements per entry), which becomes prohibitive for high-dimensional data.

PINE uses a different strategy: rather than prevent wraparounds, we can try to detect whether a wraparound has occurred.

[ROCT23] devises a probabilistic test for this purpose. A random vector over the field is generated (via a procedure described in Section 4.1.3) where each entry is sampled independently from a particular probability distribution. To test for wraparound, compute the dot product of this vector and the gradient, and check if the result is in a specific range determined by parameters in Table 1.

If the norm wraps around the field modulus, then the dot product is likely to be large. In fact, [ROCT23] show that this test correctly detects wraparounds with probability 1/2. To decrease the false negative probability (that is, the probability of misclassifying an invalid gradient as valid), we simply repeat this test a number of times, each time with a vector sampled from the same distribution.

However, [ROCT23] also show that each wraparound test has a non-zero false positive probability (the probability of misclassifying a valid gradient as invalid). We refer to this probability as the "zero-knowledge error", or in short, "ZK error". This creates a problem for privacy, as the Aggregators learn information about a valid gradient they were not meant to learn: whether its dot product with a known vector is in a particular range. [CP: We need a more intuitive explanation of the information that’s leaked.] The parameters of PINE are chosen carefully in order to ensure this leakage is negligible.

4. The PINE Proof System

This section specifies a randomized encoding of gradients and FLP circuit (Section 7.3 of [VDAF]) for checking that (1) the gradient’s squared L2-norm falls in the desired range and (2) the squared L2-norm does not wrap around the field modulus. We specify the encoding and validity circuit in a class PineValid.

The encoding algorithm takes as input the gradient and an XOF seed used to derive the random vectors for the wraparound tests. The seed must be known both to the Client and the Aggregators: Section 5 describes how the seed is derived from shares of the gradient.

Operational parameters for the proof system are summarized below in Table 2.

Parameter	Type	Description
alpha	float	Parameter in wraparound check that determines the ZK error. The higher alpha is, the lower ZK error is.
num_wr_checks	int	Number of wraparound checks to run.
num_wr_successes	int	Minimum number of wraparound checks that a Client must pass.
sq_norm_bound	Field	The square of l2_norm_bound encoded into a field element.
wr_check_bound	Field	The bound of the range check for each wraparound check.
num_bits_for_sq_norm	int	Number of bits to encode the squared L2-norm.
num_bits_for_wr_check	int	Number of bits to encode the range check in each wraparound check.
bit_checked_len	int	Number of field elements in the encoded measurement that are expected to be bits.
chunk_length	int	Parameter of the FLP.

Table 2: Operational parameters of the PINE FLP.

4.1. Measurement Encoding

The measurement encoding is done in two stages: * Section 4.1.2 involves encoding floating point numbers in the Client gradient into field elements Section 4.1.2.1, and encoding the results for L2-norm check Section 4.1.2.2, by computing the bit representation of the squared L2-norm, modulo q, of the encoded gradient. The result of this step allows Aggregators to check the squared L2-norm of the Client's gradient, modulo q, falls in the desired range of [0, sq_norm_bound]. * Section 4.1.4 involves encoding the results of running wraparound checks Section 4.1.3, based on the encoded gradient from the previous step, and the random vectors derived from a short, random seed using an XOF. The result of this step, along with the encoded gradient and the random vector that the Aggregators derive on their own, allow the Aggregators to run wraparound checks on their own.

4.1.1. Encoding Range-Checked Results

Encoding range-checked results is a common subroutine during measurement encoding. The goal is to allow the Client to prove a value is in the desired range of [B1, B2], over the field modulus q (see Figure 1 in [ROCT23]). The Client computes the "v bits", the bit representation of value - B1 (modulo q), and the "u bits", the bit representation of B2 - value (modulo q). The number of bits for the v and u bits is ceil(log2(B2 - B1 + 1)).

As an optimization for communication cost per Remark 3.2 in [ROCT23], the Client can skip sending the u bits if B2 - B1 + 1 (modulo q) is a power of 2. This is because the available v bits can naturally bound value - B1 to be B2 - B1.

4.1.2. Encoding Gradient and L2-Norm Check

We define a function PineValid.encode_gradient_and_norm(self, measurement: list[float]) -> list[Field] that implements this encoding step.

4.1.2.1. Encoding of Floating Point Numbers into Field Elements

TODO Specify how floating point numbers are represented as field elements.

4.1.2.2. Encoding the Range-Checked, Squared Norm

TODO Specify how the Client encodes the norm such that the Aggregators can check that it is in the desired range.

TODO Put full implementation of encode_gradient_and_norm() here.

4.1.3. Running the Wraparound Checks

Given the encoded gradient from Section 4.1.2 and the XOF to generate the random vectors, the Client needs to run the wraparound check num_wr_checks times. Each wraparound check works as follows.

The Client generates a random vector with the same dimension as the gradient's dimension. Each entry of the random vector is a field element of 1 with probability 1/4, or 0 with probability 1/2, or q-1 with probability 1/4, over the field modulus q. The Client samples each entry by sampling from the XOF output stream two bits at a time: * If the bits are 00, set the entry to be q-1. * If the bits are 01 or 10, set the entry to be 0. * If the bits are 11, set the entry to be 1.

Finally, the Client computes the dot product of the encoded gradient and the random vector, modulo q.

Note the Client does not send this dot product to the Aggregators. The Aggregators will compute the dot product themselves, based on the encoded gradient and the random vector derived on their own.

4.1.4. Encoding the Range-Checked, Wraparound Check Results

We define a function PineValid.encode_wr_checks(self, encoded_gradient: list[Field], wr_joint_rand_xof: Xof) -> tuple[list[Field], list[Field]] that implements this encoding step. It returns the tuple of range-checked, wraparound check results that will be sent to the Aggregators, and the wraparound check results (i.e., the dot products from Section 4.1.3) that will be passed as inputs to the FLP circuit.

The Client obtains the wraparound check results, as described in Section 4.1.3. For each check, the Client runs the range check on the result to see if it is in the range of [-wr_check_bound + 1, wr_check_bound]. Note we choose wr_check_bound, such that wr_check_bound is a power of 2, so the Client does not have to send the u bits in range check. The Client also keeps track of a success bit wr_check_g, which is a 1 if the wraparound check result is in range, and 0 otherwise.

The Client counts how many wraparound checks it has passed. If it has passed fewer than num_wr_successes of them, it should retry, by using a new XOF seed to re-generate the random vectors and re-run wraparound checks Section 4.1.3.

The range-checked results and the success bits are sent to the Aggregators, and the wraparound check results are passed to the FLP circuit.

4.2. The FLP Circuit

Evaluation of the validity circuit begins by unpacking the encoded measurement into the following components:

* The first dimension entries are the encoded_gradient, the field elements encoded from the floating point numbers.

* The next bit_checked_len entries are expected to be bits, and should contain the bits for the range-checked L2-norm, the bits for the range-checked wraparound check results, and the success bits in wraparound checks.

* The last num_wr_checks are the wraparound check results, i.e., the dot products of the encoded gradient and the random vectors.

It also unpacks the "joint randomness" that is shared between the Client and Aggregators, to compute random linear combinations of all the checks:

* The first joint randomness field element is to reduce over the bit checks at all bit entries.

* The second joint randomness field element is to reduce over all the quadratic checks in wraparound check.

* The last joint randomness field element is to reduce over all the checks, which include the reduced bit check result, the L2 norm equality check, the L2 norm range check, the reduced quadratic checks in wraparound check, and the success count check for wraparound check.

In the subsections below, we outline the various checks computed by the validity circuit, which includes the bit check on all the bit entries Section 4.2.2, the L2 norm check Section 4.2.3, and the wraparound check Section 4.2.4. Some of the auxiliary functions in these checks are defined in Section 7.

4.2.1. Range Check

In order to verify the range-checked results reported by the Client as described in Section 4.1.1, the Aggregators will verify (1) v bits and u bits are indeed composed of bits, as described in Section 4.2.2, and (2) the verify the decoded value from the v bits, and the decoded value from the u bits sum up to B2 - B1 (modulo q).

If the Client skips sending the u bits as an optimization mentioned in Section 4.1.4, then the Aggregators only need to verify the decoded value from the v bits is equal to B2 - B1 (modulo q).

4.2.2. Bit Check

The purpose of bit check on a field element is to prevent any computation involving the field element from going out of range. For example, if we were to compute the squared L2-norm from the bit representation claimed by the Client, bit check ensures the decoded value from the bit representation is at most 2^(num_bits_for_norm) - 1.

To run bit check on an array of field elements bit_checked, we use a similar approach as Section 7.3.1.1 of [VDAF], by constructing a polynomial from a random linear combination of the polynomial x(x-1) evaluated at each element of bit_checked. We then evaluate the polynomial at a random point r_bit_check, i.e., the joint randomness for bit check:

```
f(r_bit_check) = bit_checked[0] * (bit_checked[0] - 1) + \
  r_bit_check * bit_checked[1] * (bit_checked[1] - 1) + \
  r_bit_check^2 * bit_checked[2] * (bit_checked[2] - 1) + ...
```

If one of the entries in bit_checked is not a bit, then f(r_bit_check) is non-zero with high probability.

TODO Put PineValid.eval_bit_check() implementation here.

4.2.3. L2 Norm Check

The purpose of L2 norm check is to check the squared L2-norm of the encoded gradient is in the range of [0, sq_norm_bound].

The validity circuit verifies two properties of the L2 norm reported by the Client:

* Equality check: The squared norm computed from the encoded gradient is equal to the bit representation reported by the Client. For this, the Aggregators compute their shares of the

squared norm from their shares of the encoded gradient, and also decode their shares of the bit representation of the squared norm (as defined above in Section 4.2.2), and check that the values are equal.

* Range check: The squared norm reported by the Client is in the desired range [0, sq_norm_bound]. For this, the Aggregators run the range check described in Section 4.2.1.

TODO Put PineValid.eval_norm_check() implementation here.

4.2.4. Wraparound Check

The purpose of wraparound check is to check the squared L2-norm of the encoded Client gradient hasn't overflown the field modulus q.

The validity circuit verifies two properties for wraparound checks:

* Quadratic check (See bullet point 3 in Figure 2 of [ROCT23]): Recall in Section 4.1.4, the Client keeps track of a success bit for each wraparound check, i.e., whether it has passed that check. For each check, the Aggregators then verify a quadratic constraint that, either the success bit is a 0 (i.e., the Client has failed that check), or the success bit is a 1, and the range-checked result reported by the Client is correct, based on the wraparound check result (i.e., the dot product) computed by the Aggregators from the encoded gradient and the random vector. For this, the Aggregators multiply their shares of the success bit, and the difference of the range-checked result reported by the Client, and that computed by the Aggregators. We then construct a polynomial from a random linear combination of the quadratic check at each wraparound check, and evaluate it at a random point r_wr_check, the joint randomness.

* Success count check: The number of successful wraparound checks, by summing the success bits, is equal to the constant num_wr_successes. For this, the Aggregators sum their shares of the success bits for all wraparound checks.

TODO Put PineValid.eval_wr_check() implementation here.

4.2.5. Putting All Checks Together

Finally, we will construct a polynomial from a random linear combination of all the checks from PineValid.eval_bit_checks(), PineValid.eval_norm_check(), and PineValid.eval_wr_check(), and evaluate it at the final joint randomness r_final. The full implementation of PineValid.eval() is as follows:

TODO Specify the implementation of Valid from Section 7.3.2 of [VDAF].

5. The PINE VDAF

This section describes PINE VDAF for [ROCT23], a one-round VDAF with no aggregation parameter. It takes a set of Client gradients expressed as vectors of floating point values, and computes an element-wise summation of valid gradients with bounded L2-norm configured by the user parameters in Table 1. The VDAF largely uses the encoding and validation schemes in Section 4, and also specifies how the joint randomness shared between the Client and Aggregators is derived. There are two kinds of joint randomness used:

* "Verification joint randomness": These are the field elements used by the Client and Aggregators to evaluate the FLP circuit. The verification joint randomness is derived similar to the joint randomness in Prio3 Section 7.2.1.2 of [VDAF]: the XOF is applied to each secret share of the encoded measurement to derive the "part"; and the parts are hashed together, using the XOF once more, to get the seed for deriving the joint randomness itself.

* "Wraparound joint randomness": This is used to generate the random vectors in the wraparound checks that both the Clients and Aggregators need to derive on their own. It is generated in much the same way as the verification joint randomness, except that only the gradient and the range-checked norm are used to derive the parts.

In order for the Client to shard its gradient into input shares for the Aggregators, the Client first encodes its gradient into field elements, and encodes the range-checked L2-norm, according to Section 4.1.2. Next, it derives the wraparound joint randomness for the wraparound checks as described above, and uses that to encode the range-checked, wraparound check results as described in Section 4.1.4}. The encoded gradient, range-checked norm, and range-checked wraparound check results will be secret-shared to (1) be sent as input shares for the Aggregators, and (2) derive the verification joint randomness as described above. The Client then generates the proof with the FLP and secret shares it. The secret-shared proof, along with the input shares, and the joint randomness parts for both wraparound and verification joint randomness, are sent to the Aggregators.

Then the Aggregators carry out a multi-party computation to obtain the output shares (the secret shares of the encoded Client gradient), and also reject Client gradients that have invalid L2-norm. Each Aggregator first needs to derive wraparound and verification joint

randomness. Similar to Prio3 preparation Section 7.2.2 of [VDAF], the Aggregator does not derive every joint randomness part like the Client does. It only derives the joint randomness part from its secret share via the XOF, and applies its part and and other Aggregators' parts sent by the Client to the XOF to obtain the joint randomness seed. Then each Aggregator runs the wraparound checks Section 4.1.3 with its share of encoded gradient and the wraparound joint randomness, and queries the FLP with its input share, proof share, the wraparound check results, and the verification joint randomness. All Aggregators then exchange the results from the FLP and decide whether to accept that Client gradient.

Next, each Aggregator sums up their shares of the encoded gradients and sends the aggregate share to the Collector. Finally, the Collector sums up the aggregate shares to obtain the aggregate result, and decodes it into an array of floating point values.

Like Prio3 Section 7.1.2 of [VDAF], PINE supports generation and verification of multiple FLPs. The goal is to improve robustness of PINE (Corollary 3.13 in [ROCT23]) by generating multiple unique proofs from the Client, and only accepting the Client gradient if all proofs have been verified by the Aggregators. The benefit is that one can improve the communication cost between Clients and Aggregators, by instantiating PINE FLP with a smaller field, but repeating the proof generation (Flp.prove) and validation (Flp.query) multiple times.

The remainder of this section is structured as follows. We will specify the exact algorithms for Client sharding Section 5.1, Aggregator preparation Section 5.2 and aggregation Section 5.3, and Collector unsharding Section 5.4.

5.1. Sharding

TODO Specify the implementation of Vdaf.shard().

5.2. Preparation

TODO Specify the implementations of Vdaf.prep_init(), .prep_shares_to_prep(), and .prep_next().

5.3. Aggregation

TODO Specify the implementation of Vdaf.aggregate().

5.4. Unsharding

TODO Specify the implementation of Vdaf.unshard().

6. Variants

TODO Specify concrete parameterizations of VDAFs, including the choice of field, number of proofs, and valid ranges for the parameters in Table 1.

7. PINE Auxiliary Functions

TODO Put all auxiliary functions here, including range_check(), parallel_sum().

8. Security Considerations

Our security considerations for PINE are the same as those for Prio3 described in Section 9 of [VDAF].

9. IANA Considerations

TODO Ask IANA to allocate an algorithm ID from the VDAF algorithm ID registry.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[VDAF] Barnes, R., Cook, D., Patton, C., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", Work in Progress, Internet-Draft, draft-irtf-cfrg-vdaf-11, 22 August 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-11>.

10.2. Informative References

[BBCGGI19] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs", CRYPTO 2019 , 2019, <https://ia.cr/2019/188>.

[IEEE754-2019] "IEEE Standard for Floating-Point Arithmetic", 2019, <https://ieeexplore.ieee.org/document/8766229>.

[Lem12] Lemar迪chal, C., "Cauchy and the gradient method", 2012, <https://www.elibm.org/article/10011456>.

[MR17] McMahan, B. and D. Ramage, "Federated Learning: Collaborative Machine Learning without Centralized Training Data", 2017, <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>.

[ROCT23] Rothblum, G. N., Omri, E., Chen, J., and K. Talwar, "PINE: Efficient Norm-Bound Verification for Secret-Shared Vectors", 2023, <https://arxiv.org/abs/2311.10237>.

[Tal22] Talwar, K., "Differential Secrecy for Distributed Data and Applications to Robust Differentially Secure Vector Summation", 2022, <https://arxiv.org/abs/2202.10618>.

Acknowledgments

Guy Rothblum Apple Inc. gn_rothblum@apple.com

Kunal Talwar Apple Inc. ktalwar@apple.com

Authors' Addresses

Junye Chen
Apple Inc.
Email: junyec@apple.com

Christopher Patton
Cloudflare
Email: chrispatton+ietf@gmail.com