iDASH: Secure Genome Analysis 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 multiple solutions, with manually built circuits and automatically built circuits:

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 four solutions, one built manually using oblivious merge, one built with compiler using oblivious merge, one built manually using bloom filter, and one built using compiler. 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 Task 1

Task 1a: Computing Minor Allele Frequencies

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

Suppose Alice and Bob have list of alleles $l^a = (e_1^a, ..., e_n^a)$ and $l^b = (e_1^b, ..., e_n^b)$ respectively. Let's append l^b to l^a and get $l = l^a || l^b$. It is publicly known that l contains 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 inputs 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 requires only 40 AND gates for both manually generated circuits and automatically generated circuits.

The code used for this task is shown in Figure 1.

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

```
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];
  }
}
```

Figure 1: Code for Task 1a written in ObliVM-lang

Task 1b: Computing χ square statistics

For this problem, the two parties want to compute the following:

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

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 We take a direct approach: 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. We then compute the function mentioned above. All of this computation is performed using secure computation.

Our implementation for this task supports an arbitrary trade off between precision and speed. We mention two specific cases here. For each test case, an implementation that requires 7763 AND gates achieves a maximum absolute error of 1.11×10^{-4} and an implementation with 14443 AND gates achieves a maximum absolute error of 5.6×10^{-8}.

The code used for this task is shown in Figure 2.

```
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;
}
```

Figure 2: Code for Task 1b written in ObliVM-lang

3 Task 2

3.1 Estimating Size of Union of Two Sets

Before describing our solution for Task 2, we first present the solution to estimate the size of union of two sets. Suppose Alice and Bob have sets of elements $s^a = (e_1^a, ..., e_n^a)$ and $s^b = (e_1^b, ..., e_n^b)$ respectively. We want to find $|s^a \cup s^b|$. Here, we introduce two algorithms:

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

Algorithm 1 Compute size of union

```
Sort the input array $d[]$ obliviously.
$cnt = 1$
for $i = 0 : len(d) - 1$ do
    if $d[i] \neq d[i+1]$ then
        $cnt = cnt + 1$
    end if
end for
return $cnt$
```

For n elements, each with size D bits, 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 perform a bitonic merge using secure computation. In the submission, we take the second approach to achieve a better performance.

The code for this approach is presented in Figure 3.

```
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;
}
```

Figure 3: Code for oblivious merge written in ObliVM

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 capacity of a set. Let X be the number of bits set, m be the total number of bits used in the bloom filter and k be the number of hash functions used. Number of elements in the bloom filter can be estimated as

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

So, in order to compute the union of two sets, each party first builds their own bloom filter locally using the same set of hash functions. Then, in secure computation, the two parties union the bloom filter using a bitwise OR, and count number of ones in the new bit array (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 $O(n\lambda)$ number of gates, regardless of the bitlength.

The code for this approach is presented in Figure 4.

```
struct Pair<T1, T2> {
  T1 left;
  T2 right;
};
struct bit {
  int1 v;
};
struct Int@n {
  int@n v;
};
struct BF_circuit{};
Pair<bit, Int@n> BF_circuit.add@n(int@n x, int@n y) {
  bit cin;
  Int@n ret;
  bit t1, t2;
  for (public int32 i=0; i<n; i = i+1) {
    t1.v = x$i$ ∧ cin.v;
    t2.v = y$i$ ∧ cin.v;
    ret.v$i$ = x$i$ ∧ t2.v;
    t1.v = t1.v & t2.v;
    cin.v = cin.v ∧ t1.v;
  }
  return Pair{bit, Int@n}(cin, ret);
}
int@log(n + 1) BF_circuit.countOnes@n(int@n x) {
  if (n==1) return x;
  int@log(n − n/2 + 1) first = this.countOnes@(n/2)(x$0~n/2$);
  int@log(n − n/2 + 1) second = this.countOnes@(n − n/2)(x$n/2~n$);
  Pair<bit, Int@log(n − n/2)> ret = this.add@log(n − n/2 + 1)(first, second);
  int@log(n + 1) r = ret.right.v;
  r$log(n+1)-1$ = ret.left.v;
  return r;
}
int@log(n + 1) BF_circuit.merge@n(int@n x, int@n y) {
  int@n tmp;
  for (public int32 i = 0; i < n; i = i +1 ) {
    tmp$i$ = x$i$ | y$i$;
  }
  return this.countOnes@n(tmp);
}
```

Figure 4: Code for bloom filter approach written in `ObliVM`

3.2 Task 2a: Hamming Distance

Hamming distance, as defined in the question, can be computed easily given the functionality mentioned above. Let's define a record as containing the 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.