CSE 715 Fall 2025
Design and Implementation of Application for Computing with Private Data

Marina Blanton

Department of Computer Science and Engineering
University at Buffalo

Lecture 3: Building Larger Secure Computation Protocols

Computing on Private Values

We previously discussed mechanisms for performing elementary operations on private data

- (integer) addition $c = a + b$
- (integer) multiplication $c = a \cdot b$

The next question is where we take this

Let $[x]$ represent a secret-shared value x

Dot Product

Consider an operation that involves multiple additions and multiplications – a dot product

- on input vectors $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$, compute

$$c = \sum_{i=1}^{k} a_i \cdot b_i$$

- there are k multiplications and $k-1$ additions, how do we proceed?

Dot Product

Consider the following pseudo-code

1: $[c] = 0$;
2: for $i = 1$ to k do
3: $\quad [d] = [a_i] \cdot [b_i]$;
4: $\quad [c] = [c] + [d]$;
5: end for

Dot Product

Consider the following pseudo-code

1: $[c] = 0$;
2: for $i = 1$ to k do
3: $\quad [d] = [a_i] \cdot [b_i]$;
4: $\quad [c] = [c] + [d]$;
5: end for

What is the cost?

- important considerations are the amount of communication and the number of communication rounds

Dot Product

Now consider this version

1: for $i = 1$ to k in parallel do
2: $\quad [d_i] = [a_i] \cdot [b_i]$;
3: end for
4: $[c] = 0$;
5: for $i = 1$ to k do
6: $\quad [c] = [c] + [d_i]$;
7: end for

Dot Product

Now consider this version

1: for $i = 1$ to k in parallel do
2: $\quad [d_i] = [a_i] \cdot [b_i]$;
3: end for
4: $[c] = 0$;
5: for $i = 1$ to k do
6: $\quad [c] = [c] + [d_i]$;
7: end for

What is the cost?

Parallelization

Compilers support the ability to combine multiple operations into batched execution

With PICCO, there is a special loop construct

Original:

```
for (i = 0; i < k; i++) {
   d[i] = a[i] * b[i];
}
```

Parallelization

Compilers support the ability to combine multiple operations into batched execution

With PICCO, there is a special loop construct

Original:

```
for (i = 0; i < k; i++) {
   d[i] = a[i] * b[i];
}
```

Batched:

```
for (i = 0; i < k; i++) [
   d[i] = a[i] * b[i];
]
```

Parallelization

Furthermore, there is a way to specify batched execution at the level of array operations For example:

```
C = A * B;
D = A / B;
E = A > B;
```

- this executes the operation element-wise
- the sizes of the arrays in an operation must be the same

Dot Product

Getting back to the dot product, there is an even better way to reduce the cost

Communication cost of a dot product is the same as that of a single multiplication

- this works only using honest majority techniques
- recall that multiplication proceeds by distributed product computation and re-sharing
- instead of computing one product, we compute a sum of products and then reshare
- this works with both replicated and Shamir secret sharing
- example using RSS

Dot Product

The ability to call a dot product protocol from a user program has a profound impact on performance

PICCO provides a special operation

```
C = A @ B
```

for arrays `A` and `B` of the same size

Other Operations

Consider a more complex operation: matrix multiplication

$$C = A \times B$$

- A is of size $n_1 \times n_2$
- B is of size $n_2 \times n_3$
- C is of size $n_1 \times n_3$

What is the computation and how can it be optimized?

Comparisons

Non-linear operations such as less-than comparisons and equality tests are non-trivial to realize

- suppose that we have access to the individual bits
- the cost is linear in the bitlength k of the numbers being compared
- the number of rounds is also of crucial importance
  - a linear number of rounds for each comparison is slow
  - we want a constant or at least logarithmic number of rounds
  - illustration

Comparisons

To be able to work with bits, the idea is to generate random bits and work with protected bits

- to compare $[a]$ and $[b]$, we first compute the difference $[d] = [a] - [b]$
- testing $a < b$ reduces to determining whether d is negative
- testing $a = b$ reduces to determining whether d is 0
- generate a random element $[r]$ together with its bit decomposition $[r_{k-1}], \ldots, [r_0]$
- compute and open $[c] = [d] + [r]$
- perform the computation using public c and private bits $[r_{k-1}], \ldots, [r_0]$

Comparisons

Less-than-zero comparison outputs the complement of the most significant bit of a:

$[b] \leftarrow \mathsf{LTZ}([a], k)$

1. for $i = 0, \ldots, k-2$ do $[r_i] \leftarrow \mathsf{RandBit}(p)$;
2. $[r] \leftarrow \sum_{i=0}^{k-2} 2^i [r_i]$;
3. $[r'] \leftarrow \mathsf{RandInt}(\kappa + 1)$;
4. $c \leftarrow \mathsf{Open}(2^{k-1} + [a] + 2^{k-1}[r'] + [r])$;
5. $c' \leftarrow c \bmod 2^{k-1}$;
6. $[u] \leftarrow \mathsf{BitLT}(c', ([r_{k-2}], \ldots, [r_0]))$;
7. $[a'] = [c'] - [r] + 2^{k-1}[u]$;
8. $[b] \leftarrow ([a'] - [a])(2^{-(k-1)} \bmod p)$;
9. return $[b]$;

Integer Division

Fractional numbers cannot represented

Division is realized using one of iterative algorithms

- start with an approximation (a few bits of precision)
- initial approximation typically requires normalization
- compute a few more bits of the quotient in each iteration
- truncate intermediate results to keep bitlength manageable

Working with Real Numbers

Floating-point operations don't fit well the framework

- their computation is more expensive and they were introduced substantially later
- a floating-point number is represented as a number of integers (denoting the mantissa, exponent, and sign bit)
- alignment is needed as part of floating-point protocols

Working with Real Numbers

Floating-point operations don't fit well the framework

- their computation is more expensive and they were introduced substantially later
- a floating-point number is represented as a number of integers (denoting the mantissa, exponent, and sign bit)
- alignment is needed as part of floating-point protocols

Fixed-point representation is an alternative of faster performance

- addition is the same as with integers and is very cheap
- multiplication involves multiplying two integers followed by a truncation

Variable Bitlength

Recall that unlike CPU instructions, the cost of secure computation protocols is not fixed

Because the cost can be proportional to the bitlength, one optimization is to permit variables of custom bitlength

- a programmer can choose a custom bitlength according to the range of values being stored
- the cost will be proportional to the declared bitlength

Variable Bitlength

Recall that unlike CPU instructions, the cost of secure computation protocols is not fixed

Because the cost can be proportional to the bitlength, one optimization is to permit variables of custom bitlength

- a programmer can choose a custom bitlength according to the range of values being stored
- the cost will be proportional to the declared bitlength

In PICCO, this is declared as

```
int<16> a;
float<32,10> b;
```

Variable Qualifiers

When working with a mix of public and private values, we can distinguish between the types by annotating their types

```
public int a;
private float b;
private int<48> c;
int d;
```

Variable Qualifiers

When working with a mix of public and private values, we can distinguish between the types by annotating their types

```
public int a;
private float b;
private int<48> c;
int d;
```

Omission of the quantifier makes the type default to private

- remember the principle of fail-safe defaults?

Performance of Operations

The cost of individual operations is fundamentally different from that on conventional CPUs

- operations on conventional processors are 1 or a couple of CPU cycles
- the cost of secure protocols for the same operations differs by orders of magnitude

integer addition/subtraction $<$ integer multiplication $<$ comparison $<$ division $<$ floating-point arithmetic

Working with Private Data

In addition to being able to execute operations on private data, we want to be able to use them in programming language constructs

Are there any differences for working with conditional statements, loops, etc.?

Can we implement algorithms in the same way?

Conditional Statements

Consider conditional statements with private conditions, e.g.,

```
if (a < 0)
  b = c;
else
  b = b + c;
```

with private a

We would like to understand constraints on variable types and execution

Data Flow

Consider a simpler example

```
private int a;
public int b;

...
b = a;
```

Data Flow

Consider a simpler example

```
private int a;
public int b;

...
b = a;
```

What is the issue with this piece of code?

Conditional Statements

Returning to the previous example

```
if (a < 0)
  b = c;
else
  b = b + c;
```

of what type can variables `b` and `c` be?

Conditional Statements

Returning to the previous example

```
if (a < 0)
  b = c;
else
  b = b + c;
```

of what type can variables `b` and `c` be?

More generally, we require that conditional statements with private conditions have no public side effects

Conditional Statements

Now consider the fact that different variables are modified within the body of the conditional statement

```
if (a < 0)
  b = d;
else
  d = b;
```

Conditional Statements

Now consider the fact that different variables are modified within the body of the conditional statement

```
if (a < 0)
  b = d;
else
  d = b;
```

We have to ensure that both are modified on every run

Conditional Statements

The translated computation becomes

1: $[c] = ([a] \overset{?}{<} 0)$;
2: $[b_{\mathsf{temp}}] = [d]$;
3: $[d_{\mathsf{temp}}] = [b]$;
4: $[b] = [c] \cdot [b_{\mathsf{temp}}] + (1 - [c]) \cdot [b]$;
5: $[d] = [c] \cdot [d] + (1 - [c]) \cdot [d_{\mathsf{temp}}]$;

Data-Oblivious Execution

This leads us to the notion of data-oblivious execution

When working with private data, we must ensure that

- the sequence of instructions being executed is data independent
- the sequence of accessed memory locations is data independent

This means we always execute both branches of conditional statements but apply the result of one

Data-Oblivious Execution

Another illustrative example is accessing an array element at a private location

```
private int a;
private int A[100];
...
b = A[a];
```

How do we implement this operation if we don't know the value of a?

Data-Oblivious Execution

Another illustrative example is accessing an array element at a private location

```
private int a;
private int A[100];
...
b = A[a];
```

How do we implement this operation if we don't know the value of `a`?

The most common version is to touch all possible locations while extracting the relevant element

Similarly, writing into `A[a]` updates all elements of `A`

Loops

The next question is how the presence of private variables impacts loops

Consider the example

```
public int i, k;
private int a;
...
for (i = 0; i < k; i++)
  A[i] = A[i]*A[i];

for (i = 0; i < a; i++)
  A[i] = A[i]*A[i];
```

How do we execute this code with private A?

Loops

To be able to execute loops, the terminating condition must be known at runtime

Loops

To be able to execute loops, the terminating condition must be known at runtime

We have the following options

- public condition

Loops

To be able to execute loops, the terminating condition must be known at runtime

We have the following options

- public condition
- evaluate the condition privately and open the result
  - a single bit is opened to determine whether to continue

Loops

To be able to execute loops, the terminating condition must be known at runtime

We have the following options

- public condition
- evaluate the condition privately and open the result
  - a single bit is opened to determine whether to continue
- use a public upper bound when the number of iterations is not known
  - once the private condition is met, the iterations will have no effect

Parallel Loops

Similar to our earlier discussion, the cost of some loops can be reduced

Consider the following computation

1: $[b_0] = [a_0]$;
2: for $i = 1$ to $k - 1$ do
3: $\quad [b_i] = [a_i] \cdot [a_{i-1}]$;
4: end for

Parallel Loops

Similar to our earlier discussion, the cost of some loops can be reduced

Consider the following computation

1: $[b_0] = [a_0]$;
2: for $i = 1$ to $k - 1$ do
3: $\quad [b_i] = [a_i] \cdot [a_{i-1}]$;
4: end for

Since all iterations are independent of each other, we can write

1: $[b_0] = [a_0]$;
2: for $i = 1$ to $k - 1$ in parallel do
3: $\quad [b_i] = [a_i] \cdot [a_{i-1}]$;
4: end for

Parallel Loops

In PICCO, parallelizable loops are denoted with `[ ]`

```
private int a[k], b[k];
public int i;
...
b[0] = a[0];
for (i = 0; i < k; k++) [
  b[i] = a[i] * a[i-1]
]
```

Parallel Loops

In PICCO, parallelizable loops are denoted with `[ ]`

```
private int a[k], b[k];
public int i;
...
b[0] = a[0];
for (i = 0; i < k; k++) [
  b[i] = a[i] * a[i-1]
]
```

All operations are executed in a batch

- care must be taken to store all results in distinct locations

Data-Oblivious Algorithms

The requirement for computation to be data-oblivious impacts complexity of algorithms

- it is not sufficient to replace operations with the corresponding secure protocols
- the structure of the algorithm can change as well

Examples of algorithms difficult to convert to data-oblivious variants

- binary search
- graph algorithms

Data-Oblivious Sorting

Another important and commonly used functionality is sorting

Almost all sorting algorithms are non-oblivious

A notable example is a bitonic sorting network

- it takes $O(n \log^2 n)$ time to sort n items
- an important building block is an oblivious swap operation

```
if (a1 > a2) {
    temp = a1;
    a1 = a2;
    a2 = temp;
}
```

Data-Oblivious Sorting

Bitonic sorting network

- bitonic sorting recursively splits the set into two halves
- sort each half and then merge using a predefined sequence of swaps

Data-Oblivious Sorting

Bitonic sorting network

- bitonic sorting recursively splits the set into two halves
- sort each half and then merge using a predefined sequence of swaps

There are solutions to turn a non-oblivious sort algorithm to an oblivious variant

- randomly shuffle the input set
- open the result of each comparison and move items accordingly

Working with Sorted Data

An important question is how we use sorted data

- the use of sorted data still have to be oblivious

One important application of sorting is set operations

- concatenate two input sets and sort the concatenated set
  - identical elements are guaranteed to be next to each other
- any set operation can be performed by doing a linear scan through the resulting dataset
  - e.g., erase each duplicate element to realize set union
  - e.g., erase each duplicate element and elements without duplicates to realize set intersection

Working with Sorted Data

The use of sorted data may not lead to performance advantages when working with databases

Operations to consider

- retrieving an element
- updating an element

Data-Oblivious Graph Algorithms

When working with graphs, we normally want to protect its structure

- a graph G is represented by a set of vertices V and edges E
- suppose we know/disclose $n = |V|$ and $m = |E|$ or their upper bounds
- there are different graph representations to consider
  - adjacency matrix
  - adjacency list
  - customized representations for graphs of certain structure
- adjacency matrix works well for dense graphs
- adjacency list is preferred for sparse graphs

Data-Oblivious Graph Algorithms

Consider sample algorithms using adjacency matrix and adjacency list representations

- breadth first search
- algorithms from projects

Summary

We build secure protocols from elementary building blocks such as addition/subtraction and multiplication

Additional tools include generating a random element and reconstructing a secret-shared value

This permits building secure protocols for all operations

- it is important to know that the costs are fundamentally different from conventional arithmetic

Computing on private data requires structural changes to the program to make execution data-oblivious

Optimizations can easily reduce program's runtime by orders of magnitude