CSE 410 Fall 2025
Privacy-Enhancing Technologies

Marina Blanton

Department of Computer Science and Engineering
University at Buffalo

Lecture 12: Protecting Data during Computation IV
Data-Oblivious Execution

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 = d;
else
  d = b;
```

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 = d;
else
  d = b;
```

of what type do variables `b` and `c` have to be?

Conditional Statements

Returning to the previous example

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

of what type do variables `b` and `c` have to 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

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

Parallel Loops

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

Consider the following computation

```
[b_0] = [a_0];
for i = 1 to k − 1 do
    [b_i] = [a_i] · [a_{i−1}];
end for
```

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

```
[b_0] = [a_0];
for i = 1 to k − 1 in parallel do
    [b_i] = [a_i] · [a_{i−1}];
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 Algorithms

Another important and commonly used functionality is sorting

Almost all sorting algorithms are non-oblivious

A notable example is a bitonic sorting network

- bitonic mergesort is what assignment 4 used
- it takes $O(n \log^2 n)$ time to sort n items

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

Summary

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

Writing a well-performing program is very non-trivial

- built-in optimizations can easily reduce program's runtime by orders of magnitude