A verification framework for secure machine learning

Théophile Wallez

07/09/2020

Introduction

Situation:

- A server hold a machine learning model M
- A client hold an input x
- The client want to know $M(x)$

Introduction

Situation:

- A server hold a machine learning model M
- A client hold an input x
- The client want to know $M(x)$

Problem: M and x should remain secret

Introduction

Situation:

- A server hold a machine learning model M
- A client hold an input x
- The client want to know $M(x)$

Problem: M and x should remain secret
Solution: Use PPML (Privacy-Preserving Machine Learning) techniques

Introduction

Situation:

- A server hold a machine learning model M
- A client hold an input x
- The client want to know $M(x)$

Problem: M and x should remain secret
Solution: Use PPML (Privacy-Preserving Machine Learning) techniques

Problem: Cryptographic implementations are often prone to bugs

Introduction

Situation:

- A server hold a machine learning model M
- A client hold an input x
- The client want to know $M(x)$

Problem: M and x should remain secret
Solution: Use PPML (Privacy-Preserving Machine Learning) techniques

Problem: Cryptographic implementations are often prone to bugs
Solution: Use software verification techniques

Goal of this internship

Create a verified implementation in F$*$ of the secure multiparty computation protocol SPD$\mathbb{Z}_{2^k}$.

Table of Contents

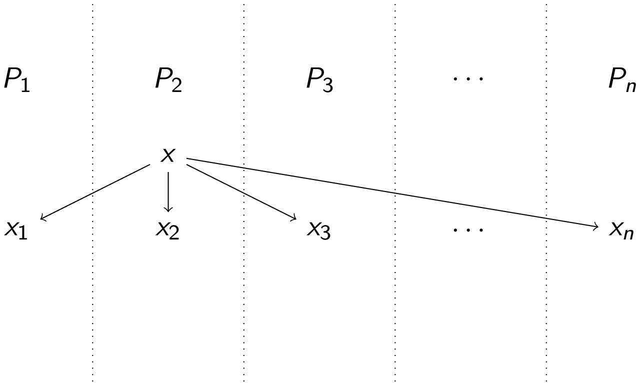
Multiparty computation modulo 2^k

P_1	P_2	P_3	$\cdots$	P_n

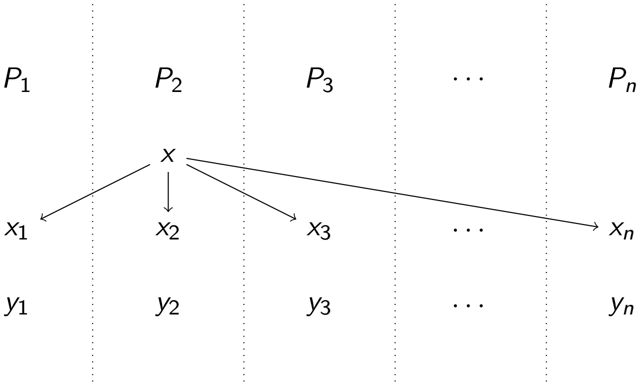
Multiparty computation modulo 2^k

P_1	P_2	P_3	$\cdots$	P_n
	x			

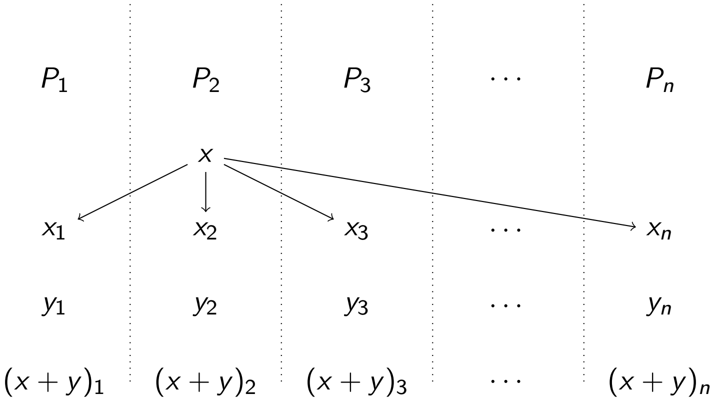
Multiparty computation modulo 2^k



Multiparty computation modulo 2^k



Multiparty computation modulo 2^k



Multiparty computation modulo 2^k: basic operations

x	x_1	x_2	x_3	$\cdots$	x_n
y	y_1	y_2	y_3	$\cdots$	y_n

Multiparty computation modulo 2^k: basic operations

x	x_1	x_2	x_3	$\cdots$	x_n
y	y_1	y_2	y_3	$\cdots$	y_n
$x+y$	x_1+y_1	x_2+y_2	x_3+y_3	$\cdots$	x_n+y_n

Multiparty computation modulo 2^k: basic operations

x	x_1	x_2	x_3	$\cdots$	x_n
y	y_1	y_2	y_3	$\cdots$	y_n
$x+y$	x_1+y_1	x_2+y_2	x_3+y_3	$\cdots$	x_n+y_n
cx	cx_1	cx_2	cx_3	$\cdots$	cx_n

Multiparty computation modulo 2^k: basic operations

x	x_1	x_2	x_3	$\cdots$	x_n
y	y_1	y_2	y_3	$\cdots$	y_n
$x+y$	x_1+y_1	x_2+y_2	x_3+y_3	$\cdots$	x_n+y_n
cx	cx_1	cx_2	cx_3	$\cdots$	cx_n
$c+x$	$c+x_1$	x_2	x_3	$\cdots$	x_n

Multiparty computation modulo 2^k: multiplication

How to compute shares for xy?

Multiparty computation modulo 2^k: multiplication

How to compute shares for xy?
Trick: use shares of random a, b, c such that $ab = c$.

Multiparty computation modulo 2^k: multiplication

How to compute shares for xy?
Trick: use shares of random a, b, c such that $ab = c$.

$$
\begin{aligned}
& xy \\
=& ((x-a)+a)((y-b)+b) \\
=& (x-a)(y-b) + (y-b)a + (x-a)b + ab
\end{aligned}
$$

SPD$\mathbb{Z}_{2^k}$, a rough idea

Problem: when opening x, an active adversary can lie about its share.

SPD$\mathbb{Z}_{2^k}$, a rough idea

Problem: when opening x, an active adversary can lie about its share.

Solution:

- Use a random shared secret α (an authentication key)
- Compute $m_x := \alpha x$ along x

SPD$\mathbb{Z}_{2^k}$, a rough idea

Problem: when opening x, an active adversary can lie about its share.

Solution:

- Use a random shared secret α (an authentication key)
- Compute $m_x := \alpha x$ along x

If an adversary lies about its share of x, it has to lie about its share of m_x and therefore guess α.

A look at F∗'s syntax

A look at F∗'s syntax

```
val add: int → int → int
let add x y = x + y
```

A look at F∗'s syntax

```
val add: int → int → int
let add x y = x + y
```

```
val map: (α → β) → list α → list β
let rec map f l =
  match l with
  | [] → []
  | h::t → (f h)::(map f t)
```

Refinement types

```
val index: list α → nat → α
let index l i = ...
```

Refinement types

```
val index: list α → nat → α
let index l i = ...
```

```
index [57;3;1000;42] 2
(* = 1000 *)
```

Refinement types

```
val index: list α → nat → α
let index l i = ...
```

```
index [57;3;1000;42] 10
(* = ? *)
```

Refinement types

```
val index: l:list α → i:nat → α
let index l i = ...
```

```
index [57;3;1000;42] 10
(* = ? *)
```

Refinement types

```
val index: l:list α → i:nat{i < length l} → α
let index l i = ...
```

```
index [57;3;1000;42] 10
(* → "Subtyping check failed" *)
```

Refinement types for proofs

```
val index: l:list α → i:nat{i < length l} → α
let index l i =


  ...
```

Refinement types for proofs

```
val index: l:list α → i:nat{i < length l} → α
let index l i =
  (* Here we can use the fact that i < length l *)
  ...
```

Refinement types for proofs

```
val index: l:list α → i:nat{1+1 = 2} → α
let index l i =
  (* Here we can use the fact that 1+1 = 2 *)
  ...
```

Refinement types for proofs

```
val index: l:list α → i:nat{1+1 = 2} → α
let index l i =
  (* Here we can use the fact that 1+1 = 2 *)
  ...
```

An instance of (){1+1=2} is a proof that 1+1=2.

The Lemma effect

```
val append_length:
  l1:list α → l2:list α →
  (){length (l1@l2) = length l1 + length l2}
```

The Lemma effect

```
val append_length:
  l1:list α → l2:list α →
  Lemma ((length (l1@l2) = length l1 + length l2))
```

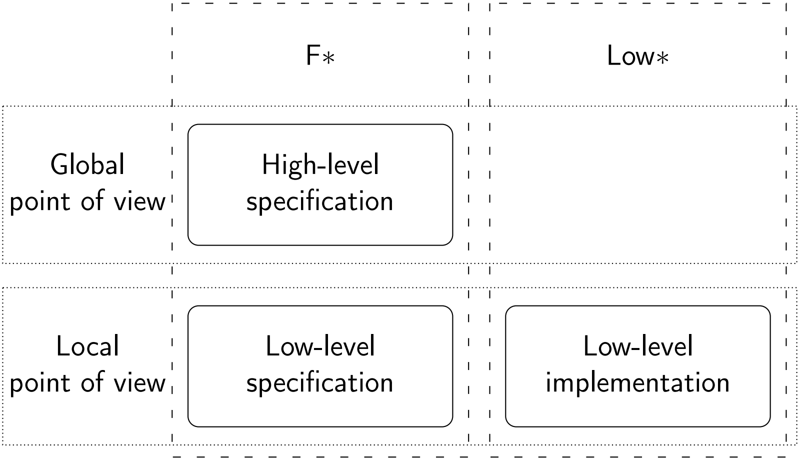
The Lemma effect

```
val append_length:
  l1:list α → l2:list α →
  Lemma ((length (l1@l2) = length l1 + length l2))
```

```
val append_eq_nil:
  l1:list α → l2:list α →
  Lemma (requires (l1@l2 == []))
        (ensures (l1 == [] ∧ l2 == []))
```

Table of Contents

Modules for this project



Types used in this project

	Local	Global
Unauthenticated	elem k	shares n k

Types used in this project

	Local	Global
Unauthenticated	elem k	shares n k
Authenticated	auth_elem k s	auth_shares n k s

Table of Contents

High-level specification

```
val add_shares_shares:
  auth_shares n k s → auth_shares n k s → auth_shares n k s
```

High-level specification: correctness theorems

```
val combine_add_shares_shares_lemma:
  a:auth_shares n k s → b:auth_shares n k s →
  Lemma (
    (combine (add_shares_shares a b))
  = (combine a) +% (combine b)
  )
```

High-level specification: correctness theorems

```
val combine_add_shares_shares_lemma:
  a:auth_shares n k s → b:auth_shares n k s →
  Lemma (
    (combine (add_shares_shares a b))
  = (combine a) +% (combine b)
  )
```

```
val auth_add_shares_shares_lemma:
  alpha:shares n s →
  a:auth_shares n k s → b:auth_shares n k s →
  Lemma
    (requires authenticated alpha a ∧ authenticated alpha b)
    (ensures authenticated alpha (add_shares_shares a b))
```

Table of Contents

How to represent communication?

f:

- receives a local α

- receives a γ from the network
- returns a δ

How to represent communication?

f:

- receives a local α
- receives a γ from the network
- returns a δ

val f: $\alpha \rightarrow (\gamma \ \rightarrow \delta)$

How to represent communication?

f:

- receives a local α
- sends a β to the network
- receives a γ from the network
- returns a δ

How to represent communication?

f:

- receives a local α
- sends a β to the network
- receives a γ from the network
- returns a δ

```
val f: α → (β * (γ → δ) )
```

The com datatype

```
type com (send:Type) (recv:Type) (ret:Type) = send * (recv → ret)
```

The com datatype

```
type com (send:Type) (recv:Type) (ret:Type) = send ∗ (recv → ret)
```

```
val open_share_dumb:
     elem k → com (elem k) (shares n k) (elem k)
let open_share_dumb x_share =
   (x_share, (λ x_shares → List.fold_right (+%) x_shares 0))
```

Low-level specification

```
val add_share_share:
  auth_elem k s → auth_elem k s → auth_elem k s
```

Low-level specification correctness theorems on local code

```
val add_share_share_correct:
    x:auth_shares n k s → y:auth_shares n k s → i:nat{i<n} →
    Lemma (
        add_share_share (List.index x i) (List.index y i)
      = List.index (add_shares_shares x y) i
    )
```

The make_broadcast function

```
val make_broadcast: llist (com α (llist α n) γ) n → llist γ n
```

Low-level specification correctness theorems on communicating code

```
val open_share_dumb:
    elem k → com (elem k) (shares n k) (elem k)
```

```
val open_share_dumb_correct:
  x:shares n k → i:nat{i<n} →
  Lemma (
    List.index (
      make_broadcast (List.map open_share_dumb x)
    ) i
  = List.fold_right (+%) x 0
  )
```

Table of Contents

Conclusion

I produced a verified functional implementation of the computing phase of the SPD$\mathbb{Z}_{2^k}$ protocol.

Conclusion

I produced a verified functional implementation of the computing phase of the SPD$\mathbb{Z}_{2^k}$ protocol.

Future work:

- Low-level implementation in Low$*$
- Implementation of the preprocessing phase
- Privacy proofs