CSE 410 Fall 2025
Privacy-Enhancing Technologies

Marina Blanton

Department of Computer Science and Engineering
University at Buffalo

Lecture 11: Protecting Data during Computation III
Building Larger Protocols

Computing on Private Values

So far we have 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 $[r_{k-1}], \ldots, [r_0]$

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

Summary

We build secure protocols starting from elementary building blocks:

- addition/subtraction
- multiplication

Additional tools include:

- generated a random element
- reconstructing a secret-shared value

This permits building secure protocols for all operations

It is important to keep in mind that the costs are fundamentally different from conventional arithmetic