Efficient polynomial evaluations with preprocessing and applications to cryptography

Grade: 9.5

Formal Methods in Computer Science and Engineering
Master's Thesis

Mario Carrillo Redondo
Supervisors: Dario Fiore and Ignacio Cascudo

Year 2023-2024

Contents

Abstract

In [20], Kedlaya and Umans proposed an algorithm that, after doing some preprocessing, enables fast evaluation of a multivariate polynomial. Recently, this algorithm has been used to achieve a breakthrough in cryptography, building an unkeyed DEPIR (Doubly Efficient Private Information Retrieval) in [22]. This construction allows, only after one run of a preprocessing algorithm on a database, efficient private access to it for any client.
The goal of this TFM is to realize an implementation (the first one to the best of our knowledge) of the algorithm by Kedlaya and Umans in order to understand its practical performance and also, to build a new polynomial commitment based on it. Our new polynomial commitment scheme allows the prover to compute proofs for evaluations in time sublinear in the size of the polynomial. We achieve this by committing to the data structure resulting from preprocessing the polynomial using a vector commitment scheme with logarithmic opening run-time. We also analyze the efficiency and the security of this construction, discuss its limitations and possible strenghtenings.

Key words
Cryptography, Polynomial Commitments, Vector Commitments, Fast Polynomial Evaluation, DEPIR

Abstract

En [20], Kedlaya y Umans propusieron un algoritmo que, después de realizar un preprocesamiento, permite la evaluación rápida de un polinomio multivariable. Recientemente, este algoritmo se ha utilizado para lograr un avance en criptografía, construyendo un DEPIR (Recuperación de Información Privada Doble y Eficiente) sin claves en [22]. Esta construcción permite, solo después de una ejecución de un algoritmo de preprocesamiento en una base de datos, el acceso privado y eficiente a la misma para cualquier cliente.
El objetivo de este TFM es realizar una implementación (la primera, según nuestro conocimiento) del algoritmo de Kedlaya y Umans para comprender su rendimiento práctico y también para construir un nuevo compromiso de polinomio basado en él. Nuestro nuevo esquema de compromiso de polinomio permite al probador computar pruebas para evaluaciones en un tiempo sublineal en el tamaño del polinomio. Logramos esto comprometiéndonos con la estructura de datos resultante del preprocesamiento del polinomio utilizando un esquema de compromiso vectorial con tiempo de apertura logarítmico. También analizamos la eficiencia y la seguridad de esta construcción, discutimos sus limitaciones y posibles fortalecimientos.

Palabras clave
Criptografía, Compromisos de Polinomio, Compromisos Vectoriales, Evaluación Rápida de Polinomios, DEPIR

Chapter 1

Introduction

1.1 Introduction to the algorithm

In [20] Kedlaya and Umans show how to preprocess polynomials $f(X_1, ..., X_m)$ over a ring R to create a data structure that allows us to later evaluate the polynomial on any given input $(\alpha_1, ..., \alpha_m) \in R^m$ extremely efficiently, in time that is sublinear in the description length of the polynomial. The result works over many types of rings R, including $R = \mathbb{Z}_q$ for an arbitrary $q \in \mathbb{N}$, as well as polynomial rings $R = \mathbb{Z}_q[Y]/(E_1(Y))$ or even $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$ for monic polynomials E_1, E_2 (and more). Assume $f(X_1, ..., X_m) \in R[X_1, ..., X_m]$ has individual degree $< d$ in each variable. The description of such polynomial consists of $N = d^m$ coefficients and hence evaluating it naively would take at least $\Omega(N \log |R|)$ time. The work in [20] shows how to preprocess such a polynomial in time $N \cdot O(m(\log m + \log d + \log\log |R|))^m \cdot poly(m, d, \log |R|)$ into a data structure of at most the above size, such that, on any future input $(\alpha_1, ..., \alpha_m) \in R^m$, we can use the data structure to evaluate $f(\alpha_1, ..., \alpha_m)$ in just $poly(m, d, \log |R|)$ time. This gives significant saving in time, at least in some select parameter ranges.

We include a sketch of the technique in [20]. Start with the special case of $R = \mathbb{Z}_q$. We reinterpret the polynomial $f(X_1, ..., X_m)$ over $\mathbb{Z}_q$ as a polynomial over the integers $\mathbb{Z}$, where the coefficients and the inputs are taken from the set $\{0, ..., q - 1\}$. The maximal value that this polynomial can take over the integers is $\leq M = d^m(q - 1)^{dm}$. Let $p_1, ..., p_h$ be the set of all primes $p_i \leq 16 \log M$, which ensures that $\prod_{i=1}^{h} p_i \geq M$. To evaluate the polynomial f over the integers, it suffices to evaluate it modulo each of the primes p_i separately and then reconstruct the answer over the integers using the Chinese Remainder Theorem (CRT [37]). So we reduced the problem of evaluating the polynomial f modulo q to that of evaluating it modulo p_i for a small set of $h = O(\log M)$ much smaller primes $p_i = O(\log M)$. Using this idea we can preprocess the polynomial as follows, consider constructing h tables, where the i-th table simply stores all the possible evaluations of the polynomial f on all p_i^m possible inputs $(\alpha_1, ..., \alpha_m)$ modulo p_i. This allows for extremely fast evaluation on any input $(\alpha_1, ..., \alpha_m) \in \mathbb{Z}_q^m$ just by looking up one entry in each table and then using CRT.

The result extends to rings $R = \mathbb{Z}_q[Y]/(E_1(Y))$, by reducing the problem over such rings to that over $\mathbb{Z}_r$ for some $r \gg q$ that depends on $|R|$. Instead of evaluating $f(\alpha_1, ..., \alpha_m)$ over R, we evaluate it over $\mathbb{Z}_r$ by taking all the coefficient/input ring elements and substituting $Y = M$ for some sufficiently large $M \in \mathbb{Z}$ and doing all the computation modulo r for some sufficiently large $r \gg M$, such that there is no wrap-around. The output is an integer whose base-M digits correspond to the coefficients of Y in the correct evaluation of $f(\alpha_1, ..., \alpha_m)$

over R. We can further extend the result to $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$ (and beyond) analogously.

1.2 Introduction to applications in cryptography

This algorithm has already been applied to solve an open problem in cryptography, the construction of a DEPIR (Doubly Efficient Private Information Retrieval) scheme, in [22]. Our theoretical contribution is a new application of this algorithm to build a polynomial commitment scheme which is efficient for the prover.

1.2.1 DEPIR

A (single server) private information retrieval (PIR [14]) scheme is a protocol between a server who holds a database $DB \in \{0, 1\}^N$ and a client who wishes to learn the i-th location $DB[i]$ of the database without revealing the index $i \in [N]$ to the server. For example, a client may want to retrieve a movie from Netflix without revealing to Netflix which one. A trivial solution is for the server to simply send the entire database DB to the client. The goal of PIR is to solve this problem with much lower communication complexity, which should be sub-linear in the database size N.

However, PIR comes with a major limitation. While it provides low communication complexity, it inherently requires the server to read the entire database DB during each protocol execution and therefore the computational complexity on the server side is at least linear in N. For example, using PIR, Netflix would need to perform a huge computation over its entire movie database to serve a single movie to a client. This limitation significantly restricts PIR's usefulness.

The DEPIR (doubly efficient PIR) gets around the above limitation and simultaneously provides low communication and server computation. To do so, it modifies the original setting by allowing the database $DB \in \{0, 1\}^N$ to be preprocessed into some static data structure $\widetilde{DB}$ that is stored on the server. This one-time preprocessing is allowed to run in time that is linear, or even slightly super-linear, in the database size N. Subsequently, any client can run a PIR protocol with the server to learn the value $DB[i]$ without revealing the index i, where both the communication and the server/client computation during the protocol are sub-linear in the database size N.

Depending on the nature of the preprocessing there are three versions of DEPIR: unkeyed, public-key and secret-key DEPIR. Unkeyed DEPIR is the simplest and strongest notion. In unkeyed DEPIR, the preprocessing algorithm that maps the database DB to the data structure $\widetilde{DB}$ is a deterministic function that the server executes on its own. Previous work in [10] and [7] gave the first constructions on keyed DEPIR, however they require a trusted party to generate the keys and to create the preprocessed data structure $\widetilde{DB}$.

The construction in [22], which is the one presented in section 4, makes use of the preprocessing algorithm by Kedlaya and Umans to achieve an unkeyed DEPIR construction under the standard ring learning with errors (RingLWE [23]) assumption (which is a well-studied hardness assumption). They also construct an updatable DEPIR that allows the server to update individual bits of the database DB and correspondingly update the preprocessed data structure $\widetilde{DB}$ in sub-linear time.

1.2.2 Prover-efficient polynomial commitment

Commitment schemes are fundamental components of many cryptographic protocols. A commitment scheme allows a committer to publish a value, called the *commitment*, which binds her to a message (*binding*) without revealing it (*hiding*). Later, she may *open* the commitment and reveal the committed message to a verifier, who can check that the message is consistent with the commitment.

In [15] they use the following informal example to explain it, consider the following game between two players P and V:

1. P wants to commit to a message m. To do so, he writes down m on a piece of paper, puts it in a box, and locks it using a padlock.
2. P gives the box to V.
3. If P wants to, he can later open the commitment by giving V the key to the padlock.

There are two basic properties of this game, which are essential to any commitment scheme:

- Having given away the box, P cannot anymore change what is inside. Hence, when the box is opened, we know that what is revealed really was the choice that P committed to originally. This is called the *binding property.*
- When V receives the box, he cannot tell what is inside before P decides to give him the key. This is called the *hiding property.*

Assume now we want to generate commitments using digital communication between players. There are several ways in which one can commit to a message:

1. Let g and h be two random generators of a group $\mathbb{G}$ of prime order p. In this scheme, known as Pedersen commitment [29], the commitment is $\mathcal{C}_{\langle g,h\rangle}(m,r) = g^m h^r$, where r is a randomly chosen value in $\mathbb{Z}_p$. Pedersen commitments are unconditionally hiding, and computationally binding under the assumption that the discrete logarithm (DL [3]) problem is hard in $\mathbb{G}$.
2. Let g and h be two random generators of a group $\mathbb{G}$ of prime order p. In this scheme, based on ElGamal encryption scheme [16], the commitment is $\mathcal{C}_{\langle g,h\rangle}(m,r) = (g^r, g^m h^r)$, where r is a randomly chosen value in $\mathbb{Z}_p$. ElGamal based commitments are computationally hiding, and perfectly binding under the assumption that the Decision Diffie-Hellman (DDH [4]) problem is hard in $\mathbb{G}$.
3. The committer may publish $H(m,r)$ for a collision resistant hash function H ([6, Section 8.1]) and a random $r \in \{0,1\}^\lambda$ for a large enough λ. This would be computationally binding and computationally hiding in the random oracle model [46].

Now, assume that we want to commit to a polynomial $p(x) \in \mathbb{Z}_q[x]$ which has degree t and coefficients $p_0, ..., p_t$. We could commit to the string $(p_0|p_1|...|p_t)$, or to some other unambiguous string representation of $p(x)$. Based on the commitment function used, this option may have a constant size commitment which determines $p(x)$. However, it limits the options for opening the commitment; opening must reveal the entire polynomial. This is not always suitable for cryptographic applications, most notably secret sharing, that require evaluations of the polynomial (i.e., $p(i)$ for $i \in \mathbb{Z}_q$) to be revealed to different parties or at different points in the protocol without revealing the entire polynomial. One solution is to commit to the coefficients, for example $\mathcal{C} = (g^{p_0}, ..., g^{p_t})^1$, which allows one to easily confirm that an opening $p(i)$ for index i is consistent with $\mathcal{C}$. However, this has the drawback that the size of the commitment is now $t + 1$ elements of $\mathbb{G}$.

In section 5.2, we present the notion of polynomial commitment schemes (introduced in [19]) in detail, the idea is that a user can commit to a polynomial $p \in \mathbb{Z}_q[x]$ over $\mathbb{Z}_q$ and later open to an evaluation $p(x)$ at any point $x \in \mathbb{Z}_q$. Previous to that, in section 5.1, we also present a similar notion, vector commitments schemes [13], where instead of committing to polynomials we commit to vectors and we want to give proofs for positions instead of for evaluations. Our main novel contribution, presented in section 5.2, is a polynomial commitment that is efficient for the committer generating evaluation proofs. The way we achieve this is by committing to the data structure outputted by Kedlaya/Umans algorithm using a vector commitment VC. Then opening for evaluations can be reduced to opening some positions in the data structure plus reconstructing the evaluation using the CRT algorithm.

More concretely, by being efficient for the committer we mean that he can compute evaluation proofs in time sublinear in the length of the polynomial. In order to achieve this we need to instantiate our construction with a vector commitment that satisfies a concrete complexity restriction shown in 5.1. In section 5.2, we also present two vector commitment schemes, Merkle Tree Vector Commitment [24] and Incrementally Aggregatable Vector Commitment [9], which satisfy this complexity constrain.

This PC construction was built in collaboration with a team formed by: Matteo Campanelli, Dario Catalano, Danilo Francati, Emanuele Giunta, Rosario Gennaro and both my supervisors Dario Fiore and Ignacio Cascudo. Right now we are still working on the strenghtenings and the applications of it.

[1]This is just an illustrative example as it doesn't satisfy hiding.

Chapter 2

Preliminaries

2.1 Notation

Define $\mathbb{N} = \{0, 1, 2, ...\}$ to be the set of natural numbers, $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$ to be the set of integers and $\mathbb{R}$ to be the set of real numbers. For any integer $n \geq 1$, define $[n] = \{1, ..., n\}$ and $[\![n]\!] = \{0, ..., n-1\}$. For an array $A \in \{0,1\}^n$, we index the array from 0 and $A[i]$ denotes the bit in position $i \in [\![n]\!]$. By default, all the logarithms are base 2 and $\log n$ stands for $\log_2 n$. For any $q \in \mathbb{N}$, let $\mathbb{Z}_q$ be the ring $\mathbb{Z}/q\mathbb{Z}$. We denote by $//$ the natural division, i.e. for any $n, m \in \mathbb{N}$, $n//m = \lfloor n/m \rfloor$. We use $\lambda \in \mathbb{N}$ to denote the security parameter. A function $\epsilon : \mathbb{N} \to \mathbb{N}$ is said to be negligible, denoted $\epsilon(\lambda) = negl(\lambda)$ if for every positive polynomial $p(\cdot)$ and all sufficiently large n it holds that $\epsilon(n) < 1/p(n)$ (more details in [43]). We say that an algorithm is probabilistic polynomial time (abbreviated PPT) if its running time is bounded by some polynomial $p(\lambda)$ (more details in [44]). A function $p(\lambda)$ is polynomial (denoted $p(\lambda) = poly(\lambda)$) if $p(\lambda) = O(\lambda^c)$ for some constant $c > 0$. For a finite set S, we write $a \xleftarrow{\$} S$ to mean a is sampled uniformly randomly from S. For an algorithm $\mathcal{A}$, we write $y \leftarrow \mathcal{A}(x)$ for the output of $\mathcal{A}$ on input x. For two distributions X, Y parametrized by λ we say that they are computationally indistinguishable, denoted $X \approx_c Y$, if for every PPT distinguisher $\mathcal{D}$ we have $|\Pr[\mathcal{D}(X) = 1] - \Pr[\mathcal{D}(Y) = 1]| = negl(\lambda)$. Across the text, when we refer to an algorithm having random access to a data structure we mean that the time complexities for those algorithms are in the RAM model [45], i.e. access to large memory is constant-time.

2.2 Auxiliary algorithms

In this section we'll present some of the algorithms that will be used in the preprocessing and the evaluation algorithms.

2.2.1 Multipoint Evaluation Algorithm

The data structure outputted by the preprocessing algorithm which allows efficient evaluations of a polynomial $f \in R[X_1, ..., X_m]$, essentially consists in a set of h primes $p_1, ..., p_h$ and a set of h tables $T_1, ..., T_h$, where the i-th table T_i contains the evaluations of the polynomial $f_i = f \bmod p_i$ in all the points in $\mathbb{Z}_{p_i}^m$. This could be easily but not efficiently implemented by evaluating the polynomials f_i in each point in $\mathbb{Z}_{p_i}^m$ independently. Luckily, the multipoint evaluation problem is well known and there exist algorithms that allow evaluation in several points faster than independent evaluation of each point. We'll start by introducing the algorithm to evaluate univariate polynomials at several points, then we'll use this algorithm to give a solution for multivariate polynomials in section 3.1.

Throughout this section, we introduce algorithms based on polynomial multiplication and division. In order to abstract from the underlying multiplication algorithm in the cost analyses, we introduce the following notation.

Definition 2.2.1. Let R be a ring (commutative, with 1). We call a function $M : \mathbb{N}\setminus\{0\} \to \mathbb{R}_{>0}$ a multiplication time for $R[X]$ if polynomials in $R[X]$ of degree less than n can be multiplied using at most $M(n)$ operations in R. We call a function $D : \mathbb{N} \setminus \{0\} \to \mathbb{R}_{>0}$ a division time for $R[X]$ if polynomials in $R[X]$ of degree less than n can be divided using at most $D(n)$ operations in R.

We'll simplify the exposition by assuming that the number n of points we want to evaluate is a power of 2. For a general n, we have two options of either adding some "phantom" points or splitting points into two roughly equal halves in the recursive calls. The idea of the univariate evaluation algorithm is to split the points we want to evaluate $\{u_0, ..., u_{n-1}\}$ into two halves of equal cardinality and to proceed recursively with each of the two halves. This leads to a binary tree of depth $\log n$ with root $\{u_0, ..., u_{n-1}\}$ and the singletons $\{u_i\}$ for $0 \le i < n$ at the leaves (see figure 2.1).

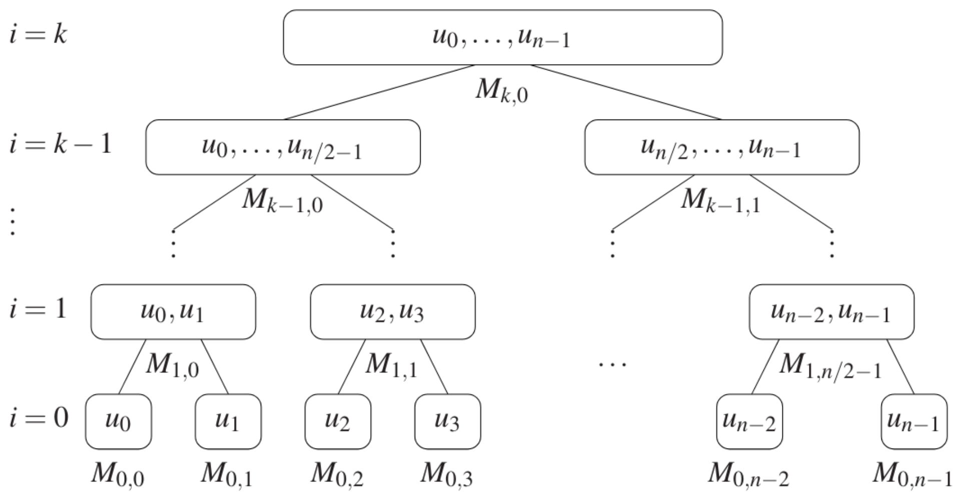


Figure 2.1: Subproduct tree for the multipoint evaluation algorithm [35, Section 10.1].

We let $m_i = x - u_i$ as above and define:

$$M_{i,j} = m_{j\cdot 2^i} \cdot m_{j\cdot 2^i+1} \cdots m_{j\cdot 2^i+(2^i-1)} = \prod_{0\le l<2^i} m_{j\cdot 2^i+l} \tag{2.1}$$

for $0 \le i \le k = \log n$ and $0 \le j < 2^{k-i}$. Thus each $M_{i,j}$ is a subproduct with 2^i factors of $m = \prod_{0\le l<n} m_l = M_{k,0}$ and satisfies for each i, j the recursive equations

$$M_{0,j} = m_j, \qquad M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}. \tag{2.2}$$

If R is an integral domain [41] and $u_0, ..., u_{n-1}$ are distinct, then $M_{i,j}$ is the monic square-free polynomial whose zero set is the j-th node from the left at level i of the tree in figure 2.1.

The following algorithm solves the more general problem of computing the subproducts $M_{i,j}$ for arbitrary moduli $m_0, ..., m_{n-1}$.

Definition 2.2.2 (Building up the subproduct tree [35, Algorithm 10.3])**.**

Input: $m_0, ..., m_{n-1} \in R[X]$ with $n = 2^k$ for some $k \in \mathbb{N}$.

Output: The polynomials $M_{i,j}$ as in 2.1 for $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$.

Algorithm:

```
1. For each $0 \leq j < n$, let $M_{0,j} = m_j$.
2. For each $1 \leq i \leq k$:
       For each $0 \leq j < 2^{k-i}$, let $M_{i,j} = M_{i-1,2j} \cdot M_{i-1,2j+1}$.
3. Output $M_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$.
```

Proposition 2.2.1. *Algorithm 2.2.2 correctly computes all subproducts $M_{i,j} \in R[X]$ and takes at most $M(m) \cdot \log n$ operations in R, where $m = \sum_{0 \leq i < r} \deg(m_i)$.*

Proof. Correctness is clear from equation 2.2. Let $d_{i,j} = \deg(M_{i,j})$ for all i and j. Step 1 uses no arithmetic operations, and the cost for the i-th iteration of step 2 is at most

$$\sum_{0 \leq j < 2^{k-i}} M(d_{i,j}) \leq M\left(\sum_{0 \leq j < 2^{k-i}} d_{i,j} \right) = M(m)$$

operations in R, since $\sum_{0 \leq j < 2^{k-i}} d_{i,j} = m$. The time estimate follows, since there are $k = \log n$ iterations.

□

We now present a divide-and-conquer algorithm that, given all subproducts $M_{i,j}$, proceeds top down along the tree in figure 2.1.

Definition 2.2.3 (Going down the subproduct tree [35, Algorithm 10.5])**.**

Input: $f \in R[X]$ with degree $< n = 2^k$ for some $k \in \mathbb{N}$, $u_0, ..., u_{n-1} \in R$ and the subproducts $M_{i,j}$ from 2.1.

Output: $f(u_0), ..., f(u_{n-1}) \in R$.

Algorithm:

```
1. If $n = 1$ then: Output $f$ $n$ times.
2. If $n > 1$ then:
   (a) Let $r_0 = f$ modulo $M_{k-1,0}$ and let $r_1 = f$ modulo $M_{k-1,1}$.
   (b) Recursively compute $r_0(u_0), ..., r_0(u_{n/2-1})$.
   (c) Recursively compute $r_1(u_{n/2}), ..., r_1(u_{n-1})$.
   (d) Output $r_0(u_0), ..., r_0(u_{n/2-1}), r_1(u_{n/2}), ..., r_1(u_{n-1})$.
```

Proposition 2.2.2. *Algorithm 2.2.3 works correctly and takes at most $D(n) \cdot \log n$ operations in R. This is at most $O(M(n) \cdot \log n)$ operations in R.*

Proof. To argue correctness of this algorithm we'll use induction. If $k = 0$, then f is constant and the algorithm outputs the correct values in step 1. Otherwise, if $1 \leq k$, then we can assume that the results of steps 2.*b* and 2.*c* are correct. Let $q_0 = f/M_{k-1,0}$ and $q_1 = f/M_{k-1,1}$. Then

$$f(u_i) = \begin{cases} q_0(u_i)M_{k-1,0}(u_i) + r_0(u_i) = r_0(u_i) & \text{if } 0 \leq i < \frac{n}{2} \\ q_1(u_i)M_{k-1,1}(u_i) + r_1(u_i) = r_1(u_i) & \text{if } \frac{n}{2} \leq i < n \end{cases}.$$

Let $T(n) = T(2^k)$ denote the cost for the recursive process. Then $T(1) = 0$ and

$$T(2^k) = 2T(2^{k-1}) + 2D(2^{k-1})$$

for $k \geq 1$, so that $T(n) = T(2^k) \leq 2k \cdot D(2^{k-1}) \leq D(n) \log n$, by Lemma 8.2 in [35]. The last claim follows from Theorem 9.6 in [35].

□

Putting both algorithms together now, we obtain the univariate multipoint evaluation algorithm:

Definition 2.2.4 (Univariate Multipoint Evaluation algorithm [35, Algorithm 10.7])**.**

Input: $f \in R[X]$ with degree $< n = 2^k$ for some $k \in \mathbb{N}$ and $u_0, ..., u_{n-1} \in R$.

Output: $f(u_0), ..., f(u_{n-1}) \in R$.

Algorithm:

1. Let $M_{i,j}$ be the outputs of algorithm 2.2.2 on input $(x - u_0), ..., (x - u_{n-1})$.
2. Output the result of algorithm 2.2.3 on input f, the points u_i and the subproducts $M_{i,j}$.

Proposition 2.2.3. *Evaluation of a polynomial in $R[X]$ of degree less than n at n points in R can be performed using at most $O(M(n) \cdot \log n)$ operations in R.*

A generalization of this algorithm for multivariate polynomials can be found in algorithm 3.1.1.

The key to make this algorithm work with the desired complexity is using a good algorithm for multiplying polynomials.

Algorithm	$\mathsf{M}(n)$
classical	$2n^2$
Karatsuba (Karatsuba & Ofman 1962)	$O(n^{1.59})$
FFT multiplication (provided that R supports the FFT)	$O(n \log n)$
Schönhage & Strassen (1971), Schönhage (1977) Cantor & Kaltofen (1991); FFT based	$O(n \log n \log\log n)$
Fürer (2009); FFT based	$n \log n \cdot 2^{O(\log^* n)}$

Figure 2.2: Various polynomial multiplication algorithms and their running times [35, Section 8.3].

The table in figure 2.2 shows the complexity of some well-known polynomial multiplication algorithms. We would be interested in applying the FFT (Fast Fourier Transform [40]) multiplication algorithm, to perform univariate multipoint evaluation in $O(n \cdot (\log n)^2)$ operations, however, this algorithm requires the ring R to support FFT which precisely means the following:

Definition 2.2.5 ([35, Definition 8.17])**.** We say that a (commutative) ring R supports the FFT if R has a primitive 2^k-th root of unity for $k \in \mathbb{N}$. Where $w \in R$ is a n-th primitive root of unity for any $n \in \mathbb{N}$ $(n \neq 0)$ if:

1. $w^n = 1_R$.
2. $n \cdot 1_R$ is a unit in R.[1]
3. $w^{n/t} - 1$ is not a zero divisor for any prime divisor t of n.[2]

This isn't satisfied in general for rings of the form $\mathbb{Z}_q$ with q a prime. For example, in $\mathbb{Z}_7$ we don't have 4-th primitive roots of unity as the only $w \in \mathbb{Z}_7$ such that $w^4 = 1$ are $w = 1$ and $w = 6$ but they don't satisfy the 3rd condition in the definition above. Luckily, the algorithm by Schönhage and Strassen achieves a good complexity (shown in figure 2.2) over more general rings that don't need to support FFT directly. This algorithm achieves to compute univariate multipoint evaluation in $O(n \cdot (\log n)^2 \cdot \log \log n)$ operations.

The way Schönhage and Strassen algorithm works is by adjoining "virtual" roots of unity, and then applying FFT multiplication. Let R be a ring such that 2 is a unit in R (which is the case for any $R = \mathbb{Z}_q$ with q a prime except for $q = 2$), $n = 2^k$ for some $k \in \mathbb{N}$, and $D = R[x]/\langle x^n + 1\rangle$. The congruences

$$x^n \equiv -1 \mod (x^n + 1), \qquad x^{2n} = (x^n)^2 \equiv 1 \mod (x^n + 1)$$

imply that $w = x \mod (x^n + 1) \in D$ is a $2n$-th root of unity. Moreover, $w^n - 1 = (-1) - 1 = -2$ is a unit in R since 2 is, and w is a primitive $2n$-th root of unity.

To multiply two polynomials $f, g \in R[x]$ with $\deg(fg) < n = 2^k \in \mathbb{N}$, it is clearly sufficient to compute fg modulo $x^n + 1$. This is called the negative wrapped convolution of f and g. We let $m = 2^{\lfloor k/2 \rfloor}$, $t = n/m = 2^{\lceil k/2 \rceil}$, and partition the coefficients of f and g into t blocks of size m, that is, we write

$$f = \sum_{0 \le j < t} f_j x^{mj}, \qquad g = \sum_{0 \le j < t} g_j x^{mj},$$

with $f_j, g_j \in R[x]$ of degree less than m for $0 \le j < t$. With $f' = \sum_{0 \le j < t} f_j y^j$, $g' = \sum_{0 \le j < t} g_j y^j \in R[x, y]$, we then have that $f = f'(x, x^m)$ and $g = g'(x, x^m)$. It is sufficient to compute $f'g'$ modulo $y^t + 1$, since

$$f'g' = h' + q'(y^t + 1) \equiv h' \mod (y^t + 1) \tag{2.3}$$

for some $h', q' \in R[x, y]$ implies that

$$fg = f'(x, x^m)g'(x, x^m) = h'(x, x^m) + q'(x, x^m)(x^{tm} + 1) \equiv h'(x, x^m) \mod (x^n + 1)$$

[1]Recall that $a \in R$ is a unit in R if there exists a $b \in R$ such that $ab = ba = 1_R$.

[2]Recall that $a \in R$ is a zero divisor in R if there exists a $b \in R$ such that $b \neq 0$ and $ab = 0$.

so computing h' would solve our problem. We want to compute $h' \in R[x, y]$ with $\deg_y h' < t$ satisfying 2.3 (which uniquely determines h'). Comparing coefficients of y^j for $j \geq t$, we see that $\deg_x q' \leq \deg_x(f'g') < 2m$ and conclude that

$$\deg_x h' \leq \max\{\deg_x(f'g'), \deg_x q'\} < 2m. \tag{2.4}$$

With $f^* = f' \mod (x^{2m}+1)$, $g^* = g' \mod (x^{2m}+1)$ and $h^* = h' \mod (x^{2m}+1)$ in $D[y]$, 2.3 implies

$$f^*g^* \equiv h^* \mod (y^t + 1) \text{ in } D[y]. \tag{2.5}$$

Since the three polynomials have degrees in x less than $2m$, by 2.4, reducing them modulo $x^{2m}+1$ is just taking a different algebraic meaning of the same coefficient array. In particular, the coefficients of $h' \in R[x][y]$ can be read off the coefficients of $h^* \in D[y]$.

Computationally, "nothing happens" in mapping h' and h^*, but we are now in a situation where we can apply the machinery of the FFT to compute 2.5, as follows. Since t equals either m or $2m$, D contains a primitive $2t$-th root of unity η, namely

$$\eta = \begin{cases} x \mod (x^{2m}+1) & \text{if } t = 2m \\ x^2 \mod (x^{2m}+1) & \text{if } t = m \end{cases} \in D = R[x]/\langle x^{2m}+1\rangle.$$

Then 2.5 is equivalent to

$$f^*(\eta y)g^*(\eta y) \equiv h^*(\eta y) \mod ((\eta y)^t + 1),$$

or

$$f^*(\eta y)g^*(\eta y) \equiv h^*(\eta y) \mod (y^t - 1), \tag{2.6}$$

since $\eta^t = -1$. Given $f^*(\eta y)$ and $g^*(\eta y)$ in $D[y]$, we can apply the FFT algorithm to compute $h^*(\eta y)$ with $O(t \log t)$ operations in D.

To see how this works more clearly check the following example:

Example 2.2.1. Let $R = \mathbb{Z}_5$, $f(x) = x^4 + 2x + 3$, and $g(x) = 2x^3 + x^2 + 4x + 2$ in $\mathbb{Z}_5[x]$. Then we can compute $fg \in \mathbb{Z}_5[x]$ witk $k = 3$ as follows. First, we compute $m = 2$, $t = 4$, $f'(x, y) = y^2+2x+3$ and $g'(x, y) = (2x+1)y+4x+2$. Then, we have that $\eta = x \mod (x^4+1)$,

$$f^*(y) = y^2 + 2\eta + 3, \qquad g^*(y) = (2\eta + 1)y + 4\eta + 2,$$
$$f^*(\eta y) = \eta^2 y^2 + 2\eta + 3, \qquad g^*(\eta y) = (2\eta^2 + \eta)y + 4\eta + 2.$$

Now the FFT algorithm would yield the following result

$$f^*(\eta y)g^*(\eta y) = (2\eta^4 + \eta^3)y^3 + (4\eta^3 + 2\eta^2)y^2 + (4\eta^3 + 3\eta^2 + 3\eta)y + 3\eta^2 + \eta + 1 = h^*(\eta y)$$

and therefore

$$h^*(y) = h^*(\eta(\eta^{-1}y)) = (2\eta + 1)y^3 + (4\eta + 2)y^2 + (4\eta^2 + 3\eta + 3)y + 3\eta^2 + \eta + 1.$$

Finally, we have that

$$h'(x, y) = (2x + 1)y^3 + (4x + 2)y^2 + (4x^2 + 3x + 3)y + 3x^2 + x + 1$$

and therefore

$$h'(x, x^2) = 2x^7 + x^6 + 4x^5 + x^4 + 3x^3 + x^2 + x + 1.$$

So, as claimed we have that:

$$h(x) = h'(x, x^2) \mod (x^8 + 1) = 2x^7 + x^6 + 4x^5 + x^4 + 3x^3 + x^2 + x + 1 = f(x)g(x).$$

2.2.2 CRT Algorithm

The Multipoint Evaluation algorithm was used to compute the data structure that allows fast evaluation of $f \in R[X_1, ..., X_m]$. Now, the only thing left to do is to reconstruct the solution using this data structure. As we'll prove later in theorem 3.1.3, to evaluate $f(\alpha)$ for $\alpha \in R^m$ it suffices to compute a $z \in \mathbb{Z}$ such that

$$z \equiv f_i(\alpha_i) \mod p_i \text{ for all } i \in [h]$$

(where $\alpha_i = \alpha \mod p_i$) and then reduce it modulo q. To compute this $z \in \mathbb{Z}$ we'll use the well known CRT algorithm:

Definition 2.2.6 (Chinese Remainder Theorem algorithm [35, Algorithm 5.4])**.**

> **Input**: $m_1, ..., m_h \in R$ pairwise coprime and $r_1, ..., r_h \in R$, where R is an Euclidean domain.
>
> **Output**: $z \in R$ such that $z \equiv r_i \mod m_i$ for all $i \in [h]$.
>
> **Algorithm**:
>
> 1. Let $m = \prod_{i=1}^{h} m_i$.
> 2. For each $i \in [h]$:
>   (a) Let $m'_i = \frac{m}{m_i}$.
>   (b) Compute $s_i, t_i \in R$ such that $s_i m'_i + t_i m_i = 1$ using the Extended Euclidean algorithm.[3]
>   (c) Let $c_i = r_i s_i \mod m_i$.
> 3. Output $z = \sum_{i=0}^{h-1} c_i m'_i$.

which works in our case because $R = \mathbb{Z}$ is an Euclidean Domain [39].

To see the correctness of this algorithm, we observe that $c_i m'_i \equiv 0 \mod m_j$ for $j \neq i$ and $c_i m'_i \equiv r_i s_i m'_i \equiv r_i \mod m_i$, and therefore $z \equiv c_i m'_i \equiv r_i \mod m_i$ for $0 \leq i < h$, as claimed.

[3]The Extended Euclidean algorithm can be found in [35, Algorithm 3.14]

Chapter 3

The algorithm

The algorithm for polynomial evaluation with preprocessing by Kedlaya and Umans [20], shows how to preprocess a multivariate polynomial $f(X_1, ..., X_m) \in R[X_1, ..., X_m]$ (for some ring R) into a static data structure such that, for any given input $\alpha \in R^m$ given later, we can use the data structure to evaluate $f(\alpha) \in R$ quickly, in time that is sublinear in the description-length of the polynomial.

First we are presenting the algorithm for the case $R = \mathbb{Z}_q$. Then we do it for $R = \mathbb{Z}_q[Y]/(E_1[Y])$ by reducing this problem to the case $R = \mathbb{Z}_{q'}$ for some q'. Using the same ideas we can build the algorithm for $R = \mathbb{Z}_q[Y, Z]/(E_1[Y], E_2[Z])$ and beyond.

3.1 Algorithm for $\mathbb{Z}_q$

We start by the simplest version of the algorithm and the one we've implemented, the case $R = \mathbb{Z}_q$. The main idea for this case is reinterpreting the polynomial $f(X_1, ..., X_m)$ over $\mathbb{Z}_q$ as a polynomial over the integers $\mathbb{Z}$, with coefficients and inputs in $\{0, ..., q-1\}$. The maximal value this polynomial can take over the integers is $\leq M = d^m(q-1)^{dm}$. Let $p_1, ..., p_h$ be the set of all primes $p_i \leq 16 \log M$, which ensures (by lemma 3.1.1) that $\prod_{i=1}^{h} p_i \geq M$. To evaluate the polynomial f over the integers, it suffices to evaluate it modulo each of the primes p_i separately and then reconstruct the answer over the integers using the CRT algorithm. So we reduced the problem of evaluating the polynomial f modulo q to that of evaluating it modulo p_i for a small set of $h = O(\log M)$ much smaller primes $p_i = O(\log M)$. Using this idea we can preprocess the polynomial as follows, consider constructing h tables, where the i-th table simply stores all the possible evaluations of the polynomial f on all p_i^m possible inputs $(\alpha_1, ..., \alpha_m)$ modulo p_i. This allows for extremely fast evaluation on any input $(\alpha_1, ..., \alpha_m) \in \mathbb{Z}_q^m$ just by looking up one entry in each table and then using CRT.

We'll start by proving the auxiliary lemma needed for the correctness of the algorithm.

Lemma 3.1.1 (Product of small primes [20, Lemma 2.4]). *For all integers $M \geq 2$, the product of the primes less than or equal to $16 \log M$ is greater than M. That is,*

$$M < \prod_{\substack{p \leq 16 \log M \\ p \text{ prime}}} p.$$

Proof. By Legendre's formula [36], we can rewrite:

$$n! = \prod_{\substack{p \leq n \\ p\ prime}} p^{\sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor}$$

In particular

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} = \prod_{\substack{p \leq n \\ p\ prime}} p^{e_p}$$

where $e_p = \sum_{i=1}^{\infty} \left(\left\lfloor \frac{n}{p^i} \right\rfloor - \left\lfloor \frac{m}{p^i} \right\rfloor - \left\lfloor \frac{n-m}{p^i} \right\rfloor \right)$.

Note that $e_p \leq 1$ for $\sqrt{n} < p \leq n$ and $e_p \leq \log_p(n)$ for all p. From this and the fact that $\binom{n}{m} \leq \binom{n}{\lfloor n/2 \rfloor}$ for all $0 \leq m \leq n$, it follows that:

$$\frac{2^n}{n+1} = \frac{1}{n+1} \sum_{m=0}^{n} \binom{n}{m} \leq \binom{n}{\lfloor n/2 \rfloor} \leq \left(\prod_{\substack{\sqrt{n} < p \leq n \\ p\ prime}} p \right) n^{\sqrt{n}} \leq \left(\prod_{\substack{p \leq n \\ p\ prime}} p \right) n^{\sqrt{n}}$$

For $M \geq 50$, we have that:

$$M \leq \frac{2^n}{(n+1)n^{\sqrt{n}}} \leq \left(\prod_{\substack{p \leq n \\ p\ prime}} p \right)$$

for $n = \lfloor 16 \log M \rfloor$, so the claim holds. For $M < 50$, we can use the program *lemma_primes.py* in [11] to check it holds too.

□

Now we'll define the algorithm that allows the evaluation of all the points in $\mathbb{Z}_p^m$ using the algorithm 2.2.4 for univariate polynomials.

Definition 3.1.1 (Multivariate Multipoint Evaluation algorithm [20, Theorem 4.1])**.**

Input: $f \in \mathbb{Z}_p[X_1, X_2, ..., X_m]$ with individual degree $< p$ in each variable.

Output: $f(\mathbb{Z}_p^m) = \{f(\alpha) \mid \alpha \in \mathbb{Z}_p^m\}$.

Algorithm:

```
1. If $m = 1$ then: Output algorithm 2.2.4 applied to $f$.
2. If $m > 1$ then:
    (a) Write $f(X_1, X_2, ..., X_m)$ as $\sum_{i=0}^{n-1} X_1^i f_i(X_2, ..., X_m)$.
    (b) For each $f_i$, recursively compute $f_i(\beta)$ on all points $\beta \in \mathbb{Z}_p^{m-1}$.
    (c) For each $\beta \in \mathbb{Z}_p^{m-1}$, output the evaluation of the univariate polynomial
        $\sum_{i=0}^{q-1} X_1^i f_i(\beta)$ on all points in $\mathbb{Z}_p$.
```

Lemma 3.1.2. *Given a polynomial $f \in \mathbb{Z}_p[X_1, X_2, ..., X_m]$ with individual degree $< p$ in each variable, the algorithm 3.1.1 outputs $f(\mathbb{Z}_p^m)$ in:*

$$m \cdot O(p^m \cdot poly(\log p))$$

operations.

Proof. We'll proceed by induction on m:

- For $m = 1$: the algorithm 2.2.4 correctly computes $f(\mathbb{Z}_p)$ in $O(p \cdot poly(\log p))$ operations using FFT based multiplication.

- For $m > 1$: by induction hypothesis we can assume that on the step 2.b the $f_i(\beta)$ on all points $\beta \in \mathbb{Z}_p^{m-1}$ are correctly computed in $(m-1) \cdot O(p^{m-1} \cdot poly(\log p))$ operations for each f_i. The last step also takes $O(p \cdot poly(\log p))$ operations using fast univariate multipoint evaluation for each $\beta \in \mathbb{Z}_p^{m-1}$. The overall time then is:

$$p \cdot (m-1) \cdot O(p^{m-1} \cdot poly(\log p)) + p^{m-1} \cdot O(p \cdot poly(\log p))$$

  which equals $m \cdot O(p^m \cdot poly(\log p))$ as claimed.

□

Having both lemmas established let's now define the preprocessing algorithm and the evaluation algorithm and prove their correctness/efficiency:

Definition 3.1.2 (Preprocessing algorithm for $R = \mathbb{Z}_q$)**.**

Input: $f \in \mathbb{Z}_q[X_1, X_2, ..., X_m]$ with individual degree $< d$ in every variable.

Output: Data structure that allows fast evaluation of f.

Algorithm:

1. Let $M := d^m q^{m(d-1)+1}$ and let $p_1, ..., p_h$ be the distinct primes less than or equal to $16 \log M$ for some $h \in \mathbb{N}$.
2. Lift $f \in \mathbb{Z}_q[X_1, ..., X_m]$ to $\overline{f} \in \mathbb{Z}[X_1, ..., X_m]$, i.e., the coefficients of $\overline{f}$ are obtained by lifting the corresponding coefficients of f from $\mathbb{Z}_q$ to $\mathbb{Z}$, where the coefficients are represented by $\{0, 1, ..., q-1\}$ in both $\mathbb{Z}_q$ and $\mathbb{Z}$.
3. For each $i \in [h]$, let $\overline{f}_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m]$ be the polynomial $\overline{f}_i = \overline{f}$ modulo p_i (i.e., take every coefficient of $\overline{f}$ modulo p_i). Moreover, compute the following for each i:
   (a) Compute the reduction $\overline{f_i}$ modulo $X_j^{p_i} - X_j$ for each $j \in [m]$.
   (b) Using algorithm 3.1.1, evaluate $\overline{f}_i(a)$ on all point $a \in \mathbb{Z}_{p_i}^m$.
   (c) Let T_i be the table of evaluations, i.e., $T_i := (\overline{f}_i(a) : a \in \mathbb{Z}_{p_i}^m)$.
4. Output the data structure consisting of $p_1, ..., p_h, T_1, ..., T_h$.

Definition 3.1.3 (Evaluation algorithm for $R = \mathbb{Z}_q$)**.**

Input: The data structure in 3.1.2 for $f \in \mathbb{Z}_q[X_1, ..., X_m]$ and $\alpha \in \mathbb{Z}_q^m$.

Output: $f(\alpha) \in \mathbb{Z}_q$.

Algorithm:

1. Lift α to $\overline{\alpha} \in \mathbb{Z}^m$, i.e., each coordinate of $\overline{\alpha}$ is obtained by lifting the corresponding coordinate of α from $\mathbb{Z}_q$ to $\mathbb{Z}$.
2. For each $i \in [h]$, let $\overline{\alpha}_i \in \mathbb{Z}_{p_i}^m$ be the point obtained by $\overline{\alpha}$ modulo p_i coordinate-wise. Moreover, for each i, the evaluation $\overline{f}_i(\overline{\alpha}_i) \in \mathbb{Z}_{p_i}$ is obtained by looking it up in table T_i.

3. Use CRT algorithm 2.2.6 to find the smallest $z \in \mathbb{Z}$ such that

$$z \equiv \overline{f}_i(\overline{\alpha}_i) \mod p_i \text{ for all } i \in [h].$$

4. Output z modulo q as the evaluation $f(\alpha)$.

Theorem 3.1.3 (Preprocessing polynomials over $\mathbb{Z}_q$)**.** *Let $q \in \mathbb{N}$ and let $f \in \mathbb{Z}_q[X_1, X_2, ..., X_m]$ be a polynomial of individual degree $< d$ in every variable. Then, the preprocessing algorithm 3.1.2 takes the coefficients of f as an input, runs in time*

$$S = poly(m, d, \log q) \cdot O(md \log q)^m$$

and outputs a data structure of size at most S, and the evaluation algorithm 3.1.3 with random access to the data structure, is given an evaluation point $\alpha \in \mathbb{Z}_q^m$ and computes $f(\alpha)$ in time

$$poly(m, d, \log q).$$

Proof.
Correctness:
The correctness of the above algorithm is argued as follows. Because the lifted polynomial $\overline{f}$ and the lifted point $\overline{\alpha}$ each have coefficients/components in $\{0, ..., q-1\}$, and $\overline{f}$ has individual degree $< d$, we can bound the lifted evaluation over $\mathbb{Z}$ by: $0 \leq \overline{f}(\overline{\alpha}) < M$. By Lemma 3.1.1, we then have $\overline{f}(\overline{\alpha}) < M < \prod_{i\in[h]} p_i$. Notice that $\overline{f}(\overline{\alpha}) \equiv \overline{f}_i(\overline{\alpha}_i) \mod p_i$ for all $i \in [h]$. . Also, by CRT, there is a unique solution for $z < \prod_{i\in[h]} p_i$ such that $z \equiv \overline{f}_i(\overline{\alpha}_i) \mod p_i$ for all $i \in [h]$. Recalling that $\overline{f}_i(\overline{\alpha}_i)$ are correctly computed by the FFT evaluation (Lemma 3.1.2), we have $z = \overline{f}(\overline{\alpha})$ by the uniqueness. Correctness follows since $z = \overline{f}(\overline{\alpha}) \mod q = f(\alpha)$.

Preprocessing time:
We next calculate the computation time of the data structure. As $h < 16 \log M = O(md \log q)$, it takes time $poly(m, d, \log q)$ to compute $M, p_1, p_2, ..., p_h$. Also, it takes time $d^m \cdot poly(m, d, \log q)$ to compute $\overline{f}_i$ for all $i \in [h]$ (and to reduce it modulo $X_j^{p_i} - X_j$ for each $j \in [m]$) since f consists of d^m coefficients. We then apply Lemma 3.1.2 to obtain T_i's, which takes time $(d^m + (16 \log M)^m) \cdot poly(m, d, \log q)$. We thus have

$$S = poly(m, d, \log q) \cdot O(md \log q)^m.$$

Evaluation time:
The evaluation time given α is straightforward. It takes time $poly(m, d, \log q)$ to perform the modular reduction (from α to $\overline{\alpha}_i$) as well as the looking up for $\overline{f}_i(\overline{\alpha}_i)$ for all $i \in [h]$. Performing the CRT and taking the result modulo q also take time $poly(m, d, \log q)$.

□

3.1.1 Improving the algorithm

In the paper, they also present an improvement of the algorithm that allows reducing the $(\log q)^m$ factor in the preprocessing algorithm run-time into a $(\log \log q)^m$ factor by using the original algorithm "recursively". The improved algorithms just have some slight changes with respect to the ones already presented, in the preprocessing instead of computing $T_i = \{\overline{f}_i(a) \mid a \in \mathbb{Z}_{p_i}^m\}$ we compute the data structure DB_i outputed by algorithm 3.1.2 for each of the $\overline{f}_i$'s and then in the evaluation algorithm we compute each of the $\overline{f}_i(\overline{\alpha}_i)$'s by invoking algorithm 3.1.3 on DB_i. The resultant algorithms with the changes highlighted in blue are the following:

Definition 3.1.4 (Preprocessing algorithm for $R = \mathbb{Z}_q$ improved)**.**

Input: $f \in \mathbb{Z}_q[X_1, X_2, ..., X_m]$ with individual degree $< d$ in every variable.

Output: Data structure that allows fast evaluation of f.

Algorithm:

1. Let $M := d^m q^{m(d-1)+1}$ and let $p_1, ..., p_h$ be the distinct primes less than or equal to $16 \log M$ for some $h \in \mathbb{N}$.
2. Lift $f \in \mathbb{Z}_q[X_1, ..., X_m]$ to $\overline{f} \in \mathbb{Z}[X_1, ..., X_m]$, i.e., the coefficients of $\overline{f}$ are obtained by lifting the corresponding coefficients of f from $\mathbb{Z}_q$ to $\mathbb{Z}$, where the coefficients are represented by $\{0, 1, ..., q-1\}$ in both $\mathbb{Z}_q$ and $\mathbb{Z}$.
3. For each $i \in [h]$, let $\overline{f}_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m]$ be the polynomial $\overline{f}_i = \overline{f}$ modulo p_i (i.e., take every coefficient of $\overline{f}$ modulo p_i). Moreover, compute the following for each i:
   (a) Apply algorithm 3.1.2 to compute the data structure for the polynomial $\overline{f}_i$ over $\mathbb{Z}_{p_i}$; let DB_i be the resulting data structure.
4. Output the data structure consisting of $p_1, ..., p_h, DB_1, ..., DB_h$.

Definition 3.1.5 (Evaluation algorithm for $R = \mathbb{Z}_q$ improved)**.**

Input: The data structure in 3.1.4 for $f \in \mathbb{Z}_q[X_1, ..., X_m]$ and $\alpha \in \mathbb{Z}_q^m$.

Output: $f(\alpha) \in \mathbb{Z}_q$.

Algorithm:

1. Lift α to $\overline{\alpha} \in \mathbb{Z}^m$, i.e., each coordinate of $\overline{\alpha}$ is obtained by lifting the corresponding coordinate of α from $\mathbb{Z}_q$ to $\mathbb{Z}$.
2. For each $i \in [h]$, let $\overline{\alpha}_i \in \mathbb{Z}_{p_i}^m$ be the point obtained by $\overline{\alpha}$ modulo p_i coordinate-wise. Moreover, for each i, the evaluation $\overline{f}_i(\overline{\alpha}_i) \in \mathbb{Z}_{p_i}$ is obtained by invoking the evaluation algorithm 3.1.3 using the data structure DB_i.
3. Use CRT algorithm 2.2.6 to find the smallest $z \in \mathbb{Z}$ such that
$$z \equiv \overline{f}_i(\overline{\alpha}_i) \mod p_i \text{ for all } i \in [h].$$
4. Output z modulo q as the evaluation $f(\alpha)$.

Now with this algorithm we achieve the following result:

Theorem 3.1.4 (Preprocessing polynomials over $\mathbb{Z}_q$ improved)**.** *Let $q \in \mathbb{N}$ and let $f \in \mathbb{Z}_q[X_1, X_2, ..., X_m]$ be a polynomial of individual degree $< d$ in every variable. Then, the preprocessing algorithm 3.1.4 takes the coefficients of f as an input, runs in time*

$$S = d^m \cdot poly(m, d, \log q) \cdot O(m(\log m + \log d + \log\log q))^m$$

and outputs a data structure of size at most S, and the evaluation algorithm 3.1.5 with random access to the data structure, is given an evaluation point $\alpha \in \mathbb{Z}_q^m$ and computes $f(\alpha)$ in time

$$poly(m, d, \log q).$$

Proof.
Correctness:
The correctness of the recursion follows by the same argument as theorem 3.1.3 and the correctness of the base case then follows from theorem 3.1.3 directly.

Preprocessing time:
We next calculate the efficiency of the data structure. For each $i \in [h]$, by theorem 3.1.3, the data structure DB_i can be computed in time $S_i = \mathit{poly}(m, d, \log p_i) \cdot O(md \log p_i)^m$, and takes at most S_i space. Since $p_i \leq 16 \log M = O(md \log q)$ for all i, the total space of all h data structures is

$$\begin{aligned} S &= O(md \log(md \log q))^m \cdot \mathit{poly}(m, d, \log q) = \\ &= d^m \cdot \mathit{poly}(m, d, \log q) \cdot O(m(\log m + \log d + \log\log q))^m. \end{aligned}$$

Evaluation time:
Finally, given $\alpha \in \mathbb{Z}_q^m$, the evaluation time is at most

$$\mathit{poly}(\log M) \cdot \mathit{poly}(m, d, \log\log M) = \mathit{poly}(m, d, \log q),$$

where $\mathit{poly}(\log M)$ comes from the CRT and $h < 16 \log M$, and $\mathit{poly}(m, d, \log\log M)$ comes from each of the evaluation of $\overline{f}_i(\overline{\alpha}_i)$ using theorem 3.1.3.

□

Note that the recursion decreases the problem size in q but at the cost of extra factors in $(\log m + \log d)^m$.

3.2 Algorithm for extension rings

We now extend the results of the previous section to work over extensions rings of $\mathbb{Z}_q$. By extension ring we mean rings of the form $\mathbb{Z}_q[Y]/(E_1(Y))$ where E_1 is an arbitrary (non-constant) monic[1] polynomial, and also extensions for several variable like $\mathbb{Z}_q[Y, Z]/(E_1[Y], E_2[Z])$ and beyond. We start by giving an algorithm to evaluate polynomials over univariate extension rings of the form $R = \mathbb{Z}_q[Y]/(E_1(Y))$, we later show how to extend this for polynomials over bivariate extensions rings of the form $R = \mathbb{Z}_q[Y, Z]/(E_1[Y], E_2[Z])$ and quickly explain how this can be generalized for polynomials over n-variate extension rings.

The main idea for $R = \mathbb{Z}_q[Y]/(E_1(Y))$ is the following. First, instead evaluating $f(\alpha)$ over R, we lift the evaluation to $\mathbb{Z}[Y]$ without reducing modulo q or modulo $E_1(Y)$. In that case, we can show that the output $\beta = f(\alpha) \in \mathbb{Z}[Y]$ is a polynomial $\beta = \sum_{i=0}^{D} \beta_i Y^i$ of degree $\leq D$ with non-negative coefficients $\beta_i < M$ for some integer bounds D, M that we compute below. But this means that we can recover the entire polynomial β given its evaluation $\beta(M) \in \mathbb{Z}$ on input $Y = M$, since the $D + 1$ coefficients of β are just the $D + 1$ digits of $\beta(M)$ when written in base M. Furthermore, $\beta(M) < r := M^{D+1}$. Therefore, instead of evaluating $f(\alpha)$ over $\mathbb{Z}[Y]$ it suffices to evaluate it over $\mathbb{Z}_r$ by reducing modulo $(Y - M)$ and modulo r; the reduction modulo r does not have any effect and the base M digits of the output $\beta(M)$ as an integer are the coefficients of β as a polynomial over $\mathbb{Z}[Y]$. We can now use the results in the previous section to preprocess the reduced version of the polynomial f over $\mathbb{Z}_r$ for fast evaluation.

[1]This means that it‘s leading coefficient is 1.

The idea for $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$ (and beyond) is similar. We first lift the evaluation to $\mathbb{Z}[Y, Z]$ and then reduce it to an evaluation over $\mathbb{Z}_{r'}[Y]/(Y^{D'} + 1)$ for an appropriate r', D', which we then solve using the algorithm from the previous paragraph.

Definition 3.2.1 (Preprocessing algorithm for $R = \mathbb{Z}_q[Y]/(E_1(Y))$)**.**

Input: $f \in R[X_1, X_2, ..., X_m]$ with individual degree $< d$ in every variable.

Output: Data structure that allows fast evaluation of f.

Algorithm:

1. Let $e = \deg(E_1)$, $M := d^m(e(q-1))^{m(d-1)+1} + 1$, $D := (e-1)(m(d-1)+1)$ and $r := M^{D+1}$.
2. Lift $f \in R[X_1, ..., X_m]$ to $\tilde{f} \in \mathbb{Z}[Y][X_1, ..., X_m]$ by lifting each coefficient of f from R to $\mathbb{Z}[Y]$.
3. Compute $\overline{f} \in \mathbb{Z}_r[X_1, ..., X_m]$ by reducing $\tilde{f}$ modulo the ideal $(r, Y - M)$.
4. Apply algorithm 3.1.4 to compute the data structure T for $\overline{f}$.
5. Output the data structure consisting of M, r, T.

Definition 3.2.2 (Evaluation algorithm for $R = \mathbb{Z}_q[Y]/(E_1(Y))$)**.**

Input: The data structure in 3.2.1 for $f \in R[X_1, ..., X_m]$ and $\alpha \in R^m$.

Output: $f(\alpha) \in R$.

Algorithm:

1. Lift α to $\tilde{\alpha} \in (\mathbb{Z}[Y])^m$ by lifting each coordinate.
2. Compute $\overline{\alpha} \in \mathbb{Z}_r^m$ from $\tilde{\alpha}$ by reducing each coordinate modulo the ideal $(r, Y - M)$.
3. Use the data structure T to compute $\overline{\beta} = \overline{f}(\overline{\alpha}) \in \mathbb{Z}_r$ and lift to $\tilde{\beta} \in \mathbb{Z}$.
4. Let $\tilde{\beta}_0, ..., \tilde{\beta}_D \in [\![M]\!]$ be the digits of $\tilde{\beta}$ written in base M so that $\tilde{\beta} = \sum_{i=0}^{D} \tilde{\beta}_i M^i$.
5. Construct the polynomial $Q(Y) = \sum_{i=0}^{D} \tilde{\beta}_i Y^i \in \mathbb{Z}[Y]$.
6. Output $Q(Y)$ modulo $(q, E_1(Y))$ as the evaluation $f(\alpha)$.

Theorem 3.2.1 (Preprocessing polynomials over $\mathbb{Z}_q[Y]/(E_1(Y))$)**.** *Let $R = \mathbb{Z}_q[Y]/(E_1(Y))$ for some $q \in \mathbb{N}$ and some arbitrary monic polynomial E_1 with degree $e > 0$, and let $f \in R[X_1, ..., X_m]$ be a polynomial of individual degree $< d$ in every variable. Then, the preprocessing algorithm 3.2.1 takes the coefficients of f as an input, runs in time*

$$S = d^m \cdot poly(m, d, \log |R|) \cdot O(m(\log m + \log d + \log\log |R|))^m$$

and outputs a data structure of size at most S, and the evaluation algorithm 3.2.2 with random access to the data structure, that is given an evaluation point $\alpha \in R^m$ and computes $f(\alpha)$ in time

$$poly(m, d, \log |R|).$$

Proof.
Correctness:
For $f \in R[X_1, ..., X_m]$ and $\alpha \in R^m$, let $\gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y]$ denote the evaluation of the lifted polynomial on the lifted input. Observe that $\gamma(Y) = \sum_{i=0}^{D} \gamma_i Y^i$ is a polynomial in the variable Y with non-negative coefficients $\gamma_i \geq 0$ and degree $\leq D = (e-1)(m(d-1)+1)$. This holds because each coordinate of $\tilde{\alpha}$ and each coefficient of $\tilde{f}$ is itself a polynomial in Y of degree $\leq (e-1)$ with coefficients in $[\![q]\!]$. Also observe that we can bound the coefficients γ_i by

$$\gamma_i \leq \sum_{i=0}^{D} \gamma_i \leq \gamma(1) < d^m(e(q-1))^{m(d-1)+1} + 1 = M.$$

Where the last inequality follows by substituting $Y = 1$ in all coefficients of $\tilde{f}$ and coordinates of $\tilde{\alpha}$, in which case each of them becomes an integer $\leq e(q-1)$; each monomial term then evaluates to an integer $\leq (e(q-1))^{m(d-1)+1}$, and we sum d^m of them. Next note that reducing $\gamma = \tilde{f}(\tilde{\alpha})$ modulo $Y - M$ is equivalent to evaluating $\gamma(M)$. Also $\gamma(M) < M^{D+1} = r$ so reducing $\gamma(M)$ modulo r and then lifting the answer to $\mathbb{Z}$ is the same as computing $\gamma(M)$ in $\mathbb{Z}$. Therefore, by the correctness of 3.1.4, the value $\tilde{\beta} = \tilde{f}(\tilde{\alpha}) \bmod (r, Y - M)$ computed in step 3 of algorithm 3.2.2 is just $\gamma(M)$. Furthermore, for the values $\tilde{\beta}_i$ computed in step 4, we have:

$$\tilde{\beta} = \sum_{i=0}^{D} \tilde{\beta}_i M^i = \gamma(M) = \sum_{i=0}^{D} \gamma_i M^i$$

with $\tilde{\beta}_i, \gamma_i \in [\![M]\!]$, which implies $\tilde{\beta}_i = \gamma_i$. Therefore, for the polynomial Q computed in step 5, we have $Q = \gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y]$ and hence $Q(Y) \bmod (q, E_1(Y)) = f(\alpha)$.

Preprocessing/Evaluation time:
The efficiency of the preprocessing and the evaluation algorithms follow almost immediately from the efficiency properties of theorem 3.1.4 applied to the ring $\mathbb{Z}_r$. This is because constructing and using the data structure for $\tilde{f}$ contributes the dominant term in the run time and data structure space. Thus the claimed efficiency holds since $\log r = poly(m, d, \log|R|)$ and hence $\log\log r = O(\log m + \log d + \log\log|R|)$.

□

Following very similar techniques to those in the proof of theorem 3.2.1, we can actually extend the result to handle evaluating polynomials over extension rings of t variables $R = \mathbb{Z}_q[Y_1, ..., Y_t]/(E_1(Y_1), ..., E_t(Y_t))$ for arbitrary q and arbitrary non-constant monic polynomials E_i, as long as $t = O(1)$. That is, given R of the above form and a polynomial $f \in R[X_1, ..., X_m]$, we can reduce the problem of preprocessing/evaluating the polynomial over R to the case of preprocessing/evaluating the polynomial over the ring $R' = \mathbb{Z}_{r'}[Y_1, ..., Y_{t-1}]/(E'_1(Y_1), ..., E'_{t-1}(Y_{t-1}))$ for some $r_1 > q$ and polynomials $E'_1, ..., E'_{t-1}$ of greater degree, as in the proof of theorem 3.2.1. This is done by (partially) evaluating at $Y_t = M_t$ for some large enough $M_t \in \mathbb{N}$ and reducing modulo r' and the polynomials $E'_1, ..., E'_{t-1}$ which are chosen large enough to avoid any "wrapping" over the modulus. Repeating this process iteratively, we ultimately reduce to invoking theorem 3.1.4 for the ring $\mathbb{Z}_{r^*}$ for some $r^* \gg q$. Because t is constant, the size of the final ring satisfies $\log r^* = poly(m, d, \log|R|)$ so we achieve the same efficiency properties as in theorem 3.2.1.

3.3 The implementation

Part of our work has been giving the first known implementation for this algorithm for the case $R = \mathbb{Z}_q$, which can be found in [11]. Implementing the algorithm in Rust felt natural. Rust is a programming language that has widely been used in the context of cryptography. For example, some nice crypto libraries such as RustCrypto [31], ring [33] or sodiumoxide [2], and some blockchains such as Parity Substrate [28], Solana [34] or Nervos CKB [25], are built on Rust. There are several reasons that make Rust well suitable to cryptography [26], some of them are:

1. Memory safety: Memory management is one of the key challenges in developing complex systems like blockchain. Rust provides memory safety without garbage collection. This means that you can write code without worrying about memory leaks, dangling pointers, or buffer overflows, which are common issues in other languages.

2. Concurrency and performance: Rust allows for concurrent execution, which is crucial for blockchain systems due to their decentralized nature.

3. Interoperability: Rust can be easily integrated with other languages. This characteristic makes it easy to create blockchain systems that interact seamlessly with existing systems and libraries.

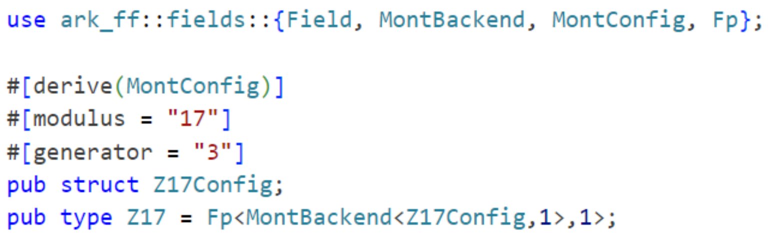
Our first idea was using the arkworks [1] set of libraries. Arkworks is a Rust ecosystem for zkSNARK programming. Libraries in the arkworks ecosystem provide efficient implementations of all components required to implement zkSNARK applications, from generic finite fields to R1CS constraints for common functionalities. This includes implementations of multivariate polynomials over finite fields which matched perfectly with our problem. However, implementing the algorithm in the arkworks environment was challenging for several reasons:

1. **Need of giving generators**: The finite field library doesn't provide a parametric way to build the $\mathbb{Z}_p$ given a prime $p \in \mathbb{N}$. In order to build the field $\mathbb{Z}_p$ you have to provide a generator of the field[2]. However, there is a nice algorithm [32, Section 11.1] to find generators in this case. A concrete example of how the arkworks syntax is used to build $\mathbb{Z}_q$ is shown in Figure 3.1.

2. **Types not buildable on run-time**: Another problem was that the structures/types for $\mathbb{Z}_p$ couldn't be built on run-time. This was fixable by creating a Rust auxiliary file where the $\mathbb{Z}_p$'s where generated until a high bound and accessing them from the original Rust file with the algorithm. To build this auxiliary file we did a Python script that iterated over the primes until a high bound, founded generators for their field and generated the Rust code, which could be then automatically written in the auxiliary Rust file. This approach might be problematic, maybe we are given a polynomial $f \in \mathbb{Z}_q[X_1, X_2, ..., X_m]$ with q, m, d such that $16 \log M$ is higher than our bound and therefore, some of the $\mathbb{Z}_{p_i}$'s wouldn't be built. However, in the context of this algorithm this isn't a huge problem as whenever you want to build the data structure for a new polynomial f you can just pass the Python script the new q, m, d and let it write the new $\mathbb{Z}_p$'s needed.

[2]Recall that a generator of $\mathbb{Z}_p$ is an element $g \in \mathbb{Z}_p$ such that $\{g^n \mod p \mid n \in \mathbb{N}\} = \mathbb{Z}_p^*$ $(= \mathbb{Z}_p \setminus \{0\}$ for p prime).

3. **Types not usable on run-time**: At some point we realized that using the polynomials in arkworks didn't work for this algorithm. The reason is that in this algorithm we have to work with polynomials over several $\mathbb{Z}_{p_i}$ where the primes p_i for $i \in \{1, ..., h\}$ are computed in run-time, and even if the structures/types for $\mathbb{Z}_{p_i}$ are pregenerated, Rust doesn't support dependent types [38] so you couldn't use them parametrically.



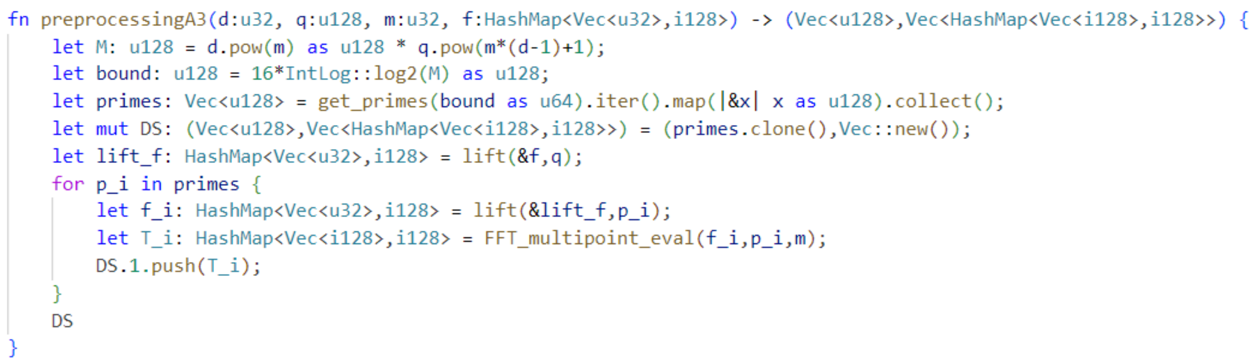
```
use ark_ff::fields::{Field, MontBackend, MontConfig, Fp};

#[derive(MontConfig)]
#[modulus = "17"]
#[generator = "3"]
pub struct Z17Config;
pub type Z17 = Fp<MontBackend<Z17Config,1>,1>;
```

Figure 3.1: Building the type for Z_{17} in ark-ff.

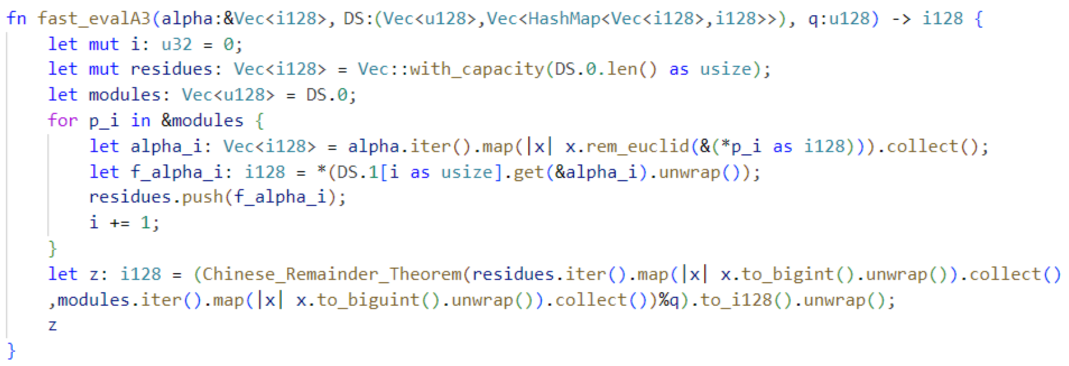
For this reason we had to discard arkworks and use our own type for polynomials. As polynomials, we wanted a data structure that made it easy to access the monomial of a given vector of powers. The straightforward idea is using dictionaries (HashMaps in Rust), where the keys are the vector of powers and the values are the coefficients. For example, the polynomial $f(x_0, x_1) = 3x_0^2x_1^2 + 3x_0^2x_1 + 2x_0x_1^2 + x_0x_1 + 4x_0 + x_1^2 + 1$ would be represented by the HashMap $\{[2,2] : 3,\ [2,1] : 3,\ [2,0] : 0,\ [1,2] : 2,\ [1,1] : 1,\ [1,0] : 4,\ [0,2] : 1,\ [0,1] : 0,\ [0,0] : 1\}$.

The implementation of the preprocessing algorithm, which is shown in Figure 3.1.2, was mainly challenging because of the FFT_multipoint_eval function, which implements the multivariate multipoint evaluation algorithm in 3.1.1. In order to make this algorithm work, we first needed to implement the algorithm that computes univariate multipoint evaluation. The univariate multipoint evaluation algorithm, which we explained in 2.2.4, is relatively easy to implement. However, if we want it to have the right efficiency, we need to use the right algorithm for the underlying polynomial multiplications in $R[X_1, ..., X_m]$, i.e. we need to implement a FFT-based polynomial multiplication, and this was the really hard part.



```
fn preprocessingA3(d:u32, q:u128, m:u32, f:HashMap<Vec<u32>,i128>) -> (Vec<u128>,Vec<HashMap<Vec<i128>,i128>>) {
    let M: u128 = d.pow(m) as u128 * q.pow(m*(d-1)+1);
    let bound: u128 = 16*IntLog::log2(M) as u128;
    let primes: Vec<u128> = get_primes(bound as u64).iter().map(|&x| x as u128).collect();
    let mut DS: (Vec<u128>,Vec<HashMap<Vec<i128>,i128>>) = (primes.clone(),Vec::new());
    let lift_f: HashMap<Vec<u32>,i128> = lift(&f,q);
    for p_i in primes {
        let f_i: HashMap<Vec<u32>,i128> = lift(&lift_f,p_i);
        let T_i: HashMap<Vec<i128>,i128> = FFT_multipoint_eval(f_i,p_i,m);
        DS.1.push(T_i);
    }
    DS
}
```

Figure 3.2: Preprocessing algorithm 3.1.2 Rust.



```rust
fn fast_evalA3(alpha:&Vec<i128>, DS:(Vec<u128>,Vec<HashMap<Vec<i128>,i128>>), q:u128) -> i128 {
    let mut i: u32 = 0;
    let mut residues: Vec<i128> = Vec::with_capacity(DS.0.len() as usize);
    let modules: Vec<u128> = DS.0;
    for p_i in &modules {
        let alpha_i: Vec<i128> = alpha.iter().map(|x| x.rem_euclid(&(*p_i as i128))).collect();
        let f_alpha_i: i128 = *(DS.1[i as usize].get(&alpha_i).unwrap());
        residues.push(f_alpha_i);
        i += 1;
    }
    let z: i128 = (Chinese_Remainder_Theorem(residues.iter().map(|x| x.to_bigint().unwrap()).collect()
    ,modules.iter().map(|x| x.to_biguint().unwrap()).collect())%q).to_i128().unwrap();
    z
}
```

Figure 3.3: Evaluation algorithm 3.1.3 in Rust.

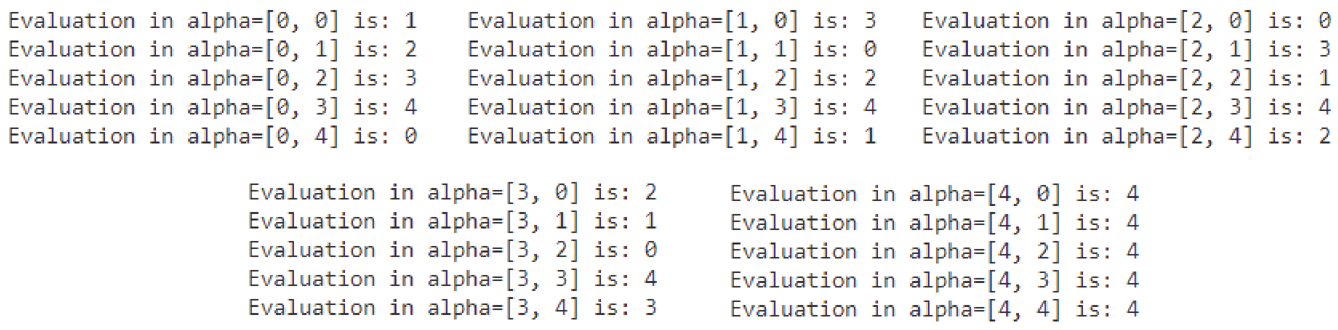
As we've explained in the preliminaries, implementing FFT-based multiplication is "easy" if the ring R over which the polynomials are defined contains certain primitive roots of unity, i.e. holds definition 2.2.5. This is not the case in general for $R = \mathbb{Z}_p$ with p prime. There are two ways to overcome this:

1. We can either lift $f, g \in \mathbb{Z}_p[X_1, ..., X_m]$ to $\overline{f}, \overline{g} \in \mathbb{C}[X_1, ..., X_m]$ and perform FFT-based multiplication directly in $O(n \cdot \log n)$ operations in $\mathbb{C}$ (where $\deg f, \deg g < n$).

2. Or we can apply the algorithm by Schönhage and Strassen, which adjoins "virtual" roots of unity, and then applies FFT multiplication. This algorithm would take $O(n \cdot \log n \cdot \log \log n)$ operations in $\mathbb{Z}_p$ (where $\deg f, \deg g < n$).

Ideally, we would like to apply the second approach but implementing Schönhage/Strassen's algorithm requires good libraries to represent the polynomials, as we need to apply FFT based multiplication over roots of unity $w \in R[x]/\langle x^{2m} + 1\rangle$. As we weren't aware of good libraries to implement this, we followed the first approach and we used the polynomen [30] Rust library to implement the FFT based multiplication and the univariate multipoint evaluation.

Example 3.3.1. Let's give the implementation a test with a toy example to check it's working well. Let $f(x_0, x_1) = x_0x_1 + 2x_0 + x_1 + 1 \in \mathbb{Z}_5[x_0, x_1]$. If we apply the preprocessing algorithm (shown in 3.2) to f and then apply the evaluation algorithm (shown in 3.3) to evaluate f on every point $\alpha \in \mathbb{Z}_5^2$ we get the following results:



```
Evaluation in alpha=[0, 0] is: 1    Evaluation in alpha=[1, 0] is: 3    Evaluation in alpha=[2, 0] is: 0
Evaluation in alpha=[0, 1] is: 2    Evaluation in alpha=[1, 1] is: 0    Evaluation in alpha=[2, 1] is: 3
Evaluation in alpha=[0, 2] is: 3    Evaluation in alpha=[1, 2] is: 2    Evaluation in alpha=[2, 2] is: 1
Evaluation in alpha=[0, 3] is: 4    Evaluation in alpha=[1, 3] is: 4    Evaluation in alpha=[2, 3] is: 4
Evaluation in alpha=[0, 4] is: 0    Evaluation in alpha=[1, 4] is: 1    Evaluation in alpha=[2, 4] is: 2

                 Evaluation in alpha=[3, 0] is: 2    Evaluation in alpha=[4, 0] is: 4
                 Evaluation in alpha=[3, 1] is: 1    Evaluation in alpha=[4, 1] is: 4
                 Evaluation in alpha=[3, 2] is: 0    Evaluation in alpha=[4, 2] is: 4
                 Evaluation in alpha=[3, 3] is: 4    Evaluation in alpha=[4, 3] is: 4
                 Evaluation in alpha=[3, 4] is: 3    Evaluation in alpha=[4, 4] is: 4
```

Figure 3.4: Implementation evaluations.

Now, we can compare the output with the real evaluations of the polynomial in table 3.1 and check that they match

		α_0				
		0	1	2	3	4
α_1	0	1	3	0	2	4
	1	2	0	3	1	4
	2	3	2	1	0	4
	3	4	4	4	4	4
	4	0	1	2	3	4

Table 3.1: Real evaluations.

In terms of efficiency, the algorithm seems to still be far from being practical. We ran experiments for $d \in \{1, ..., 10\}$, for $q \in \{2, 3, 5, 7, 11\}$ and for $m \in \{1, 2, 3\}$. We had to stop test there as the times/memory needed for preprocessing where already huge. For $m = 3$, $d = 3$ and $q = 5$ the algorithm already takes 30 hours to preprocess and needs 59.957 GB of memory. All the data from the test's can be found in [11] in the file *test.json*.

Chapter 4

DEPIR

The algorithm by Kedlaya and Umans has already been applied to solve an open problem in cryptography. The PIR (private information retrieval) allows a client to read data from a public database held on a remote server, without revealing to the server which locations he is reading. In a doubly efficient PIR (DEPIR), the database is first preprocessed, but the server can subsequently answer any client's query in time that is sub-linear in the database size.

Ring LWE:
The construction on [22] is built on top of the Ring LWE (Learning With Errors) assumption, which is stated as follows:

Definition 4.0.1 (The RingLWE assumption [23]). Let $n = n(\lambda) \in \mathbb{Z}$ and $q = q(\lambda) \in \mathbb{Z}$ be integers. Define the ring $R = \mathbb{Z}_q[Z]/(Z^n + 1)$ and let $\mathcal{X} = \mathcal{X}(\lambda)$ denote an error distribution over the ring R. The RingLWE$_{n,q,\mathcal{X}}$ assumption, states that for any $l = poly(\lambda)$ it holds that

$$\{(a_i, a_i \cdot s + e_i)\}_{i\in[l]} \approx_c \{(a_i, u_i)\}_{i\in[l]}$$

where $s \leftarrow R$, $a_i \leftarrow R$, $e_i \leftarrow \mathcal{X}$ and $u_i \leftarrow R$.

ASHE:
On top of this assumption they define ASHE (Algebraic Somewhat Homomorphic Encryption). An ASHE is a symmetric-key CPA-secure encryption scheme, where the plaintext space is $\mathbb{Z}_d$ for a prime d of our choosing, and the ciphertext space is some corresponding polynomial ring R. An ASHE allows us to homomorphically evaluate multivariate polynomials $f(X_1, ..., X_m)$ with degree $< d$ in each variable over encrypted data, just by evaluating the same polynomial (appropriately lifted) over ciphertexts. An ASHE consists of five PPT algorithms: **Setup**, **Gen**, **Enc**, **Dec**, **Lift**. In the case of the Ashe for the DEPIR construction they are the following:

- $params :=$ **Setup**$(1^\lambda, 1^d, 1^D, N)$: Set the gap parameter $t := D\log d + \log N + \log d + 1$ and choose $n = poly(\lambda, t)$, $q = \lambda^{poly(t)}$ and a β-bounded error distribution $\mathcal{X}$ as defined in [22, Section 2.2] so that $q > (2\beta n)^t > 2Nd(d(\beta + 1)n)^D$ and q is relatively prime to d. Define the rings

$$Q := \mathbb{Z}_q[Z]/(Z^n + 1), \quad R := Q[Y]/(Y^D + 1) \cong \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1).$$

- $s \leftarrow$ **Gen**(1^λ): Sample $s \xleftarrow{\$} Q$ uniformly at random.

- $ct \leftarrow$ **Enc**(s, μ): Reinterpret $\mu \in \mathbb{Z}_d$ as an element of Q. Sample $a \xleftarrow{\$} Q$ and $e \xleftarrow{\$} \mathcal{X}$. Let
$$b = a \cdot s + d \cdot e + \mu \in Q.$$
Define $ct \in R$ as the formal polynomial with a symbolic variable Y via:
$$ct(Y) = -a \cdot Y + b.$$

- $\mu :=$ **Dec**(s, ct): Interpret $ct \in R$ as a formal polynomial $ct(Y) \in Q[Y]/(Y^D + 1)$ and compute $g = ct(s) \in Q$ to be its evaluation on $s \in Q$. Interpret $g \in Q$ as a formal polynomial $g(Z) \in \mathbb{Z}_q[Z]/(Z^n + 1)$ and let $h = g(0) \in \mathbb{Z}_q$ be its constant term. Reinterpret h as an element of $\mathbb{Z}_d$ and output it.

- $\overline{\mu} :=$ **Lift**(μ): Reinterpret $\mu \in \mathbb{Z}_d$ as an element of R.

Also, this scheme satisfies the following definitions of correctness, security and efficiency over the Ring LWE assumption[1]:

- **Correctness:** We require that for all plaintexts $\mu_1, ..., \mu_m \in \mathbb{Z}_d$ and for any polynomial $f(X_1, ..., X_m)$ over $\mathbb{Z}_d$ consisting of at most N terms and total degree $< D$, it holds that:
$$\Pr\left(\textbf{Dec}(s, ct') = f(\mu_1, ..., \mu_m) : \begin{array}{r} params := \textbf{Setup}(1^\lambda, 1^d, 1^D, N); \\ s \leftarrow \textbf{Gen}(params); \\ ct_j \leftarrow \textbf{Enc}(s, \mu_j) \text{ for all } j \in [m]; \\ \overline{f} := \textbf{Lift}(f); \\ ct' := \overline{f}(ct_1, ..., ct_m) \end{array} \right) = 1.$$

- **Security:** We require the standard symmetric-key IND-CPA security [17, Definition 6.8] for the encryption scheme (Gen,Enc,Dec) when $params :=$ Setup$(1^\lambda, 1^d, 1^D, N)$ for any $N = poly(\lambda)$, $d = poly(\lambda)$ and $D = poly(\lambda)$.

- **Efficiency:** We require that the description length of ring elements, the run-time of the ring operations and the run-time of Setup, Gen, Enc, Dec and Lift are all bounded by $poly(\lambda, D, d, \log N)$.

DEPIR:

Over this ASHE we can now construct the DEPIR solution. At a high level, an unkeyed DEPIR scheme is a protocol between a Server and an arbitrary Client. The protocol consists of four algorithms, **Prep**, **Query**, **Resp** and **Dec**, which are performed in two phases of the protocol, preprocessing and query, illustrated as follows:

Preprocessing. Server has a database $DB \in \{0, 1\}^N$ and runs the deterministic preprocessing algorithm $\widetilde{DB}$:=**Prep**$(1^\lambda, DB)$. It stores the static data structure $\widetilde{DB}$ in random-access memory.

Query. Client knows the database size N, has index $i \in [\![N]\!]$ and wants to learn the entry $DB[i]$ without revealing i.

1. Client runs $(ct, s) \leftarrow$**Query**$(1^\lambda, N, i)$ to generate query ciphertext ct that it sends to Server, and a query-specific secret decoding key s that it keeps locally.

[1]A proof for this can be found in [22, Theorem 3.2]

2. Server responds with the answer $ans \leftarrow \mathbf{Resp}(\widetilde{DB}, ct)$ using random-access to the data structure $\widetilde{DB}$.
3. Client decodes the answer using the algorithm $b := \mathbf{Dec}(s, ans)$ to learn the bit $b = DB[i]$.

The algorithms (**Prep**,**Query**,**Resp**,**Dec**) should satisfy the following correctness, security and efficiency definitions:

Correctness: Honest execution of **Prep**, **Query**, **Resp** and **Dec** successfully recovers requested data items with probability 1. That is, for every $DB \in \{0,1\}^N$ and every $i \in [\![N]\!]$, it holds that

$$\Pr\left(\mathbf{Dec}(s, ans) = DB[i] : \begin{array}{c} \widetilde{DB} := \mathbf{Prep}(1^\lambda, DB); \\ (ct, s) \leftarrow \mathbf{Query}(1^\lambda, N, i); \\ ans := \mathbf{Resp}(\widetilde{DB}, ct) \end{array}\right) = 1.$$

Security: No efficient adversary can distinguish the queries output by **Query** on input index i_0 and i_1. Namely, we can define the following game between a challenger and an adversary $\mathcal{A}$:

1. $(i_0, i_1, 1^N, aux) \leftarrow \mathcal{A}(1^\lambda)$: $\mathcal{A}$ selects a size N, a challenge index pair $i_0, i_1 \in [\![N]\!]$ and auxiliary information aux.
2. $b \leftarrow \{0,1\}$; $(s, ct) \leftarrow \mathbf{Query}(1^\lambda, N, i_b)$: The challenger selects a random bit b and generates a sample query ct for the chosen index i_b.
3. $b' \leftarrow \mathcal{A}(aux, ct)$: $\mathcal{A}$ outputs a guess for b, given the query ct.

We require that for every PPT adversary $\mathcal{A}$, there exists a negligible function $negl$ such that the distinguishing advantage of $\mathcal{A}$ in the above security game is

$$|\Pr[b' = b] - 1/2| \leq negl(\lambda).$$

Efficiency: Suppose that **Resp** is given random accesses to $\widetilde{DB}$. We say the scheme (**Prep**, **Query**, **Resp**, **Dec**) is doubly efficient if **Prep** runs in time $poly(\lambda, N)$ and **Query**, **Resp**, **Dec** run in time sublinear in N. Ideally, we want **Prep** to run in quasi-linear time in N and **Query**, **Resp**, **Dec** run in polylogarithmic time in N.

Now, let's present the DEPIR construction in [22]. The construction relies on some parameters d, m such that d is prime and $d^m > N$. The construction consists of 3 steps that carefully fit together:

1. Express the function $f_{DB}(i) = DB[i]$ as an m-variate polynomial of individual degree $< d$ over $\mathbb{Z}_d$, where the inputs are the base-d digits of the index i.

2. Construct a basic PIR scheme by using ASHE to homomorphically evaluate the polynomial f_{DB} over the encryptions of the base-d digits of i; the ciphertext consists of m ring elements and the server computation consists of evaluating the "lifted" polynomial $\overline{f}_{DB}$ over them.

3. Preprocess the polynomial $\overline{f}_{DB}$ into a data structure $\widetilde{DB}$ that enables fast online evaluation using Theorem 3.2.1.

This DEPIR construction is implemented by the following four algorithms:

Parameters: The database size N, determines some parameters $d, m \in \mathbb{N}$ such that d prime and $d^m \geq N$. Let $params := \text{ASHE.Setup}(1^\lambda, 1^d, 1^D, d^m)$ with $D = dm$, which determines a ring $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$. Without loss of generality, we implicitly assume all algorithms have access to these parameters, which they can derive from λ, N.

Prep$(1^\lambda, DB)$:

1. Let $f_{DB} := \text{ToPoly}_{d,m}(DB)$. This algorithm interpolates the coefficients of a m-variate polynomial over $\mathbb{Z}_d$ with individual degree $< d$ in each variable such that $f_{DB}(\text{base}_{d,m}(i)) = DB[i]$ for all $i \in [\![N]\!]$. The total degree of f_{DB} is $< dm$ and the total number of terms is $\leq d^m$.
2. Lift $f_{DB} \in \mathbb{Z}_d[X_1, ..., X_m]$ to $\overline{f}_{DB} \in R[X_1, ..., X_m]$ via $\overline{f}_{DB} := \text{ASHE.Lift}(f_{DB})$.
3. Invoke the preprocessing algorithm (3.1.2 for general R) on the polynomial $\overline{f}_{DB}$ over R and let the resulting data structure be $\widetilde{DB}$.
4. Output $\widetilde{DB}$.

Query$(1^\lambda, N, i)$:

1. Let $(i_1, ..., i_m) = \text{base}_{d,m}(i)$ be the base-d digits of i.
2. Sample $s \leftarrow \text{ASHE.Gen}(1^\lambda)$.
3. For each $j \in [m]$, encrypt i_j by invoking $ct_j \leftarrow \text{ASHE.Enc}(s, i_j)$.
4. Output $(ct = (ct_1, ..., ct_m), s)$.

Resp$(\widetilde{DB}, ct)$:

1. Parse $ct = (ct_1, ..., ct_m)$.
2. Invoke the evaluation algorithm (3.1.3 for general R) to evaluate $ans = \overline{f}_{DB}(ct_1, ..., ct_m)$ using random-access to the data structure $\widetilde{DB}$.
3. Output ans.

Dec(s, ans):

1. Output $\text{ASHE.Dec}(s, ans)$.

And, as expected, this construction satisfies the three definitions above, correctness, security and efficiency.

- **Correctness:** Consider any $DB \in \{0,1\}^N$ and any $i \in [\![N]\!]$ with $(i_1, ..., i_m) = \text{base}_{d,m}(i)$. Let $\widetilde{DB} := \text{Prep}(1^\lambda, DB)$, $(ct, s) \leftarrow \text{Query}(1^\lambda, N, i)$, $ans := \text{Resp}(\widetilde{DB}, ct)$ and $b = \text{Dec}(s, ans)$. The polynomial f_{DB} computed during the preprocessing satisfies $f_{DB}(i_1, ..., i_m) = DB[i]$. Also, by the correctness of polynomial evaluation with preprocessing (Theorem 3.2.1), we have that the value ans computed during Resp satisfies $ans = \overline{f}_{DB}(ct_1, ..., ct_m)$. Hence, by the definition of correctness for ASHE, we have that:

$$b = \text{ASHE.Dec}(s, ans) = \text{ASHE.Dec}(s, \overline{f}_{DB}(ct_1, ..., ct_m)) = f_{DB}(i_1, ..., i_m) = DB[i].$$

- **Security:** The security of the DEPIR follows directly from that of ASHE, since the adversary only sees m ASHE ciphertexts.
- **Efficiency:** We calculate the computation time and output size for each algorithm. Recall that, by the definition of ASHE efficiency, the bit-length of ring elements and the run-time of the ring operations are bounded by $poly(\lambda, d, m)$.
  - Prep: The interpolation takes time $d^m \cdot m \cdot poly(\log d)$ and ASHE.Lift takes time $d^m \cdot poly(\lambda, d, m)$. By Theorem 3.2.1, computing the data structure $\widetilde{DB}$ takes time:

$$\begin{aligned} & d^m \cdot m^m \cdot poly(m, d, \log |R|) \cdot O(\log m + \log d + \log\log |R|)^m = \\ & = d^m \cdot m^m \cdot poly(m, d, \lambda) \cdot O(\log m + \log d + \log \lambda)^m, \end{aligned}$$

    which therefore dominates the run-time of Prep and also bounds the size of $\widetilde{DB}$.
  - Query: The run-time of Query is bounded by that of ASHE.Gen and that of running m copies of ASHE.Enc, which is bounded by $poly(\lambda, d, m)$ by the definition of ASHE efficiency.
  - Resp: By Theorem 3.2.1, the run-time of Resp is bounded by $poly(m, d, \log |R|) = poly(\lambda, d, m)$.
  - Dec: By the definition of ASHE efficiency, the run-time of Dec is bounded by $poly(\lambda, d, m)$.

Updatable DEPIR:

Later in the paper they also add updatability to the scheme, which means that the server can efficiently update bits of the database, by setting $DB[i] := b$, and correspondingly update the preprocessed data structure $\widetilde{DB}$ in sublinear time.

Briefly, the updatable DEPIR data structure $\widetilde{DB}$ consists of a hierarchy of $L = \log N$ levels of exponentially increasing size. Each level $l \in \{0, ..., L\}$ contains a database DB_l consisting of 2^l location/value pairs (i, b) sorted according to the location $i \in [N]$ and a preprocessed data structure $\widetilde{DB_l}$ for DB_l under the basic DEPIR scheme. Initially, all the data is contained inside the $N = 2^L$ pairs $(i, DB[i])$ in the largest level L, and the remaining levels are empty. To update $DB[i] := b$, the server first puts the pair (i, b) in a temporary buffer and, after every 2^l updates, the server will take all the pairs (i, b) contained in the first l levels $DB_0, ..., DB_l$ and in the temporary buffer, and sort them according to location i into the database DB_l at level l, taking only the freshest copy from the smallest level if there are conflicting location/value pairs (i, b) with the same i at different levels. The server then applies the basic DEPIR-preprocessing to this database DB_l to create $\widetilde{DB_l}$, and empties all the data from levels $0, ..., l-1$. The DEPIR-preprocessed data structure $\widetilde{DB_l}$ at every level l allows the client to privately execute arbitrary RAM computation over the data in DB_l by making a sequence of basic DEPIR queries to the server. Therefore, to query for a location i in the updatable DEPIR scheme, the client makes a sequence of queries to the basic DEPIR to perform a binary search for the location i in every level $l \in [L]$ and takes the freshest such tuple (i, b) found in the smallest level.

A downside of this updatable DEPIR scheme is that the PIR protocol now requires $O(\log N)$ rounds of interaction for the client to run binary search by making DEPIR queries.

However, they achieve to reduce this to the optimal 2 rounds by using RAM-FHE [18]. In particular, RAM-FHE allows the client to send an encrypted index i to the server, and the server can homomorphically perform binary search over the data DB_l in each level l, by using random access to the data structure $\widetilde{DB}_l$ to efficiently derive the encrypted output, without additional interaction.

Chapter 5

Prover-efficient polynomial commitment

5.1 Vector commitments

One of the key building blocks of our new polynomial commitment is a vector commitment to the data structure produced by the preprocessing algorithm. The vector commitment schemes were introduced in [12]. A vector commitment allows one to commit to an ordered sequence of values in such a way that it is later possible to open the commitment only with respect to a specific position.

A vector commitment scheme consists of four algorithms: **VC.Setup**, **VC.Commit**, **VC.OpenPos** and **VC.VerifyPos**.

- **VC.Setup**$(1^\lambda, s)$: Given the security parameter λ and the size s of the committed vector (with $s = poly(\lambda)$), the setup outputs some public parameters **PK** which implicitly defines the message space $\mathcal{M}$.
- **VC.Commit**$(\mathbf{PK}, m; r)$: Given $m \in \mathcal{M}^s$, the public parameters **PK** and (possibly) randomness r, the committing algorithm outputs a commitment string $\mathcal{C}$ and an auxiliary information aux.
- **VC.OpenPos**$(\mathbf{PK}, m_i, i, aux)$: Given $m_i \in \mathcal{M}$ and $i \in [\![s]\!]$, this algorithm is run by the committer to produce a proof w_i that m_i is the i-th committed message.
- **VC.VerifyPos**$(\mathbf{PK}, \mathcal{C}, i, m_i, w_i)$: Given $m_i \in \mathcal{M}$ and $i \in [\![s]\!]$, the verification algorithm accepts (i.e., it outputs 1) only if w_i is a valid proof that $\mathcal{C}$ was created to a sequence $m \in \mathcal{M}^s$ such that $m[i] = m_i$. Otherwise it outputs 0.

Now, we define a vector commitment scheme to be secure if it satisfies the following properties:

- **Correctness**: Let $PK \leftarrow$ **VC.Setup**$(1^\lambda, s)$ and $m \in \mathcal{M}^s$. For any $(\mathcal{C}, aux) \leftarrow$ **VC.Commit**$(PK, m; r)$ for a (possibly) randomness r:

$$\Pr\left(\mathbf{VC.VerifyPos}(PK, \mathcal{C}, i, m_i, \mathbf{VC.OpenPos}(PK, m_i, i, aux)) = 1\right) = 1$$

  i.e. the output of the **VC.OpenPos** algorithm is successfully verified by the **VC.VerifyPos** algorithm.

- **Position Binding**: For any PPT adversary $\mathcal{A}$:

$$\mathbf{Adv}^{\text{PosBind}}_{\mathcal{A},VC}(\lambda) = \Pr\begin{pmatrix} PK \leftarrow \mathbf{VC.Setup}(1^\lambda, s),\ (\mathcal{C}, i, m_i, w_i, m_i', w_i') \leftarrow \mathcal{A}(PK) : \\ \mathbf{VC.VerifyPos}(PK, \mathcal{C}, i, m_i, w_i) = 1 \wedge \\ \mathbf{VC.VerifyPos}(PK, \mathcal{C}, i, m_i', w_i') = 1 \ \wedge\ m_i \neq m_i' \end{pmatrix} = negl(\lambda)$$

Intuitively, correctness holds if honestly generated proofs for positions are accepted by the verification algorithm and position binding holds if generating proofs for different messages in the same position is infeasible (which implies that the committer can't lie with the proofs).

Vector commitments can also be required to be hiding. Informally, a vector commitment is hiding if an adversary cannot distinguish whether a commitment was created to a sequence $(m_1, ..., m_s)$ or to $(m_1', ..., m_s')$, even after seeing some openings (at positions i where the two sequences agree).

5.2 Polynomial commitments

The polynomial commitment schemes were introduced in [19]. A polynomial commitment allows one to commit to a polynomial in a way that it is later possible to open evaluations (generate proofs that evaluations are correct) without leaking the entire polynomial.

A polynomial commitment scheme consists of four algorithms: **Setup**, **Commit**, **OpenEval** and **VerifyEval**.

- **Setup**$(1^\lambda, m, d)$: Given the security parameter λ, the number of variables m and a bound for the degree of each individual variable d, the setup outputs some public parameters **PK** which implicitly define the polynomial space

$$\{p \in R[X_1, ..., X_m] \mid p \text{ has individual degree} < d \text{ in each variable}\}.$$

- **Commit**$(\mathbf{PK}, p(X); r)$: Outputs a commitment $\mathcal{C}$ to a polynomial $p(X)$ for the public key **PK** and (possibly) randomness r, and some associated decommitment information aux. (In some constructions, aux is null).
- **OpenEval**$(\mathbf{PK}, p(X), i, aux)$: Outputs w_i, where w_i is a witness for the evaluation $p(i)$ of $p(X)$ at the index i and aux is the decommitment information.
- **VerifyEval**$(\mathbf{PK}, \mathcal{C}, i, p(i), w_i)$: Verifies that $p(i)$ is indeed the evaluation at the index i of the polynomial committed in $\mathcal{C}$. If so it outputs 1, otherwise it outputs 0.

Now, we define a polynomial commitment scheme to be secure if it satisfies the following properties:

- **Correctness**: Let $PK \leftarrow \mathbf{Setup}(1^\lambda, m, d)$ and $p(X) \in R[X_1, ..., X_m]$ with degree $< d$ in each variable. For any $(\mathcal{C}, aux) \leftarrow \mathbf{Commit}(PK, p(X); r)$ for a (possibly) randomness r:

$$\Pr\left(\mathbf{VerifyEval}(PK, \mathcal{C}, i, p(i), \mathbf{OpenEval}(PK, p(X), i, aux)) = 1\right) = 1$$

  i.e. the output of the **OpenEval** algorithm is successfully verified by the **VerifyEval** algorithm.

- **Commitment Binding**: For any PPT adversary $\mathcal{A}$:

$$\mathbf{Adv}^{\text{ComBind}}_{\mathcal{A},PC}(\lambda) = \Pr\begin{pmatrix} PK \leftarrow \mathbf{Setup}(1^\lambda, m, d),\ (p(X), r, p'(X), r') \leftarrow \mathcal{A}(PK) : \\ \mathbf{Commit}(PK, p(X); r) = \mathbf{Commit}(PK, p'(X); r') \wedge \\ p(X) \neq p'(X) \end{pmatrix} = negl(\lambda)$$

- **Evaluation Binding**: For any PPT adversary $\mathcal{A}$:

$$\mathbf{Adv}^{\text{EvBind}}_{\mathcal{A},PC}(\lambda) = \Pr\begin{pmatrix} PK \leftarrow \mathbf{Setup}(1^\lambda, m, d),\ (\mathcal{C}, i, y, w_i, y', w_i') \leftarrow \mathcal{A}(PK) : \\ \mathbf{VerifyEval}(PK, \mathcal{C}, i, y, w_i) = 1 \wedge \\ \mathbf{VerifyEval}(PK, \mathcal{C}, i, y', w_i') = 1 \ \wedge \ y \neq y' \end{pmatrix} = negl(\lambda)$$

As with vector commitments, we can also require polynomial commitments to be hiding. Informally, a polynomial commitment is hiding if an adversary cannot distinguish whether a commitment was created to a polynomial $p \in R[X_1, ..., X_m]$ or to $p' \in R[X_1, ..., X_m]$ both with individual degree $< d$ in each variable, even after seeing some openings (at $x \in R^m$ where both polynomials agree).

Some extra properties, based on the ones for functional commitments in [13] are the following:

- **Weak Evaluation Binding**: For any PPT adversary $\mathcal{A}$:

$$\mathbf{Adv}^{\text{wEvBind}}_{\mathcal{A},PC}(\lambda) = \Pr\begin{pmatrix} PK \leftarrow \mathbf{Setup}(1^\lambda, m, d),\ (p(X), r, i, y, w_i) \leftarrow \mathcal{A}(PK) : \\ (\mathcal{C}, aux) \leftarrow \mathbf{Commit}(PK, p(X); r) \wedge \\ \mathbf{VerifyEval}(PK, \mathcal{C}, i, y, w_i) = 1 \ \wedge \ y \neq p(i) \end{pmatrix} = negl(\lambda)$$

- **Strong Evaluation Binding**: For any PPT adversary $\mathcal{A}$:

$$\mathbf{Adv}^{\text{sEvBind}}_{\mathcal{A},PC}(\lambda) = \Pr\begin{pmatrix} PK \leftarrow \mathbf{Setup}(1^\lambda, m, d),\ (\mathcal{C}, \{i_j, y_j, w_j\}_{j=1}^{N}) \leftarrow \mathcal{A}(PK) : \\ \forall j \in [N] : \mathbf{VerifyEval}(PK, \mathcal{C}, i_j, y_j, w_j) = 1 \wedge \\ \nexists\, p \in R[X_1, ..., X_m] \text{ with degree} < d \text{ in each variable} : \\ \forall j \in [N] : p(i_j) = y_j \end{pmatrix} = negl(\lambda)$$

  Where the last condition is equivalent to saying that there doesn't exist a solution to the following linear system:

$$\mathbf{M} \cdot \mathbf{z} = \mathbf{y}$$

  where $\mathbf{M}$ is the $(N \times d^m)$-size matrix consisting of all the monomials evaluated in the i_j's, $\mathbf{z}$ is the d^m-size vector of coefficients of the polynomial and $\mathbf{y}$ is the N-size vector consisting of the y_j's.

Now, let's prove that this definitions satisfy that **Strong Evaluation Binding** $\implies$ **Evaluation Binding** $\implies$ **Weak Evaluation Binding** $\implies$ **Commitment Binding** as expected.

Proposition 5.2.1 (**Weak Evaluation Binding** $\implies$ **Commitment Binding**). *If a PC satisfies correctness and weak evaluation binding then it also satisfies commitment binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks commitment binding. Let $PK \leftarrow \mathbf{Setup}(1^\lambda, m, d)$, then by assumption $(p(X), r, p'(X), r') \leftarrow \mathcal{A}(PK)$ such that $\mathbf{Commit}(PK, p(X); r) = \mathbf{Commit}(PK, p'(X); r') = (\mathcal{C}, aux)$ and $p(X) \neq p'(X)$. As $p(X) \neq p'(X)$ there must exist an $i \in R^m$ such that $p(i) \neq p'(i)$. Let $\mathcal{B}$ be a PPT such that

$(p(X), r, i, p'(i), w_i') \leftarrow \mathcal{B}(PK)$ where $w_i' \leftarrow \textbf{OpenEval}(PK, p'(X), i, aux)$. Then, by correctness we have that:

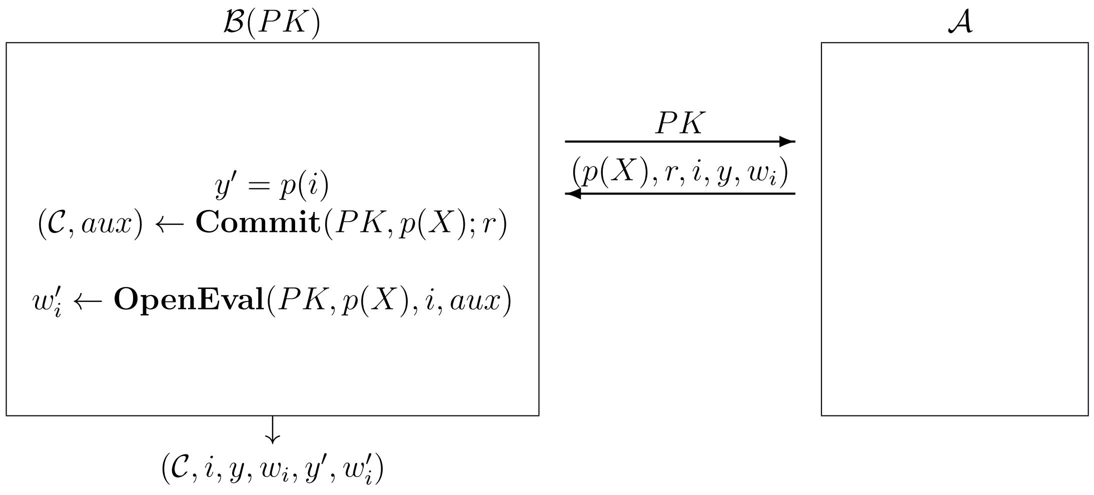
$$\textbf{VerifyEval}(PK, \mathcal{C}, i, p'(i), w_i') = 1$$

but $p'(i) \neq p(i)$.

$\square$

Proposition 5.2.2 (**Evaluation Binding $\implies$ Weak Evaluation Binding**). *If a PC satisfies correctness and evaluation binding then it also satisfies weak evaluation binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks weak evaluation binding. Let $\mathcal{B}$ be the following PPT:



$\mathcal{B}(PK)$
$\mathcal{A}$
PK
$(p(X), r, i, y, w_i)$
$y' = p(i)$
$(\mathcal{C}, aux) \leftarrow \textbf{Commit}(PK, p(X); r)$
$w_i' \leftarrow \textbf{OpenEval}(PK, p(X), i, aux)$
$(\mathcal{C}, i, y, w_i, y', w_i')$

By correctness of the PC we have that $\textbf{VerifyEval}(PK, \mathcal{C}, i, y', w_i') = 1$. Also, if $\mathcal{A}$ wins the game then $y \neq p(i) = y'$ and $\textbf{VerifyEval}(PK, \mathcal{C}, i, y, w_i) = 1$ and therefore:

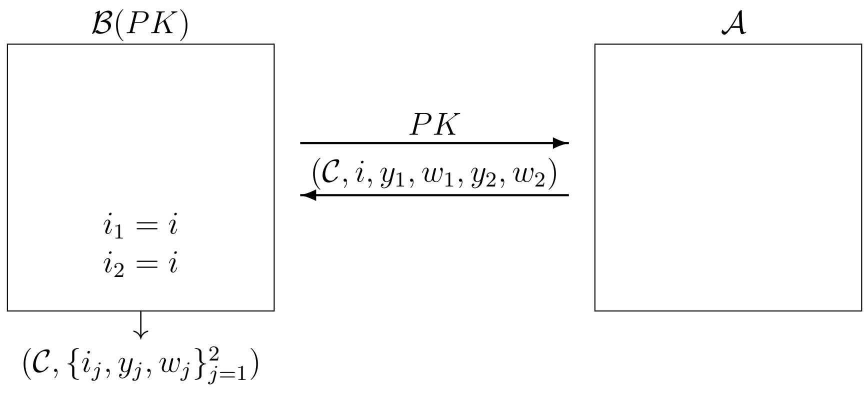
$$\textbf{Adv}^{\text{EvBind}}_{\mathcal{B},PC}(\lambda) = \textbf{Adv}^{\text{wEvBind}}_{\mathcal{A},PC}(\lambda)$$

By the assumption then $\mathcal{B}$ breaks evaluation binding.

$\square$

Proposition 5.2.3 (**Strong Evaluation Binding $\implies$ Evaluation Binding**). *If a PC satisfies strong evaluation binding then it also satisfies evaluation binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks evaluation binding. Let $\mathcal{B}$ be the following PPT:



$\mathcal{B}(PK)$
$\mathcal{A}$
PK
$(\mathcal{C}, i, y_1, w_1, y_2, w_2)$
$i_1 = i$
$i_2 = i$
$(\mathcal{C}, \{i_j, y_j, w_j\}_{j=1}^2)$

If $\mathcal{A}$ wins then $\forall j \in \{1, 2\}$, $\textbf{VerifyEval}(PK, \mathcal{C}, i_j, y_j, w_j) = 1$. Also, in that case $y_1 \neq y_2$ and therefore it can't exist $p(X) \in R[X]$ such that $p(i_1) = y_1 \ \wedge \ p(i_2) = y_2$ (because then $y_1 = p(i_1) = p(i) = p(i_2) = y_2$). Therefore:

$$\textbf{Adv}^{\text{sEvBind}}_{\mathcal{B},PC}(\lambda) = \textbf{Adv}^{\text{EvBind}}_{\mathcal{A},PC}(\lambda)$$

By the assumption then $\mathcal{B}$ breaks strong evaluation binding.

$\square$

5.3 Prover-efficient polynomial commitment

Based on the preprocessing algorithm, we've constructed a polynomial commitment scheme in which the prover computing an evaluation proof for $p(x)$ can run in time sublinear in $|p|$. This can be done thanks to preprocessing $p(x)$ at commitment time, an commiting to the resulting data structure DS using a vector commitment. Then, we can send proofs for the positions in DS that the verifier needs to compute $p(x)$ as proofs for the polynomial commitment, and he can verify evaluations using CRT.

To implement this construction we need the following building blocks:

- **Vector commitment scheme**: $VC = (VC.\text{Setup},\ VC.\text{Commit},\ VC.\text{OpenPos},\ VC.\text{VerifyPos})$ which has to satisfy a special efficiency property for the opening algorithm (stated after).
- **Polynomial preprocessing scheme**: $PP = (PP.\text{PreProcess},\ PP.\text{Lookup},\ PP.\text{Reconstruct})$ where:
  - $PP.\text{PreProcess}(p(X))$: Given an m-variable polynomial $p(X)$ of individual degree $< d$ over a ring $R[X]$, outputs a data structure DB of size S. Notice S is a function of (R, m, d).
  - $PP.\text{Lookup}(R, m, d, x)$: Given the description of a ring R, the number of variables m, the degree d, and an evaluation point $x \in R^m$, outputs a tuple of k indices $i_1, ..., i_k \in [\![S]\!]$.
  - $PP.\text{Reconstruct}(R, x, DB[i_1], ..., DB[i_k])$: Given the evaluation point x and k values of DS, outputs the result of the evaluation $y = p(x)$.

  We'll also need this polynomial preprocessing scheme to satisfy the following properties:

  1. **Correctness**: For any $p(X) \in R[X_1, ..., X_m]$ with individual degree $< d$ and for any $x \in R^m$ we have that:

  $$\Pr\begin{pmatrix} DB \leftarrow PP.\text{PreProcess}(p(X)),\ (i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x) : \\ PP.\text{Reconstruct}(R, x, DB[i_1], ..., DB[i_k]) = p(x) \end{pmatrix} = 1.$$

  2. Both $PP.\text{Lookup}$ and $PP.\text{Reconstruct}$ are deterministic algorithms.

Given this building blocks, our polynomial commitment scheme PC_{VC} is the following:

- **Setup**$(1^\lambda, m, d)$:
  1. Generates the ring R.
  2. Let S be the upper bound on the size of the data structure DB produced by the preprocessing algorithm.
  3. Let $PK_{DB} \leftarrow VC.\text{Setup}(1^\lambda, S)$.
  4. Output $PK = (PK_{DB}, R, m, d)$.

- **Commit**$(PK, p(X); r)$: Given $p(X)$ a polynomial of m variables and individual degree $< d$ in each variable.

  1. Let $DB \leftarrow PP.\text{PreProcess}(p(X))$ (with $|DB| = S$).
  2. Let $(\mathcal{C}_{DB}, aux_{DB}) \leftarrow VC.\text{Commit}(PK_{DB}, DB; r)$.
  3. Output $\mathcal{C}_p = \mathcal{C}_{DB}$ and let $aux_p = (DB, aux_{DB})$.

- **OpenEval**$(PK, p(X), x, aux_p)$[1]: Given $x \in R^m$.

  1. Let $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$.
  2. For each $j \in [k]$, let $w_j \leftarrow VC.\text{OpenPos}(PK_{DB}, DB[i_j], i_j, aux_{DB})$.
  3. Output $w_x = (DB[i_1], w_1, ..., DB[i_k], w_k)$.

- **VerifyEval**$(PK, \mathcal{C}_p, x, y, w_x)$: Given $y \in R$.

  1. Parse $w_x = (v_1, w_1, ..., v_k, w_k)$.
  2. Let $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$.
  3. Output
$$\left(\bigwedge_{j=1}^{k} VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j) = 1\right) \wedge y = PP.\text{Reconstruct}(R, x, v_1, ..., v_k).$$

Now, let's check which of the properties defined above this scheme satisfies.

Proposition 5.3.1 (Correctness). *If VC satisfies correctness then the PC_{VC} scheme satisfies correctness.*

Proof. Let $PK \leftarrow \text{Setup}(1^\lambda, m, d)$ and $(\mathcal{C}_p = \mathcal{C}_{DB}, aux_p = (DB, aux_{DB})) \leftarrow \text{Commit}(PK, p(X); r)$. Let $w_x \leftarrow \text{OpenEval}(PK, p(X), x, aux_p)$. By definition of OpenEval, we can rewrite $w_x = (v_1, w_1, ..., v_k, w_k)$ where $\forall j \in [k]$, $v_j = DB[i_j]$ and $w_j = VC.\text{OpenPos}(PK_{DB}, i_j, v_j, aux_{DB})$ with $(i_1, ..., i_k) = PP.\text{Lookup}(R, m, d, x)$. Then:

$$\text{VerifyEval}(PK, \mathcal{C}_p, x, p(x), w_x) = \text{VerifyEval}(PK, \mathcal{C}_p, x, p(x), (v_1, w_1, ..., v_k, w_k)) =$$
$$= \left(\bigwedge_{j=1}^{k} VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j)\right) \wedge (p(x) = PP.\text{Reconstruct}(R, x, v_1, ..., v_k)) \overset{(1)}{=}$$
$$\overset{(1)}{=} (p(x) = PP.\text{Reconstruct}(R, x, v_1, ..., v_k)) = 1$$

where in (1) we use VC's correctness.

□

Proposition 5.3.2 (Evaluation Binding). *If VC satisfies position binding then the PC_{VC} scheme satisfies evaluation binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks PC_{VC} evaluation binding. This means for $PK \leftarrow \text{Setup}(1^\lambda, m, d)$, $\mathcal{A}(PK)$ outputs $(\mathcal{C}_p, x, y, w_x, y', w'_x)$ such that:

$$\text{VerifyEval}(PK, \mathcal{C}_p, x, y, w_x) = 1 \ \wedge \ \text{VerifyEval}(PK, \mathcal{C}_p, x, y', w'_x) = 1 \ \wedge \ y \neq y'$$

Notice for both executions of the VerifyEval algorithm, we compute the same tuple of indices $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$ as we have the same input x. If we parse $w_x = (v_1, w_1, ..., v_k, w_k)$ and $w'_x = (v'_1, w'_1, ..., v'_k, w'_k)$ we have two options:

[1]Notice in our case $p(X)$ is not needed as an input.

1. If $(v_1, ..., v_k) = (v'_1, ..., v'_k)$ then

$$y = PP.\text{Reconstruct}(R, x, v_1, ..., v_k) = PP.\text{Reconstruct}(R, x, v'_1, ..., v'_k) = y'$$

but $y \neq y'$ (contradiction).

2. If $(v_1, ..., v_k) \neq (v'_1, ..., v'_k)$ then $\exists j \in [k]$ such that:

$$VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j) = 1 \wedge$$
$$\wedge\ VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v'_j, w'_j) = 1\ \wedge\ v_j \neq v'_j$$

and therefore we break VC position binding.

□

Now, we'll prove that as expected the algorithms 3.1.2 and 3.1.3 are a valid instantiation of the PP scheme, i.e. they ensure both correctness and determinism in the PP.Lookup and PP.Reconstruct algorithms.

Proposition 5.3.3. *The polynomial preprocessing scheme PPKU = (PPKU.PreProcess, PPKU.Lookup, PP.Reconstruct) defined as follows:*

- *PPKU.PreProcess($p(X)$): Given $p(X) \in \mathbb{Z}_q[X_1, ..., X_m]$ with individual degree $< d$ in each variable, output algorithm 3.1.2 applied to $p(X)$.*
- *PPKU.Lookup($\mathbb{Z}_q, m, d, x$): Given $x \in \mathbb{Z}_q^m$, lift it to $\mathbb{Z}$ and reduce it modulo each of the h primes. For each $i \in [h]$ output the index in DB corresponding to p_i and to $\overline{\alpha_i}$.*
- *PPKU.Reconstruct($\mathbb{Z}_q, x, DB[i_1], ..., DB[i_k]$): Given k positions of the database, reconstruct the solution using the CRT algorithm as in 3.1.3 and output it modulo q.*

is a correct polynomial preprocessing scheme.

Proof. Trivial by correctness and determinism of algorithms 3.1.2 and 3.1.3.

□

However, if we instantiate the PC_{VC} scheme with the polynomial preprocessing scheme for these algorithms, we don't satisfy strong evaluation binding. The intuition behind this is that the space of the data structures is bigger than the space of the polynomials we preprocess and therefore, there should exist data structures that verify evaluations but that don't match any polynomial. More concretely for $\mathbb{Z}_q$, the space of polynomials is $P = \{f \in \mathbb{Z}_q[X_1, ..., X_m] \mid f \text{ has individual degree} < d \text{ in each variable}\}$ and has size $|P| = q^{d^m}$; the space of the data structures should have the same size as the following space $DS = \prod_{i=1}^{h}\{f_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m] \mid f_i \text{ has individual degree} < d \text{ in each variable}\}$ because we can 1-to-1 correspond the T_i tables with polynomials in $\mathbb{Z}_{p_i}[X_1, ..., X_m]$ with individual degree $< d$ in every variable [2] and therefore has size:

$$|DS| = \left|\prod_{i=1}^{h}\{f_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m] \mid f_i \text{ has individual degree} < d \text{ in each variable}\}\right| =$$
$$= \prod_{i=1}^{h} |\{f_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m] \mid f_i \text{ has individual degree} < d \text{ in each variable}\}| =$$
$$= \prod_{i=1}^{h} p_i^{d^m} = \left(\prod_{i=1}^{h} p_i\right)^{d^m} > M^{d^m}$$

[2] The reason why this is a 1-to-1 correspondence is that given a polynomial f_i we can determine it's table by evaluating on every point in $\mathbb{Z}_{p_i}^m$ and given a table T_i we can determine it's polynomial by using Lagrange interpolation [42].

where $M = d^m q^{m(d-1)+1}$. Therefore:

$$|DS| > |P| \geq |\{PP.PreProcess(f) \mid f \in P\}|.$$

Proposition 5.3.4 (Strong Evaluation Binding). *The PC_{VC} scheme **doesn't** satisfy strong evaluation binding.*

Proof. Let $d = 2$, $m = 2$ and $PK = (PK_{DB}, \mathbb{Z}_q) \leftarrow \text{Setup}(1^\lambda, m, d)$. Let $f(X_1, X_2) = -2X_1X_2 + X_1 + X_2 + 1 \in \mathbb{Z}_q[X_1, X_2]$. Let $\mathcal{A}$ be the following PPT:

$\mathcal{A}(PK)$:

```
Let $DB = PPKU.\text{PreProcess}(f)$

For each $i$ in $PPKU.\text{Lookup}(\mathbb{Z}_q, m, d, (0,2))$ :
    Replace $DB[i]$ with 1

Let $\mathcal{C} = (VC.\text{Commit}(PK_{DB}, DB; r), d, m)$
Let $w_{(0,0)} = \text{OpenEval}(PK, f, (0,0), DB)$
Let $w_{(0,1)} = \text{OpenEval}(PK, f, (0,1), DB)$
Let $w_{(1,0)} = \text{OpenEval}(PK, f, (1,0), DB)$
Let $w_{(1,1)} = \text{OpenEval}(PK, f, (1,1), DB)$
Let $w_{(0,2)} = \text{OpenEval}(PK, f, (0,2), DB)$
```

$$\downarrow$$

$$(\mathcal{C}, \{((0,0), f(0,0), w_{(0,0)}), ((0,1), f(0,1), w_{(0,1)}),$$
$$((1,0), f(1,0), w_{(1,0)}), ((1,1), f(1,1), w_{(1,1)}), ((0,2), 1, w_{(0,2)})\})$$

Notice that as $(0,2) \bmod p_i = (0,2)$ for all $i \in [h] \setminus \{1\}$ and $(0,2) \bmod p_1 = (0,0)$ which satisfies that $f_2(0,0) = 1$ (which is the exact value we "replaced" in the data structure) the points $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ should still verify after replacing $DB[i]$ in $i \in PPKU.\text{Lookup}(\mathbb{Z}_q, m, d, (0,2))$. Therefore, it holds that:

$$\text{VerifyEval}(PK, \mathcal{C}, (0,0), 1, w_{(0,0)}) = \text{VerifyEval}(PK, \mathcal{C}, (0,1), 2, w_{(0,1)}) =$$
$$= \text{VerifyEval}(PK, \mathcal{C}, (1,0), 2, w_{(1,0)}) = \text{VerifyEval}(PK, \mathcal{C}, (1,1), 1, w_{(1,1)}) = 1.$$

Also, $PPKU.\text{Reconstruct}((0,2), 1, ..., 1) = 1$ and therefore $\text{VerifyEval}(PK, \mathcal{C}, (0,2), 1, w_{(0,2)}) = 1$. However, the system of equations in the strong evaluation binding definition, which is the following

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_{1,1} \\ p_{1,0} \\ p_{0,1} \\ p_{0,0} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 1 \end{pmatrix},$$

doesn't have any solution in $\mathbb{Z}_q$. Therefore, $\mathbf{Adv}^{\text{sEvBind}}_{\mathcal{A}, PC_{VC}}(\lambda) = 1$.

□

5.3.1 Efficiency of the PC_{VC} construction

As said in the introduction, we want the prover to be able to compute evaluation proofs for $p \in R[X_1, ..., X_m]$ (with individual degree $< d$ in each variable) in time sublinear in $|p| = d^m$, therefore, we have to instantiate the construction above with an appropriate VC. If we denote by $COE(|R|, m, d)$ the complexity of the OpenEval algorithm, by $CLU(|R|, m, d)$ the complexity of the PP.Lookup algorithm and by $COP(|\mathcal{M}|, s)$ the complexity of the VC.OpenPos algorithm for $\mathcal{M}$ the message space of the VC and s is the size of the vector we commit to, we want to satisfy that:

$$COE(|R|, m, d) = CLU(|R|, m, d) + k \cdot COP(|\mathcal{M}|, s) < d^m$$

If we take $R = \mathcal{M} = \mathbb{Z}_q$ and $s = S = poly(m, d, \log q) \cdot O(md \log q)^m$ then the equation simplifies to:

$$\begin{aligned} COE(q, m, d) &= CLU(q, m, d) + k \cdot COP(q, S) = \\ &= poly(m, d, \log q) + h \cdot COP(q, S) = \\ &= poly(m, d, \log q) + O(md \log q) \cdot COP(q, S) < d^m. \end{aligned} \tag{5.1}$$

Some vector commitment constructions that satisfy this constrain are: Merkle Tree VC [24] and Incrementally Aggregatable VC [9].

Merkle Tree VC:
Given a vector $(m_1, ..., m_s) \in \mathcal{M}^s$ one way we could easily commit to it is by hashing every value with a collision resistant hash $h : \mathcal{M} \to \mathcal{Y}$, i.e. committing to $\mathcal{C} = (h(m_1), ..., h(m_s))$. Then we can verify positions just by hashing the position we are asked for and by the collision resistant property [3] we would have a position binding commitment. However, this commitment requires the verifier to store the s values $(h(m_1), ..., h(m_s))$. Fortunately, we can improve this solution by making use of a clever data structure called Merkle trees which can be used to hash the whole vector. The resulting hash function H is called a Merkle tree hash.

The Merkle tree hash uses a collision resistant hash function h that outputs values in a set $\mathcal{Y}$. The input of h is either a single element in $\mathcal{M}$ or a pair of elements in $\mathcal{Y}$. The Merkle tree hash H, derived from h, is defined over $(\mathcal{M}^s, \mathcal{Y})$. For simplicity, let's assume that s is a power of two (if not, we can just add dummy points to the closest power of two). The Merkle tree hash works as in figure 5.1: to hash $(m_1, ..., m_s) \in \mathcal{M}^s$, first apply h to each of the s input elements to get $(y_1, ..., y_s) \in \mathcal{Y}^s$. Then build a hash tree from these elements, as shown in the figure. More precisely, the hash function is defined as follows:

Definition 5.3.1 (Merkle tree hash).

Input: $(m_1, ..., m_s) \in \mathcal{M}^s$ with s a power of two and $h : \mathcal{M} \cup \mathcal{Y}^2 \to \mathcal{Y}$ a hash function.

Output: $y \in \mathcal{Y}$.

Algorithm:

1. For each $i \in [s]$, let $y_i = h(m_i)$.
2. For each $i \in [s-1]$, let $y_{i+s} = h(y_{2i-1}, y_{2i})$.
3. Output $y_{2s-1} \in \mathcal{Y}$.

[3]Intuitively, being collision resistant means that it is computationally hard to find $m_0, m_1 \in \mathcal{M}$ such that $h(m_0) = h(m_1) \land m_0 \neq m_1$.

The nice thing about the Merkle tree hash is that given a value $y = H((m_1, ..., m_s), h)$, it is quite easy to prove that an $m \in \mathcal{M}$ is the element at a particular position in $(m_1, ..., m_s)$. For example, to prove that $m = m_3$ in figure 5.1, one provides the intermediate hashes $w = (y_4, y_9, y_{14})$ (shaded in the figure). The verifier can then compute

$$\hat{y}_3 \leftarrow h(m), \quad \hat{y}_{10} \leftarrow h(\hat{y}_3, y_4), \quad \hat{y}_{13} \leftarrow h(y_9, \hat{y}_{10}), \quad \hat{y}_{15} \leftarrow h(\hat{y}_{13}, y_{14})$$

and accept that $m = m_3$ if $y = \hat{y}_{15}$.

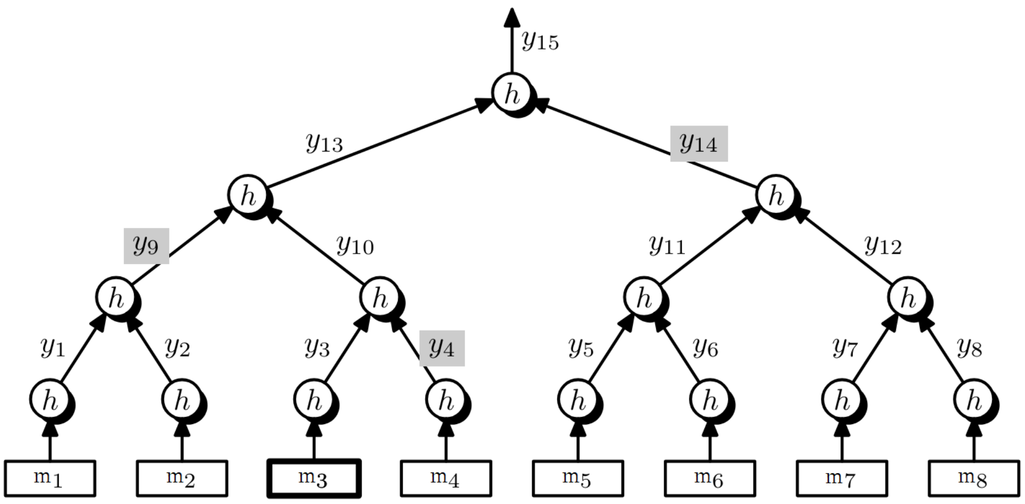


Figure 5.1: A Merkle tree with eight leaves. The values y_4, y_9, y_{14} prove authenticity of m_3 [6, Section 8.9].

The Merkle Tree VC scheme is defined by the following four algorithms:

- **MT.Setup**$(1^\lambda, s)$:
  1. Check that $s = 2^k$ for a $k \in \mathbb{N}$. If not let $s = \min\{2^k \mid s \leq 2^k \ \wedge \ k \in \mathbb{N}\}$.
  2. Let $\mathcal{M}$ be the message space and let $h : \mathcal{M} \cup \mathcal{Y}^2 \rightarrow \mathcal{Y}$ be a hash function.
  3. Output $PK = (\mathcal{M}, h, k)$.
- **MT.Commit**(PK, m):
  1. Let $\mathcal{C} \in \mathcal{Y}$ be the result of Merkle tree hashing m and h, i.e. $\mathcal{C} = H(m, h)$, and store the Merkle tree in aux.
  2. Output $(\mathcal{C}, aux)$.
- **MT.OpenPos**(PK, m_i, i, aux):
  1. For each $j \in [k]$:
     - If $j = 1$ then: let $\hat{y}_{p(i)} = y_{p(i)}$ and let $i' = p(i)$[4].
     - If $j > 1$ then: let $\hat{y}_{p((i'+1)//2+s)} = y_{p((i'+1)//2+s)}$ and let $i' = p((i'+1)//2 + s)$.
     - Add $\hat{y}_{i'}$ to w_i.
  2. Output w_i.

[4]Where $p(i) = \begin{cases} i-1, & \text{if } i \text{ is even} \\ i+1, & \text{if } i \text{ is odd} \end{cases}$.

- **MT.VerifyPos**$(PK, \mathcal{C}, m_i, i, w_i)$:

```
1. For each $j \in [k+1]$:
    – If $j = 1$ then: let $\hat{y}_i = h(m_i)$ and let $i' = i$.
    – If $j > 1$ then:
        ∗ If $i'$ is even then: let $\hat{y}_{(i'+1)//2+s} = h(\hat{y}_{i'-1}, \hat{y}_{i'})$.
        ∗ If $i'$ is odd then: let $\hat{y}_{(i'+1)//2+s} = h(\hat{y}_{i'}, \hat{y}_{i'+1})$.
        ∗ Let $i' = (i'+1)//2 + s$.
2. Output $\mathcal{C} = \hat{y}_{i'}$.
```

Then the opening algorithm has a complexity of: $O(k) = O(\log s)$. In particular, the time cost for opening $S = \mathit{poly}(m, d, \log q) \cdot O(md \log q)^m$ (the size of the data structure in Theorem 3.1.3) would be $O(\log S) = \log(m, d, \log q) \cdot m(\log m + \log d + \log\log q)$. Therefore, for a sufficiently large m we would satisfy equation 5.1:

$$\begin{aligned} COE(q, m, d) &= \mathit{poly}(m, d, \log q) + O(md \log q) \cdot COP(q, S) = \\ &= \mathit{poly}(m, d, \log q) + O(md \log q) \cdot \log(m, d, \log q) \cdot m(\log m + \log d + \log\log q) < d^m. \end{aligned}$$

Now let's prove this scheme also satisfies correctness and position binding, which implies correctness and evaluation binding of the polynomial commitment.

Proposition 5.3.5 (Correctness). *The Merkle Tree VC scheme satisfies correctness.*

Proof. Let $PK \leftarrow \text{MT.Setup}(1^\lambda, s)$ and $(\mathcal{C}, aux) \leftarrow \text{MT.Commit}(PK, m)$. Let $w_i \leftarrow \text{MT.OpenPos}(PK, m_i, i, aux)$. If we denote by i_j^{OP} and by i_j^{VP} the value of i' at the end of the j-th loop of MT.OpenPos and MT.VerifyPos respectively then it holds that: [5]

$$i_j^{OP} = p(i_j^{VP}), \quad \forall j \in [k] \tag{5.2}$$

First, let's prove that we can compute $\hat{y}_{i_j^{VP}}$ $\forall j \in [k+1]$ in the MT.VerifyPos algorithm:

- For $j = 1$, it's clear we can compute $\hat{y}_{i_1^{VP}} = h(m_i)$.
- For $j > 1$, we have that in the j-th loop:

$$\hat{y}_{i_j^{VP}} = \hat{y}_{(i'+1)//2+s} = \begin{cases} h(\hat{y}_{i'-1}, \hat{y}_{i'}), & \text{if } i' \text{ is even} \\ h(\hat{y}_{i'}, \hat{y}_{i'+1}), & \text{if } i' \text{ is odd} \end{cases} =$$

$$= \begin{cases} h(\hat{y}_{p(i')}, \hat{y}_{i'}), & \text{if } i' \text{ is even} \\ h(\hat{y}_{i'}, \hat{y}_{p(i')}), & \text{if } i' \text{ is odd} \end{cases} = \begin{cases} h(\hat{y}_{p(i_{j-1}^{VP})}, \hat{y}_{i_{j-1}^{VP}}), & \text{if } i' \text{ is even} \\ h(\hat{y}_{i_{j-1}^{VP}}, \hat{y}_{p(i_{j-1}^{VP})}), & \text{if } i' \text{ is odd} \end{cases} \overset{5.2}{=}$$

$$\overset{5.2}{=} \begin{cases} h(\hat{y}_{i_{j-1}^{OP}}, \hat{y}_{i_{j-1}^{VP}}), & \text{if } i' \text{ is even} \\ h(\hat{y}_{i_{j-1}^{VP}}, \hat{y}_{i_{j-1}^{OP}}), & \text{if } i' \text{ is odd} \end{cases}$$

  which means that to compute $\hat{y}_{i_j^{VP}}$ we just need $\hat{y}_{i_{j-1}^{VP}}$ which was computed in the previous loop and $\hat{y}_{i_{j-1}^{OP}}$ which is contained in w_i by definition.

Now, let's prove that the $\hat{y}_{i_j^{VP}}$ are nodes in the Merkle Tree, i.e. $\hat{y}_{i_j^{VP}} = y_{i_j^{VP}}$ $\forall j \in [k+1]$:

- For $j = 1$, it's clear $\hat{y}_{i_1^{VP}} = \hat{y}_i = h(m_i) = y_i$.

[5]This can easily be proved by induction over j.

- For $j > 1$, we have that in the j-th loop, if we write $i' = 2k$ or $i' = 2k - 1$ for a $k \in \mathbb{N}$ then:

$$\hat{y}_{i_j^{VP}} = \hat{y}_{(i'+1)//2+s} = \hat{y}_{k+s} = \begin{cases} h(\hat{y}_{i'-1}, \hat{y}_{i'}), & \text{if } i' = 2k \\ h(\hat{y}_{i'}, \hat{y}_{i'+1}), & \text{if } i' = 2k-1 \end{cases} = h(\hat{y}_{2k-1}, \hat{y}_{2k})$$

  which matches the equation in 5.3.1.

Finally, we just need to prove that the final i' in MT.VerifyPos is the right one, i.e. $i_{k+1}^{VP} = 2s - 1$. It can easily be proved that $\forall j \in [k+1]$:

$$i_j^{VP} = l_j + \sum_{h=0}^{j-2} 2^{k-h}$$

where $1 \leq l_j \leq 2^{k-(j-1)}$. Which implies that for $j = k+1$ we have:

$$i_{k+1}^{VP} = 1 + \sum_{h=0}^{k-1} 2^{k-h} = \sum_{h=0}^{k} 2^h = 2^{k+1} - 1 = 2s - 1$$

Therefore, the algorithm correctly computes y_{2s-1} and MT.VerifyPos$(PK, \mathcal{C}, m_i, i, w_i) = 1$.

□

Proposition 5.3.6 (Position Binding). *If h is a collision resistant hash function then the Merkle Tree VC scheme satisfies position binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks Merkle Tree VC position binding. This means for $PK \leftarrow$ MT.Setup$(1^\lambda, s)$, $\mathcal{A}$ outputs $(\mathcal{C}, i, m_i, w_i, m_i', w_i')$ such that:

$$\text{MT.VerifyPos}(PK, \mathcal{C}, m_i, i, w_i) = 1 \ \wedge \ \text{MT.VerifyPos}(PK, \mathcal{C}, m_i', i, w_i') = 1 \ \wedge \ m_i \neq m_i'$$

If we again denote by i_j^{VP} the value of i' at the end of the j-th loop of MT.VerifyPos, as both m_i and m_i' verify we would have that:

$$\hat{y}_{i_{k+1}^{VP}} = h(y_0, y_1) = \mathcal{C} = h(y_0', y_1') = \hat{y}'_{i_{k+1}^{VP}}$$

where $\hat{y}_{i_{k+1}^{VP}}$ corresponds to the execution with m_i and $\hat{y}'_{i_{k+1}^{VP}}$ corresponds to the execution with m_i'. Then, either:

- $(y_0, y_1) \neq (y_0', y_1')$: in this case we would have $h(y_0, y_1) = h(y_0', y_1')$ but $(y_0, y_1) \neq (y_0', y_1')$ which implies h wouldn't be a collision resistant hash function.

- $(y_0, y_1) = (y_0', y_1')$: we can repeat the same argument for $y_0 = y_0'$ and $y_1 = y_1'$ until we reach $j = 1$. In that case we would have $h(m_i) = h(m_i')$ but $m_i \neq m_i'$ which implies h wouldn't be a collision resistant hash function.

Therefore, in any case we would break collision resistance of h.

□

Incrementally Aggregatable VC:

In [5] and [21] the authors proposed new constructions of vector commitments that enjoy a property called subvector openings. A VC with subvector openings (called SVC, for short) allows one to open a commitment at a collection of positions $I = \{i_1, ..., i_m\}$ with a constant-size proof (independent of the vector's length n and the subvector's length m). The formal definition of SVC can be found in [21, Definition 7], but it's just the one presented in section 5.1 but instead of opening for a single position i and a single vector value v_i we open to a set of positions $I = \{i_1, ..., i_m\}$ and a subvector of values $v_I = (v_{i_1}, ..., v_{i_m})$. Formally, subvectors were defined in [21] as follows:

Definition 5.3.2. Let $\mathcal{M}$ be a set, $n \in \mathbb{N}$ be a positive integer and $I = \{i_1, ..., i_{|I|}\} \subseteq [n]$ be an ordered index set. For a vector $v \in \mathcal{M}^n$, the I-subvector of v is $v_I = (v_{i_1}, ..., v_{i_{|I|}})$.

Also, let $I, J \subseteq [n]$ be two sets, and let v_I, v_J be two subvectors of some vector $v \in \mathcal{M}^n$. The ordered union of v_I and v_J is the subvector $v_{I \cup J} = (v_{k_1}, ..., v_{k_m})$ is the ordered sets union of I and J.

In [9], a new property called incremental aggregation was proposed for SVC. In a nutshell, aggregation means that different subvector openings (say, for sets of positions I and J) can be merged together into a single concise (i.e. constant-size) opening (for positions $I \cup J$). This operation must be doable without knowing the entire committed vector. Moreover, aggregation is incremental if aggregated proofs can be further aggregated (i.e. two openings for $I \cup J$ and K can be merged into one for $I \cup J \cup K$, and so on an unbounded number of times) and disaggregated (i.e. given an opening for set I one can create one for any $K \subset I$). More formally, a vector commitment scheme VC with subvector openings is called aggregatable if there exists algorithms **VC.Agg** and **VC.Disagg** such that:

- **VC.Agg**$(PK, (I, v_I, w_I), (J, v_J, w_J))$: Given as input two triples (I, v_I, w_I) and (J, v_J, w_J) where I and J are sets of indices, $v_I \in \mathcal{M}^{|I|}$ and $v_J \in \mathcal{M}^{|J|}$ are subvectors, and w_I and w_J are opening proofs, outputs a proof w_K that is supposed to prove opening of values in positions $K = I \cup J$.
- **VC.Disagg**(PK, I, v_I, w_I, K): Given as input a triple (I, v_I, w_I) and a set of indices $K \subset I$, outputs a proof w_K that is supposed to prove opening of values in positions K.

These algorithms must also guarantee the following properties:

- **Aggregation Correctness**: Aggregation is correct if for all $\lambda \in \mathbb{N}$, all honestly generated $PK \leftarrow \text{VC.Setup}(1^\lambda, s)$, any commitment $\mathcal{C}$ and triple (I, v_I, w_i) such that $\text{VC.VerifyPos}(PK, \mathcal{C}, I, v_I, w_I) = 1$, the following two properties hold:
  1. for any triple (J, v_J, w_J) such that $\text{VC.VerifyPos}(PK, \mathcal{C}, J, v_J, w_J) = 1$,
  $$\Pr\begin{pmatrix} w_K \leftarrow \textbf{VC.Agg}(PK, (I, v_I, w_I), (J, v_J, w_J)) : \\ \text{VC.VerifyPos}(PK, \mathcal{C}, K, v_K, w_K) = 1 \end{pmatrix} = 1$$
  where $K = I \cup J$ and v_K is the ordered union $v_{I \cup J}$ of v_I and v_J.
  2. for any subset of indices $K \subset I$,
  $$\Pr\begin{pmatrix} w_K \leftarrow \textbf{VC.Disagg}(PK, I, v_I, w_I, K) : \\ \text{VC.VerifyPos}(PK, \mathcal{C}, K, v_K, w_K) = 1 \end{pmatrix} = 1$$
  where $v_K = (v_{i_l})_{i_l \in K}$, for $v_I = (v_{i_1}, ..., v_{i_{|I|}})$.
- **Aggregation Conciseness**: There exists a fixed polynomial $p(\cdot)$ in the security parameter such that all openings produced by **VC.Agg** and **VC.Disagg** have length bounded by $p(\lambda)$.

Assume now we want to aggregate several openings for sets of positions $I_1, ..., I_m$ into a single opening $\bigcup_{j=1}^{m} I_j$. The syntax in this definition allows pairwise aggregation which can be sequentially applied to handle several aggregations. However, this would be costly since it would require executing **VC.Agg** on increasingly growing sets. In [9] they overcome this by using a divide-and-conquer based algorithm **VC.AggManyToOne** (shown in Figure 5.2) which achieves a complexity given as a recurrence relation:

$$T(m) = 2T\left(\frac{m}{2}\right) + f_a(b \cdot m)$$

where b is an upper bound to the size of the sets and $f_a(k)$ is the complexity of **VC.Agg** on two sets of total size k.

Analogously, the definition above allows disaggregation of one subset K of I at once. However, in the paper they also present an algorithm **VC.DisaggOneToMany** which allows to disagregate an opening for a set I into several openings for sets $I_1, ..., I_m$ that form a partition of I in an efficient way using a divide-and-conquer methodology. This algorithm (shown in Figure 5.2) achieves a complexity given by the following recurrence relation:

$$T(m) = 2T\left(\frac{m}{2}\right) + 2f_d(m/2)$$

where $f_d(|I|)$ is the complexity of **VC.Disagg**.

```
$\mathsf{VC.AggManyToOne}(PK, (I_j, v_{I_j}, w_j)_{j\in[m]})$
    if $m = 1$ return $w_1$
    $m' \leftarrow m/2$
    $L \leftarrow \cup_{j=1}^{m'} I_j, \quad R \leftarrow \cup_{j=m'+1}^{m} I_j,$
    $w_L \leftarrow \mathsf{VC.AggManyToOne}(PK, (I_j, v_{I_j}, w_j)_{j=1,\dots,m'})$
    $w_R \leftarrow \mathsf{VC.AggManyToOne}(PK, (I_j, v_{I_j}, w_j)_{j=m'+1,\dots,m})$
    $w_{L\cup R} \leftarrow \mathsf{VC.Agg}(PK, (L, v_L, w_L), (R, v_R, w_R))$
    return $w_{L\cup R}$
```

```
$\mathsf{VC.DisaggOneToMany}(PK, B, I, v_I, w_I)$
    if $n = |I| = B$ return $w_I$
    $n' \leftarrow n/2$
    $L \leftarrow \cup_{j=1}^{n'} i_j, \quad R \leftarrow \cup_{j=n'+1}^{m} i_j,$
    $w'_L \leftarrow \mathsf{VC.Disagg}(PK, I, v_I, w_I, L)$
    $w'_R \leftarrow \mathsf{VC.Disagg}(PK, I, v_I, w_I, R)$
    $w_L \leftarrow \mathsf{VC.DisaggOneToMany}(PK, B, L, v_L, w'_L)$
    $w_R \leftarrow \mathsf{VC.DisaggOneToMany}(PK, B, R, v_R, w'_R)$
    return $w_L || w_R$
```

Figure 5.2: Extensions of Aggregation and Disaggregation in [9].

Given this multi-aggregation/disaggregation algorithms they now define two new algorithms, **VC.PPCom** and **VC.FastOpen** (shown in Figure 5.3), that allow faster opening times via preprocessing some proofs at commitment time. This preprocessing method works with flexible choice of a parameter B that allows for different time-memory tradeoffs. In a nutshell, ranging from 1 to n, a larger B reduces memory but increases opening time while a smaller requires larger storage but gives the fastest opening time.

Let B be an integer that divides n and let $n' = n/B$. The core of the preprocessing is that, during the commitment stage, one can create openings for n' subvectors of v that cover the all vector (i.e. B contiguous positions). Let $w_{P_1}, ..., w_{P_{n'}}$ be such openings; these elements are stored as auxiliary information. Then, in the opening phase, in order to compute the opening for a subvector v_I of m positions, one should:

1. Find the subset of openings w_{P_j} such that, for some S, $I \subseteq \bigcup_{j\in S} P_j$.
2. Possibly disaggregate some of them and then aggregate in order to compute w_I.

```
VC.PPCom(PK, B, v)
1: $(\mathcal{C}, \mathsf{aux}) \leftarrow \text{VC.Commit}(PK, v)$
2: $w^* \leftarrow \text{VC.OpenPos}(PK, v, [n], \mathsf{aux})$
3: $w \leftarrow \mathsf{VC.DisaggOneToMany}(PK, B, [n], v, w^*)$
4: $\mathsf{aux}^* := (w_1, \ldots, w_{n'}, v)$
5: return $\mathcal{C}, \mathsf{aux}^*$
```

```
VC.FastOpen(PK, B, aux*, I)
1: Let $P_j := \{(j-1)B + i : i \in [B]\}, \forall j \in [n']$
2: Let $I := \{i_1, \ldots, i_m\}$
3: Let $S$ minimal set s.t. $\bigcup_{j \in S} P_j \supseteq I$
4: for $j \in S$ do :
5:     $I_j \leftarrow I \cap P_j$
6:     $w'_j \leftarrow \mathsf{VC.Disagg}(PK, P_j, v_{P_j}, w_j, I_j)$
7: endfor
8: $w_I \leftarrow \mathsf{VC.AggManyToOne}(PK, ((I_j, v_{I_j}, w'_j))_{j \in S})$
9: return $w_I$
```

Figure 5.3: Generic algorithms for committing and opening with precomputation in [9].

In the paper, they achieve two constructions that satisfy this incrementally aggregation property, further details of the constructions and their correctness/position binding proofs can be found in [9, Section 5]. We are going to use the second one as our building block as it has a better complexity when opening, it's complexity opening is

$$O(l \cdot m' \cdot (\log m')^2)$$

where l is the length of the representation in bits of the elements of the vector and m' is the number of positions we are opening (using $B = 1$, which is the fastest but the one that requires the most memory). Notice, we have to adapt the equation 5.1 a bit for this case, as we are opening the h openings all at once, the resulting equation here would be

$$\begin{aligned}
&COE(q, m, d) = poly(m, d, \log q) + COP(q, S, m') = \\
&= poly(m, d, \log q) + O(O(\log q) \cdot O(md \log q) \cdot (\log O(md \log q))^2) = \\
&= poly(m, d, \log q) + O(\log q \cdot (md \log q) \cdot (\log m + \log d + \log\log q)^2) < d^m
\end{aligned}$$

(for a sufficiently large m) where we use that $l = O(\log q)$ and $m' = h = O(md \log q)$.

This VC perfectly fits our polynomial commitment construction idea of preprocessing the polynomial at commitment time and then having fast proofs, we can slightly change the commitment algorithm to compute both the preprocessed data structure DB and the preprocessed proofs $w_{P_1}, ..., w_{P_{n'}}$.

5.3.2 Towards Strong Evaluation Binding

In this section, we present some strengthenings of the PC_{VC} construction that satisfy the strong evaluation binding property. First, we'll present the notion of knowledge soundness (also known as extractability) which intuitively says that given a commitment that verifies for some values we can always find a polynomial fitting it using an extractor. Knowledge soundness, as we'll prove later, is even stronger than strong evaluation binding, so achieving this notion will suffice to achieve strong evaluation binding. Then we'll present two constructions, $PC_{VC-SNARK}$ and $PC_{PC-SNARK}$, that achieve this knowledge soundness property.

Definition 5.3.3 (Knowledge Soundness). We say a polynomial commitment scheme PC = (**Setup**, **Commit**, **OpenEval**, **VerifyEval**) satisfies knowledge soundness if for any

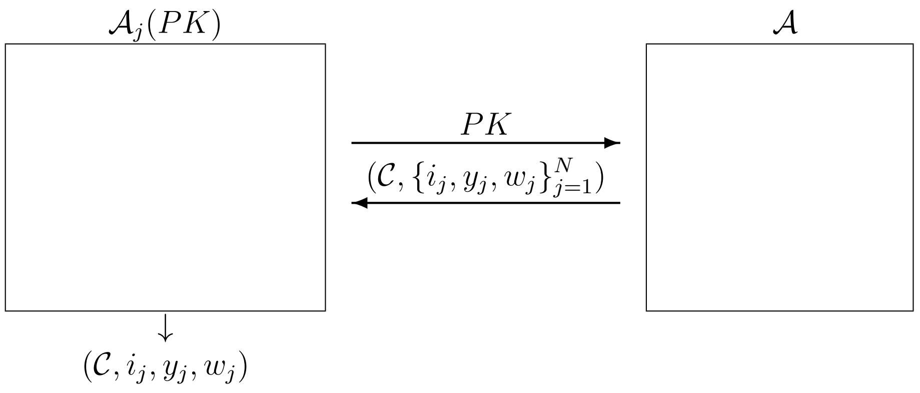
PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}_{\mathcal{A}}$ such that:

$$\Pr\begin{pmatrix} PK \leftarrow \mathbf{Setup}(1^\lambda, m, d),\ (\mathcal{C}, x, y, w_x) \leftarrow \mathcal{A}(PK),\ (p(X); r) \leftarrow \mathcal{E}_{\mathcal{A}}(PK) : \\ \mathbf{VerifyEval}(PK, \mathcal{C}, x, y, w_x) = 1 \wedge \\ (\mathcal{C} \neq \mathbf{Commit}(PK, p(X); r)\ \vee\ p(x) \neq y) \end{pmatrix} = negl(\lambda)$$

Now as claimed let's prove this is stronger than strong evaluation binding:

Proposition 5.3.7 (Knowledge Soundness $\implies$ Strong Evaluation Binding). *If a PC satisfies knowledge soundness then it also satisfies strong evaluation binding.*

Proof. Let's proceed by contradiction. Assume a PPT $\mathcal{A}$ breaks strong evaluation binding. Let $\mathcal{A}_j$ be the following PPT:


$\mathcal{A}_j(PK)$
$\mathcal{A}$
PK
$(\mathcal{C}, \{i_j, y_j, w_j\}_{j=1}^N)$
$(\mathcal{C}, i_j, y_j, w_j)$

for each $j \in [N]$. Notice, for each $j \in [N]$, $(\mathcal{C}, i_j, y_j, w_j) \leftarrow \mathcal{A}_j(PK)$ such that

$$\text{VerifyEval}(PK, \mathcal{C}, i_j, y_j, w_j) = 1.$$

Assume now that PC was knowledge sound. For each $j \in [N]$ there exists a PPT extractor $\mathcal{E}_{\mathcal{A}_j}$ such that $(p_j(X); r_j) \leftarrow \mathcal{E}_{\mathcal{A}_j}(PK)$ with:

$$\mathcal{C} = \text{Commit}(PK, p_j(X); r_j)\ \wedge\ p_j(i_j) = y_j.$$

Then if $p_i(X) \neq p_j(X)$ for some $i, j \in [N]$, we would have $\mathcal{C} = \text{Commit}(PK, p_j(X); r_j) = \text{Commit}(PK, p_i(X); r_i)$ breaking commitment binding. Otherwise, we would have $p_i(X) = p_j(X)$ for every $i, j \in [N]$, breaking evaluation binding.

□

Before showing the two constructions satisfying this knowledge soundness property, first, we need to introduce a building block that both constructions use, SNARKs (Succint Non-interactive Argument of Knowledge). By this term, we denote a proof system which is:

- **Succint**: the size of the proof is very small compared to the size of the statement or the witness, i.e. the size of the computation itself.
- **Non-interactive**: it does not require rounds of interaction between the prover and the verifier.
- **Argument**: we consider it secure only for provers that have bounded computational resources, which means that provers with enough computational power can convince the verifier of a wrong statement.

- **Knowledge-sound**: it is not possible for the prover to construct a proof without knowing a certain so-called witness for the statement; formally, for any prover able to produce a valid proof, there is an extractor capable of extracting a witness ("the knowledge") for the statement.

A SNARK protocol is described by three algorithms, **Gen**, **Prove** and **Verify**, that work as follows:

- **Gen**$(1^\lambda, \mathcal{R})$: the CRS generation algorithm takes as input some security parameter λ, a relation $\mathcal{R}$ and outputs a common string reference string crs.
- **Prove**(crs, u, w): the prover algorithm takes as input the crs, a statement u and a witness w. It outputs some argument π.
- **Verify**(crs, u, π): the verifier algorithm takes as input a statement u, together with an argument π and crs. It outputs $b = 1$ (accept) if the proof was accepted and $b = 0$ (reject) otherwise.

Now, we introduce some properties for SNARKs protocols:

- **Completeness**: Given a true statement for the relation $\mathcal{R}$, a honest prover P with a valid witness should convince the verifier V. Formally, for all $\lambda \in \mathbb{N}$ and for all $(u, v) \in \mathcal{R}$:
$$\Pr\begin{pmatrix} crs \leftarrow \textbf{Gen}(1^\lambda, \mathcal{R}),\ \pi \leftarrow \textbf{Prove}(crs, u, w) : \\ \textbf{Verify}(crs, u, \pi) = 1 \end{pmatrix} = 1$$
- **Knowledge Soundness**: The notion of knowledge soundness (which is essentially the same as the one we've just presented for polynomial commitments) implies that there is an extractor that can compute the witness whenever the adversary produces a valid argument. The extractor gets full access to the adversary's state, including any random coins. Formally, we require that for all PPT adversaries $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}_{\mathcal{A}}$ such that:
$$\Pr\begin{pmatrix} crs \leftarrow \textbf{Gen}(1^\lambda, \mathcal{R}),\ (u, \pi) \leftarrow \mathcal{A}(crs),\ w \leftarrow \mathcal{E}_{\mathcal{A}}(crs) : \\ \textbf{Verify}(crs, u, \pi) = 1\ \wedge\ (u, w) \notin \mathcal{R} \end{pmatrix} = negl(\lambda)$$
- **Succintness**: A non-interactive argument where the verifier runs in $poly(\lambda + |u|)$ time and the proof size is $poly(\lambda)$ is called a succint SNARK. If we also restrict the common reference string to be $poly(\lambda)$ we say the non-interactive argument is a fully succint SNARK.

Let's now show the constructions, $PC_{VC-SNARK}$ and $PC_{PC-SNARK}$.

First construction: $PC_{VC-SNARK}$
This one is just a slight modification of the original PC_{VC} construction where we add a SNARK proof in the commitment $\mathcal{C}_p$ showing that

$$\mathcal{C}_{DB} = VC.\text{Commit}(PK_{DB}, PP.\text{PreProcess}(p(X)); r)$$

for a (possible) randomness r. This would clearly fix the strong evaluation attack shown in proposition 5.3.4, because even if the space of data structures is bigger than the space of polynomials, we are proving that the commitment corresponds to a polynomial. Notice this would still be efficient for the prover as the OpenEval algorithm complexity isn't changed.

We need a SNARK for the following relation:

$\mathcal{R}^{VC,PP}_{R,m,d} = \{(\mathcal{C}_{DB}, (p(X); r)) \in CDS \times (P \times R) \mid \mathcal{C}_{DB} = VC.Commit(PK_{DB}, PP.\text{PreProcess}(p(X)); r)$ for a (possible) randomness r where $PK_{DB} = VC.\text{Setup}(1^\lambda, S)$ for a $\lambda \in \mathbb{N}\}$

where P is the space of polynomials, R is the space of randomnesses and CDS the space of the commitments to data structures in DS (the space of the data structures). Using this building block the new construction, $PC_{VC-SNARK}$, would be the following:

- **Setup**$(1^\lambda, m, d)$:
  1. Generates the ring R.
  2. Let S be the upper bound on the size of the data structure DB produced by the preprocessing algorithm.
  3. Let $PK_{DB} \leftarrow VC.\text{Setup}(1^\lambda, S)$.
  4. Let $crs \leftarrow SNARK.\text{Gen}(1^\lambda, \mathcal{R}^{VC,PP}_{R,m,d})$.
  5. Output $PK = (crs, PK_{DB}, R, m, d)$.

- **Commit**$(PK, p(X); r)$: Given $p(X)$ a polynomial of m variables and individual degree $< d$ in each variable.
  1. Let $DB \leftarrow PP.\text{PreProcess}(p(X))$ (with $|DB| = S$).
  2. Let $(\mathcal{C}_{DB}, aux_{DB}) \leftarrow VC.\text{Commit}(PK_{DB}, DB; r)$.
  3. Let $\pi \leftarrow SNARK.\text{Prove}(crs, \mathcal{C}_{DB}, (p(X); r))$.
  4. Output $\mathcal{C}_p = (\mathcal{C}_{DB}, \pi)$ and let $aux_p = (DB, aux_{DB})$.

- **OpenEval**$(PK, p(X), x, aux_p)$: Given $x \in R^m$.
  1. Let $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$.
  2. For each $j \in [k]$, let $w_j \leftarrow VC.\text{OpenPos}(PK_{DB}, DB[i_j], i_j, aux_{DB})$.
  3. Output $w_x = (DB[i_1], w_1, ..., DB[i_k], w_k)$.

- **VerifyEval**$(PK, \mathcal{C}_p, x, y, w_x)$: Given $y \in R$.
  1. Parse $w_x = (v_1, w_1, ..., v_k, w_k)$.
  2. Let $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$.
  3. Output

$$\left(\bigwedge_{j=1}^{k} VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j) = 1\right) \wedge$$
$$\wedge \ (y = PP.\text{Reconstruct}(R, x, v_1, ..., v_k)) \ \wedge \ (SNARK.\text{Verify}(crs, \mathcal{C}_{DB}, \pi) = 1).$$

Proposition 5.3.8 (Correctness). *If $SNARK$ satisfies completeness and VC satisfies correctness then the $PC_{VC-SNARK}$ scheme satisfies correctness.*

Proof. Let $PK \leftarrow \text{Setup}(1^\lambda, m, d)$ and $(\mathcal{C}_p = (\mathcal{C}_{DB}, \pi), aux_p = (DB, aux_{DB})) \leftarrow \text{Commit}(PK, p(X); r)$. Let $w_x \leftarrow \text{OpenEval}(PK, p(X), x, aux_p)$. By definition of OpenEval, we can rewrite $w_x = (v_1, w_1, ..., v_k, w_k)$ where $\forall j \in [k],\ v_j = DB[i_j]$ and $w_j = VC.\text{OpenPos}(PK_{DB}, i_j, v_j, aux_{DB})$ with $(i_1, ..., i_k) = PP.\text{Lookup}(R, m, d, x)$. Then:

$$\text{VerifyEval}(PK, \mathcal{C}_p, x, p(x), w_x) = \text{VerifyEval}(PK, \mathcal{C}_p, x, p(x), (v_1, w_1, ..., v_k, w_k)) =$$

$$= \left(\bigwedge_{j=1}^{k} VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j) \right) \wedge\ (p(x) = PP.\text{Reconstruct}(R, x, v_1, ..., v_k)) \wedge$$

$$\wedge\ (SNARK.\text{Verify}(crs, \mathcal{C}_{DB}, \pi) = 1) \overset{(1)}{=} 1$$

where in (1) we use VC's correctness, $SNARK$'s completeness and PP's correctness.

□

Proposition 5.3.9 (Knowledge Soundness). *If $SNARK$ satisfies completeness and knowledge soundness and VC satisfies correctness and (weak) position binding, then the $PC_{VC-SNARK}$ scheme satisfies knowledge soundness (and thus strong evaluation binding).*

Proof. Let $PK \leftarrow \text{Setup}(1^\lambda, m, d)$. Let $\mathcal{A}$ be a PPT such that $(\mathcal{C}_p = (\mathcal{C}_{DB}, \pi), x, y, w_x = (v_1, w_1, ..., v_k, w_k)) \leftarrow \mathcal{A}(PK)$ with $\text{VerifyEval}(PK, \mathcal{C}_p, x, y, w_x) = 1$. Let $\mathcal{B}$ be a PPT such that $(\mathcal{C}_{DB}, \pi) \leftarrow \mathcal{B}(crs)$. By completeness of the $SNARK$, $SNARK.\text{Verify}(crs, \mathcal{C}_{DB}, \pi) = 1$. By knowledge soundness of the $SNARK$, there exists a PPT $\mathcal{E}_{\mathcal{B}}$ such that the extracted witness $(p(X); r) \leftarrow \mathcal{E}_{\mathcal{B}}(crs)$ satisfies that

$$(\mathcal{C}_{DB}, aux_{DB}) = VC.\text{Commit}(PK_{DB}, PP.\text{PreProcess}(p(X)); r)$$

for some aux_{DB}.
Then, by validity of the verification it holds that for $(i_1, ..., i_k) \leftarrow PP.\text{Lookup}(R, m, d, x)$:

$$VC.\text{VerifyPos}(PK_{DB}, \mathcal{C}_{DB}, i_j, v_j, w_j) = 1 \text{ for all } j \in [k] \wedge$$
$$\wedge\ y = PP.\text{Reconstruct}(x, v_1, ..., v_k).$$

Therefore, if any of the values $v_j \neq PP.\text{PreProcess}(p(X))[i_j]$ we can break (weak) position binding of VC. Otherwise, if all the values $v_j = PP.\text{PreProcess}(p(X))[i_j]$ the correctness of PP implies that $y = p(x)$, i.e. it can't be $y \neq p(x)$.

□

Second construction: $PC_{PC-SNARK}$

This construction is a bit different than the previous ones, here instead of committing to the data structure resulting from preprocessing a polynomial $f \in \mathbb{Z}_q[X_1, ..., X_m]$ using a vector commitment, we use a polynomial commitment PC to commit to each of the $f_i = f \mod p_i$ in the algorithm 3.1.2. Notice this works for the Kedlaya and Umans algorithm for $R = \mathbb{Z}_q$ and is not a general construction using a polynomial preprocessing scheme as a building block. Another building block for this construction is a $SNARK$, the reason why we need it is because we need to ensure that the commitments to each of the $f_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m]$ are indeed the reductions from a polynomial $f \in \mathbb{Z}_q[X_1, ..., X_m]$ modulo their respective p_i.

Let $PC = (PC.\text{Setup}, PC.\text{Commit}, PC.\text{OpenEval}, PC.\text{VerifyEval})$ where $PC.\text{Setup}(1^\lambda, p, m, d)$ creates a commitment key to commit and evaluate polynomials in $\mathbb{Z}_p[X_1, ..., X_m]$ with individual degree $< d$ in each variable. Also, the concrete relation we want to proof with the

$SNARK$ is the following:

$\mathcal{R}^{PC}_{R,m,d} = \{((\mathcal{C}_1, ..., \mathcal{C}_h), (f_1, r_1, ..., f_h, r_h, f)) \in (CP^h) \times (P_{p_1} \times R_1 \times ... \times P_{p_h} \times R_h \times P_q) \mid$
$f_i = f \mod p_i, \text{ for each } i \in [h], \wedge \mathcal{C}_i = PC.\text{Commit}(PK_i, f_i; r_i)$
for a (possible) randomness r_i where $PK_i = PC.\text{Setup}(1^\lambda, p_i, m, d)$ for a $\lambda \in \mathbb{N}$, for each $i \in [h]\}$

where $P_p = \{f \in \mathbb{Z}_p[X_1, ..., X_m] \mid f \text{ has individual degree } < d\}$ for each p, R_i are the spaces of randomnesses and CP is the space of the commitments to polynomials in any P_p. Using this building blocks the new construction, $PC_{PC-SNARK}$, would be the following:

- **Setup**$(1^\lambda, q, m, d)$:
  1. Take h primes $p_1, ..., p_h$, following the guidelines of the preprocessing algorithm.[6]
  2. Let S be the upper bound on the size of the data structure DB produced by the preprocessing algorithm.
  3. For each $i \in [h]$, let $PK_i \leftarrow PC.\text{Setup}(1^\lambda, p_i, m, d)$.
  4. Let $crs \leftarrow SNARK.\text{Gen}(1^\lambda, \mathcal{R}^{PC}_{R,m,d})$.
  5. Output $PK = (crs, PK_1, ..., PK_h, q, m, d)$.
- **Commit**$(PK, f(X); r_1; ...; r_h)$: Given $f(X)$ a polynomial in $\mathbb{Z}_q$ of m variables and individual degree $< d$ in each variable.
  1. For each $i \in [h]$:
     - Let $f_i = f \mod p_i \in \mathbb{Z}_{p_i}[X_1, ..., X_m]$.
     - Let $(\mathcal{C}_i, aux_i) \leftarrow PC.\text{Commit}(PK_i, f_i; r_i)$.
  2. For each $i \in [h]$, for all $a \in \mathbb{Z}^m_{p_i}$:
     - Compute $y_{i,a} = f_i(a)(= f(a) \mod p_i)$.
     - Compute the evaluation proof $w_{i,a} \leftarrow PC.\text{OpenEval}(PK_i, f_i(X), a, aux_i)$.
     - Store $T_i[a] = (y_{i,a}, w_{i,a})$.
  3. Let $\pi \leftarrow SNARK.\text{Prove}(crs, (\mathcal{C}_1, ..., \mathcal{C}_h), (f_1, r_1, ..., f_h, r_h, f))$.
  4. Output $\mathcal{C} = (\mathcal{C}_1, ..., \mathcal{C}_h, \pi)$ and let $aux = (T_1, aux_1, ..., T_h, aux_h)$.
- **OpenEval**$(PK, f(X), x, aux)$: Given $x \in \mathbb{Z}^m_q$.
  1. For each $i \in [h]$, let $x_i = x \mod p_i$.
  2. Output $w_x = (y_{1,x_1}, w_{1,x_1}, ..., y_{h,x_h}, w_{h,x_h})$.
- **VerifyEval**$(PK, \mathcal{C}, x, y, w_x)$: Given $y \in \mathbb{Z}^m_q$.
  1. Parse $w_x = (v_1, w_1, ..., v_h, w_h)$.
  2. Find the smallest integer z such that

$$z = v_i \mod p_i, \text{ for each } i \in [h].$$

     using the CRT algorithm.

[6]Recall it holds that $d^m q^{m(d-1)+1} = M < p_1 \cdot ... \cdot p_h = p^*$.

3. Output

$$\left(\bigwedge_{j=1}^{h} PC.\text{VerifyEval}(PK_i, \mathcal{C}_i, x_i, v_i, w_i) = 1 \right) \wedge$$
$$\wedge \ (y = z \mod q) \ \wedge \ (SNARK.\text{Verify}(crs, (\mathcal{C}_1, ..., \mathcal{C}_h), \pi) = 1).$$

Proposition 5.3.10 (Correctness). *If $SNARK$ satisfies completeness and PC satisfies correctness then the $PC_{PC-SNARK}$ scheme satisfies correctness.*

Proof. Let $PK \leftarrow \text{Setup}(1^\lambda, q, m, d)$ and $(\mathcal{C} = (\mathcal{C}_1, ..., \mathcal{C}_h, \pi), aux = (T_1, aux_1, ..., T_h, aux_h)) \leftarrow \text{Commit}(PK, f(X); r_1; ...; r_h)$. Let $w_x \leftarrow \text{OpenEval}(PK, f(X), x, aux)$. By definition of OpenEval, we can rewrite $w_x = (v_1, w_1, ..., v_h, w_h)$ where $\forall i \in [h]$, $v_i = f_i(x_i)$ and $w_i = PC.\text{OpenEval}(PK_i, f_i(X), x_i, aux_i)$ with $x_i = x \mod p_i$. Then:

$$\text{VerifyEval}(PK, \mathcal{C}, x, f(x), w_x) = \text{VerifyEval}(PK, \mathcal{C}, x, f(x), (v_1, w_1, ..., v_h, w_h)) =$$
$$= \left(\bigwedge_{i=1}^{h} PC.\text{VerifyEval}(PK_i, \mathcal{C}_i, x_i, v_i, w_i) \right) \wedge \ (f(x) = z \mod q) \wedge$$
$$\wedge \ (SNARK.\text{Verify}(crs, (\mathcal{C}_1, ..., \mathcal{C}_h), \pi) = 1) \overset{(1)}{=} 1$$

where in (1) we use PC's correctness, $SNARK$'s completeness and correctness of algorithms 3.1.2 and 3.1.3. □

Proposition 5.3.11 (Knowledge Soundness). *If $SNARK$ satisfies completeness and knowledge soundness and PC satisfies correctness and weak evaluation binding, then the $PC_{PC-SNARK}$ scheme satisfies knowledge soundness (and thus strong evaluation binding).*

Proof. Let $PK \leftarrow \text{Setup}(1^\lambda, q, m, d)$. Let $\mathcal{A}$ be a PPT such that

$$(\mathcal{C} = (\mathcal{C}_1, ..., \mathcal{C}_h, \pi), x, y, w_x = (y_{1,x_1}, w_{1,x_1}, ..., y_{h,x_h}, w_{h,x_h})) \leftarrow \mathcal{A}(PK).$$

Let $\mathcal{B}$ be a PPT such that $((\mathcal{C}_1, ..., \mathcal{C}_h), \pi) \leftarrow \mathcal{B}(crs)$. By completeness of the $SNARK$, $SNARK.\text{Verify}(crs, (\mathcal{C}_1, ..., \mathcal{C}_h), \pi) = 1$. By knowledge soundness of the $SNARK$, there exists a PPT $\mathcal{E}_\mathcal{B}$ such that the extracted witness $(f_1, r_1, ..., f_h, r_h, f) \leftarrow \mathcal{E}_\mathcal{B}(PK)$ satisfies that

$$f_i = f \mod p_i \ \wedge \ (\mathcal{C}_i, aux_i) = PC.\text{Commit}(PK_i, f_i; r_i), \text{ for each } i \in [h]$$

with $f \in \mathbb{Z}_q[X_1, ..., X_m]$ with degree $< d$ in each individual variable.
Therefore, if any of the values $v_i \neq f_i(x_i)$, where $x_i = x \mod p_i$ and $v_i = y_{i,x_i}$, we can break weak evaluation binding of the i-th PC instance. Otherwise, if all the values $v_i = f_i(x_i)$, then from the fact that $f \in \mathbb{Z}_q[X_1, ..., X_m]$ with degree $< d$ in each individual variable we have that:

$$v_i = f_i(x_i) = f(x_i) \mod p_i < M < p^* = p_1 \cdot ... \cdot p_h$$

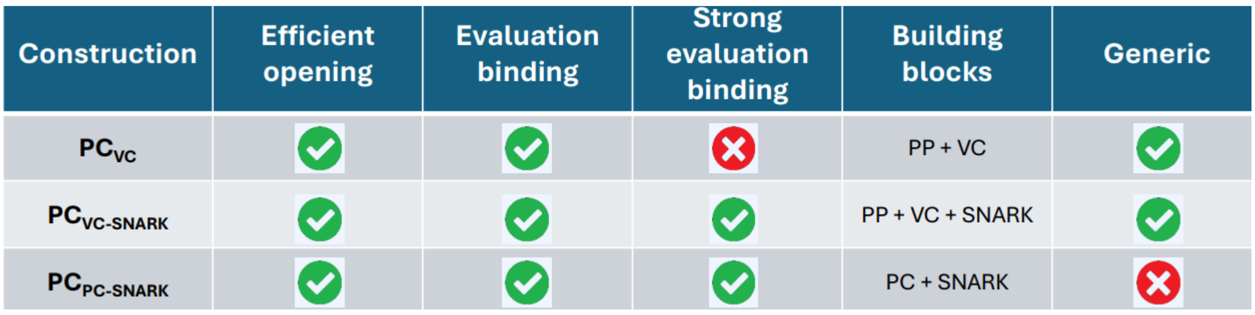
and then by uniqueness of CRT reconstruction we have $y = f(x)$.

□

Chapter 6

Conclusion

The following table summarizes the results we achieved for the prover efficient polynomial commitment constructions:



Construction	Efficient opening	Evaluation binding	Strong evaluation binding	Building blocks	Generic
PC_{VC}	✓	✓	✗	PP + VC	✓
$PC_{VC\text{-}SNARK}$	✓	✓	✓	PP + VC + SNARK	✓
$PC_{PC\text{-}SNARK}$	✓	✓	✓	PC + SNARK	✗

Figure 6.1: Comparison between constructions.

As shown in section 3.3 although the preprocessing algorithm has interesting theoretical aplications, it is still not practical. However while working on this, we also found a paper [27], where the authors propose some optimizations for the DEPIR scheme in [22]. Their paper focuses on improving the algorithm for $R = \mathbb{Z}_q[X,Y]/(E_1(X), E_2(Y))$, which is the ciphertext space of the DEPIR protocol, in terms of both runtime and server storage. For a database of size $N \approx 2^{20}$ the naive implementation of the protocol would require the server to store at least 10^{78} elements (a few bytes each). Their implementation requires less than $5.3 \cdot 10^{11}$ elements to be stored by the server, which is a huge improvement.

Further interesting research topics might be checking if the techniques to add updatability in the DEPIR scheme (or similar) can be used in our polynomial commitment, and also to find applications for our polynomial commitment.

Acknowledgement

First of all, I would like to thank both my supervisors Dario Fiore and Ignacio Cascudo. They gave me the opportunity to work in this project and to discover the world of research from the inside. My personal goal for this internship/TFM was to see if the academic path was for me. Now I know this is what I want to do and I'll always be grateful for that. I'm also thankful for the care they've taken into reading and correcting this thesis details and for the references/explanations they've given when I was stuck.

I'm also thankful to the whole team involved in the polynomial commitment project: Matteo Campanelli, Dario Catalano, Danilo Francati, Emanuele Giunta, Rosario Gennaro and my supervisors. Having the opportunity to be inside such a big project with so talented people was amazing and made me a better researcher.

I would also like to thank my brother who has always been an inspiration for me (especially to follow this path), my mother for her constant support and my friends from Toledo: Marcos, Sego, Nacho, Javi, Sergio, etc.

Finally, I want to mention the people I've met during the master: Sinhue, Pablo, etc. and during my time at IMDEA: Elvira, Marga, Ale Naser, Harlan, Guru, Chana, etc. Sharing these months with you all has been beautiful, you can't imagine how grateful I am to all of you. Hope everything works out and our paths meet again in the future.

Bibliography

[1] ARKWORKS CONTRIBUTORS. arkworks zksnark ecosystem. https://github.com/arkworks-rs, 2022.

[2] ASHHAMI, D., AND DENIS, F. sodiumoxide. https://github.com/sodiumoxide/sodiumoxide.

[3] BLAKE, I. F., AND GAREFALAKIS, T. On the complexity of the discrete logarithm and diffie–hellman problems. *Journal of Complexity 20*, 2 (2004), 148–170. Festschrift for Harald Niederreiter, Special Issue on Coding and Cryptography.

[4] BONEH, D. The decision diffie-hellman problem. In *Algorithmic Number Theory* (Berlin, Heidelberg, 1998), J. P. Buhler, Ed., Springer Berlin Heidelberg, pp. 48–63.

[5] BONEH, D., BÜNZ, B., AND FISCH, B. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Advances in Cryptology – CRYPTO 2019* (Cham, 2019), A. Boldyreva and D. Micciancio, Eds., Springer International Publishing, pp. 561–586.

[6] BONEH, D., AND SHOUP, V. *A Graduate Course in Applied Cryptography.* 2015.

[7] BOYLE, E., ISHAI, Y., PASS, R., AND WOOTTERS, M. Can we access a database both locally and privately? In *TCC (2)* (2017), Springer, pp. 662–693.

[8] CAMPANELLI, M., FAONIO, A., FIORE, D., QUEROL, A., AND RODRÍGUEZ, H. Lunar: a toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. Cryptology ePrint Archive, Paper 2020/1069, 2020. https://eprint.iacr.org/2020/1069.

[9] CAMPANELLI, M., FIORE, D., GRECO, N., KOLONELOS, D., AND NIZZARDO, L. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *Advances in Cryptology – ASIACRYPT 2020* (Cham, 2020), S. Moriai and H. Wang, Eds., Springer International Publishing, pp. 3–35.

[10] CANETTI, R., HOLMGREN, J., AND RICHELSON, S. Towards doubly efficient private information retrieval. In *TCC (2)* (2017), Springer, pp. 694–726.

[11] CARRILLO, M. Tfm code. https://github.com/MariocarrilloGH/TFM, 2024.

[12] CATALANO, D., AND FIORE, D. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013* (2013), vol. 7778, p. 54.

[13] CATALANO, D., FIORE, D., AND TUCKER, I. Additive-homomorphic functional commitments and applications to homomorphic signatures. Cryptology ePrint Archive, Paper 2022/1331, 2022. https://eprint.iacr.org/2022/1331.

[14] Chor, B., Kushilevitz, E., Goldreich, O., and Sudan, M. Private information retrieval. *J. ACM 45*, 6 (nov 1998), 965–981.

[15] Damgård, I. *Commitment Schemes and Zero-Knowledge Protocols.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 63–86.

[16] Elgamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, 4 (1985), 469–472.

[17] Goldwasser, S., and Bellare, M. Lecture notes on cryptography.

[18] Hamlin, A., Holmgren, J., Weiss, M., and Wichs, D. Fully homomorphic encryption for rams. Cryptology ePrint Archive, Paper 2019/632, 2019. `https://eprint.iacr.org/2019/632`.

[19] Kate, A., Zaverucha, G. M., and Goldberg, I. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010* (Berlin, Heidelberg, 2010), M. Abe, Ed., Springer Berlin Heidelberg, pp. 177–194.

[20] Kedlaya, K. S., and Umans, C. Fast polynomial factorization and modular composition. *SIAM Journal on Computing 40*, 6 (2011), 1767–1802. `https://doi.org/10.1137/08073408X`.

[21] Lai, R. W. F., and Malavolta, G. Subvector commitments with application to succinct arguments. In *Advances in Cryptology – CRYPTO 2019* (Cham, 2019), A. Boldyreva and D. Micciancio, Eds., Springer International Publishing, pp. 530–560.

[22] Lin, W.-K., Mook, E., and Wichs, D. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. Cryptology ePrint Archive, Paper 2022/1703, 2022. `https://eprint.iacr.org/2022/1703`.

[23] Lyubashevsky, V., Peikert, C., and Regev, O. On ideal lattices and learning with errors over rings. vol. 60, pp. 1–23.

[24] Merkle, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87* (Berlin, Heidelberg, 1988), C. Pomerance, Ed., Springer Berlin Heidelberg, pp. 369–378.

[25] Nervos. Nervos ckb. `https://github.com/nervosnetwork/ckb`.

[26] Obregon, A. The rise of rust in blockchain development. `https://medium.com/@AlexanderObregon/the-rise-of-rust-in-blockchain-development-eddbad0d9424`.

[27] Okada, H., Player, R., Pohmann, S., and Weinert, C. Towards practical doubly-efficient private information retrieval. In *Financial Cryptography and Data Security 2024 (FC '24)* (Nov. 2023).

[28] Parity Technologies. Parity substrate. `https://github.com/paritytech/substrate`.

[29] Pedersen, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91* (Berlin, Heidelberg, 1992), J. Feigenbaum, Ed., Springer Berlin Heidelberg, pp. 129–140.

[30] POLYNOMEN CONTRIBUTORS. polynomen. https://daingun.gitlab.io/automatica/polynomen/.

[31] RUST CRYPTO ORG. Rust crypto. https://github.com/RustCrypto.

[32] SHOUP, V. *A Computational Introduction to Number Theory and Algebra*, 2 ed. Cambridge University Press, 2008.

[33] SMITH, B. ring. https://github.com/briansmith/ring.

[34] SOLANA LABS. Solana. https://github.com/solana-labs/solana.

[35] VON ZUR GATHEN, J., AND GERHARD, J. *Modern Computer Algebra*, 3 ed. Cambridge University Press, 2013.

[36] WIKIPEDIA CONTRIBUTORS. Legendre's formula — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Legendre%27s_formula&oldid=1184628007, 2023.

[37] WIKIPEDIA CONTRIBUTORS. Chinese remainder theorem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Chinese_remainder_theorem&oldid=1218592501, 2024.

[38] WIKIPEDIA CONTRIBUTORS. Dependent type — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Dependent_type&oldid=1214224408, 2024.

[39] WIKIPEDIA CONTRIBUTORS. Euclidean domain — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Euclidean_domain&oldid=1214273601, 2024.

[40] WIKIPEDIA CONTRIBUTORS. Fast fourier transform — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Fast_Fourier_transform&oldid=1221693816, 2024.

[41] WIKIPEDIA CONTRIBUTORS. Integral domain — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Integral_domain&oldid=1198224596, 2024.

[42] WIKIPEDIA CONTRIBUTORS. Lagrange polynomial — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Lagrange_polynomial&oldid=1218658172, 2024.

[43] WIKIPEDIA CONTRIBUTORS. Negligible function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Negligible_function&oldid=1214342485, 2024.

[44] WIKIPEDIA CONTRIBUTORS. Pp (complexity) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PP_(complexity)&oldid=1200382155, 2024.

[45] WIKIPEDIA CONTRIBUTORS. Random-access machine — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Random-access_machine&oldid=1214064057, 2024.

[46] Wikipedia contributors. Random oracle — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Random_oracle&oldid=1226845624`, 2024.