Implementing iDASH Competition over ObliVM

Xiao Shaun Wang, Chang Liu, Kartik Nayak, Justin Wagner, Yan Huang and Elaine Shi

February 28, 2015

1 Overview

We implemented all the problems using the secure computation framework ObliVM [1] [2]. For each question, we provide 2 or 3 solutions:

1. Task1a: we provide two solutions, one built manually, one built using compiler. Both of them achieve exact result with no error.
2. Task1b: we provide three solutions, two built manually, one built using compiler. The second manually built circuit can achieve a trade off between efficiency and accuracy.
3. Task2a/b: we provide three solutions, one built manually using oblivious merge, one built with compiler using oblivious merge, and one built manually using bloom filter. The one with bloom filter can be tuned for a trade off between efficiency and accuracy.

All the implementations will be open sourced at https://github.com/wangxiao1254/idash_competition after the deadline of the competition.

2 Task1

2.1 Task1a, Computing Minor Allele Frequencies

The problem of computing Minor Allele Frequencies(MAF) can be abstracted as follows:

Suppose Alice and Bob each has a list of alleles $l^a = (e_1^a, ..., e_n^a)$ and $l^b = (e_1^b, ..., e_n^b)$. Let's append l^b to l^a and get $l = l^a || l^b$. It is known in public that in l, there will be at most two types of alleles from (A, T, C, G). We want to compute the frequency of allele in l that appears less frequently.

Our Solution The two parties first aggregate their own input into two numbers: (f_1^a, f_2^a) for Alice and (f_1^b, f_2^b) for Bob, ordered by allele type. Then, in the secure computation, the two parties first aggregate the frequency by

$$(f_1, f_2) = (f_1^a + f_1^a, f_2^b + f_2^b),$$

and then report the smaller number between f_1 and f_2

For this task, each test case only requires 40 AND gates for both manually generated circuits and automatically generated circuits.

[1] http://www.oblivm.com

Code used for Task1a. Here we also include the code used for Task1a written in `ObliVM-lang`.

```
struct Task1aAutomated@m{};
void Task1aAutomated@m.funct(int@m[public 1] alice_data, int@m[public 1] bob_data,
  int@m[public 1] ret, public int@m total_instances, public int32 test_cases) {
  int@m total = total_instances;
  int@m half = total_instances / 2;
  for (public int32 i = 0; i < test_cases; i = i + 1) {
    ret[i] = alice_data[i] + bob_data[i];
    if (ret[i] > half)
      ret[i] = total - ret[i];
  }
}
```

2.2 Task1b, Computing χ square statistics

The problem can be abstracted as follows: The two parties want to compute the following results:

$$n \times \frac{(ad-bc)^2}{rsgk},$$

where $r = a + b, s = c + d, g = a + c, k = b + d, n = r + s$; and a, b, c, d are additively secret shared by two parties.

Our Solution On the high level, we take a direct approach: In secure computation, we first add shares from two parties and get a, b, c, d, r, s, g, k and then convert it into floating point numbers securely, and finally compute the function mentioned above using floating point numbers securely.

For this task, each test case only requires 7763 AND gates achieving maximum absolute error of 1.11×10^{-4} and 14443 AND gates achieving maximum absolute error of 5.6×10^{-8}. In fact, our implementation support arbitrary trade off between precision and speed, however, we only showed two cases for the convenience of testing.

Code used for Task1b. Here we also include the code used for Task1b written in `ObliVM-lang`.

```
struct Task1bAutomated@n{};
float32[public n] Task1bAutomated@n.func(
  float32[public n][public 3] alice_case, float32[public n][public 3] alice_control,
  float32[public n][public 3] bob_case, float32[public n][public 3] bob_control) {
  float32[public n] ret;
  for (public int32 i = 0; i < n; i = i + 1) {
    float32 a = alice_case[i][0] + bob_case[i][0];
    float32 b = alice_case[i][1] + bob_case[i][1];
    float32 c = alice_control[i][0] + bob_control[i][0];
    float32 d = alice_control[i][1] + bob_control[i][1];
    float32 g = a + c, k = b + d;
    float32 tmp = a*d - b*c;
    tmp = tmp*tmp;
    ret[i] = tmp / (g * k);
  }
  return ret;
}
```

3 Task2

3.1 Estimating Size of Union of Two Sets

Before going into how to solve Task2, let us first present two solutions to calculate the size of union of two sets. Suppose Alice and Bob each has a set of elements $s^a = (e_1^a, ..., e_n^a)$ and $s^b = (e_1^b, ..., e_n^b)$. The problem we want to solve is to compute $|s^a \cup s^b|$.

Using oblivious sort and oblivious merge A straightforward way of computing the union is to use oblivious sorting, as detailed in Algorithm 1.

Algorithm 1 Compute size of union

```
1: sort the input array d[] obliviously.
2: cnt = 1
3: for i = 0 : len(d) − 1 do
4:     if d[i]! = d[i + 1] then
5:         cnt = cnt + 1
6:     end if
7: end for
8: return cnt
```

For n elements each of bitlength D, this approach requires a circuit of size $O(Dn \log^2 n)$ using bitonic sorting network [1], and can be further reduced to $O(Dn \log n)$ if two parties sort their data locally and only perform an bitonic merge using secure computation. In the submission, we take the second approach to achieve a better performance.

Code used for oblivious merge. Here we also include the code used of this algorithm written in `ObliVM-lang`.

```
struct Task2Automated@m@n{};
int@n Task2Automated@m@n.funct(int@m[public 1] key, public int32 length) {
  this.obliviousMerge(key, 0, length);
  int@n ret = 1;
  for (public int32 i = 1; i < length; i = i + 1) {
    if (key[i-1] != key[i])
      ret = ret + 1;
  }
  return ret;
}
void Task2Automated@m@n.obliviousMerge(int@m[public 1] key, public int32 lo, public int32 l) {
  if (l > 1) {
    public int32 k = 1;
    while (k < l) k = k << 1;
      k = k >> 1;
      for (public int32 i = lo; i < lo + l - k; i = i + 1)
        this.compare(key, i, i + k);
      this.obliviousMerge(key, lo, k);
      this.obliviousMerge(key, lo + k, l - k);
  }
}
void Task2Automated@m@n.compare(int@m[public 1] key, public int32 i, public int32 j) {
  int@m tmp = key[j];
  int@m tmp2 = key[i];
  if ( key[i] < key[j] )
    tmp = key[i];
  tmp = tmp ∧ key[i];
  key[i] = tmp ∧ key[j];
  key[j] = tmp ∧ tmp2;
}
```

Using bloom filter It is known that bloom filters can be used to check the existence of an element in a set. However, bloom filters can also be used to estimate the size of a set. Let X be the number of bits set as one in a bloom filter, m be the total number of bits used in the bloom filter and k be the number of hash functions used. The size of a bloom filter can be estimated as

$$-\frac{m \ln(1 - \frac{X}{m})}{k}.$$

So, if we would like to compute the size of union of two sets, each party can first build their own bloom filter locally using the same set of hash functions and then, in secure computation, the two parties first union the bloom filter by bit-wise or, and then count number of ones appeared in the new bit array, which is X mentioned above. Note that after getting X, the remaining part of the computation can be done in cleartext. Using bloom filter, we can compute the size of union using $O(n)$ number of gates, regardless of the bitlength.

3.2 Task2a, Hamming Distance

The website simplify the definition of hamming distance, which can be computed easily given the functionality mentioned above. Suppose a record contains an associated position and the value in a form of (pos, val). Each party holds a set of records like this, namely S^a and S^b. The hamming distance is equivalent to $|S^a \cup S^b| - |S^a \cap S^b|$, that is the sum of number of elements not shared by two parties. Note that $|S^a \cup S^b| - |S^a \cap S^b| = 2 \times |S^a \cup S^b| - |S^a| - |S^b|$. So we can use the aforementioned algorithms to compute hamming distance.

Note that in order to do oblivious merge, each record has to be of the same bitlength. Instead of padding every record to the maximum possible length, we hash each record to a fixed length bit string.

3.3 Task2b, Edit Distance

Edit distance can also be reduced to calculating size of certain sets. First, we compute `d1` that is defined similarly to the definition on website as follows:

```
d1 = 0;
for every pos in VCF files:
if there are two records x, y at pos,
   d1 += max(D(x), D(y))
else if there is only one record at pos,
   d1+=D(x)
```

In order to compute `d1`, we construct two new sets: for every record (pos, val), each party insert $(pos, i), i \in [1, len(val)]$ to a new set and get set S_1^a, S_1^b for Alice and Bob. Then it can be seen that `d1` $= |S_1^a \cup S_1^b|$.

Now, let's define `d2` as follows:

```
d2 = 0;
for every pos in VCF files:
if there are two records x, y at pos and they are same,
   d2 += D(x)
```

In order to compute `d2`, the two parties construct another two new sets: For every record (pos, val), each party insert $(pos, val, i), i \in [1, len(val)]$ to the set and get set S_2^a, S_2^b. Then it can be seen that `d2` $= |S_2^a \cap S_2^b|$.

After we define `d2` and `d1`, it is clear and `d = d1-d2`.

References

[1] K. E. Batcher. Sorting Networks and Their Applications. AFIPS '68 (Spring), 1968.

[2] Xiao Shaun Wang, Chang Liu, Karthik Nayak, Yan Huang, and Elaine Shi. Oblivm: A generic, customizable, and reusable secure computation architecture. In *IEEE Symposium on Security and Privacy*. IEEE, 2015.