safe-illustration

August 31, 2022

0.1 The SAFE Protocol

0.1.1 Introduction

This notebook illustrates a run of the SAFE Protocol described in (1).

SAFE is a protocol that allows for the secure aggregation of $N > 1$ feature vectors in a semi-honest security model.

The Toy implementation of SAFE is merely illustrative– mock data of a small size will be used to illustrate the basic mechanics of the SAFE protocol.

n.b. The code implementation in this notebook can be improved– for instance, it would be preferable to implement the protocol in an *object-oriented* manner for code that is less visually noisy. However, to illustrate the inner workings of each step of the protocol more clearly, I opt for a procedural approach. See the notebook where I apply the protocol to the use case for an object-oriented implementation

0.1.2 Setting Up Toy Implementation

To illustrate how the SAFE Protocol works, we will mock the data of 3 users who contribute feature vectors.

Set Up Raw Vector Creation First, let's create the raw feature vectors for each user.

These vectors represent distributions of the raw data the SAFE protocol wants to keep private whilst allowing for useful computation over.

[ ]:
```
# SET UP - Create raw feature vectors

import torch
import numpy as np
# for reproducibility
torch.manual_seed(0)

NUMBER_OF_USERS = 3
NUMBER_OF_FEATURES_PER_USER_VECTOR = 2
VECTOR_SHAPE = (NUMBER_OF_FEATURES_PER_USER_VECTOR, 1)

raw_feature_vectors = {}
```

```
for i in range(NUMBER_OF_USERS):
    raw_feature_vectors[i] = torch.rand(VECTOR_SHAPE)
    print(f"Here is the feature vector for user {i}: ")
    print(raw_feature_vectors[i])
    print("\n")
```

```
Here is the feature vector for user 0:
tensor([[0.4963],
        [0.7682]])


Here is the feature vector for user 1:
tensor([[0.0885],
        [0.1320]])


Here is the feature vector for user 2:
tensor([[0.3074],
        [0.6341]])
```

0.1.3 SHARE CREATION AND DISTRIBUTION

Next, each user will generate (N-1) shares from the interval [-D, D], where N is the number of users.

Each share is a K-dimensional vector, where K is the number of features each user's raw feature vector has.

Each user uses these N-1 shares to generate their own Nth share.

Then, they will distribute the N-1 shares to the other N-1 users, whilst keeping their Nth share.

[ ]:
```
# SHARE CREATION

import random
from token import NUMBER
from unicodedata import name

random.seed(0)

D_VALUE = 10.0 # defining the interval from which users drawn random share␣
 ↪vectors.

print("Generating the shares for each user \n")
each_users_shares_to_distribute = {} # mapping from i to list of tensors.
each_users_Nth_share = {} # mapping from i to 4-D tensor
for user in range(NUMBER_OF_USERS):
```

```
    # Each user first generates N-1 shares : a share to be sent to every other␣
↪user.
    each_users_shares_to_distribute[user] = []
    for other_user in range(NUMBER_OF_USERS - 1):
        share_to_distribute = [random.uniform(-D_VALUE, D_VALUE) for i in␣
↪range(NUMBER_OF_FEATURES_PER_USER_VECTOR)]
        share_to_distribute = torch.reshape(torch.tensor(share_to_distribute),␣
↪VECTOR_SHAPE)
        each_users_shares_to_distribute[user].append(share_to_distribute)
    # Then, each user calculates their own Nth share from the N-1 Shares the␣
↪draw from the interval
    sum_of_shares = torch.zeros(VECTOR_SHAPE)
    for share in each_users_shares_to_distribute[user]:
        sum_of_shares.add_(share)
    each_users_Nth_share[user] = raw_feature_vectors[user] - sum_of_shares

print("Let's take a look at shares each user will distribute:\n")
for user in range(NUMBER_OF_USERS):
    print(f"User {user}: \n")
    for share in each_users_shares_to_distribute[user]:
        print(share)
    print("\n")
    print("And here's their Nth share:\n")
    print(each_users_Nth_share[user])
    print("\n")
```

```
Generating the shares for each user

Let's take a look at shares each user will distribute:

User 0:

tensor([[6.8884],
        [5.1591]])
tensor([[-1.5886],
        [-4.8217]])


And here's their Nth share:

tensor([[-4.8036],
        [ 0.4308]])


User 1:

tensor([[ 0.2255],
```

```
        [-1.9013]])
tensor([[ 5.6760],
        [-3.9337]])


And here's their Nth share:

tensor([[-5.8130],
        [ 5.9671]])


User 2:

tensor([[-0.4681],
        [ 1.6676]])
tensor([[8.1623],
        [0.0937]])


And here's their Nth share:

tensor([[-7.3868],
        [-1.1273]])
```

Now, let's distribute the shares among the users.

[ ]:
```python
# SHARE Distribution

print("Distributing shares...\n")
each_users_received_shares = {}
# set up the map
for i in range(NUMBER_OF_USERS):
    each_users_received_shares[i] = []
# fill the map
for user in range(NUMBER_OF_USERS):
    other_user_ids = [ind for ind in range(NUMBER_OF_USERS) if ind != user]
    # get the shares for user_0 to send to other users
    shares_to_send = [share for share in each_users_shares_to_distribute[user]]
    for other_user, share_to_send in zip(other_user_ids, shares_to_send):
        each_users_received_shares[other_user].append(share_to_send)
print("Let's see what the first three users got:")
for i in range(3):
    print(f"User_{i} received:")
    print(*each_users_received_shares[i], sep="\n")
    print("\n")
```

```
# sanity check
assert len(each_users_received_shares[0]) == NUMBER_OF_USERS - 1
```

```
Distributing shares…

Let's see what the first three users got:
User_0 received:
tensor([[ 0.2255],
        [-1.9013]])
tensor([[-0.4681],
        [ 1.6676]])


User_1 received:
tensor([[6.8884],
        [5.1591]])
tensor([[8.1623],
        [0.0937]])


User_2 received:
tensor([[-1.5886],
        [-4.8217]])
tensor([[ 5.6760],
        [-3.9337]])
```

0.1.4 Obfuscated Vector Generation

Each user can now calculate their own ofuscated feature vector.

[ ]:
```
"""
GENERATING OBFUSCATED FEATURE Vectors:
"""
each_users_masked_vector = {}

for user in range(NUMBER_OF_USERS):
    sum_of_received_shares = torch.zeros(VECTOR_SHAPE)
    for share in each_users_received_shares[user]:
        sum_of_received_shares.add_(share)
    each_users_masked_vector[user] = each_users_Nth_share[user] +␣
 ↪sum_of_received_shares

print("Here's what each user will send to the aggregator:\n")
for user, masked_vector in each_users_masked_vector.items():
    print(f"User {user}:")
    print(masked_vector)
```

```
    print("\n")
```

```
Here's what each user will send to the aggregator:

User 0:
tensor([[-5.0462],
        [ 0.1971]])


User 1:
tensor([[ 9.2377],
        [11.2199]])


User 2:
tensor([[-3.2994],
        [-9.8827]])
```

0.1.5 Aggregating Feature Vectors

Now we will mock an aggregator who receives each feature vector and computes a target function. In the actual protocol, each user would 'send' their masked vector to an aggregator. For simplicity's sake, we will simply act as an aggregator that has already received these vectors, and simply compute the target function.

[ ]:
```
# Computing Target Function: Aggregate Sum

target_result = sum(raw_feature_vectors[user] for user in raw_feature_vectors.
 ↪keys())

masked_vector_result = sum(each_users_masked_vector[user] for user in␣
 ↪each_users_masked_vector.keys())

print(f"Sum of masked vectors: \n\n {masked_vector_result}\n")
print(f"Sum of raw vectors: \n\n {target_result}")
```

```
Sum of masked vectors:

 tensor([[0.8922],
        [1.5343]])

Sum of raw vectors:

 tensor([[0.8922],
        [1.5343]])
```

0.1.6 Utility of Protocol

As the output from the code block above shows, the aggregate sum of the masked vectors is equivalent to the aggregate sum of the raw vectors of the users.

Also, it follows from this that simple mean of both vector sets are equal as well.

SAFE thus does allow one to compute aggregate functions over user data without having direct access to the user data.

0.1.7 Bibliography

(1) Chaulwar A, Huth M. Secure Bayesian Federated Analytics for Privacy-Preserving Trend Detection. 2021; https://arxiv.org/abs/2107.13640.