A Note on Packed Secret Sharing and Implementations

Gabrielle Beck

June 19, 2022

Packed secret sharing is a technique that was introduced by Franklin and Yung [4] and is often useful in honest majority MPC protocols. The main purpose of this note is to briefly discuss what packed secret sharing is, the kinds of computation it supports, and to discuss some implementation decisions one can make when implementing in practice. We'll first give the *briefest* refresher imaginable for what MPC is so you (hopefully) don't need to look at too many outside resources. Then we'll discuss packed secret sharing (PSS) and finally talk lightly about some implementation details. This is meant to accompany and give some background to the github code in the repo this document is located in. The code is not precisely clean enough to easily grasp and would probably require the reader to spend some time looking at other repos outside this one to understand it fully without this note. And if you still need to look at stuff after this, hopefully the reference section should guide that search.

1 What is MPC? Why do we care?

MPC or multiparty computation is a technique for computing the output of a function *privately* so that it's not possible for any party to learn about another party's input, beyond some unavoidable leakage that comes from the utility of the function itself. [1]. One of the classic examples of an MPC

[1] This is important to remember and is probably not spoken about enough - the formal definition of an MPC protocol does allow leakage in a party's input that depends on the known inputs and the output of the function. I.e. the adversary is allowed to output any arbitrary function of its view and that includes the input of corrupted parties

protocol is the so called "Millionaires Problem" where a group of people want to figure out who makes the most money, but they don't want anyone else to know exactly how much they're making. They can use an MPC protocol to figure this out without anyone ever having to say they make 30k/500k for example. They can also be used for doing mutual matchmaking, i.e. you can do a protocol with someone else to see if you mutually like one another with the guarantee the other party won't know you said yes if they said no. In larger scale applications, they can be used to compute aggregate statistics on medical data, to train machine learning models or to utilize ML algorithms on private input, and so on.

MPC protocols can be differentiated on many dimensions like the types of adversaries that are protected against, the output of the function when malicious parties are present, etc. Our main concern in this document is going to be the number of adversaries tolerated by the MPC protocol. In general, this number can be less than half the people who are participating or greater than half. A protocol that is secure when the number of adversaries is less than half is called an *honest majority* protocol. Protocols that remain secure for more parties than this are called *dishonest majority* protocols. This distinction is usually very important when discussing MPC, since the protocols tend to differ widely at even the most basic of levels for these methods. For example, typically, in a dishonest majority protocol meant to be conducted among n parties, every party has *additive* shares of some private value $[x]$ so that party P_i has share x_i and

$$x = \bigoplus_{i=1}^{n} x_i \tag{1}$$

One can see in this case that even if $n - 1$ parties are adversarial, if the input of the last party is unknown then the output x is also unknown. More specifically, if the distribution of the last value is uniform and unknown then x is perfectly hidden (this is not a rigorous argument, just to give some general intuition). On the other hand, honest majority protocols tend to use Shamir secret sharing. Shamir secret sharing is a secret-sharing technique that uses polynomials and takes advantage of the fact that n points uniquely defines a polynomial of degree less than or equal to $n - 1$. In it's most basic form, n parties hold a sharing of a degree t polynomial $p(x)$ where the secret value is located at $p(e)$ and their sharings consist of the evaluation of that polynomial at set points $\alpha_1 ... \alpha_n$. The most common way to do this is to have

$e = 0$ and $\alpha_i = i \; \forall i \in [n]$ so that the secret is $p(0)$ and party P_i has $p(i)$. Any group of t parties is unable to uncover the secret, while any group of greater than or equal to $t + 1$ can recover the secret using Lagrange interpolation: if the points each party holds are given as (α_i, y_i) interpolation is done by computing

$$f(x) = \sum_{i=1}^{k} y_i \cdot L_i(x) \tag{2}$$

where $L_i(x)$ is a function such that $\forall j \neq i$, $L_i(\alpha_j) = 0$ and $L_i(\alpha_i) = 1$. It reminds me actually of kronecker delta although that's probably inaccurate. The exact form of L_i is

$$L_i(x) = \prod_{\forall j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} \tag{3}$$

You *can* use more than $t+1$ points to reconstruct and if all shares provided are correct the polynomial will be of the correct degree, but typically this is not done when conducting an MPC protocol: in the limited number of works I have seen, exactly $t+1$ points are used and then the points provided by other parties are checked for correctness by evaluating the interpolated polynomial on those points. Of course, some of the points you use for interpolation can also be malicious but at this point you know that at least one party must have provided an incorrect sharing and you can potentially abort the protocol.

Of course, other sharing schemes exist like replicated secret sharing. But, I would personally argue, the most common sharing schemes are these two and traditionally are used in the settings described above (in dishonest and honest majority protocols respectively) even if it is by no means a necessity.

One final question you may have at this point, if you are new to MPC entirely, is why do we need to discuss secret sharing techniques at all? How do they relate to MPC protocols? There are a couple of ways to think about MPC protocols conceptually. Folks who work purely in MPC tend to look for protocols to compute MPC generically: that is they are trying to come up with protocols Π that can be used to securely compute *any* function f. The common approach for this is to treat the function f we want to compute as a circuit C, either boolean or arithmetic. The general structure of an MPC is then split among three different phases: every party secret shares their input, we evaluate the gates in the circuit privately while maintaining the invariant that each party holds correct sharings of the input and output wires, and then we reveal the output. The steps of correctly and privately computing the output of gates, in the most efficient way possible, is really the meat of an MPC protocol. If you're interested, I would recommend taking a look at MPC literature (maybe starting with the MPC protocols BGW, GMW, and BMR). However, that's not the main topic of this note, so we'll table a discussion of that going forward.

2 Packed Secret Sharing

Packed secret sharing, aka PSS, can be seen as a generalization of shamir secret sharing. Instead of simply embedding a secret value at $e = 0$, we have a set of secret points $e_1 ... e_\ell$ so that the secret s_i is embedded at point e_i. This allows a single shared polynomial to hide multiple secret values instead of just one. This has some rather serious implications on how the MPC protocol is conducted and changes some of the most basic aspects of the secret sharing scheme itself. To address the former point, in general packed secret sharing is good for SIMD style computation. In short, it's not natively easy to do different operations in different positions of a packed share: you must do the same operation in all locations. For example, adding two packed shares with respective secret vectors $(s_1 \ldots s_\ell)$ and $(\bar{s_1} \ldots \bar{s_\ell})$ adds every component coordinate wise so the result is a secret sharing of $(s_1 + \bar{s_1} \ldots s_\ell + \bar{s_\ell})$. There are ways around the inefficiencies that result from this, see [3] and the related work cited therein. Most techniques involve modifying the circuit to withstand the SIMD requirement from earlier or looking for inherent repetitiveness in the circuit itself.

Addressing the latter point, whereas in regular secret sharing the degree of our polynomial is *equal* to the privacy threshold, in PSS there is a decoupling of these variables. The degree of the sharing is given by D, t is once again the traitor threshold, and ℓ represents the number of secrets embedded in a single secret sharing. How D relates to t and ℓ may differ depending on the protocol. In general, D must be at least $t + l - 1$, although it may be larger. The important point to remember is that while $D + 1$ parties can reconstruct, security only holds for at most t malicious actors.

3 Implementing Shamir Secret Sharing and PSS

To implement a secret sharing scheme we need a way of both secret-sharing private values and a way to do recovery when given the correct number of points. There are a few main ways I know of for how to implement these methods in software: one of them uses regular lagrangian interpolation split into a preprocessing and online phase, the other uses FFT and carefully chooses the x coordinates assigned to each party. Both methods and their efficiencies have been discussed and/or implemented in different contexts [5, 2, 1]. I'm not saying anything new here, just restating in a different way what's been observed by others explicitly applied to the setting of packed secret sharing.

3.1 "Vanilla" Lagrange Interpolation

One of the most simple methods of implementing shamir sharing is to use the concept of lagrange interpolation from earlier. Commonly in academic papers, if we want to share secret points $s_1 \dots s_\ell$, embedded at x coordinates $e_1 ... e_\ell$ in a scheme with degree D polynomials, we construct the polynomial $p(x)$ according to the equation below:

$$p(x) = q(x) \prod_{i=1}^{\ell} (x - e_i) + \sum_{i=1}^{\ell} s_i L_i(x) \tag{4}$$

Where $q(x)$ is a random degree $D - l$ polynomial and $L_i(x)$ is the polynomial of equation (3) applied to the points $e_1 \dots e_\ell$. i.e. $L_i(e_i) = 1$ and $L_i(e_j) = 0$ for $j \neq i$. Finally once we have $p(x)$, we evaluate it on points $\alpha_1 \dots \alpha_n$ to get the actual secret share values we need to hand out to other parties.

It's clear to see from (4) that correctness is satisfied so that for all i, $p(e_i) = s_i$. Security follows because any given subset of at most t points $(\alpha_{i_1}, y_{i_1}), \dots (\alpha_{i_t}, y_{i_t})$ is consistent with all polynomials of degree $t + \ell - 1$ that have *any* set of secret values $(e_1, s_1) \dots (e_1, s_\ell)$. The t points do not place any constraints or correlation among the ℓ secrets.

The only problem is this technique is not very efficient at all. We can pre-compute some of these values like the product $\prod_{i=1}^{l} (x - e_i)$ and the polynomials $L_i(x)$. However, that still leaves a polynomial multiplication and at least ℓ^2 multiplications to recover $p(x)$ and then n evaluations of $p(x)$ to finally get the relevant points. One way that we can do better is to take the pre-computation idea a step further. First, we make sure we sample $p(x)$ in point evaluation form rather than by coefficients. We know that $p(x)$ should be a degree D polynomial, so we can represent it with $D+1$ pairs $\{(x_i, y_i)\}_{i\in[D+1]}$. The benefits to this form is it makes constructing the polynomial very easy: just randomly sample $D+1-\ell$ points, add them to the set of secret points, and we have $p(x)$. Even better, these $D+1-\ell$ points can actually be the $p(\alpha_1)\dots p(\alpha_{D+1-\ell})$ we need to hand out to other parties. And we got them without having to do any evaluations at all! We still need to do evaluation on $p(\alpha_{D+1-\ell+1})\dots p(\alpha_n)$ in order to properly secret share. Fortunately, we don't need to actually recover the coefficients of $p(x)$ to do this. Let's look back at the lagrangian interpolation formula from earlier and consider evaluation of $p(x)$ on the point α_j:

$$p(\alpha_j) = \sum_{i=1}^{k} y_i \cdot L_i(\alpha_j) \tag{5}$$

If we always use the same x coordinates $(e_1 \dots e_\ell, \alpha_1, \dots \alpha_{D+1-\ell})$ for defining our deg D polynomial, then $L_i(\alpha_j)$ are fixed field elements we can compute for a one time cost to secret share multiple values. To be precise, we compute the following matrix and store it somewhere in memory:

$$\begin{bmatrix} L_1(\alpha_{D+1-\ell+1}) & \dots & L_{D+1}(\alpha_{D+1-\ell+1}) \\ \vdots & \ddots & \\ L_1(\alpha_N) & \dots & L_{D+1}(\alpha_N) \end{bmatrix}$$

When we want to secret share the vector $(s_1 \dots s_\ell)$ we sample $D+1-\ell$ random points $y_i \leftarrow \mathbb{F}$ for $i \in [D+1-\ell]$ and compute

$$\begin{bmatrix} L_1(\alpha_{D+1-\ell+1}) & \dots & L_{D+1}(\alpha_{D+1-\ell+1}) \\ \vdots & & \\ \vdots & \ddots & \\ \vdots & & \\ L_1(\alpha_N) & \dots & L_{D+1}(\alpha_N) \end{bmatrix} \begin{bmatrix} s_1 \\ \vdots \\ s_\ell \\ y_1 \\ \vdots \\ y_{D+1-\ell} \end{bmatrix} = \begin{bmatrix} y_{D+1-\ell+1} \\ \vdots \\ \vdots \\ \vdots \\ y_N \end{bmatrix}$$

The secret shared values are then $(\alpha_1, y_1) \dots (\alpha_N, y_N)$. This makes the cost of secret sharing sampling some random values and then multiplying a vector by a matrix of size $(N - (D + 1 - \ell)) \times D + 1$.

3.1.1 Quick note on Recovery

For recovery we do something similar to the steps of secret sharing. We pre-compute a matrix of similar size accept the points we use are different: our interpolation is defined with respect to $\alpha_1 \dots \alpha_{D+1}$ and the points we want to recover are located at $e_1 \dots e_\ell$ and $\alpha_{D+2} \dots \alpha_N$. The y-coordinates of $\alpha_{D+2} \dots \alpha_N$ are recovered in order to check that the points supplied by all parties are consistent. Notice this is not a robust method of recovery, but in some scenarios just knowing that at least one party supplied an incorrect share is enough (think security with abort). If not, you can look into using something like Berlekamp-Welch.

3.1.2 Asymptotic and Concrete Complexity Concerns

One problem with this method is that it requires a good amount of storage when we are secret sharing with large numbers of parties N and also requires a lot of time in the pre-computation step as well. In terms of asymptotics, the cost of our online phase is also $O(N^2)$. It's possible we could do better than this, at least asymptotically - which leads to our next method...

3.2 FFT

As earlier works have noted, there is a way that we can do faster computation in the online phase and also have smaller storage requirements [7, 5]. This method involves utilizing the Fast Fourier transform, more specifically the discrete variant. In order to use DFT techniques, we choose $e_1, \dots e_\ell, \alpha_1, \dots \alpha_N$ so that they are all powers of an m^{th} root of unity where $m = 2^j$ for some j. At a high level, this choice makes some computations faster, as work that is done to evaluate on some particular point can be reused to compute other points - so much so that the overall asymptotic complexity for both computation and storage will now be $O(N\log N)$.

Hang on, because now it's time to get into the weeds. Let's knock out an easy question first: how are we going to use this technique in some random arbitrary field $\mathbb{F}$? Isn't there no guarantee that $\mathbb{F}$ will have a m^{th} root of unity? Good question! You're right, so we won't be using an arbitrary field and instead we'll construct one ourselves using $\mathbb{Z}_p$. In general cryptography, using $\mathbb{Z}_p$ is not really a good idea anymore (mainly if you're relying on it for any type of discrete log assumption or are worried about being confined to small subgroups). But this shouldn't be a problem for MPC. We'll pick a bound such that for all the values on N and ℓ that we care about $z \geq \lceil \log_2(N + \ell) \rceil$. Then we can choose a p so that 2^z divides $p - 1$. This will allow us to find a generator h of the subgroup H of $(\mathbb{Z}_p^*, \cdot)$ where the size of H is 2^z [2].And voila! If $j = z$, h is our root of unity. If $j < z$, we instead use $g = h^{2^{z-j}}$.

We'll now discuss briefly how DFT works and can let us quickly evaluate roots of unity on a polynomial. The main idea is to use a divide and conquer approach and reuse the answers to sub-problems across multiple computations.

Say we want to evaluate the following degree 2^j-1 polynomial

$$f(x) = \sum_{i=0}^{2^j-1} a_i x^i \tag{6}$$

on $1, \omega, \omega^1, \dots \omega^{2^j-1}$ where ω is an $m = 2^{j^{th}}$ root of unity. Then we rewrite $f(x)$ as the combination of two smaller polynomials half the size of f:

$$f_{even}(x) = \sum_{i=0}^{2^{j-1}-1} a_{2i} x^i \tag{7}$$

$$f_{odd}(x) = \sum_{i=0}^{2^{j-1}-1} a_{2i+1} x^i \tag{8}$$

One can see that f can be computed from f_{even} and f_{odd} using the following equation:

$$f(x) = f_{even}(x^2) + x f_{odd}(x^2) \tag{9}$$

Okay, so what you might ask? How does this help us? Remember that we want to compute f on the set $\{\omega^i\}_{i \in \{0 \dots 2^j-1\}}$. The input to f_{even} and f_{odd} is not x but x^2. This means we have effectively halved our original problem: for

[2] To do this, just find a generator of $\mathbb{Z}_p^*$ and raise it to the power of $(p-1)/2^z$

any pair of values $\omega^k, \omega^{k+2^{j-1}}$, with $k \in \{0, ..., 2^{j-1}\}$ the result of evaluation on f_{odd} and f_{even} should be the *same*. Of course, this divide step can be recursed on until we get a degree 0 polynomial. The number of recursions that needs to be done in total is logarithmic in m and the overall runtime to evaluate m points is $m\log(m)$. If m is close to N, we get the bound from earlier.

There are some implementation concerns with FFT. Mathieu Poumeyrol has an in depth blog post covering different ways to implement it in Rust [6]. One of the things he comments on is that using recursive function calls doesn't necessarily play well with caching and the CPU (in terms of even the overhead coming from constructing new stack frames and where accesses are happening in the stack). We can even do better, and perform DFT *in place*. The key is that we re-order the coefficients a_i of f so that coefficient a_i where $i = b_0 \dots b_{j-1} \in \{0, 1\}^{j-1}$ is placed at index $b_{j-1} \dots b_0$. When we do this, $\forall z \in \{0, \dots, 2^{j-1}\}, i \in \{0, \dots, j-1\}$ the coefficients a_z, $a_{z+2^{j-1-i}}$ are at a distance of 2^i from one another in the array. If we want to evaluate so that the output of the array contains $f(\omega^0) \dots f(\omega^{2^j-1})$, at each iteration $i = 0 \dots j-1$ of a loop we do the following: consider those values at indices that are 2^i away from each other in the array. These positions in the array currently hold partial evaluations of $f(\omega^k)$ and $f(\omega^{k+2^i})$ from the previous $0 \dots i-1$ iterations. On iteration i, we are evaluating the sub-problem for ω^k that is $f_{even}((\omega^k)^{2^{j-i}}) + (\omega^k)^{2^{j-1-i}} \cdot f_{odd}((\omega^k)^{2^{j-i}})$ where f_{even} is some polynomial containing all coefficients in the set $\{2^{j-i} \cdot z + r | z \in \mathbb{Z}\}$ and f_{odd} contains coefficients in the set $\{2^{j-i} \cdot z + (r + 2^{j-i-1}) | z \in \mathbb{Z}\}$ for some $r \in \{0, \dots, 2^{j-i-1} - 1\}$. Luckily, position k in the array already contains the result of f_{even} from previous iterations and position $k + 2^i$ contains the result of f_{odd}! This is because $(\omega^k)^{2^{j-i}} = (\omega^{k+2^i})^{2^{j-i}}$ and the reordering we did earlier. So at each step, we only need to combine prior results and do a multiplication. An example of the rearrangement and partial calculations is given in Figure 1.

Stepping back, the reason we care about DFT is because we can modify the original Lagrange interpolation to use it instead of doing matrix multiplications: to secret share a regular shamir share, we just need to sample coefficients and do a single DFT computation to get all parties points. For recovery, we can use the method specified by Soro and Lacan [7]. Unfortunately, because we are using packed secret sharing, we cannot do secret sharing by sampling coefficients directly. Instead we need to first pick $D+1$ points, do the recovery step of Soro and Lacan and then do DFT. Still,

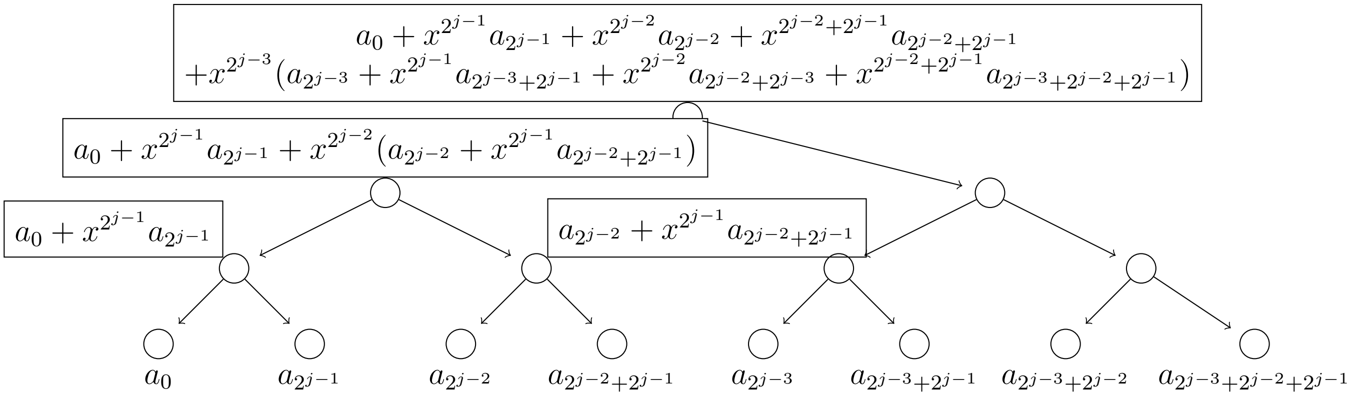
$a_0 + x^{2^{j-1}} a_{2^{j-1}} + x^{2^{j-2}} a_{2^{j-2}} + x^{2^{j-2}+2^{j-1}} a_{2^{j-2}+2^{j-1}}$
$+x^{2^{j-3}}(a_{2^{j-3}} + x^{2^{j-1}} a_{2^{j-3}+2^{j-1}} + x^{2^{j-2}} a_{2^{j-2}+2^{j-3}} + x^{2^{j-2}+2^{j-1}} a_{2^{j-3}+2^{j-2}+2^{j-1}})$
$a_0 + x^{2^{j-1}} a_{2^{j-1}} + x^{2^{j-2}}(a_{2^{j-2}} + x^{2^{j-1}} a_{2^{j-2}+2^{j-1}})$
$a_0 + x^{2^{j-1}} a_{2^{j-1}}$
$a_{2^{j-2}} + x^{2^{j-1}} a_{2^{j-2}+2^{j-1}}$
a_0
$a_{2^{j-1}}$
$a_{2^{j-2}}$
$a_{2^{j-2}+2^{j-1}}$
$a_{2^{j-3}}$
$a_{2^{j-3}+2^{j-1}}$
$a_{2^{j-3}+2^{j-2}}$
$a_{2^{j-3}+2^{j-2}+2^{j-1}}$

Figure 1: A tree, picturing the rearrangement of coefficients for $j = 3$ when calculating the FFT of $f(x) = \sum_{i=0}^{7} a_i x^i$ and some values that are produced at each level for $x = \omega^i$ for $i \in \{0, \dots, 3\}$.

asymptotically, this should be more efficient than the normal method that we described earlier. However, as noted by [5] the constants associated with FFT appear to be relatively large. Even attempting to get rid of explicitly rearranging coefficients for DFT, or doing other smaller optimizations (like picking points given to parties to ensure that DFT evaluation can use a smaller root of unity 2^{j-1} vs 2^j for some operations) is not enough to make FFT more efficient than the regular method for reasonable parameters. In some tests, for example, secret sharing still costs more in running time until around 500 parties are used. This number of parties is something that I would argue is still relatively unreasonable for current MPC applications. So regular secret sharing techniques are probably your best shot unless you are prepared to do some serious innovation.

References

[1] Honeybadger mpc implementation. `https://github.com/initc3/HoneyBadgerMPC`.

[2] Libscapi. `https://github.com/cryptobiu/libscapi`.

[3] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits.

In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 663–693. Springer, 2021.

[4] Matthew Franklin and Moti Yung. Communication complexity of secure computation. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 699–710, 1992.

[5] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.

[6] Mathieu Poumeyrol. Optimizing threshold secret sharing. `https://medium.com/snips-ai/optimizing-threshold-secret-sharing-c877901231e5`, 2016.

[7] Alexandre Soro and Jérôme Lacan. Fnt-based reed-solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5. IEEE, 2010.