ACSAC 2024 Artifact Documentation: Efficient Secure Aggregation for Privacy-Preserving Federated Machine Learning

Rouzbeh Behnia
University of South Florida

Arman Riasi
Virginia Tech

Reza Ebrahimi
University of South Florida

Sherman S. M. Chow
The Chinese University of Hong Kong

Balaji Padmanabhan
University of Maryland, College Park

Thang Hoang
Virginia Tech

I. Abstract

`e-SeaFL` prototype provides a privacy-preserving federated machine learning system that protects users' local gradients in a federated setting. `e-SeaFL` consists of users who contribute to a federated model by sending their local updates to a central aggregation server, an aggregation server that performs secure aggregation and computes proofs of honest aggregation for each iteration and assisting nodes that help the aggregation server to unmask the final model. `e-SeaFL` is primarily implemented in Python with approximately $1,500$ lines of code. The source code is designed to run on Linux-based operating systems and macOS. Our code is available at https://github.com/vt-asaplab/e-SeaFL/tree/ACSAC_2024

II. Description & Requirements

A. Security, privacy, and ethical concerns

Our artifact uses standard cryptographic libraries and widely-used Python packages, all of which can be easily installed using `pip`. The execution of this artifact does not pose any risks to the evaluators' system security, data privacy, or ethical concerns.

B. How to access

Our open-source implementation is available at https://github.com/vt-asaplab/e-SeaFL/tree/ACSAC_2024.

C. Software dependencies

We used the `coincurve` library (https://github.com/ofek/coincurve) to implement public key primitives based on elliptic curves (EC), including ECDSA signatures and EC Diffie-Hellman (ECDH) key exchange protocols. We used AES via `OpenSSL` as a `PRF` to generate masks. To implement communication between all parties, we used the standard Python `socket` library.

III. Set-up

A. Installation

We have provided a detailed `README.md` file, available at https://github.com/vt-asaplab/e-SeaFL/blob/main/README.md, which contains all necessary instructions to set up and run our source code. In summary, the `auto_setup.sh` script can automatically install all required libraries and packages needed to run the `e-SeaFL` code.

B. Configuring and Running

We provid `script.sh` to run the `e-SeaFL` system. To configure and execute `e-SeaFL` system, you can set the following parameters using `script.sh`.

- `-u <number_of_users>`: (Required) Sets the number of users in the federated learning process. It must be greater than 2.
- `-a <number_of_assisting_nodes>`: (Required) Specifies the number of assisting nodes.
- `-c <commitment>`: (Required) Sets operational mode:
  - `0`: without model integrity check.
  - `1`: with model integrity check.
- `-o <printAggOutput>`: (Optional) Controls output:
  - `0`: No print (default).
  - `1`: Print result.
  - `2`: Print summary.
- `-b <bandwidth_mode>`: (Optional) Controls output:
  - `0`: No print (default).
  - `1`: Print outbound bandwidth.
  - `2`: Only print outbound bandwidth.
- `-p <server_port>`: (Optional) Sets the server port:
  - Default: `9000`, with a valid range from `2000` to `60000`.

C. Basic test

We provide two quick and basic tests to verify the functionality of our `e-SeaFL` system. Please ensure that the `./auto_setup.sh` script has been successfully executed before proceeding.

- **T1:** In this test, we set the number of users to 25, configure 3 assisting nodes, and disable the integrity check. The test can be run using the following command:
  `$ ./script.sh -u 25 -a 3 -c 0 -o 2 -p 10050`
  This test will output the computation delay for both the setup and aggregation phases, in both semi-honest and malicious settings, for users, assisting nodes, and the server. It prints the aggregated vector in a summary format.
- **T2:** In this test, we set the number of users to 20, configure 3 assisting nodes, and enable the integrity check. The test can be run using the following command:
  `$ ./script.sh -u 20 -a 3 -c 1 -o 2 -p 10070`
  This test will output the delays similar to test T1, along with the commitment creation time for the integrity check and verification process. It prints the aggregated vector in a summary format.

IV. Evaluation Workflow

Ensure that the `./auto_setup.sh` script has been successfully executed before proceeding.

- **Experiment E1:**
  - **Execution Instructions:** Run the following command, adjusting the parameter value after `-u` to set the **number of users**. The options are {200, 400, 600, 800, 1000}. For example:
    `$ ./script.sh -u 200 -a 3 -c 0 -p 8000`
  - **Results Interpretation:** The script will generate computation delays for the server, users, and assisting nodes, as depicted in Figures 2 and 4. Specifically:
    - **Client Results**:
      - **Setup phase**: Corresponds to Figure 2-b, covering both semi-honest (SH) and malicious (MS) settings.
      - **Aggregation phase**: Corresponds to Figure 4-b, covering both semi-honest (SH) and malicious (MS) settings.
    - **Assisting Node Results**:
      - **Setup phase**: Corresponds to Figure 2-c, covering both semi-honest (SH) and malicious (MS) settings.
      - **Aggregation phase**: Corresponds to Figure 4-c, covering both semi-honest (SH) and malicious (MS) settings.
    - **Server Results**:
      - **Setup phase**: Corresponds to Figure 2-a, covering both semi-honest (SH) and malicious (MS) settings.
      - **Aggregation phase**: Corresponds to Figure 4-a, covering both semi-honest (SH) and malicious (MS) settings.
- **Experiment E2:**
  - **Execution Instructions:** Run the following command and enable the integrity check by setting the value after `-c` to 1. The `-u` parameter can be adjusted similarly to Experiment E1. For example:
    `$ ./script.sh -u 200 -a 3 -c 1 -p 9000`
  - **Results:** This script generates similar results to experiment E1, with the addition of integrity check delays and verification process data.
- **Experiment E3:**
  - **Execution Instructions:** Run the following command, adjusting the parameter value after `-a` to set the number of assisting nodes. Available options are {3, 7, 11, 15, 19}. For example:
    `$ ./script.sh -u 10 -a 7 -c 0 -p 11000`
  - **Results Interpretation:** This script will generate computation delay data for users based on the number of assisting nodes, as depicted in Figure 6. Specifically:
    - **Client Results**:
      - **Setup phase**: Corresponds to Figure 6-a, covering both semi-honest (SH) and malicious (MS) settings.
      - **Aggregation phase**: Corresponds to Figure 6-b, covering both semi-honest (SH) and malicious (MS) settings.
- **Experiment E4:** This experiment details the methodology and calculations used to derive the *outbound bandwidth* results presented in Figures 3 and 5.
  - **Parameters:** We implemented public key primitives based on elliptic curves (EC), including ECDSA signature and EC Diffie-Hellman (ECDH) key (refer to section 4.2.1 of the submitted paper). We used `secp256k1` curve with 256-bit group order for both ECDSA and ECDH protocols (as stated on line 686 of the paper). Therefore, the compressed public key size is 33 B. We set $k = 3$ assisting nodes (line 687), and the weight vector $\mathbf{w}$ comprises 16,000 gradients, each of 4 B (line 823). The iteration number t is 4 B (line 1108), the length of the user list is also 4 B (line 1109), and the signature size transmitted for the malicious setting is 64 B (line 1109), albeit varying due to the DER (Distinguished Encoding Rules) formatting as detailed further.
  - **Execution Instructions:** Run the following command, adjusting the parameter after `-u` to set the number of users and the parameter after `-b` for outbound bandwidth output. Available user options are {200, 400, 600, 800, 1000} and bandwidth settings are {0, 1, 2}. Examples:
    `$ ./script.sh -u 200 -a 3 -c 0 -b 2 -p 15000`
    This script will generate the outbound bandwidth for the server, users, and assisting nodes, as depicted in Figures 3 and 5.
  - **Results Interpretation (*Setup Phase* – Figure 3):** Running the script confirms the analytical communication cost outlined below; the returned values match exactly. Specifically:
    - **Server (Figure 3-a):**
      - **Semi-Honest (SH):** The server's outbound bandwidth in the semi-honest setting is 0 because the server does not transmit any data, as stated on lines 903-904 of

the paper.

- **Malicious (MS):** In the malicious setting, the server's outbound bandwidth ranges from 6,600 to 33,000 bytes for 200 to 1,000 users. As described in Algorithm 1, Phase 1, Step 7, the server sends a public key, $\mathsf{pk}_{\Pi}^{\mathsf{S}}$, of size 33 bytes to all users, resulting in an outbound bandwidth of $33 * n$ = 6,600 to 33,000 B for n = 200 to 1,000 users (lines 896 to 902 of the paper).

* **Client (Figure 3-b):**
  - **Semi-Honest (SH):** The user's outbound bandwidth in the semi-honest setting is 33 B for 200 to 1,000 users. As described in Algorithm 1, Phase 1, Step 1, the user sends a public key, $\mathsf{pk}_{\Sigma}^{\mathsf{P}_i}$, of size 33 B to $k = 3$ assisting nodes, resulting in an outbound bandwidth of $33 * 3 = 99$ bytes (lines 905 to 914 of the paper).
  - **Malicious (MS):** In the malicious setting, the user's outbound bandwidth is 231 B for 200 to 1,000 users. As described in Algorithm 1, Phase 1, Step 1, the user sends $(\mathsf{pk}_{\Pi}^{\mathsf{P}_i}, \mathsf{pk}_{\Sigma}^{\mathsf{P}_i})$ keys to the 3 assisting nodes and $\mathsf{pk}_{\Pi}^{\mathsf{P}_i}$ to the server, each of size 33 B, resulting in an outbound bandwidth of $(33 + 33) * 3 + 33 = 231$ B (lines 905 to 914 of the paper).

* **Assisting node (Figure 3-c):**
  - **Semi-Honest (SH):** The assisting node's outbound bandwidth in the semi-honest setting ranges from 6,600 to 33,000 B for 200 to 1,000 users. As described in Algorithm 1, Phase 1, Step 2, the assisting node sends a public key, $\mathsf{pk}^{\mathsf{A}_j}\Sigma$, of size 33 B to all users, resulting in an outbound bandwidth of $33 * n$ = 6,600 to 33,000 B for n = 200 to 1,000 users (lines 921 to 927 of the paper).
  - **Malicious (MS):** In the malicious setting, the assisting node's outbound bandwidth ranges from 13,233 to 66,033 B for 200 to 1,000 users. As described in Algorithm 1, Phase 1, Step 2, the assisting node sends $(\mathsf{pk}_{\Pi}^{\mathsf{A}_j}, \mathsf{pk}_{\Sigma}^{\mathsf{A}_j})$ to all users and $\mathsf{pk}_{\Pi}^{\mathsf{A}_j}$ to the server, each of size 33 B, resulting in an outbound bandwidth of $(33 + 33) * n + 33$ = 13,233 to 66,033 B for n = 200 to 1,000 users (lines 921 to 923 of the paper).

– **Results Interpretation (*Aggregation Phase* – Figure 5):** Running the script confirms the manual computations for both settings. In the *semi-honest* setting, the returned value matches exactly. In the *malicious* setting, the returned value also matches but with a minor deviation due to the additional bytes used in the DER formatting of the ECDSA signature. The length of the ECDSA signature can vary in practice because it encodes the integers `r` and `s` with a prefix that indicates their lengths, which can lead to signatures being slightly larger—typically by about 6-9 bytes—than the fixed 64 bytes expected in a raw signature format. Specifically:

* **Server (Figure 5-a):**
  - **Semi-Honest (SH):** The server's outbound bandwidth in the semi-honest setting ranges from 12,800,000 to 64,000,000 B for 200 to 1,000 users. As described in Algorithm 2, Phase 2, Step 4, the server broadcasts final updates $\mathbf{w}_t$ to all users, which consist of 16,000 gradients of size 4 B, resulting in an outbound bandwidth of $16,000 * 4 * n = 12,800,000$ to 64,000,000 B for $n = 200$ to 1,000 users (lines 1064 to 1071 of the paper).
  - **Malicious (MS):** In the malicious setting, the server's outbound bandwidth ranges from 12,800,064 to 64,000,064 B for 200 to 1,000 users. As described in Algorithm 2, Phase 2, Step 4, the server transmits the final updates $\mathbf{w}_t$ along with a signature of size 64 B to all users, resulting in an outbound bandwidth of $16,000 * 4 * n + 64 = 12,800,064$ to 64,000,064 B for $n = 200$ to 1,000 users (lines 1064 to 1071 of the paper).

* **Client (Figure 5-b):**
  - **Semi-Honest (SH):** The user's outbound bandwidth in the semi-honest setting is 64,016 B for 200 to 1,000 users. As described in Algorithm 2, Phase 1, Step 3, the user sends the iteration number t of size 4 B and the masked update $\mathbf{y}_t^{\mathsf{P}_i}$, consisting of 16,000 masked gradients each of size 4 B, to the server and transmits t to 3 assisting nodes, resulting in an outbound bandwidth of $(4 + (16,000 * 4)) + (4 * 3) = 64,016$ B (lines 1080 to 1086 of the paper).
  - **Malicious (MS):** In the malicious setting, the user's outbound bandwidth is 64,272 B for 200 to 1,000 users. As described in Algorithm 2, Phase 1, Step 3, the user sends the iteration number t of size 4 B and the masked update $\mathbf{y}_t^{\mathsf{P}_i}$, consisting of 16,000 masked gradients each of size 4 B, along with a signature of size 64 B to the server, and transmits t along with a signature to 3 assisting nodes, resulting in an outbound bandwidth of $(4 + (16,000 * 4) + 64) + ((4 + 64) * 3) = 64,272$ B (lines 1080 to 1086 of the paper).

* **Assisting node (Figure 5-c):**
  - **Semi-Honest (SH):** The assisting node's outbound bandwidth in the semi-honest setting is 64,008 B for 200 to 1,000 users. As described in Algorithm 2, Phase 2, Step 2, the assisting node sends the iteration number t of size 4 B, the length of the user list of size 4 B, and the masked values $\mathbf{a}_t^{\mathsf{A}_j}$, consisting of 16,000 values each of size 4 B to the server, resulting in an outbound bandwidth of $4 + 4 + (16,000 * 4) = 64,008$ B (lines 1105 to 1110 of the paper).
  - **Malicious (MS):** In the malicious setting, the assisting node's outbound bandwidth is 64,072 B for 200 to 1,000 users. As described in Algorithm 2, Phase 2, Step 2, the assisting node sends the iteration number t of size 4 B, the length of the user list of size 4 B, and the masked values $\mathbf{a}_t^{\mathsf{A}_j}$, consisting of 16,000 values each of size 4 B, along with a signature of size 64 B

to the server, resulting in an outbound bandwidth of $4 + 4 + (16,000 * 4) + 64 = 64,072$ B (lines 1105 to 1110 of the paper).

V. Notes on Reusability

We note that our implementation code for `e-SeaFL` is a proof-of-concept prototype and is not ready for production use.