APPENDIX

In a normal workflow, Recommender first selects a learning method for its learning process. After that, Recommender grabs users' id and their grading of preference for products to train the model. After training procedure, the resulting model can be used to predict how much a user may like a product previously ungraded for this user. In our scenario, we are concerned of service provider knowing the exact grading for products of a single user. As a matter of fact, we find several learning methods implemented in Recommender would lead to leakage of user's grading (e.g. learn_mf_bias, learn_basic_mf, learn_mf_neighbor and learn_social). In the following paragraphs, we go through the leakage in learn_mf_bias and learn_basic_mf and how we find them with the help of PrivacyScope in detail.

We start with an easy to understand leakage in learn_mf_bias and followed by a more complicated leakage in learn_basic_mf. In the learn_mf_bias learning method, when the hyper parameter of the learning algorithm is set to particular value, sensitive data can be reversed from the output. According to current setting in machine learning as a service [39], it is commonplace for service provider to control the hyper parameters of a learning algorithm. Listing 2 shows the function called when Recommender selects learn_mf_bias as its learning method. The sensitive data is transferred into this function through learning_param. The outcome of this function is data struct lfactors which contains the learnt model. During the process inside function learn_mf_bias, it calls another function calculate_average_ratings with sensitive data tset and output data struct lfactors as in Listing 3. With help of PrivacyScope, we find out that the ratings_average field of lfactors could potentially leak sensitive data.

PrivacyScope reports an explicit leakage because it finds out tset->ratings_sum labeled as sensitive input crosses security boundary and is transferred out through lfactors->ratings_average. The authors manually verify the finding and figure out tset->ratings_sum contains sum of the first user's rating if training_set_size is set to 1. What's more, if the service provider set the hyper parameter to receive one rating at a time and use it to train a model, the service provider can infer all the individual ratings for every user through simple computation on the lfactors->ratings_average.

```
struct learned_factors* learn_mf_bias (
    learning_algorithm_params_t* learning_param)
{
    /* learning_param contains secret input in its
    member tset*/
  struct training_set* tset = learning_param->tset;
    /* lfactors contains resulting learnt model*/
  struct learned_factors* lfactors =
    init_learned_factors (&params);
  ...
  calculate_average_ratings (tset, lfactors);
  ...
  return lfactors;
}
```

Listing 2: function learn_mf_bias

```
calculate_average_ratings (struct training_set* tset
    , learned_factors_t* lfactors)
{
  /* training_set_size is a hyper parameter selected
     by service provider */
  double average_rating = (double) tset->ratings_sum
     / ( (double) training_set_size);
  average_rating = (average_rating - 1) / 4.0;
  /* ratings_average leak sensitive data */
  lfactors->ratings_average = log ( (double)
    average_rating / (1 - average_rating) );
}
```

Listing 3: function calculate_average_ratings

In learn_basic_mf, it does not have ratings_average issue, however, during our porting of Recommender to Intel SGX enclave, we find out that it contains improper usage of random generator. And this leads to same initial matrix values for a new model training. We have reported this bug to the Github code owner and he has confirmed the bug. We further investigate will any leakage happens if the bug is intentional planted by service provider. By assuming this, PrivacyScope finds out logics hidden inside this function which leads to sensitive data leakage.

We explain the improper usage of random generator here first. Normal use of srand() function should output a random number like srand(time(0)), however, Recommender use srand(0) to set the seed of the random number generator to 0 every time. This causes the random generator to create the same output everytime. For example, in Listing 4, we uses srand(0) to set the seed of random number generator to 0. And we print out the random number it generates with rand(). Every time this program will print the same number sequence.

```
    srand(0);
    for(int i = 0; i<5; i++)
        printf(" %d ", rand());
    return 0;
```

Listing 4: improper srand usage

Recommender uses generate_random_matrix to create random matrix for initial training state, however, generate_random_matrix sets seed to 0 always and uses box_muller to get random number. The number sequence box_muller generates is always the same because of the same seed is used. We show generate_random_matrix and box_muller in Listing 5and Listing 6 respectively.

```
double** generate_random_matrix(int nrow,int ncol,
    int seed)
{
  ...
  srand(seed);
  ...
        /* normal random variate generator */
         matrix[i][j] = box_muller(0,0.1);
  ...
  return matrix;
}
```

Listing 5: function generate_random_matrix

```
double box_muller(double m, double s)
{
  ...
  x1 = 2.0 * rand()/RAND_MAX - 1.0;
  x2 = 2.0 * rand()/RAND_MAX - 1.0;
  ...
}
```

Listing 6: random generator

Now we know the generate_random_matrix in Recommender always create the same initial state for training, we can continue to see how PrivacyScope find sensitive data leakage. We first set all the initial states set by generate_random_matrix to a manmade concrete value. For example, in Listing 7, lfactors->item_factor_vectors and lfactors->user_factor_vectors are generated by function generate_random_matrix, so we set concrete values for them. After this, we run the program analysis on the learning method.

```
struct learned_factors* init_learned_factors (struct
     model_parameters * params)
{
    ...
  lfactors->item_factor_vectors =
    generate_random_matrix (params->items_number,
    params->dimensionality, params->seed);
  lfactors->user_factor_vectors =
    generate_random_matrix (params->users_number,
    params->dimensionality, params->seed);
    ...
  return lfactors;
}
```

Listing 7: initialization function of learning model

PrivacyScope reports item_factors and user_factors fields of data struct lfactors leak sensitive data. After manual verification of function learn_basic_mf in Listing 8, we confirm the leakage. We find r_iu stores sensitive data and it is passed to e_iu because r_iu_estimated is a concrete value. For the loops, the symbolic execution engine unrolls them to a limit and the limit is set to 4. The sensitive e_iu then is passed to function compute_factors as predicted_error in Listing 10. Sensitive predicted_error is computed with concrete values and service provider selected hyper parameters to set the value of item_factors and user_factors. However, item_factors and user_factors are labeled as output, thus this breaks the rule set by PrivacyScope. In the case where service provider controls the hyper parameters and knows the number sequence of random number generator, he can choose to take user ratings one at a time and inferring the rating by knowing item_factors and user_factors.

```
struct learned_factors* learn_basic_mf(
    learning_algorithm_params_t* learning_params)
{
  struct learned_factors* lfactors =
    init_learned_factors(&learning_params->params);
    ...
  /* every field of data struct params are hyper
    parameters selected by service provider*/
  for (k = 0; k < learning_params->params.
    iteration_number; k++)
  {
    for (r = 0; r < learning_params->params.
    training_set_size; r++)
    {
        /* r_iu stores the sensitive input */
       r_iu = learning_params->tset->ratings->
    entries[r].value;
            ...
        /* r_iu_estimated is concrete value as shown
     in Listing 9*/
       r_iu_estimated = estimate_item_rating(
    item_factors, user_factors, learning_params->
    params.dimensionality);
        /* e_iu is labeled sensitive */
       e_iu = r_iu - r_iu_estimated;
       compute_factors(item_factors, user_factors,
    learning_params->params.lambda, learning_params
    ->params.step, e_iu, learning_params->params.
    dimensionality);
    ...
  return lfactors;
}
```

Listing 8: function learn_basic_mf

```
double estimate_item_rating(double* user_vector,
    double* item_vector, size_t dim)
{
  double sum = 0;
  /* item_factors, user_factors are set as concrete
    values because they are generated by
    generate_random_matrix */
  for (size_t i = 0; i < dim; i++)
    sum += user_vector[i] * item_vector[i];
  return sum;
}
```

Listing 9: function estimate_item_rating

```
void compute_factors( double* item_factors, double*
    user_factors, double lambda, double step, double
     predicted_error, size_t dimensionality)
{
  /* step, lambda, dimensionality are hyper
    parameters selected by service provider */
  for ( size_t i = 0; i < dimensionality; i++)
  {
      /* item_factors and user_factors leak
    sensitive data*/
    item_factors[i] = item_factors[i] + step * (
    predicted_error * user_factors[i] - lambda *
    item_factors[i]);
    user_factors[i] = user_factors[i] + step * (
    predicted_error * item_factors[i] - lambda *
    user_factors[i]);
  }
}
```

Listing 10: function compute_factors

In the normal workflow of Kmeans, it first sets up hyper parameters like maximum iteration number, number of objects, number of centroid points, method to calculate distance. Then, it populates objects into data struct config. After that, Kmeans performs kmeans algorithm on the data struct config and output the learnt centroid points. as in Listing 11.

```
void enclave_kmeans(char *objects, char* result){
  ...
  /* populate objects */
  for (i = 0; i < config->num_objs - 1; i++)
  {
    config->objs[i] = &(objects[i]);
  }
  ...
  /* algorithm converges when the assignments no
    longer change or max_iteration is reached */
  while (1)
  {
    ...
    /* Assignment step: Assign each observation to
    the cluster whose mean has the least squared
    Euclidean distance */
    update_r(config);
    /* Update step: Calculate the new means to be
    the centroids of the observations in the new
    clusters */
    update_means(config);
    ...
  }
  ...
}
}
```

Listing 11: Kmeans code snippet

In this case, the sensitive data we want to protect is the objects. We verify effectiveness of PrivacyScope by inserting assignment statements of objects to result explicitly before the enclave ends at line 19 of Listing 11. We also embed implicit leakage at line 19 of Listing 11 to verify the effectiveness of PrivacyScope. PrivacyScope reports leakage in both cases. What worth mentioning here is that PrivacyScope is built on top of Clang static analyzer's symbolic engine, so PrivacyScope inherits its unsoundness. The symbolic engine unrolls the loop with a default limit of 4. Although this value is configurable, an enclave writer can bypass the detection easily and this is a well-known challenge in finding bugs with symbolic execution.