iDASH Secure Genome Analysis Competition Using ObliVM

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

University of Maryland, College Park
{wangxiao,liuchang,kartik,elaine}@cs.umd.edu, yh33@indiana.edu

This is a short note in supplement to our ObliVM paper [4].

Overview

For problems in iDASH Secure Genome Analysis Competition [1], we implemented them using the secure computation framework ObliVM [4, 2]. For each question, we provide multiple solutions, with manually built circuits and automatically built circuits:

1. Task 1a: we provide two solutions, one built manually, one built using compiler. Both of them achieve exact result with no error.
2. Task 1b: 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. Tasks 2a and 2b: 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.

Detailed instructions on how to run the program can be found in `README.md`. All the implementations will be open sourced at `https://github.com/wangxiao1254/idash_competition` after the deadline of the competition.

Manual vs. compiler-generated code. For optimal speed, please run our manually optimized code. Our compiler-generated code results in 1% to 2% larger circuit sizes than the manually optimized code. The difference in actual running time is slightly larger because the compiler generates longer instruction sequences (in Java) to execute the circuits than our manually optimized code. Further compiler backend optimizations can be performed to compress our target instruction sequence – this will be part of our future work.

Nevertheless, we include our compiler-generated implementations to demonstrate the ease-of-use of ObliVM's source language.

1 Task 1: Secure Distributed GWAS

For all tasks, we will briefly repeat the problem statement. For a more detailed description of their application contexts, we refer the readers to the competition website [1].

Task 1a: Computing Minor Allele Frequencies

Problem statement.
Input for Alice: List of alleles $l^A = (e_1^A, ..., e_n^A)$ where each e_i^A is from $\{A, T, C, G\}$.
Input for Bob: List of alleles $l^B = (e_1^B, ..., e_n^B)$ where each e_i^B is from $\{A, T, C, G\}$.

The problem is to compute the Minor Allele Frequencies(MAF) over the combined input $l = l^A || l^B$. The definition of MAF is as follows: Let f_1 and f_2 be the frequency of two types of alleles in the combined input l (it is guaranteed that only two types of alleles will appear). MAF is defined to be as $min\{f_1, f_2\}$.

Our solution. The two parties first compute the frequency of two types of alleles of their own lists and get (f_1^A, f_2^A) for Alice and (f_1^B, f_2^B) for Bob, ordered by allele type. Then, over a secure computation protocol, the two parties first sum up 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.

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

```
1  struct Task1aAutomated@m@n{};
2  void Task1aAutomated@m@n.funct(int@m[public n] alice_data, int@m[public n] bob_data,
3    int@m[public n] ret, public int@m total_instances) {
4    int@m total = total_instances;
5    int@m half = total_instances / 2;
6    for (public int32 i = 0; i < n; i = i + 1) {
7      ret[i] = alice_data[i] + bob_data[i];
8      if (ret[i] > half)
9        ret[i] = total - ret[i];
10   }
11 }
```

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

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

Task 1b: Computing χ^2 statistics

Problem statement.
Input for Alice: two lists of alleles $l_{case}^A = (e_1^A, ..., e_n^A)$, $l_{control}^A = (e_1'^A, ..., e_n'^A)$ where each $e_i^A, e_i'^A$ is from $\{A, T, C, G\}$.

Input for Bob: two lists of alleles $l^B_{case} = (e^B_1, ..., e^B_n)$, $l^B_{control} = (e'^B_1, ..., e'^B_n)$ where each e^B_i, e'^B_i is from $\{A, T, C, G\}$.

First let us define a, b to be the frequency of two types of alleles in $l_{case} = l^A_{case} || l^B_{case}$ and let c, d be the frequency of two types of alleles in $l_{control} = l^A_{control} || l^B_{control}$. (it is also guaranteed that only two types of alleles will appear). The problem is to compute the χ^2 statistics over l_{case} and $l_{control}$, defined as:

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

where $r = a + b, s = c + d, g = a + c, k = b + d, n = r + s$;

Our solution. Each party first locally computes the frequency of alleles on their own list: a^A, b^A, c^A, d^A for Alice and a^B, b^B, c^B, d^B for Bob. a^A, b^A are the frequency of two types of alleles in l^A_{case} and c^A, d^A are the frequency of two types of alleles in $l^A_{control}$. a^B, b^B, c^B, d^B is defined similarly. In the secure computation we first compute $a = a^A + a^B, b = b^A + b^B, c = c^A + c^B, d = d^A + d^B$, then we evaluate the equation mentioned above directly using floating point numbers.

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

```
1  struct Task1bAutomated@n{};
2  float32[public n] Task1bAutomated@n.func(
3    float32[public n][public 3] alice_case, float32[public n][public 3] alice_control,
4    float32[public n][public 3] bob_case, float32[public n][public 3] bob_control) {
5    float32[public n] ret;
6    for (public int32 i = 0; i < n; i = i + 1) {
7      float32 a = alice_case[i][0] + bob_case[i][0];
8      float32 b = alice_case[i][1] + bob_case[i][1];
9      float32 c = alice_control[i][0] + bob_control[i][0];
10     float32 d = alice_control[i][1] + bob_control[i][1];
11     float32 g = a + c, k = b + d;
12     float32 tmp = a*d - b*c;
13     tmp = tmp*tmp;
14     ret[i] = tmp / (g * k);
15   }
16   return ret;
17 }
```

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

Results. 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}.

Algorithm 1 Compute the size of union(d[]: a list of elements)

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

2 Task 2: Secure comparison between genomic data

2.1 A Building Block: Estimating Set Union Cardinality

Before describing our solution for Task 2, we first present a building block 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 merge. A strawman approach of computing cardinality of the union is to use oblivious sorting, as detailed in Algorithm 1.

Suppose there are totally n elements each of size D bits. This strawman approach involves an oblivious sorting on the input data, which requires a circuit of size $O(Dn \log^2 n)$, using bitonic sorting network [3]. In our implementation, we utilize the local computation: the two parties first sort their data locally, and then they perform an oblivious merge in secure computation. This improved approach only requires a circuit of size $O(Dn \log n)$. The `ObliVM-lang` code for this approach is presented in Figure 3 and Figure 4.

```
1  struct Task2Automated@m@n{};
2  int@n Task2Automated@m@n.funct(int@m[public n] key) {
3    this.obliviousMerge(key, 0, n);
4    int32 ret = 1;
5    for (public int32 i = 1; i < n; i = i + 1) {
6      if (key[i-1] != key[i])
7        ret = ret + 1;
8    }
9    return ret;
10 }
11 void Task2Automated@m@n.obliviousMerge(int@m[public n] key, public int32 lo, public int32 l) {
12   if (l > 1) {
13     public int32 k = 1;
14     while (k < l) k = k << 1;
```

Figure 3: Code for oblivious merge written in `ObliVM`

```
15        k = k >> 1;
16      for (public int32 i = lo; i < lo + l - k; i = i + 1)
17        this.compare(key, i, i + k);
18      this.obliviousMerge(key, lo, k);
19      this.obliviousMerge(key, lo + k, l - k);
20    }
21  }
22  void Task2Automated@m@n.compare(int@m[public n] key, public int32 i, public int32 j) {
23    int@m tmp = key[j];
24    int@m tmp2 = key[i];
25    if ( key[i] < key[j] )
26      tmp = key[i];
27    tmp = tmp ∧ key[i];
28    key[i] = tmp ∧ key[j];
29    key[j] = tmp ∧ tmp2;
30  }
```

Figure 4: Code for oblivious merge written in `ObliVM`, continued

N	M	relative error (95% confidence)	relative error (99% confidence)
100 to 10000	15000	0.7% to 1.3%	1.5% to 1.7%
1000 to 30000	30000	0.8% to 0.9%	1.1% 1.2%
173851	N	0.40%	0.50%
173851	1.5N	0.31%	0.40%

Table 1: **Parameters and accuracy for bloom filter cardinality estimation.** N is the (maximum) number of elements, M is the array size. We choose $k = 1$ number of hashes in this table. A relative error of ϵ with $y\%$ confidence means that on a random run giving an estimate $\hat{N}$ of N, with probability $y\%$, it holds that $(\hat{N} - N)/N < \epsilon$.

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

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

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 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.

The following table gives typical parameter choices (and for values of N given in the example dataset) and their accuracy guarantees:

The code for this approach is presented in Figure 5.

```
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 5: Code for bloom filter approach written in `ObliVM`

Failure probability	#bits for hashing (N = 8754)	#bits for hashing (N = 173851)
2^{-10}	35	44
2^{-20}	45	54
2^{-30}	55	64
2^{-40}	65	74
2^{-50}	75	84
2^{-60}	85	94
2^{-70}	95	104
2^{-80}	105	114

Table 2: Correctness failure probabilities (i.e., hash collision probabilities) for oblivious merge. Note that even when hash collision happens, we obtain an approximate answer.

2.2 Task 2a, Hamming Distance

Problem statement. In this problem, Alice and Bob each holds a list of records in VCF files, where each record is of the format $(ref, svtype, alt)$. We want to compute the hamming distance defined on the competition website [1]:

```
d = 0;
for all records in the VCF files, which have SVTYPE = SNP or SUB: if given
a chrom and pos, there is only one record in one of the VCF file (e.g., x !=
null), then we set the other record as NULL (e.g., y == null)
if (x == null) || (y == null) || (x.ref == y.ref && x.alt != y.alt)
   d += 1;
end for
```

Our solution. Each party constructs a set containing all the records from their own input: S^A for Alice and S^B for Bob. Hamming distance defined above 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. In the code, we hash it to 64-bit numbers, which gives a failure probability of 2^{-31}. The collision failure probabilities and the corresponding number of bits in the hash function is shown in Table 2.

2.3 Task 2b, Edit Distance

Problem statement. In this problem, Alice and Bob each hold a list of records in VCF files, where each record is of the format $(ref, svtype, alt)$. We want to compute the edit distance defined on the competition website [1]:

```
d = 0;
for all records x,y at the same locus in the VCF files:
1.  if x == y, continue;
2.  if x != y, d += max(D(x), D(y))
end for
where D(x):
if x.svtype == snp, D(x) = 1
if x.svtype == sub, D(x) = len(x)
if x.svtype == ins, D(x) = len(x)
if x.svtype == del, D(x) = len(x)
```

Our solution. We first compute `d1` defined as follows:

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

In order to compute `d1`, each party constructs a new set as follows: for every record $(ref, svtype, alt)$, each party inserts $(ref, i), i \in [1, len(alt)]$ to a new set and get set S_1^A, S_1^B for Alice and Bob. Then we compute `d1` $= |S_1^A \cup S_1^B|$.

Then we compute `d2` defined as follows:

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

In order to compute `d1`, each party constructs a new set as follows: for every record $(ref, svtype, alt)$, each party inserts $(ref, svtype, alt, i), i \in [1, len(alt)]$ to a new set and get set S_2^A, S_2^B for Alice and Bob. Then we compute `d2` $= |S_2^A \cup S_2^B|$.

The final result can be computed by `d = d2 - d1`.

Acknowledgments

This research is partially supported by an NSF grant CNS-1314857, a Sloan Fellowship, Google Research Awards, and a subcontract from the DARPA PROCEED program.

Supplemental Information

A ObliVM Overview

ObliVM is a programming framework for secure computation, offering the following features:

- **Ease-of-use.** Non-specialist programmers can easily write programs in our source language ObliVM-lang, a familiar imperative-style language.
- **Efficiency.** ObliVM compiles these programs into concise circuit representations suitable for secure computation. Much as MapReduce is a programming paradigm for parallel computation, ObliVM offers user- and compiler- friendly programming abstractions for secure computation.
- **Formal security.** ObliVM offers a security type system to ensure that programs supplied by nonspecialist developers will be executed securely without leaking information. At a high level, the security type system guarantees that a program's execution traces is oblivious to secret inputs.

Capabilities of ObliVM. ObliVM supports richer and much more sophisticated applications than the challenges given in this competition. We have developed a variety of demo applications in machine learning, streaming algorithms, graph algorithms, common data structures, and common utilities. Using ObliVM, we have demonstrated queries on GB databases where we can compute the answers in a reasonable amount of time.

For more information about ObliVM and additional resources, please refer to our paper

`http://www.cs.umd.edu/~elaine/docs/oblivm.pdf`

and our website

`http://oblivm.com`

B Thank You and Suggestions to the Organizers

We think this competition is such a fantastic idea. It definitely helps to bridge communities, and helps state-of-the-art secure computation technology find its way in high-impact application domains. We are grateful to the organizers for organizing this competition, and for detailed discussions and feedback throughout.

We really hope that more competitions like this will be held in the future. If a competition like this is to be repeated, we urge that the organizers increase the difficulty of the challenges — e.g., consider much more sophisticated tasks and bigger data sizes. We feel that the simple and small-scale nature of the problems in this competition is insufficient to fully demonstrate the true power of state-of-the-art secure computation frameworks such as ObliVM.

References

[1] `http://humangenomeprivacy.org/2015`.

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

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

[4] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, 2015.