Secure Aggregation for Federated Learning in Flower

Kwing Hei Li
University of Cambridge

Pedro Porto Buarque de Gusmão
University of Cambridge

Daniel J. Beutel
University of Cambridge

Nicholas D. Lane
University of Cambridge

ABSTRACT

Federated Learning (FL) allows parties to learn a shared prediction model by delegating the training computation to clients and aggregating all the separately trained models on the server. To prevent private information being inferred from local models, Secure Aggregation (SA) protocols are used to ensure that the server is unable to inspect individual trained models as it aggregates them. However, current implementations of SA in FL frameworks have limitations, including vulnerability to client dropouts or configuration difficulties.

In this paper, we present Salvia, an implementation of SA for Python users in the Flower FL framework. Based on the SecAgg(+) protocols for a semi-honest threat model, Salvia is robust against client dropouts and exposes a flexible and easy-to-use API that is compatible with various machine learning frameworks. We show that Salvia's experimental performance is consistent with SecAgg(+)'s theoretical computation and communication complexities.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; • **Computing methodologies → Machine learning approaches**.

KEYWORDS

Federated Learning, Secure Aggregation, Secure Multi-party Computation

1 INTRODUCTION

Federated Learning (FL) [19, 27] is a recent machine learning (ML) paradigm that allows a centralized *server* to compute a global model by aggregating local models trained by a set of *clients*. Though not having direct access to users' data, a malicious server can still infer patterns of private data through inference attacks on clients' local models [2, 12, 28].

Secure aggregation (SA) [6], in general, refers to any protocol that allows a group of mutually distrustful parties, each holding a private value, to compute an aggregate value without revealing any information about their private value to each other. This is especially relevant in the context of FL as we would want the server to perform the aggregation step with SA. That way, the server cannot access clients' trained models and obtain information about their private data.

Current implementations of SA in FL frameworks generally fall under one of two main categories:

Trusted Execution Environment: The data-sensitive computation for aggregating models is delegated to an isolated processing environment, which is supported by trusted hardware running parallel to the operating system, e.g. the Intel Software Guard Extensions [13]. The server is only able to inspect the final result of the computation, but not any intermediate results or inputs to the computation performed in the isolated environment.

FL frameworks such as PySyft [23] and OpenFL [22] provide support for this kind of SA, by using a lightweight library OS, Graphene [8], to integrate its code with the secure hardware. However, a ML engineer may find it difficult to configure the program files to use the hardware; Configuration steps are long and complicated, and there is a lack of documentation and examples for using these trusted hardware to perform SA in FL. In addition, various attacks [16, 18] targeting supposedly-secure hardware have been discovered in recent years.

Multi-party Computation: Privacy of locally-trained models is achieved by applying techniques from cryptography, e.g. Yao's garbled circuits [31], homomorphic encryption [10] or secret sharing [7]. Instead of relying on trusted hardware, the server operates directly on encrypted or masked models to calculate the aggregated result without revealing individual clients' contributions.

FL frameworks such as Crypten [15] provide support for this kind of SA. Though these SA methods can be designed to expose an easy-to-use API to engineers, most implementations of multi-party computation SA protocols cannot work around dropouts, a phenomenon all too common with cross-device FL. On top of that, these protocols often incur significant computation and communication overhead, making them infeasible in larger FL experiments.

In summary, most implementations or proposed solutions for SA in common FL frameworks have one or more of the following limitations that hinder their usability:

- Not trivial to configure and use. This is the case for most frameworks, especially those that rely on trusted execution environments for SA, like PySyft and OpenFL.
- Dependent on certain trusted hardware and prone to existing or future attacks on it. This is the case for all frameworks that use trusted hardware for SA, like PySyft and OpenFL.

- Inability to tolerate client dropouts. This is the case for most frameworks that utilize multi-party computation protocols for SA, like Crypten.
- Computation and communication overhead too significant to be used in larger FL settings. This is the case for many proposed multi-party computation protocols, where the protocol itself is computationally expensive [21].

In this paper, we present Salvia [17], an implementation of SA in the open-source FL framework Flower that aims to address each of the above limitations by:

- Exposing a flexible and easily-configurable API that works with Flower's existing Strategy abstraction.
- Utilizing a multi-party computation protocol that does not require trusted hardware.
- Tolerating various percentages of client dropouts while providing strong security guarantees.
- Using a SA algorithm that has low theoretical computation and communication complexities.

We explain Salvia's system design and implementation details in section 2 and section 3, respectively. We also present experiments that explore the algorithmic aspects of Salvia's computation and communication costs in section 4. Lastly, we discuss Salvia's current limitations and directions for future work in section 5.

2 SYSTEM DESIGN

We present Salvia's design goals that address limitations of other SA implementations, and summarize the assumptions made for our implementation. We also present Flower and SecAgg(+), which are the chosen underlying framework and SA protocol for Salvia's design, respectively.

2.1 Design Goals

Many frameworks provide support for SA, though there are weaknesses in their designs that hinder engineers from using them. Based on these observations, we present five main design goals for implementing Salvia:

1) Usability: Given the difficulty in using SA for some FL frameworks, Salvia should be intuitive and easy to use.

2) Flexibility: Given the complexity of FL systems, Salvia should provide a flexible API for users to configure parameters of the protocol to fit their experiment/deployment goals.

3) Compatibility: Given the robust and diverse range of existing ML frameworks, Salvia should be compatible with the most commonly-used ones.

4) Reliability: Given that dropouts of mobile devices in FL is common, Salvia should be robust against such behaviour.

5) Efficiency: Given that real-world FL is often used on large systems, Salvia should not incur significant communication and computation overhead to the FL training.

2.2 Assumptions

We present the assumptions made for our implementation:

Semi-honest threat model: Parties cannot deviate from the protocol specification. However, corrupted parties may cooperate outside the protocol to exchange information, e.g. secret shares.

Liveness properties on the response time of clients: After sending a request to a client, the server eventually receives a response or detects a disconnection from the client.

Secure links between clients and server: Links are encrypted and authenticated in advance.

2.3 Framework Selection – Flower

Flower (Figure 1) [5] is a recent FL framework that provides higher-level abstractions enabling researchers to extend and implement FL ideas on a reliable stack. It is one of the very few frameworks that can support heterogeneous clients running on different ML frameworks (including TensorFlow [1] and PyTorch [20]) and using different programming languages. Choosing Flower as Salvia's underlying FL framework allows Salvia to be compatible with many existing ML frameworks; Each client can freely choose which ML framework to use for their local training pipelines independently. Flower also has a large suite of built-in Strategies representing state-of-the-art FL algorithms for users to freely extend, modify, and use for their experiments. We also note that Flower did not have any support for SA in the past, which is a significant limitation in terms of privacy promises associated with FL.
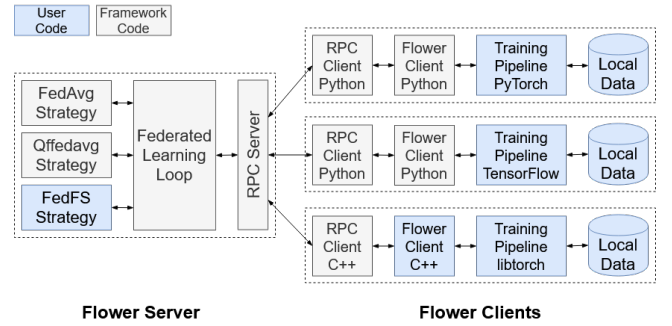

Figure 1: Flower's Basic Architecture (without Salvia)

2.4 Protocol Selection – SecAgg(+)

We chose the SecAgg [25] and SecAgg+ [4] protocols for semi-honest scenarios (parties cannot deviate from the protocol) as the base algorithm of Salvia. These multi-party computation protocols rely on cryptographic primitives to generate private masks for encrypting locally-trained models seen as a single vector of integer weights. These masks cancel each other out when the encrypted vectors are aggregated.

We use n and l to denote the number of clients participating in the protocol and the model size, respectively. We use k to denote the number of other clients each client communicates with (including itself) in the protocol, using the server as a relay.

Like SecAgg and SecAgg+, there are five stages in Salvia:

Stage 0 – Setup Parameters: The server sends values of the protocol's parameters to each client.

Stage 1 – Ask Keys: Each client generates private-public keys and shares the public keys via the server.

Stage 2 – Share Keys: Each client generates secret shares of its private key and a randomly generated seed, and shares it with its k neighbors via the server.

Stage 3 – Ask Vectors: Each client creates masks for its model vector, generated with its private key and randomly generated seed. It then sends the masked vector to the server.

Stage 4 – Unmask Vectors: The server asks clients to contribute secret shares they have received in the Share Keys Stage to remove the masks of the aggregated masked vector.

The main difference between SecAgg and SecAgg+ lies in the value of k. For the former, the value of k is the same as n. As a result, all clients consider each other to be close neighbors, and secret shares are generated for all other clients. For the latter, k is any value smaller than n (usually $O(\log n)$). Each client thus produces shares for its closest k neighbors (including itself), thus producing a $(k-1)$-connected communication graph. As both protocols differ mainly by their k value [1] and that their underlying algorithms work similarly, we use SecAgg(+) to represent both protocols.

The computation and communication overhead complexities of Salvia are summarized in Table 1.

Table 1: SecAgg(+)'s Overhead

	Computation	Communication
Server	$O(nk(k+l))$	$O(n(k+l))$
Client	$O(k(k+l))$	$O(k+l)$

We chose these two protocols as the base algorithm of Salvia for the following reasons:

Flexible API: SecAgg(+) lends itself easily to provide a variety of parameters to be configured, allowing us to design a set of APIs that is flexible to the user's needs. Depending on the parameters used, the protocols can tolerate various percentages of corrupt users and dropouts, and is suitable for a wide range of FL scenarios of various complexities. This enables us to specifically address both our **Flexibility** and **Reliability** design goals.

Simple Configuration: The execution of these protocols for both the server and clients do not rely on any special hardware support. This allows much of the logic to be directly implemented into a FL framework, decreasing the amount of configuration work a user has to set up to use Salvia. This enables us to specifically address our **Usability** design goal.

Low Overhead: SecAgg's communication and computation overhead is lower than other traditional multi-party computation SA protocols. This overhead is even more significantly lowered when SecAgg+ is used, allowing a smaller value of k to be used without sacrificing significant security guarantees. For example, the server's communication and computation complexities are linear to n. Furthermore, the client's communication and computation complexities do not increase with n. This enables us to specifically address our **Efficiency** design goal.

Clear Specifications: SecAgg(+) served as the inspiration for many other SA multi-party computation algorithms, e.g. CCESA [9],

TurboAgg [29] and FastSecAgg [14]. We are also not aware of other common FL frameworks that provide SA via the SecAgg(+) protocols. Implementing SecAgg(+) in a FL framework could provide valuable insights on the implementations of other similar protocols.

3 IMPLEMENTATION

We now describe details of implementing Salvia within Flower.

3.1 Salvia's Architecture

Salvia's architecture consists of three major components: Salvia-compatible Strategies, the server-side logic (Figure 2), and the client-side logic (Figure 3).

To use Salvia, the user must provide a Salvia-compatible Strategy that provides the configuration parameters of the protocol when starting the server, see subsection 3.3 and subsection 3.4 for details.

Salvia's server-side logic provides functions that are called by the FL Loop, the heart of Flower's core architecture (Figure 1). In a normal FL round, the loop asks the Strategy to produce configuration parameters, sends those parameters to the clients via the RPC server and client, receives the trained clients' model vectors and delegates the result aggregation to the Strategy. If the user chooses to use SA in their FL training stage, a special *sec_agg_fit* function is executed in the Loop, which first asks the Strategy to provide parameters that customize core aspects of the SA algorithm. Unlike a normal FL, the aggregation computation is performed within the Loop instead of being delegated to the Strategy. Though this limits the flexibility of the aggregation step (subsection 3.3), this is necessary as the aggregation computation involves complicated logic to remove the masks of the aggregated model vector, and should not be exposed to the user directly.
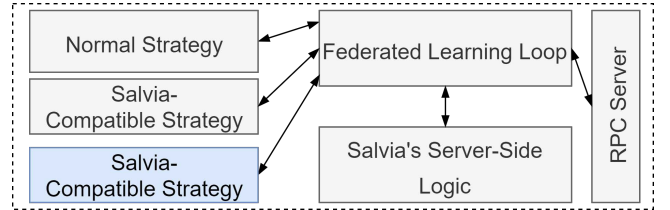

Figure 2: Server-side Architecture with Salvia
The Salvia-compatible Strategy can be one of the standard Strategies provided by Flower (grey), or one that is implemented by the user themselves (blue).

Salvia's client-side logic is implemented in a wrapper class of the Flower client. Depending on the header of the messages received, the client inspects its contents, executes the corresponding SecAgg(+) function, and responds to the server's request for each stage of the SecAgg(+) protocol.

3.2 Cryptographic Primitives

A suite of functions providing cryptographic primitives are used in both the server and client-side logic of Salvia. We opted not to directly provide our own implementations of these cryptographic primitives at the current stage. Instead, we make use of cryptography modules that are widely used in the Python community to
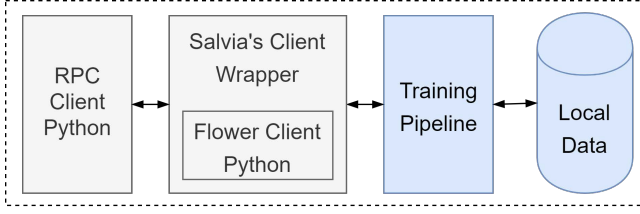
---

[1]We acknowledge the fact that there exists an optimization that only works for SecAgg, but not SecAgg+ in the Unmask Vectors Stage. However, since both protocols have many features in common, we consider SecAgg+ as a generalized version of SecAgg.

Figure 3: Client-side Architecture with Salvia

limit the risk of incorrect implementation of critical cryptography functions. We discuss the cryptographic primitives used in Salvia.

Private-Public Key Generation: This is used to create a common seed between two clients for generating pairwise masks, and a mutual key shared between two clients for authenticated encryption of secret shares. We used functions provided by the Elliptic curve cryptography module in Cryptography [3] for this primitive.

Authenticated Encryption: To distribute secret shares during the Share Keys stage, the message is encrypted by an authenticated encryption function to guarantee confidentiality and integrity of the message. We used functions provided by the Fernet module in Cryptography for this primitive.

t-**out-of-**n **Secret Sharing Scheme:** Depending whether a client dropped out after the Ask Vectors Stage, the server asks all clients for a secret share they received, either from the original client's first private key, or its private seed. If at least t out of n shares are received by the server, then the secret is reconstructed [26]. Otherwise, the shares cannot give any extra information about the secret. We used the Shamir's secret sharing module from the PyCryptodome [11] library for this primitive.

Random Number Generator: Each client produces a random number that acts as the seed for generating its own private mask of its model vector. We used the random function provided by the Python [30] *os* library for this primitive.

Pseudo Random Number Generator: To ensure pairs of clients generate the same pairwise masks, a pseudo-random number generator is required so that the same mask is produced among clients when given identical seeds. We used the standard random function in Python's standard library for this primitive.

3.3 Federation Strategies

In Flower, users can experiment with state-of-the-art algorithms and modify the behavior of their FL workload through the Strategy abstraction. A Strategy customizes core aspects of the FL process, e.g. client sampling and training parameters. Currently, Flower provides a comprehensive suite of FL Strategies, representing the wide range of FL algorithms used by the FL community, e.g. FedAvg [19] and FedProx [24]. In addition, users can extend or modify these built-in Strategies, or even implement their own. We address our **Usability** and **Compatibility** design goals by allowing Salvia to be compatible with this Strategy abstraction, i.e. users can choose Strategies easily and flexibly to be used in conjunction with Salvia.

For a Strategy to be Salvia-compatible, the Strategy must also be a subclass of the *SecAggStrategy* abstract base class. This means that the user needs to provide a definition of *get_sec_agg_param()*,

a function called by the FL Loop at the beginning of the round to obtain a dictionary of SecAgg(+)-related parameters, see subsection 3.4. After assigning default values for undefined parameters, the FL Loop verifies that all parameters are valid for the SecAgg(+) protocol and passes the dictionary to clients in the Setup Parameters Stage. Since the FL Loop automatically fills in default values for undefined parameters, the implementation of the *get_sec_agg_param()* could be as simple as returning an empty dictionary.

Like a normal Strategy, a Salvia-compatible one provides arbitrary logic and parameters to customize the client sampling and client's model fitting process. However, as mentioned in subsection 3.1, it cannot tune the weighted aggregation process which occurs together with the Unmask Vectors Stage in the FL Loop, see subsection 3.5.

3.4 API Parameters

To achieve our **Flexibility** goal, users can specify values of Salvia-related parameters for configuring the SA protocol via the dictionary returned by the *get_sec_agg_param()* function of their Strategy. These parameters are categorized into these classes:

Minimum Number of Clients: To prevent the server from computing an aggregated vector of too few people (an aggregated vector from a single client in the extreme case), the user can specify the minimum number of clients required to be available by the end of the protocol. If the number of available clients drops below this limit, the server and clients refuse to continue the protocol for security reasons.

The user can control this limit through one of two ways, by providing an exact value of this limit with the parameter *min_num*, or by specifying a fraction with respect to the number of clients sampled in *min_frac*. If both are provided, the least restrictive of the two is used.

Secret Sharing: The generation and reconstruction functions of the secret sharing cryptographic primitive is controlled by two parameters provided by the user. *share_num* specifies the number of shares generated for each secret, and *threshold* specifies the minimum number of shares required to reconstruct the secret. Incidentally, *share_num* also represents the value of k, i.e. the number of neighbors each client communicates with (including itself). Therefore, if the value of *share_num* is identical to the number of clients sampled, the protocol executed is SecAgg.

Quantization: For masks generated by cryptographic primitives to work properly, client's locally-trained model vector must be quantized beforehand. This quantization step can be configured through the *clipping_range* and *target_range* parameters. For each real value of the trained model vector, clients first clip the value within the range [−*clipping_range*, *clipping_range*] and then map it to an integer between 0 and (*target_range*-1) uniformly. Similarly, an inverse translation is performed on the server's side by converting each integer element of the quantized aggregated vector to the float type, and mapping it to a real value in the range [−*clipping_range*, *clipping_range*] uniformly to produce the real-valued aggregated model vector.

Though information is lost after quantization which might affect the accuracy of the FL, this procedure is compulsory for the cryptographic primitives to work properly. A user can decrease

the amount of information lost from quantization by providing a sufficiently large value of *target_range*.

Weighted Aggregation: *max_weights_factor* denotes the maximum weight that could be applied to a client's model vector for weighted aggregation. The *mod_range* parameter specifies the modulus of the aggregation computation, and also the range of values elements in client's vector masks can take, see subsection 3.5.

3.5 Weighted Aggregation

Fundamentally, the SecAgg and SecAgg+ protocols only allow the server to calculate an unweighted average of all received model vectors. However, we often want to calculate a weighted average of all vectors, e.g. a client who has trained on more data should have a larger influence on the final global model than one who has trained with fewer. This is not trivial because the server cannot perform arbitrary operations on clients' masked vectors without modifying the masks themselves.

We address this issue by applying the technique described in the appendix of [25]: Salvia's client-side logic automatically multiplies the client's model vector by the amount of training data used, and appends that factor to the front of the vector, producing a modified vector to be masked. On the server's side, at the end of the Unmask Vectors Stage, the server pops the first element of the vector, representing the sum of all factors, and divides the rest of the vector by that value to obtain a weighted average of all input vectors.

The *max_weights_factor* prevents clients from multiplying their vector by too large a factor that risks 'overflowing' the field and affecting the aggregation result. If a client trains its model by an amount of data that exceeds *max_weights_factor*, the vector is only multiplied by *max_weights_factor* instead of the actual amount of data trained. In addition, after all Salvia-related parameters are defined, the FL Loop checks that *mod_range* is at least as large as (*max_weights_factor* × *target_range* × number of sampled clients) so that it is impossible for the aggregated vector to exceed the chosen modulus field.

An interesting side-effect of this design choice is that we can execute unweighted aggregation by setting *max_weights_factor* to 1, as each client automatically multiplies their vector by the same factor of value 1.

4 EVALUATION

To verify that Salvia's behavior matches our expectations in theoretical complexity, we evaluate the changes of Salvia's computation and communication overhead with the number of clients or the model vector size. We ran our FL simulations on a Linux workstation with an Intel Xeon E-2136 CPU (3.30GHz), with 256 GB of RAM. In our simulations, all entries of our local vectors were of size 24 bits. We ignore communication latency in our simulations. Moreover, all dropouts simulated happened after stage 2, i.e. Share Keys Stage. This is because this would impose the most significant overhead as the server not only needs to regenerate their secret, but also compute their pairwise masks generated between their neighbors.

For our simulations, *share_num* and *threshold* were set to 51 and 26, respectively. These parameters were chosen to reference SecAgg+'s proven correctness and security guarantees, where we can tolerate up to 5% dropouts and 5% corrupted clients with correctness holding with probability $1 - 2^{-20}$ and security holding with probability $1 - 2^{-40}$. Though not shown in our following simulations, users can choose larger values for *share_num* and *threshold* to tolerate higher percentages of dropouts and corrupted clients.

We observe that all our experiment results are consistent with the theoretical computation and communication overhead complexities for SecAgg(+) from Table 1, thus achieving our **Efficiency** design goal.

Varying the number of clients: We measured the CPU running times of the server and a client, and the total data transfer per client, as the number of sampled clients increases. Fixing the model vector size to 100k entries, we plotted the results measured through sampling 100 clients to sampling up to 500 clients in Figure 4a, 4b, and 4c. We also measured how the performance would change after client dropouts by repeating the experiments with a 5% dropouts.

We observe that the server's running time increases linearly with the number of sampled clients, which matches the expected computation cost's complexity as the server repeats the same operations for each available client, e.g. reconstructing secrets and generating masks. The server's running time increases whenever there are 5% clients dropping out, as the server has to perform extra computations to calculate all *k* pairwise masks for each client dropped out in the Unmask Vectors Stage. On the other hand, the client's running time remains constant with the number of sampled clients regardless whether there are dropouts. This is because each client only communicates with *k* other neighbors and the actual number of sampled clients does not affect the client-side logic. Lastly, we note that the total data transferred per client remains unchanged as each client is only communicating with exactly *k* neighbors regardless of the total number of clients and dropouts.

Varying vector size: We measured the CPU running times of the server and a client, and the total data transfer per client, as the model vector size increases. Fixing the number of sampled clients to 100, we plotted the results measured through aggregating a vector of size 100k entries to aggregating one of size 500k entries in Figure 4d, 4e, and 4f. Like before, we repeated our experiments with a 5% dropouts.

We observe that both the server's and clients' running times increase linearly with the model vector size. This matches both expected computation cost's complexities because all computation costs involving model vectors are linear to the vectors' sizes, e.g. generating masks and unmasking vectors. In addition, like before, the server's running time increases and the client's running time remains unchanged when there are 5% clients dropping out. We also observe that the total data transferred per client increases linearly with the model vector size as expected, because each client sends the vector to the server for aggregation.

5 LIMITATIONS AND FUTURE WORK

We recognize a few limitations of our current work and we point out future work to address these issues.
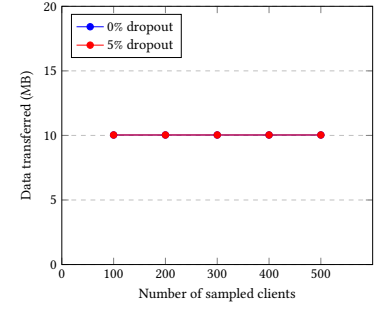
At this moment, the secret sharing scheme function is one of the most significant bottlenecks on Salvia's computation cost. In the future, we wish to provide our own implementation of the
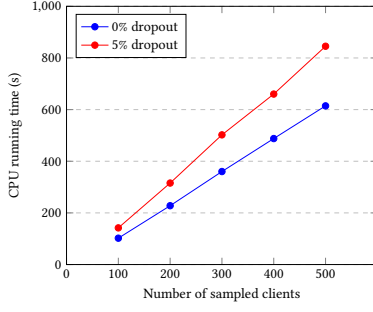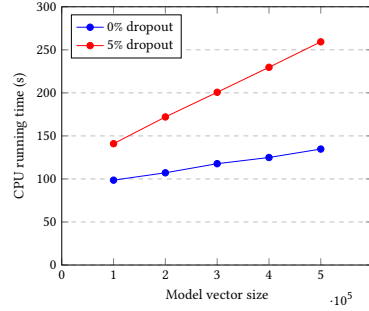
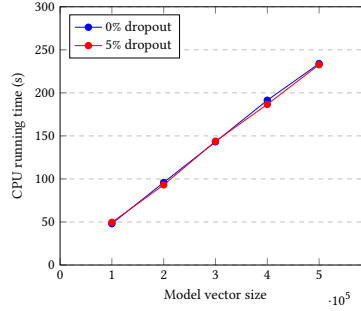**(a) Running time of server with increasing number of clients**
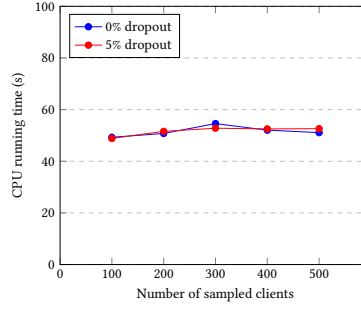
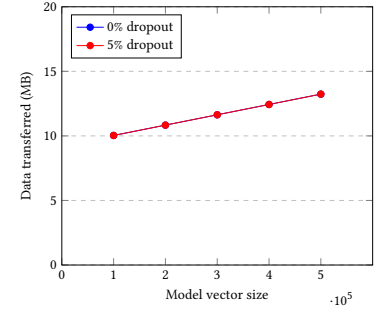**(b) Running time of client with increasing number of clients**

(c) Total data transfer per client with increasing number of clients

(d) Running time of server with increasing vector size

(e) Running time of client with increasing vector size

(f) Total data transfer per client with increasing vector size

Figure 4: Experiment Results

secret sharing mechanism that is more efficient while not sacrificing security guarantees.

In addition, our support for SA on Flower is only limited to Python users. We aim to rewrite cryptographic functions that are specific to the Python library, such that it can be used in FL scenarios where clients are running under different programming languages on their backends, e.g. C++.

Lastly, we would want to allow users to customize more aspects of their FL experiments. For example, Salvia fixes clients' weights factor to be the amount of training data used. In addition, users cannot customize the quantization step. We hope to extend our implementation to allow users to provide arbitrary logic specifying how clients compute the weights factor and how quantization is performed via the Salvia-compatible Strategy.

6 CONCLUSION

We presented Salvia, an open-source implementation of SA based on the SecAgg(+) protocols in the FL framework Flower. Leveraging Flower's ML framework-agnostic property, Salvia is compatible with a diverse range of ML frameworks. We explained how Salvia's API provides a set of flexible and easily-configurable parameters and how it works together with Flower's Strategy abstraction. We showed that Salvia can handle client dropouts, and its performance is consistent with SecAgg(+)'s theoretical computation and communication complexities. Future work includes improving the efficiency of the secret-sharing functions, extending Salvia for clients

running in other programming languages, and providing support for configuring the weights factor via the Strategy abstraction.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.
[2] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Oct 2016). https://doi.org/10.1145/2976749.2978318
[3] Python Cryptographic Authority. 2021. Cryptography (version 3.4.8). https://github.com/pyca/cryptography

[4]   James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1253–1269. https://doi.org/10.1145/3372297.3417885

[5]   Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. 2020. Flower: A Friendly Federated Learning Research Framework. *CoRR* abs/2007.14390 (2020). arXiv:2007.14390 https://arxiv.org/abs/2007.14390

[6]   Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2016. Practical Secure Aggregation for Federated Learning on User-Held Data. *CoRR* abs/1611.04482 (2016). arXiv:1611.04482 http://arxiv.org/abs/1611.04482

[7]   Elette Boyle, Kai-Min Chung, and Rafael Pass. 2015. Large-Scale Secure Computation: Multi-party Computation for (Parallel) RAM Programs. In *CRYPTO*. Springer, 742–762. https://doi.org/10.1007/978-3-662-48000-7_36

[8]   Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[9]   Beongjun Choi, Jy-yong Sohn, Dong-Jun Han, and Jaekyun Moon. 2020. Communication-Computation Efficient Secure Aggregation for Federated Learning. *CoRR* abs/2012.05433 (2020). arXiv:2012.05433 https://arxiv.org/abs/2012.05433

[10] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2011. Multiparty Computation from Somewhat Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2011 (01 2011), 535. https://doi.org/10.1007/978-3-642-32009-5_38

[11] Helder Eijs. 2020. PyCryptodome (version 3.9.9). https://github.com/Legrandin/pycryptodome

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1322–1333. https://doi.org/10.1145/2810103.2813677

[13] Intel Corporation 2007. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.

[14] Swanand Kadhe, Nived Rajaraman, Onur Ozan Koyluoglu, and Kannan Ramchandran. 2020. FastSecAgg: Scalable Secure Aggregation for Privacy-Preserving Federated Learning. *CoRR* abs/2009.11248 (2020). arXiv:2009.11248 https://arxiv.org/abs/2009.11248

[15] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. 2020. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*.

[16] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[17] Kwing Hei Li. 2021. SecAgg Experimental. https://github.com/hei411/flower/tree/secagg_experimental.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[19] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Jerry Zhu (Eds.). PMLR, 1273–1282. https://proceedings.mlr.press/v54/mcmahan17a.html

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[21] Vibhor Rastogi and Suman Nath. 2010. Differentially Private Aggregation of Distributed Time-Series with Transformation and Encryption. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 735–746. https://doi.org/10.1145/1807167.1807247

[22] G Anthony Reina, Alexey Gruzdev, Patrick Foley, Olga Perepelkina, Mansi Sharma, Igor Davidyuk, Ilya Trushkin, Maksim Radionov, Aleksandr Mokrov, Dmitry Agapov, Jason Martin, Brandon Edwards, Micah J. Sheller, Sarthak Pati, Prakash Narayana Moorthy, Shih han Wang, Prashant Shah, and Spyridon Bakas. 2021. OpenFL: An open-source framework for Federated Learning. arXiv:2105.06413 [cs.LG]

[23] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. 2018. A generic framework for privacy preserving deep learning. *CoRR* abs/1811.04017 (2018). arXiv:1811.04017 http://arxiv.org/abs/1811.04017

[24] Anit Kumar Sahu, Tian Li, Maziar Sanjabi, Manzil Zaheer, Ameet Talwalkar, and Virginia Smith. 2018. On the Convergence of Federated Optimization in Heterogeneous Networks. *CoRR* abs/1812.06127 (2018). arXiv:1812.06127 http://arxiv.org/abs/1812.06127

[25] Aaron Segal, Antonio Marcedone, Benjamin Kreuter, Daniel Ramage, H. Brendan McMahan, Karn Seth, K. A. Bonawitz, Sarvar Patel, and Vladimir Ivanov. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *CCS*. https://eprint.iacr.org/2017/281.pdf

[26] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[27] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 909–910. https://doi.org/10.1109/ALLERTON.2015.7447103

[28] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2017.41

[29] Jinhyun So, Basak Guler, and Amir Salman Avestimehr. 2020. Turbo-Aggregate: Breaking the Quadratic Aggregation Barrier in Secure Federated Learning. *CoRR* abs/2002.04156 (2020). arXiv:2002.04156 https://arxiv.org/abs/2002.04156

[30] Guido Van Rossum and Fred L Drake Jr. 1995. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.

[31] Andrew C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 160–164. https://doi.org/10.1109/SFCS.1982.38