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 $M(x)$

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
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 holds a machine learning model $M$
- A client holds an input $x$
- The client wants 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 holds a machine learning model $M$
- A client holds an input $x$
- The client wants 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 \( \text{SPDZ}_{2k} \).
Table of Contents

Background
- Multiparty computation modulo $2^k$
- SPD $\mathbb{Z}_{2^k}$
- F*

Architecture of this implementation

High-level specification

Low level spec

Conclusion
Multiparty computation modulo $2^k$
Multiparty computation modulo $2^k$
Multiparty computation modulo $2^k$
Multiparty computation modulo $2^k$
Multiparty computation modulo $2^k$
Multiparty computation modulo $2^k$: basic operations

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$\cdots$</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$\cdots$</td>
<td>$y_n$</td>
</tr>
</tbody>
</table>
Multiparty computation modulo $2^k$: basic operations

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>...</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>...</td>
<td>$y_n$</td>
</tr>
<tr>
<td>$x + y$</td>
<td>$x_1 + y_1$</td>
<td>$x_2 + y_2$</td>
<td>$x_3 + y_3$</td>
<td>...</td>
<td>$x_n + y_n$</td>
</tr>
</tbody>
</table>
### Multiparty computation modulo $2^k$: basic operations

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$\cdots$</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$\cdots$</td>
<td>$y_n$</td>
</tr>
<tr>
<td>$x + y$</td>
<td>$x_1 + y_1$</td>
<td>$x_2 + y_2$</td>
<td>$x_3 + y_3$</td>
<td>$\cdots$</td>
<td>$x_n + y_n$</td>
</tr>
<tr>
<td>$cx$</td>
<td>$cx_1$</td>
<td>$cx_2$</td>
<td>$cx_3$</td>
<td>$\cdots$</td>
<td>$cx_n$</td>
</tr>
</tbody>
</table>
**Multiparty computation modulo $2^k$: basic operations**

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$\cdots$</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td></td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$\cdots$</td>
<td>$y_n$</td>
</tr>
<tr>
<td>$x + y$</td>
<td>$x_1 + y_1$</td>
<td>$x_2 + y_2$</td>
<td>$x_3 + y_3$</td>
<td>$\cdots$</td>
<td>$x_n + y_n$</td>
<td></td>
</tr>
<tr>
<td>$cx$</td>
<td></td>
<td>$cx_1$</td>
<td>$cx_2$</td>
<td>$cx_3$</td>
<td>$\cdots$</td>
<td>$cx_n$</td>
</tr>
<tr>
<td>$c + x$</td>
<td>$c + x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$\cdots$</td>
<td>$x_n$</td>
<td></td>
</tr>
</tbody>
</table>
Multiparty computation modulo $2^k$: multiplication

How to compute shares for $xy$?
How to compute shares for $xy$?

Trick: use shares of random $a$, $b$, $c$ such that $ab = c$. 

\[
xy = (x - a)(y - b) + (y - b)a + (x - a)b + ab
\]
How to compute shares for $xy$?
Trick: use shares of random $a$, $b$, $c$ such that $ab = c$.

$$xy = ((x - a) + a)((y - b) + b) = (x - a)(y - b) + (y - b)a + (x - a)b + ab$$
SPD$Z_{2^k}$, a rough idea

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

Solution:
- Use a random shared secret $\alpha$ (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 $\alpha$ (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 $\alpha$. 
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

```ocaml
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

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

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

index [57;3;1000;42] 2
(* = 1000 *)
```
Val index : list $\alpha \rightarrow \text{nat} \rightarrow \alpha$

Let index l i = ...

Index [57;3;1000;42] 10

(* = ? *)
Refinement types

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

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

```ocaml
val index : l:list \(\alpha \rightarrow i:\text{nat}\{i < \text{length } l\} \rightarrow \alpha\)
let index l i = ...

index [57;3;1000;42] 10
(* \rightarrow "Subtyping check failed" *)
```
Refinement types for proofs

```ocaml
cval index : l : list α → i : nat {i < length l} → α
define index l i =
...
```
Refinement types for proofs

```ocaml
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 *)
...

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

```plaintext
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

\[
\begin{align*}
\text{val append\_length:} \\
\text{l1:list} \alpha \rightarrow \text{l2:list} \alpha \rightarrow \\
\text{()\{length (l1@l2) = length l1 + length l2\}}
\end{align*}
\]
The Lemma effect

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

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

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

Background

Architecture of this implementation
  Modules
  Types

High-level specification

Low level spec

Conclusion
Modules for this project

F*

High-level specification

Low*

Local point of view

Low-level specification

Local point of view

Low-level implementation

Global point of view
## Types used in this project

<table>
<thead>
<tr>
<th>Unauthenticated</th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>elem $k$</td>
<td></td>
<td>shares $n$ $k$</td>
</tr>
</tbody>
</table>
Types used in this project

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unauthenticated</td>
<td>elem k</td>
<td>shares n k</td>
</tr>
<tr>
<td>Authenticated</td>
<td>auth_elem k s</td>
<td>auth_shares n k s</td>
</tr>
</tbody>
</table>
Table of Contents

Background

Architecture of this implementation

High-level specification
  Specification
  Correctness theorems

Low level spec

Conclusion
High-level specification

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

val auth_add_shares_shares_lemma:
  alpha:shares n s \rightarrow
  a:auth_shares n k s \rightarrow b:auth_shares n k s \rightarrow
  Lemma ( requires authenticated alpha a \land authenticated alpha b )
  (ensures authenticated alpha (add_shares_shares a b))
Table of Contents

Background

Architecture of this implementation

High-level specification

Low level spec
  Representing communication
  Specification
  Correctness theorems

Conclusion
How to represent communication?

\[ f: \]
- receives a local \( \alpha \)
- receives a \( \gamma \) from the network
- returns a \( \delta \)
How to represent communication?

\[
\text{f:}
\begin{align*}
\&\quad \text{receives a local } \alpha \\
\&\quad \text{receives a } \gamma \text{ from the network} \\
\&\quad \text{returns a } \delta
\end{align*}
\]

\[
\text{val f: } \alpha \rightarrow (\gamma \rightarrow \delta)
\]
How to represent communication?

\( f \):
- Receives a local \( \alpha \)
- Sends a \( \beta \) to the network
- Receives a \( \gamma \) from the network
- Returns a \( \delta \)
How to represent communication?

\[ f: \]
- receives a local \( \alpha \)
- sends a \( \beta \) to the network
- receives a \( \gamma \) from the network
- returns a \( \delta \)

\[ \text{val } f: \alpha \rightarrow (\beta \ast (\gamma \rightarrow \delta)) \]
The com datatype

define 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)
The com datatype

```
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

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

```ml
val make_broadcast : llist (com α (llist α n) γ) n → llist γ n
```
Low-level specification correctness theorems on communicating code

Definition:

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

Definition:

```ml
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

Background

Architecture of this implementation

High-level specification

Low level spec

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

Future work:
- Low-level implementation in Low$^*$
- Implementation of the preprocessing phase
- Privacy proofs