The SPDZ Protocol, Part 1

Secure Computation using Precomputed Triples

by Morten Dahl on September 3, 2017

This post is still very much a work in progress.

TL;DR: *this is the first in a series of posts explaining a state-of-the-art protocol for secure computation.*

In this blog post we'll go through the state-of-the-art SPDZ protocol for secure computation. Unlike the protocol used in a previous blog post (/2017/04/17/private-deep-learning-with-mpc/), SPDZ allows us to have as few as two parties computing on private values. Moreover, it has received significant scientific attention over the last few years and as a result several optimisations are known that can used to speed up our computation.

In this series we'll go through and describe the state-of-the-art SPDZ protocol for secure computation. Unlike the protocol used in a previous blog post (/2017/04/17/private-deep-learning-with-mpc/), SPDZ allows us to have as few as two parties computing on private values and it allows us to move parts of the computation to an *offline* phase in order to gain a more performant *online* phase. Moreover, it has received significant scientific attention over the last few years that resulted in various optimisations and efficient implementations.

The code for this section is available in this associated notebook (https://github.com/mortendahl/privateml/blob/master/image-analysis/Basic%20SPDZ.ipynb).

Background

The protocol was first described in SPZD'12 (https://eprint.iacr.org/2011/535) and DKLPSS'13 (https://eprint.iacr.org/2012/642), but have also been the subject of at least one series of blog posts (https://bristolcrypto.blogspot.fr/2016/10/what-is-spdz-part-1-mpc-circuit.html). Several implementations exist, including one (https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/) from the cryptography group (http://www.cs.bris.ac.uk/Research/CryptographySecurity/) at the University of Bristol providing both high performance and full active security.

As usual, all computations take place in a finite ring, often identified by a prime modulus `Q`. As we will see, this means we also need a way to encode the fixed-point numbers used by the CNNs as integers modulo a prime, and we have to take care that these never "wrap around" as we then may not be able to recover the correct result.

Moreover, while the computational resources used by a procedure is often only measured in time complexity, i.e. the time it takes the CPU to perform the computation, with interactive computations such as the SPDZ protocol it also becomes relevant to consider communication and round complexity. The former measures the number of bits sent across the network, which is a relatively slow process, and the latter the number of synchronisation points needed between the two parties, which may block one of them with nothing to do until the other catches up. Both hence also have a big impact on overall executing time.

Concretely, we have an interest in keeping `Q` is small as possible, not only because we can then do arithmetic operations using only a single word sized operations (as opposed to arbitrary precision arithmetic which is significantly slower), but also because we have to transmit less bits when sending field elements across the network.

Note that while the protocol in general supports computations between any number of parties we here present it for the two-party setting only. Moreover, as mentioned earlier, we aim only for passive security and assume a crypto provider that will honestly generate the needed triples.

Note that while the protocol in general supports computations between any number of parties we here use and specialise it for the two-party setting only. Moreover, as mentioned earlier, we aim only for passive security and assume a crypto provider that will honestly generate the needed triples.

Setting

We will assume that the training data set is jointly held by a set of *input providers* and that the training is performed by two distinct *servers* (or *parties*) that are trusted not to collaborate beyond what our protocol specifies. In practice, these servers could for instance be virtual instances in a shared cloud environment operated by two different organisations.

The input providers are only needed in the very beginning to transmit their training data; after that all computations involve only the two servers, meaning it is indeed plausible for the input providers to use e.g. mobile phones. Once trained, the model will remain jointly held in encrypted form by the two servers where anyone can use it to make further encrypted predictions.

For technical reasons we also assume a distinct *crypto producer* that generates certain raw material used during the computation for increased efficiency; there are ways to eliminate this additional entity but we won't go into that here.

Finally, in terms of security we aim for a typical notion used in practice, namely *honest-but-curious (or passive) security*, where the servers are assumed to follow the protocol but may otherwise try to learn as much possible from the data they see. While a slightly weaker notion than *fully malicious (or active) security* with respect to the servers, this still gives strong protection against anyone who may compromise one of the servers *after* the computations, despite what they do. Note that for the purpose of this blog post we will actually allow a small privacy leakage during training as detailed later.

Secure Computation with SPDZ

Sharing and reconstruction

Sharing a private value between the two servers is done using the simple additive scheme (/2017/06/04/secret-sharing-part1/#additive-sharing). This may be performed by anyone, including an input provider, and keeps the value perfectly private (https://en.wikipedia.org/wiki/Information-theoretic_security) as long as the servers are not colluding.

```
def share(secret):
    share0 = random.randrange(Q)
    share1 = (secret - share0) % Q
    return [share0, share1]
```

And when specified by the protocol, the private value can be reconstruct by a server sending his share to the other.

```
def reconstruct(share0, share1):
    return (share0 + share1) % Q
```

Of course, if both parties are to learn the private value then they can send their share simultaneously and hence still only use one round of communication.

Note that the use of an additive scheme means the servers are required to be highly robust, unlike e.g. Shamir's scheme (/2017/06/04/secret-sharing-part1/) which may handle some servers dropping out. If this is a reasonable assumption though, then additive sharing provides significant advantages.

```
class PrivateValue:

    def __init__(self, value, share0=None, share1=None):
        if not value is None:
            share0, share1 = share(value)
        self.share0 = share0
        self.share1 = share1

    def reconstruct(self):
        return PublicValue(reconstruct(self.share0, self.share1))
```

Linear operations

Having obtained sharings of private values we may next perform certain operations on these. The first set of these is what we call linear operations since they allow us to form linear combinations of private values.

The first are addition and subtraction, which are simple local computations on the shares already held by each server. And if one of the values is public then we may simplify.

```
class PrivateValue:

    ...

    def add(x, y):
        if type(y) is PublicValue:
            share0 = (x.share0 + y.value) % Q
            share1 =  x.share1
            return PrivateValue(None, share0, share1)
        if type(y) is PrivateValue:
            share0 = (x.share0 + y.share0) % Q
            share1 = (x.share1 + y.share1) % Q
            return PrivateValue(None, share0, share1)

    def sub(x, y):
        if type(y) is PublicValue:
            share0 = (x.share0 - y.value) % Q
            share1 =  x.share1
            return PrivateValue(None, share0, share1)
        if type(y) is PrivateValue:
            share0 = (x.share0 - y.share0) % Q
            share1 = (x.share1 - y.share1) % Q
            return PrivateValue(None, share0, share1)
```

```
x = PrivateValue(5)
y = PrivateValue(3)

z = x + y
assert z.reconstruct() == 8
```

Next we may also perform multiplication with a public value by again only performing a local operation on the share already held by each server.

```
class PrivateValue:

    ...

    def mul(x, y):
        if type(y) is PublicValue:
            share0 = (x.share0 * y.value) % Q
            share1 = (x.share1 * y.value) % Q
            return PrivateValue(None, share0, share1)
```

Note that the security of these operations is straight-forward since no communication is taking place between the two parties and hence nothing new could have been revealed.

```
x = PrivateValue(5)
y = PublicValue(3)

z = x * y
assert z.reconstruct() == 15
```

Multiplication

Multiplication of two private values is where we really start to deviate from the protocol used previously (/2017/04/17/private-deep-learning-with-mpc/). The techniques used there inherently need at least three parties so won't be much help in our two party setting.

Perhaps more interesting though, is that the new techniques used here allow us to shift parts of the computation to an *offline phase* where *raw material* that doesn't depend on any of the private values can be generated at convenience. As we shall see later, this can be used to significantly speed up the *online phase* where training and prediction is taking place.

This raw material is popularly called a *multiplication triple* (and sometimes *Beaver triple* due to their introduction in Beaver'91 (https://scholar.google.com/scholar?cluster=14306306930077045887)) and consists of independent sharings of three values `a`, `b`, and `c` such that `a` and `b` are uniformly random values and `c == a * b % Q`. Here we assume that these triples are generated by the crypto provider, and the resulting shares distributed to the two parties ahead of running the online phase. In other words, when performing a multiplication we assume that `Pi` already knows `a[i]`, `b[i]`, and `c[i]`.

```
def generate_mul_triple():
    a = random.randrange(Q)
    b = random.randrange(Q)
    c = (a * b) % Q
    return PrivateValue(a), PrivateValue(b), PrivateValue(c)
```

Note that a large portion of efforts in current (https://eprint.iacr.org/2016/505) research (https://eprint.iacr.org/2017/1230) and the full reference implementation (https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/) is spent on removing the crypto provider and instead letting the parties generate these triples on their own; we won't go into that here but see the resources pointed to earlier for details.

To use multiplication triples to compute the product of two private values `x` and `y` we proceed as follows. The idea is simply to use `a` and `b` to respectively mask `x` and `y` and then reconstruct the masked values as respectively `alpha` and `beta`. As public values, `alpha` and `beta` may then be combined locally by each server to form a sharing of `z == x * y`.

```
class PrivateValue:

    ...

    def mul(x, y):
        if type(y) is PublicValue:
            ...
        if type(y) is PrivateValue:
            a, b, a_mul_b = generate_mul_triple()
            # local masking followed by communication of the reconstructed values
            alpha = (x - a).reconstruct()
            beta  = (y - b).reconstruct()
            # local re-combination
            return alpha.mul(beta) + \
                   alpha.mul(b) + \
                   a.mul(beta) + \
                   a_mul_b
```

If we write out the equations we see that `alpha * beta == xy - xb - ay + ab`, `a * beta == ay - ab`, and `b * alpha == bx - ab`, so that the sum of these with `c` cancels out everything except `xy`. In terms of complexity we see that communication of two field elements in one round is required.

Finally, since `x` and `y` are perfectly hidden (https://en.wikipedia.org/wiki/Information-theoretic_security) by `a` and `b`, neither server learns anything new as long as each triple is only used once. Moreover, the newly formed sharing of `z` is "fresh" in the sense that it contains no information about the sharings of `x` and `y` that were used in its construction, since the sharing of `c` was independent of the sharings of `a` and `b`.

Encoding Values

Signed integers

```
def encode_integer(integer):
    element = integer % Q
    return element

def decode_integer(element):
    integer = element if element <= Q//2 else element - Q
    return integer
```

Fixedpoint numbers

The last step is to provide a mapping between the rational numbers used by the CNNs and the field elements used by the SPDZ protocol. As typically done, we here take a fixed-point approach where rational numbers are scaled by a fixed amount and then rounded off to an integer less than the field size `Q`.

```
def encode(rational, precision=6):
    upscaled = int(rational * 10**precision)
    field_element = upscaled % Q
    return field_element

def decode(field_element, precision=6):
    upscaled = field_element if field_element <= Q/2 else field_element - Q
    rational = upscaled / 10**precision
    return rational
```

In doing this we have to be careful not to “wrap around” by letting any encoding exceed `Q`; if this happens our decoding procedure will give wrong results.

To get around this we’ll simply make sure to pick `Q` large enough relative to the chosen precision and maximum magnitude. One place where we have to be careful is when doing multiplications as these double the precision. As done earlier we must hence leave enough room for double precision, and additionally include a truncation step after each multiplication where we bring the precision back down. Unlike earlier though, in the two server setting the truncation step can be performed as a local operation as pointed out in SecureML (https://eprint.iacr.org/2017/396).

```
def truncate(x, amount=6):
    y0 = x[0] // 10**amount
    y1 = Q - ((Q - x[1]) // 10**amount)
    return [y0, y1]
```

With this in place we are now (in theory) set to perform any desired computation on encrypted data.

Next Steps

← PREVIOUS POST (/2017/08/13/SECRET-SHARING-PART3/)

NEXT POST → (/2017/09/10/THE-SPDZ-PROTOCOL-PART2/)

● (/feed.xml) ● (https://twitter.com/mortendahlcs)

● (https://github.com/mortendahl)

● (https://www.linkedin.com/in/mortendahlcs)

● (mailto:mortendahlcs@gmail.com)

Copyright © 2019