A verification framework for secure machine learning

Théophile Wallez, supervised by Karthikeyan Bhargavan and Prasad Naldurg
Prosecco, INRIA Paris

20/08/2020

The general context

Machine learning applications consume vast amounts of user data (including personally identifiable information) to produce analytical models, which are subsequently used to assist in making classification decisions on new data without human intervention. While machine learning applications are getting more and more sophisticated, security and privacy issues in this context have received lesser attention. In privacy preserving machine learning (PPML), such a machine-learning classifier (server) should treat user queries opaquely, and should not learn anything about the query issued by a client or its resulting response (i.e., the resulting class). A client should only learn the correct response to its query and not learn anything about the model parameters on the servers or about the data used to train the model.

One approach to address the privacy problem is through the design of sophisticated cryptographic protocols, based on secure multi-party computation, homomorphic encryption, functional encryption or garbled circuits. All these techniques are expensive, but recent advances make them realizable. For example, the novel SPD$\mathbb{Z}_{2^k}$ [4] protocol allows efficient computations modulo 2^k between several parties, using a less efficient preprocessing phase generating correlated random numbers.

Implementation of cryptographic protocols can be subtle and bugs can be introduced at different layers. The implementation might compute the wrong result (e.g. an arithmetic error in the OpenSSL Poly1305 implementation [3]), there might be a memory vulnerability (e.g. a buffer overflow in the OpenSSL Chacha20-Poly1305 implementation [1]) or information might be leaked via side channels (e.g. the ECDSA nonces leaked in the cache in OpenSSL [9]).

Software verification can help to reduce the attack surface of such implementations. For example, the proof assistant F$*$ [6] has been used to build HACL$*$ [10], a fully-verified and efficient cryptographic library. HACL$*$ is used inside production nowadays in Mozilla Firefox and Wireguard, and is the basis of Everest [2], a project for a verified implementation for HTTPS.

The research problem

The design, implementation and analysis of protocols for PPML is an exciting new area of research. However, none of the existing implementations are verified.

One of the core building blocks of recent proposals are MPC protocols [5]. We build the first verified formal implementation of SPD$\mathbb{Z}_{2^k}$ and our goal is to prove its correctness and security in F$*$. Beyond PPML, we believe that this implementation is of stand-alone interest to any sophisticated cryptographic application built using MPC that needs to replace a trusted third party, including digital currency, encrypted databases, online auctions, and e-voting protocols etc.

Your contribution

We build several components necessary for the implementation of the computation phase of the SPD$\mathbb{Z}_{2^k}$ protocol. We first build a high-level specification of the protocol, describing how the computation runs from a global point of view, and prove its correctness. We then build a low-level specification of the protocol, describing how the computation runs from a local point of view, and prove its correctness using the global specification.

In addition to the implementation of verified MPC, we also revisit the nature and scope of the foundational definitions of PPML. One of our findings that we present here is that it is difficult to get the definitions right in practice. We show how we can extract information from an SVM model, even in a so called privacy preserving implementation, arguing for a different notion of privacy in this context.

Arguments supporting its validity

We produced a concise and readable specification of the SPD$\mathbb{Z}_{2^k}$ protocol which can easily be checked by a human, and proved that the result of the computations with SPD$\mathbb{Z}_{2^k}$ is correct. The low-level specification can be extracted to OCaml and actually run on a computer, therefore producing a reference implementation for the SPD$\mathbb{Z}_{2^k}$ protocol.

Summary and future work

In order to complete this work, we can write a low-level implementation in Low$*$ [7], a low-level subset of F$*$ which can then be extracted to readable C code. The current implementation can only be extracted to OCaml and is not designed to be efficient. The protocol is only proved to be secure on the paper, a mechanized proof would be a nice addition, using ideas similar to Dolev-Yao symbolic models or Wys$*$ [8]. The SPD$\mathbb{Z}_{2^k}$ protocol has a preprocessing phase and a computation phase, the preprocessing phase is currently not implemented.

Acknowledgments

I would like to express my gratitude to Karthikeyan Bhargavan and Prasad Naldurg for supervising me during this internship, for being available even when the exterior conditions were sub-optimal (during the confinement, or when I broke my leg).

Thanks to my roommates Paul-Nicolas Madelaine, Hugo Manet, Camille Masson-Durcudoy and Adélaïde Subts for allowing me to live a rather enjoyable confinement.

Contents

1 Background

1.1 F∗

F∗ [6] is a general-purpose functional programming language with effects aimed at program verification. F∗'s type system includes dependent types, monadic effects, refinement types, and a weakest precondition calculus. Together, these features allow expressing precise and compact specifications for programs. Verification is fully automated using an SMT solver. F∗ programs can be extracted to efficient OCaml, F#, C, WASM, or ASM code, which allows verifying the functional correctness and security of realistic applications.

A refinement type is a type associated with a property which is true when there is an instance of this type. For example, an object of the type n:nat{n%2 = 0} is an even natural number. It allows to define precisely what conditions should meet the input of a function, for example the function to get the ith element of a list:

```
val index: l:list α → i:nat{i < List.Tot.length l} → α
```

is a total function and the compiler verifies that the index passed is in bound.

Often, properties used in refinement types are related to the object of this type, but do not have to necessarily. The type i:nat{i < List.Tot.length l} is a natural number along with a property that it is less than List.Tot.length l, the type i:nat{1+1=2} is a natural number along with a property that 1+1=2.

This way, we can define theorems as a refinement type on unit: an object of the type (){1+1=2} is a theorem that 1+1=2.

As in Coq, types are theorems and instances of these types are proofs. However F∗ theorems are not defined using the Curry-Howard isomorphism but with refinement types.

In F∗'s standard library, there are several basic datatypes.

Some types to represent numbers with various properties:

```
(* 'int' represents an integer *)
assume new type int
(* 'nat' represents a natural number *)
type nat = i:int{i ≥ 0}
(* 'pos' represents a positive number *)
type pos = i:int{i > 0}
```

And the standard list and option types, along with a type for fixed-size lists.

```
(* 'option a' represents an optional value of type 'a' *)
type option (a: Type) =
  | None : option a
  | Some : v: a → option a

(* 'list a' represents a list of elements with types 'a' *)
type list (a: Type) =
  | Nil : list a
  | Cons : hd: a → tl: list a → list a

(* 'llist a n' represents a list of elements with type 'a' and size 'n' *)
let llist a (n:nat) = l:list a {length l = n}
```

The Lemma effect allows to write theorems in a more elegant way than using a refinement on the unit type:

```
(* For all lists 'l1', 'l2', the length of the concatenation of 'l1' and 'l2' is
 * equal to the sum of the lengths of 'l1' and 'l2' *)
val append_length:
  l1:list α → l2:list α →
  Lemma ((length (l1@l2) = length l1 + length l2))

(* For all lists 'l1', 'l2', if their concatenation is the empty list, then
 * both of 'l1' and 'l2' are the empty list *)
val append_eq_nil:
  l1:list α → l2:list α →
  Lemma (requires (l1@l2 == []))
        (ensures (l1 == [] ∧ l2 == []))
```

In F*, since the proofs are written using an SMT solver, they can be really concise. In fact, well-written proofs often consist of a few instantiations of some theorems from which it is possible do deduce the final result. To save the programmer some work, some theorems are automatically instantiated when a term matches some pattern during the proof.

For example, the theorem

```
val append_l_nil:
  x:list α →
  Lemma (requires ⊤)
        (ensures (x@[] == x))
        [SMTPat (x@[])]
```

will be automatically instantiated every time a list is concatenated with the empty list.

1.2 Secure multiparty computation, a rough idea

Suppose we have n parties $P_1, P_2, \ldots, P_n$, and each party holds a value in $\mathbb{Z}_{2^k}$ (P_i holds the value $x^{(i)}$) and they want to compute the sum of their values $\sum_{i=1}^{n} x^{(i)}$ without revealing anything else about their values.

Here is one way to do this.

For each j, P_j splits its value into shares $x_1^{(j)}, x_2^{(j)}, \ldots, x_n^{(j)}$ such that

$$x^{(j)} = \sum_{i=1}^{n} x_i^{(j)}$$

and sends $x_i^{(j)}$ to P_i. One way to do this for P_j would be to choose $x_1^{(j)}, \ldots, x_{j-1}^{(j)}, x_{j+1}^{(j)}, \ldots, x_n^{(j)}$ randomly, and compute

$$x_j^{(j)} = x^{(j)} - \sum_{i \neq j} x_i^{(j)}$$

, so that the values revealed to other parties are purely random and do not reveal any information.

Subsequently, each party P_j computes

$$S_j = \sum_{i=1}^{n} x_j^{(i)}$$

and broadcasts this value to every party.

Now, each party can compute

$$\sum_{j=1}^{n} S_j = \sum_{j=1}^{n} \sum_{i=1}^{n} x_j^{(i)} = \sum_{i=1}^{n} \sum_{j=1}^{n} x_j^{(i)} = \sum_{i=1}^{n} x^{(i)}$$

Each party now knows the sum of $x^{(i)}$ without having to reveal their values. This protocol is summarized in Figure 1.

This protocol is secure against $n-1$ passive adversaries. Without loss of generality, assume we are P_1. If there are exactly $n-1$ passive adversaries, then they can deduce $x^{(1)}$ using the result of the computation, and the protocol does not reveal more information. If there are less than $n-1$ passive adversaries, without loss of generality, P_2 is not a passive adversary, then

$$S_1 = \sum_{i=1}^{n} x_1^{(i)} = x_1^{(1)} + x_1^{(2)} + \sum_{i=3}^{n} x_1^{(i)}$$

and since $x_1^{(2)}$ is purely random, the adversaries cannot deduce $x^{(1)}$ from the broadcasted values.

This protocol is not secure against active adversaries: the adversaries can compromise the computation by revealing garbage values, and $n-1$ active adversaries can deduce x_1, by sending $x_1^{(2)} = x_1^{(3)} = \cdots = x_1^{(n)} = 0$ to P_1 who will subsequently broadcast $S_1 = x_1^{(1)}$, and the adversaries will be able to reconstruct $x^{(1)}$.

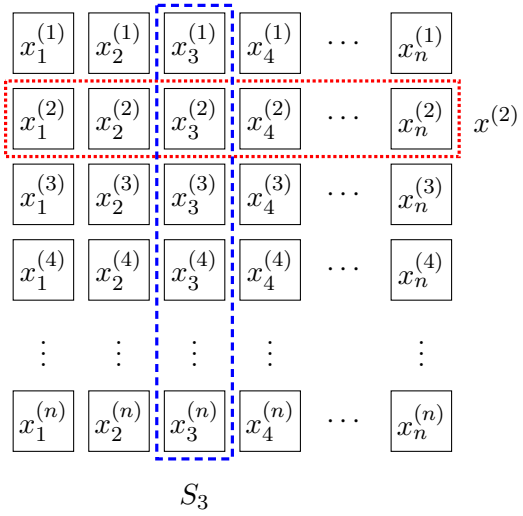

Figure 1: Drawing of the multiparty sum protocol.

1.3 SPD$\mathbb{Z}_{2^k}$

SPD$\mathbb{Z}_{2^k}$ [4] works in a similar way with some additional features, and is secure against $n-1$ active adversaries.

In the previous example, values are computed modulo 2^k, in SPD$\mathbb{Z}_{2^k}$ we compute them modulo 2^{k+s} where s is a security parameter. There is a secret value $\alpha \in \mathbb{Z}_{2^s}$ shared between parties, called the MAC key. The MAC for a value x is given by $m_x := \alpha x$, is shared between parties and computed at the same time as x. For example, m_{x+y} is computed at the same time as $x+y$, using the equality $m_{x+y} = m_x + m_y$.

When opening the value x, if an active adversary lies about its share of x and m_x and the opened value is $x' \neq x$ with a MAC m'_x, then it also needs to ensure that $\alpha x' = m'_x$, which gets harder as s grows since α is unknown and it has to guess the value of $\alpha(x'-x)$.

If x, y are shared values and c is a public constant, then it is possible to compute authenticated shares of $x+c$, cx and $x+y$ locally without any communication.

For multiplication of two shares, more work is needed: SPD$\mathbb{Z}_{2^k}$ use shares computed in a preprocessing phase to do multiplication of two shares. A multiplication triplet consists of three authenticated shares of a, b, c such that a and b are random and $c = ab$. Then, by noticing that

$$xy = ((x-a)+a)((y-b)+b) = (x-a)(y-b) + (y-b)a + (x-a)b + ab$$

we can open the values of $x-a$ and $y-b$ (which do not reveal any information since a and b are random) and compute authenticated shares of xy by using the three elementary operations on authenticated shares, since $(x-a)$ and $(y-b)$ are constants, and $ab = c$ is a shared value.

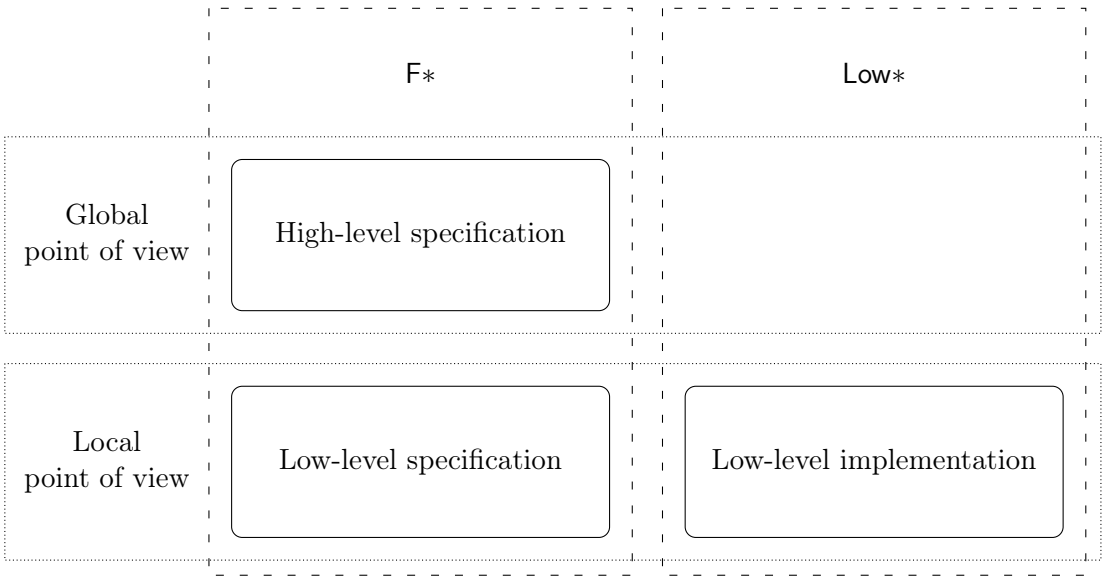

Figure 2: Diagram showing the various parts of this verified implementation.

2 An approach for a secure, fully verified and efficient implementation of a cryptographic protocol

We follow an approach (shown in Figure 2) similar to HACL∗.

2.1 High-level specification

First, the protocol is described from a global point of view, by manipulating the state of every party at the same time. This allows us to describe the protocol in a succinct way, without bothering with some details such as communication (for example, opening a shared value is easy, we just have to compute a sum of numbers). In this specification, the correctness of the protocol is verified, for example the protocol to compute addition of two shared numbers correctly produces shares for the sum of those numbers.

In this internship, the high-level specification was fully implemented and verified.

2.2 Low-level specification

Then, the protocol is described from a local point of view, by manipulating the state of one party. We can see this as a functional implementation of the protocol. It means that we can use it in the real world (e.g. by extracting it to OCaml) but it might be slow. Here, we need to consider details such as communication. This specification is proved to correspond to the high-level one. We can also prove its security, for example that secret values are not leaked to the adversaries.

In this internship, the low-level specification was fully implemented and verified to correspond to the high-level specification.

2.3 Low-level implementation

The protocol is finally implemented in a fast imperative language. We prove that it is equivalent to the low-level specification. We also prove that no conditions or memory accesses depend on a secret, ensuring that it is resistant against cache and other timing attacks.

3 High-level specification

3.1 Datatypes

First, we define a type for elements of the $\mathbb{Z}_{2^k}$ semiring.

```
(* 'elem k' is an element of Z_{2^k} *)
type elem (k:pos) = n:nat{n < pow2 k}
```

An element of $\mathbb{Z}_{2^k}$ is represented as a natural number less than 2^k, canonically. It corresponds to a machine unsigned integer on k bits on which operations can overflow.

Several operations are defined on numbers in $\mathbb{Z}_{2^k}$.

```
(* Negation *)
val ( ¬% ): #k:pos → elem k → elem k
(* Addition *)
val ( +% ): #k:pos → elem k → elem k → elem k
(* Subtraction *)
val ( −% ): #k:pos → elem k → elem k → elem k
(* Multiplication *)
val ( *% ): #k:pos → elem k → elem k → elem k
(* Add bits *)
val upcast: #k:pos → k':pos{k ≤ k'} → elem k → elem k'
(* Remove bits *)
val downcast: #k:pos → k':pos{k' ≤ k} → elem k → elem k'
```

For example, the negation takes an implicit parameter k corresponding to the number of bits, takes an element of $\mathbb{Z}_{2^k}$, and returns an element of $\mathbb{Z}_{2^k}$. Sometimes, we need to extend a number and add bits, or to truncate a number and remove bits: this is done with the downcast and upcast functions. For example, the upcast functions takes an implicit parameter k corresponding to the number of bits as input, takes a k' greater than k corresponding to the numbor of bits as output, takes an input on k bits, and returns an output on k' bits.

Next, we define several datatypes representing the values in SPD$\mathbb{Z}_{2^k}$.

An unauthenticated share with k bits of value and s bits of security is just an elem (k+s). However for authenticated shares we need two numbers: the share of the value and the share of the MAC. This is done in the following record type:

```
(* 'auth_elem k s' is an authenticated share with 'k' bits of value and 's' bits of security *)
type auth_elem (k:pos) (s:nat) = {v:elem (k+s); m: elem (k+s)}
```

The shares of a value is a list of shares, with a length corresponding to the number of parties. There are two versions, an un-authenticated one and an authenticated one.

```
(* 'shares n k' represents the shares between 'n' parties of an unauthenticated element of
 * 'k' bits *)
type shares (n:pos) (k:pos) = l:llist (elem k) n
(* 'auth_shares n k s' represents the shares between 'n' parties of an authenticated element
 * with 'k' bits of value and 's' bits of security *)
type auth_shares (n:pos) (k:pos) (s:nat) = l:llist (auth_elem k s) n
```

To ease the use of multiplication triplet shares, a record type is defined, along with its "list" version.

```
(* 'multiplication_triplet k s' represents one share of a multiplication triplet with 'k' bits of
 * value and 's' bits of security *)
type multiplication_triplet (k:pos) (s:nat)={a:auth_elem k s; b:auth_elem k s; c:auth_elem k s}
(* 'multiplication_triplet_shares n k s' represents the shares between 'n' parties of a
 * multiplication triplet *)
type multiplication_triplet_shares (n:pos) (k:pos) (s:nat) = l:llist (multiplication_triplet k s) n
```

3.2 Functional specification

Addition between two shared values looks like this:

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

This function takes the list of shares for x (of type auth_shares n k s), the list of shares for y (of type auth_shares n k s), and returns the list of shares corresponding to $x + y$ (of type auth_shares n k s).

Some operations (such as the addition of a constant) depend on the choice of a non-canonical party, which is passed as an argument.

```
val add_cst_shares:
    party:nat{party < n} → shares n (k+s) →
    elem (k+s) → auth_shares n k s → auth_shares n k s
```

This function takes the chosen party identity (a nat less than n), the list of shares for the MAC key α (of type shares n (k+s) since α is not authenticated), the constant c (of type elem (k+s)), the list of shares for x (of type auth_shares n k s), and returns the list of shares corresponding to $c + x$ (of type auth_shares n k s).

The multiplication of two shared values takes an additional argument corresponding to the multiplication triplet shares.

```
val mul_shares_shares:
    party:nat{party < n} → shares n (k+s) → multiplication_triplet_shares n k s →
    auth_shares n k s → auth_shares n k s → auth_shares n k s
```

3.3 Correctness theorems

To write the correctness theorem, we use the following two helper functions:

```
val combine_auth_shares: auth_shares n k s → auth_elem k s
val authenticated: shares n (k+s) → auth_shares n k s → bool
```

The function combine_auth_shares takes the list of shares for x (of type auth_shares n k s) and returns the reconstruction of the value and its MAC (of type auth_elem k s). The function authenticated takes the list of shares for α (of type shares n (k+s) since α is not authenticated), the list of shares for x (of type auth_shares n k s), and returns a boolean checking whether x is correctly authenticated with respect to the key α.

This theorem ensures the correctness of the value produced by the add_shares_shares protocol.

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

We can read it as follows: for a list of authenticated shares a and a list of authenticated shares b, the value associated with the shares given by the protocol add_shares_shares corresponds to the sum of the value of a and the value of b.

The following theorem ensures the correctness of the MAC produced by the protocol add_shares_shares:

```
val auth_add_shares_shares_lemma:
    alpha:shares n (k+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))
```

We can read it as follows: for a list of shares alpha, a list of authenticated shares a, and a list of authenticated shares b, if a and b are correctly authenticated with respect to alpha, so are the shares produced by the protocol add_shares_shares.

The correctness theorem for the multiplication protocol is a bit more complex.

```
val combine_mul_shares_shares_lemma:
  party:nat{party < n} → alpha:shares n (k+s) → triplet:multiplication_triplet_shares n k s →
  x:auth_shares n k s → y:auth_shares n k s →
  Lemma
    (requires
        let a_shares = triplet_shares_a triplet in
        let b_shares = triplet_shares_b triplet in
        let c_shares = triplet_shares_c triplet in
        (combine_auth_shares a_shares).v *% (combine_auth_shares b_shares).v
      = (combine_auth_shares c_shares).v
    )
    (ensures
        (downcast k (combine_auth_shares (mul_shares_shares party alpha triplet x y)).v)
      = (downcast k (combine_auth_shares x).v) *% (downcast k (combine_auth_shares y).v)
    )
```

First, some assumption about the multiplication triplet is needed (which is that $ab = c$). Second, the multiplication protocol correctly computes the multiplication on the k value bits, but not on the s security bits! This is because inside the protocol, some values are opened and their security bits are not revealed (as this might leak information). Actually, the theorem about add_shares_shares is stronger than needed because we do not need to know that it also computes the addition on the s security bits in order to be correct.

4 Low-level specification

4.1 Representing communication with the com datatype

Let's see how to represent communication in a simple protocol. Two parties A and B hold a value (a natural number). Party A sends its value to B, and party B sends its value to A. Then, they both compute the sum of their values. In other words, each party sends its value and then receives another value to compute its sum.

One way to implement it would be:

```
val example_add: nat → nat * (nat → nat)
let example_add x =
    (x, (λ y → x+y))
```

The function example_add returns a pair: the first element is its value, corresponding to the data to send and the second element is a callback (or a continuation), which can be called when the value from the other party is received.

This way of representing communication can be generalized and is implemented in the com datatype:

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

com send recv ret represents a function sending a value of type send and waiting on a value of type recv to return a value of type ret. This communication is asynchronous.

A com type can be handled by the following imperative pseudo-code:

```
val com_resolve: com α β γ → γ
let com_resolve (value, cont) =
    network_send value;
    let x = network_receive () in
    cont x
```

A simple example of communication on shares would be:

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

This function broadcasts one share and gets back a list of shares (including its own), which is used to reconstruct the shared value.

In SPD$\mathbb{Z}_{2^k}$, the communication is mostly done in a way such that each party broadcasts a value, therefore we use the following type:

```
type com_broadcast (n:pos) (ty:Type) (ret:Type) = com ty (llist ty n) ret
```

where a party sends a value of type ty, then receives n values of type ty corresponding to the values sent by each party (including its own).

We can resolve a com_broadcast from a global point of view with the following function:

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

Briefly, it takes a list of parties waiting to broadcast an α to return a γ, and implements broadcasting to return a list of γ corresponding to the result computed by each party.

A correctness theorem for the above open_share_dumb function might look like this:

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

It says that if each party calls open_share_dumb on its share of x, and the broadcasting is done, then each party holds the sum of the shares of x.

We notice that the com datatype is actually a functor and we can therefore use the following function:

```
val fmap: (γ → δ) → com α β γ → com α β δ
```

Briefly, fmap takes a function of type $\gamma \to \delta$ and applies it to the return value of the communication.

The following theorem is useful to handle the fmaps in the correctness proofs:

```
val make_broadcast_fmap:
    f:(γ → δ) → x:llist (com α (llist α n) γ) n →
    Lemma (make_broadcast (List.Tot.map (fmap f) x) == List.Tot.map f (make_broadcast x))
```

Briefly, it says that applying fmap f on a communication and resolving the communication is the same as resolving the communication and applying f.

4.2 A false good idea: the coml datatype

When the protocol needs to communicate multiple times, we have to deal with types that look like com a b (com c d (com e f g)) which are not convenient. I thought it would be a good idea to write the list of send and receive types along with the return type, like this: coml [(a,b); (c,d); (e,f)] g, using the following function on types:

```
val coml: list (Type * Type) → Type → Type
let rec coml l ret =
  match l with
  | [] → ret
  | (hs,hr)::t → com hs hr (coml t ret)
```

And since fmap can be used to nest communication (by specifying δ , its type could be (c → com d e f) → com a b c → com a b (com d e f)), we can define a "nested fmap" function:

```
val fmapl: #l1:list (Type*Type) → #l2:list (Type*Type) →
           (α → coml l2 β) → coml l1 α → (coml (l1@l2) β)
```

Even if on the paper it looks nice, in practice it was a terrible idea since F* had a lot of trouble doing the type inference and in practice I spent more time dealing with the communication types than doing the actual proofs. I ended up removing coml and fmapl and things worked a lot better.

4.3 Functional specification

The protocol to add one share of two shared values is done using the following function:

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

Since this protocol is done locally without any communication, it takes as arguments two shares and returns a share.

The protocol to open a shared value has a more complex type:

```
val open_share: (elem (k+s) * auth_elem k s * auth_elem k s) →
    com_broadcast n (elem (k+s)) (
        com_broadcast n commitment_hidden (
            com_broadcast n (commitment_reveal * elem (k+s)) (
                option (elem k)
            )
        )
    )
```

It takes a share of the MAC key α, an authenticated share of a fresh random number r, and a share of the value to open x. It does a bunch of communication and returns an option (elem k). The reason for this return type is that the open protocol only reveals k-bits values, and the protocol can fail if the MAC check failed (i.e. an active attacker is trying to compromise the computation).

4.4 Correctness theorems

The correctness theorem for add_share_share looks like this:

```
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.Tot.index x i) (List.Tot.index y i)
      = List.Tot.index (add_shares_shares x y) i
    )
```

In other words, if the party i runs the protocol add_share_share with its shares of x and y then it gets the same share as the one described in the global protocol (add_shares_shares).

The correctness theorem for open_share looks like this:

```
val open_share_correct:
    alpha:shares n (k+s) → r:auth_shares n k s → x:auth_shares n k s → i:nat{i<n} →
    Lemma (
        let res = make_broadcast (make_broadcast (make_broadcast (
```

```
            List.Tot.map open_share (List.Pure.zip3 alpha r x)
        ))) in
        match List.Pure.index res i with
        | Some resi → resi == downcast k (combine_auth_shares x).v
        | None → ⊤
    )
```

It is quite similar to the one for open_share_dumb, but with more communication to resolve and an option to open.

5 Privacy-preserving machine learning, a wrong definition?

5.1 Preliminaries

The definition of PPML is that the server learns nothing about the input of the client, and the client learns nothing about the model of the server except the classification result on its input. But how much information can the client extract with this exception? We studied this in the context of an SVM classifier.

Consider an SVM server who knows a matrix F and a vector b. A client can make the request $R(x) = \arg\max_i (Fx+b)_i$. Using the result R, we explore how the client could extract as much information as possible on F and b.

We can first notice that R is invariant under the scaling of F and b.

5.2 Compute the sorting permutation of Fr for any vector r

The general idea: fix a vector x, compute $R(\alpha r + x)$ for enough α and you can get a bunch of inequalities like $(Fr)_i < (Fr)_j$. If you have enough inequalities you can compute the sorting permutation. Otherwise you change x and compute a new set of inequalities, until you have enough information to recover the sorting permutation.

In the rest of the section, x and r are fixed, and we use the abuse of notation: $R(\alpha) := R(\alpha r + x)$.

Lets define $h_i(\alpha) = (F(\alpha r + x) + b)_i$. Then, $R(\alpha) = \arg\max_i h_i(\alpha)$. We can notice that $h_i(\alpha) = \alpha(Fr)_i + (Fx + b)_i$ is actually an affine function. Hence for each y, $\{\alpha \mid R(\alpha) = y\}$ is an interval, and if $\alpha_1 < \alpha_2$ and $R(\alpha_1) \neq R(\alpha_2)$ then $(Fr)_{R(\alpha_1)} < (Fr)_{R(\alpha_2)}$ (which are the slopes of the affine functions). We can easily see this on a drawing of affine functions.

Therefore, there exists $\alpha_1 < \alpha_2 < \cdots < \alpha_k$ and distinct $y_0, y_1, \ldots, y_k$ such that

$$R(\alpha) = \begin{cases} y_0 & \text{if } \quad \alpha < \alpha_1 \\ y_1 & \text{if } \alpha_1 < \alpha < \alpha_2 \\ \cdots & \\ y_k & \text{if } \alpha_k < \alpha \end{cases}$$

and $(Fr)_{y_i} < (Fr)_{y_{i+1}}$.

Now we only need to compute this decomposition. Here is a way to do this:

- Find a lower-bound for α_1 (e.g. compute $R(-10^k)$ for $0 < k < 10$ and suppose α_1 is not too low)
- Find a good approximation of α_1 using binary search (supposing you also know a higher bound)
- Use the same process for α_2 and so on

In fact to get the order, we do not need to know the α_i precisely, it suffices to know $\beta_0 < \cdots < \beta_k$ such that $R(\beta_i) = y_i$ to have a proof that $(Fr)_{y_i} < (Fr)_{y_{i+1}}$.

From a practical point of view, we might wonder two things: how many SVM queries do we need to find the α_i and y_i for a given x, and how much information it exposes. In Figure 3, we can see a histogram of the number of queries made to the SVM to get the α_i and y_i for a given x. This is a multimodal distribution and the different modes actually correspond to the number of reconstructed y_i. We see that we need about 100 queries per iteration. In Figure 4, we plot the percentage of knowledge we have about the sorting permutation of Fr, as a function of the number of tested x. Each x gives some relation $(Fr)_{y_i} < (Fr)_{y_{i+1}}$ and we compute the transitive closure of these relations to get the percentage of knowledge we have about the sorting permutation of Fr. This can be seen as the percentage of (i,j) for which we know the relationship between $(Fr)_i$ and $(Fr)_j$. We see that we need about 50 iterations to have 10% of knowledge, about 2000 iterations to have 50% of knowledge and 10^7 iterations gives about 90% of knowledge.

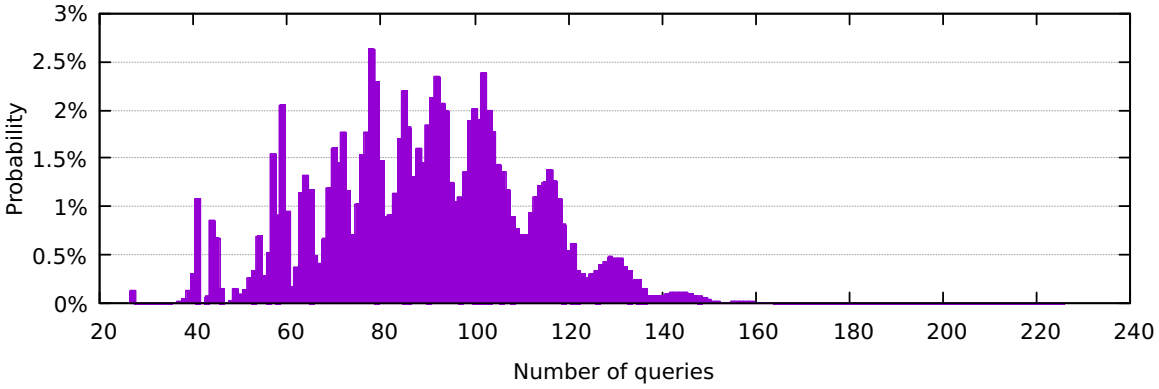

Figure 3: Number of queries to the SVM to reconstruct an $R(\alpha)$ function for a given x.

5.3 A simple usage of this first construction

If r^j is a vector such that

$$r_i^j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

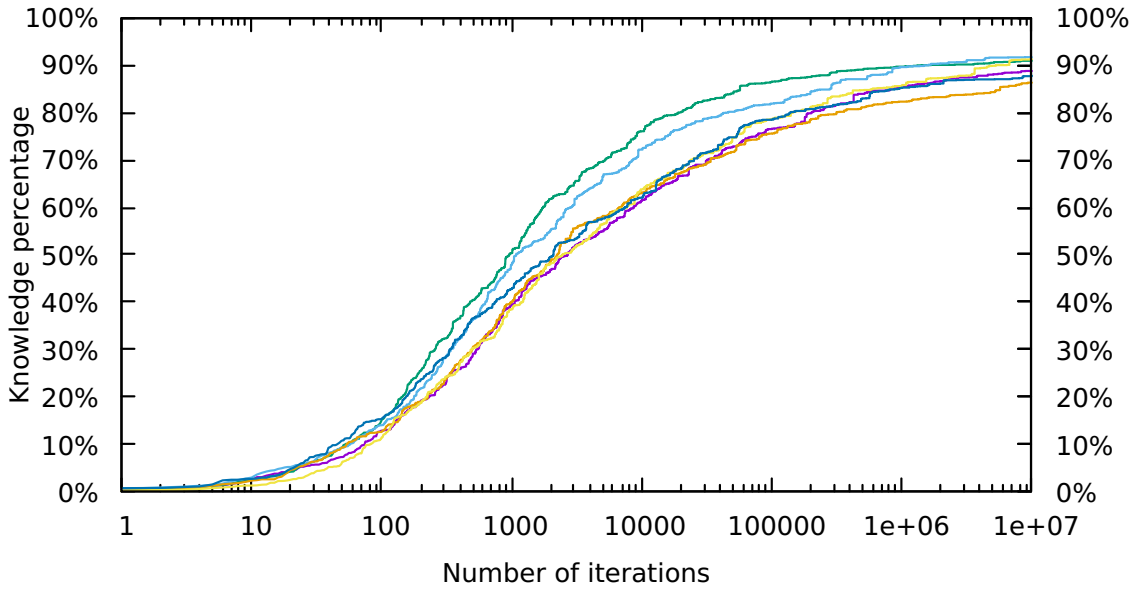

Figure 4: Evolution of the fraction of known value of $(Fr)_i < (Fr)_j$ in function of the number of random x vector tested, on several random SVM models.

then $(Fr^j)_i = f_{i,j}$, so the previous construction gives the sorting permutation of every column of F.

5.4 A more advanced usage

If $r^{j,k,\alpha}$ is a vector such that

$$r_i^{j,k,\alpha} = \begin{cases} 1 & \text{if } i = j \\ \alpha & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

then $(Fr^{j,k,\alpha})_i = f_{i,j} + \alpha f_{i,k}$, which is an affine function in α.

Fix some a, b, for every α with the construction you can know if $(Fr^{j,k,\alpha})_a < (Fr^{j,k,\alpha})_b$. If $f_{a,k} \neq f_{b,k}$ then you can find $\alpha_{a,b}$ such that $(Fr^{j,k,\alpha_{a,b}})_a = (Fr^{j,k,\alpha_{a,b}})_b$ (using binary search).

From this equality, you can deduce $f_{a,j} - f_{b,j} = \alpha_{a,b}(f_{b,k} - f_{a,k})$. It means that if you know a column well, you can have a lot of information on the other columns.

Note that in this usage we do not need to know exactly the sorting permutation of $(Fr^{j,k,\alpha})$, knowing a lot of inequalities might be sufficient.

I tried to generalize this usage with two parameters: $r^{j,k,l,\alpha,\beta}$ but that didn't give anything interesting.

6 Future work

6.1 Security proofs

The protocol implementation is currently proved to be correct, but not proved to be secure (although the SPD$\mathbb{Z}_{2^k}$ paper [4] proves its security by hand). We could use some variant of the Dolev-Yao model, or Wys$*$ [8], a DSL that models MPC protocols in F$*$, to prove its security formally.

6.2 Low-level implementation

The low-level specification can be extracted to a functional language such as OCaml but this implementation might be slow. A low-level implementation could be written in Low$*$ [7], a subset of F$*$ that is memory-safe and can be extracted to readable C code.

6.3 Implementation of the preprocessing phase

The SPD$\mathbb{Z}_{2^k}$ protocol depends on the preprocessing of correlated random numbers, to authenticate new values and to multiply two shared values. This is done using an oblivious transfer protocol, and could be implemented as future work.

7 Conclusion

During this internship, I learned to use F$*$ and used it to implement a part of a novel MPC protocol.

Previously, I had some experience with tactic-based provers such as Coq or HOL4. It was interesting to learn F$*$ with its SMT-based proofs, which were a lot different than my previous experience with theorem provers. In the end, once I found the right definition and the right theorem statements, the proofs could be very short, so the proof files looked at lot cleaner than Coq or HOL4 proofs.

This internship was not done in the best conditions: during the confinement and a work-at-home situation, and I managed to break my leg the first week after the confinement. Hopefully, I had the chance to have accommodating supervisors and roommates so this internship went well.

During this internship, I realized that an internship is not only about thinking hard with your brain and hammering a keyboard with your fingers, but also discovering new people, chatting during a coffee break or during lunch. I missed the social aspect of the internship.

References

[1] CVE-2016-7054. MITRE, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054.

[2] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*, May 2017.

[3] Hanno Boeck. Wrong results with Poly1305 functions, 2016. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413.

[4] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018. https://eprint.iacr.org/2018/482.

[5] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. Cryptology ePrint Archive, Report 2019/599, 2019. https://eprint.iacr.org/2019/599.

[6] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019.

[7] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, September 2017.

[8] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A DSL for verified secure multi-party computations. In Flemming Nielson and David Sands, editors, *8th International Conference on Principles of Security and Trust (POST)*, volume 11426 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2019.

[9] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. https://eprint.iacr.org/2014/140.

[10] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536, 2017. https://eprint.iacr.org/2017/536.