Polynomial Inversion Algorithms in Constant Time for Post-Quantum Cryptography

Abhraneel Dutta
Emrah Karagoz
Edoardo Persichetti
Pakize Sanal

Florida Atlantic University (USA)

Overview

- Constant Time Polynomial Inversion
- Our Contributions
- Constant-Time Fermat Little Theorem and Extended GCD Based Inversion
- IT Variant Inversion
- Software Implementation
- Observations and Future Work

Polynomial Inversion in Post-Quantum Cryptosystem

- Polynomial inversion is a crucial operation in many post-quantum cryptosystems.
- For example, polynomial inversion is performed during the key generation algorithm of both BIKE and LEDAcrypt KEMs.
- It is important that there exists an efficient algorithm capable of running in constant time, to prevent timing side-channel attacks.
- A constant-time algorithm runs for the same time regardless of the input data size.

Our Contributions

- We analyze the performance of both the constant-time algorithms, specifically for the cryptosystems like BIKE and LEDACrypt, based on their mathematical foundations.
- The variants of FLT-based inversions used for Elliptic Curve Scalar Multiplication have not been explored in the context of a Constant Time setup for post-quantum cryptosystems.
- Computationally perform better with less number of polynomial multiplications compared to constant-time IT inversion.
- A performance comparison is conducted through benchmarking using software implementation.

Bernstein-Yang Inversion (SafeGCD)

- The key generation in BIKE computes the multiplicative inverse of a secret polynomial $h_0 \in R = \mathbb{F}_2[x]/\langle x^p + 1\rangle$ where $x^p + 1 = (x+1)(\sum_{i=0}^{p-1} x^i)$ with $ord_p(2) = p - 1$ *i.e.* 2 is a primitive element of $GF(p)$.
- Extended Euclidean Algorithm ($extGCD$) takes two polynomials f and g and outputs $(gcd(f, g), u, v)$ where $gcd(f, g) = u \cdot f + v \cdot g$ where $f, g, u, v \in \mathbb{F}_2[x]$.
- For the case of BIKE set up, $extGCD(x^p + 1, h_0) = (1, u, v)$.
- $$\begin{aligned} & u \cdot (x^p + 1) + v \cdot h_0 = 1 \\ \Rightarrow & v \cdot h_0 \equiv 1 \; mod \; (x^p + 1) \\ \Rightarrow & h_0^{-1} = v \text{ in } R \end{aligned}$$

Motivation

- Traditional *extGCD* algorithm is not suitable for cryptographic applications since it usually contains branches.
- Inputs are the secrets, but an attacker can still collect information about the inputs through running time differences.
- A constant time *extGCD* is required to prevent such timing attack.
- Bernstein and Yang provides a constant time version of *extGCD*

Division Steps or *divstep* function

- The *divstep* function is defined as:

$$divstep : \mathbb{Z} \times \mathbb{F}_2[x] \times \mathbb{F}_2[x] \rightarrow \mathbb{Z} \times \mathbb{F}_2[x] \times \mathbb{F}_2[x]$$

$$divstep(\delta, f, g) = \begin{cases} (1-\delta, g, \frac{(g(0)f - f(0)g)}{x}) & \text{if } \delta > 0 \text{ and } g(0) \neq 0 \\ (1+\delta, f, \frac{(f(0)g - g(0)f)}{x}) & \text{otherwise} \end{cases}$$

- Bernstein and Yang showed that dividing a degree m_0-polynomial by a degree m_1 polynomial with $m_0 > m_1 \geq 0$ is equivalent to computing $2m_0 - 2m_1$ many *divsteps*.
- For two coprime polynomials R_0 of degree m_0 and R_1 of degree m_1 with $m_0 > m_1$, it takes $2m_0 - 1$ many *divstep* calls to compute R_1^{-1} mod R_0.

Algorithm Analysis

- This algorithm reverses the polynomial in terms of coefficients. For an input f it performs $f' \leftarrow f(1/x)x^{deg(f)}$.
- For the two inputs $f = x^p + 1$, $g = h_0(1/x) \cdot x^p$ and their degree difference be $\delta = 1$, the algorithm computes h_0^{-1}.
- This algorithm performs constant number of division steps $(2p - 1)$ or *divstep* function calls to compute the inverse of the input polynomial.

Algorithm Analysis

- The transition of the two polynomials f, g under the *divstep* operation is described as a matrix-vector multiplication.
- A transition matrix $T(\delta, f, g)$ performs a transition $(f, g) \to (f_1, g_1)$.

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = T(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix} \text{ where} T(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{x} & -\frac{f(0)}{x} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0 \\ \begin{pmatrix} 1 & 0 \\ -\frac{f(0)}{x} & \frac{g(0)}{x} \end{pmatrix} & \text{otherwise} \end{cases}$$

- The i-th step transition matrix is $T_i = T(\delta_i, f_i, g_i)$. After n-steps, the input polynomial becomes

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = T_{n-1} \cdots T_0 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}$$

Divstep Speed Up (*jumpdivstep*)

Divide and conquer algorithm to compute (δ_n, f_n, g_n) and the n-step transition matrix $T_{n-1} \cdots T_0$:

- The *jumpdivstep* function is recursive in nature which splits the problem into two parts and after recursive calls when it reaches its base case the *divstep* function does its work.
- Jump j steps from δ, f, g to δ_j, f_j, g_j by calling the same algorithm recursively.
- Similarly, jump another $n - j$ steps from δ_j, f_j, g_j to δ_n, f_n, g_n.
- The splitting point can be chosen for any non-null portion of the maximum degree n, although $j = n/2$ is likely to be optimal.

Complexity Analysis

- The *divstep* algorithm performs $O(n(m+n))$ operations for n divsteps and m degree polynomial (the polynomial with the higher degree).
- With the *jumpdivstep* speed up function and FFT based polynomial multiplication, the number of operations can be bounded by $(n+m)\log(n+m) + c'n(\log(n))^2$ when n is sufficiently large.
- For $n = m$ the number of operations is $O(n(\log n)^2)$

IT Inversion

- Computes inverse of a polynomial $a(x) \in R^*$ using Fermat's Little Theorem as:

$$a(x)^{-1} \equiv a(x)^{2^{p-1}-2} \equiv (a(x)^{2^{p-2}-1})^2 \ mod \ (x^p + 1)$$

- Rewrite $2^{p-2} - 1 = \sum_{i \in supp(p-2)} (2^{2^i} - 1) \cdot 2^{(p-2) \ mod \ 2^i}$
- If the binary representation of $p-2$ is $(p-2)_2 = (a_i)_{i=0}^{p-1}$ where $a_i \in \mathbb{F}_2$ then *support* of $p-2$ is defined as $supp(p-2) = \{i \in \{0, 1, \ldots, p-1\} | \ a_i \neq 0\}$
- Note: For $a(x) \in R^*$ computing $a(x)^{2^k}$ for some $k \in \mathbb{Z}^*$ is equivalent to performing a permutation or cyclic shift to its coefficients. It can be called as a *k-squaring* of a polynomial.
- $a(x)^{2^k} = (\sum_{i \in supp(a)} x^i)^{2^k} = \sum_{i \in supp(a)} (x^i)^{2^k} = \sum_{i \in supp(a)} x^{i \cdot 2^k \ mod \ p}$

Constant-Time IT Inversion

- Based Fermat's Little Theorem.
- Computes inverse of a polynomial over a polynomail ring $R = \mathbb{F}_2[x]/\langle(x^p+1)\rangle$ where $x^p+1=(x+1)(\sum_{i=0}^{p-1} x^i)$.
- For $p = 13$, $p-2 = 11 = (1011)_2$ and $a(x) \in \mathcal{R}^*$ let's compute $a(x)^{-1} = (a(x)^{2^{11}-1})^2$

i	$\|p-2\|[i]$	$a^{2^{2^i}-1}$	$a^{\sum_{k_j=i,\|p-2\|[i]=1} 2^{2^{k_j}}-1}$
0	1	$a^{2^{2^0}-1}$	$a^{2^{2^0}-1}$
1	1	$(a^{2^{2^0}-1})^{2^{2^0}+1} = a^{2^{2^1}-1}$	$a^{2^{2^0}-1} \cdot a^{(2^{2^1}-1)\cdot 2^{2^0}} = a^{2^{2^0+2^1}-1}$
2	0	$(a^{2^{2^1}-1})^{2^{2^1}+1} = a^{2^{2^2}-1}$	
3	1	$(a^{2^{2^2}-1})^{2^{2^2}+1} = a^{2^{2^3}-1}$	$a^{2^{2^0+2^1}-1} \cdot a^{(2^{2^3}-1)\cdot 2^{2^0+2^1}} = a^{2^{2^0+2^1+2^3}-1} = a^{2^{11}-1}$

Complexity Analysis

- The algorithm takes $\lfloor log(p-2) \rfloor + \lfloor wt(p-2) \rfloor - 1$ polynomial multiplications and $\lfloor log(p-2) \rfloor + \lfloor wt(p-2) \rfloor - 1$ many k-squarings and one polynomial squaring in R.
- The value of k in k-squaring depends on p but not on $a(x)$.
- For a fixed prime p, the permutation $\sigma_k : j \to j \cdot 2^k \ mod \ p$ can be precomputed for all relevant values of k which also depends only on p.

ITI Variants

- **Purpose**: Explore exponentiation algorithms that reduce polynomial multiplications in calculating the inverse of input polynomials compared to IT inversion.
- **Key Feature**: Algorithms factor and decompose exponents for constant-time implementation, regardless of polynomial degree.
- **Efficiency**:
  - Minimizes the number of polynomial multiplications.
  - Achieves a constant-time structure.
- **Implementation**:
  - Applicable to any polynomial degree.
  - Can be implemented in Constant-Time setup

CEA Algorithm (Factoring Method)

- T. Chang, E. Lu, Y. Lee, Y. Leu, and H. Shyu presented an algorithm that factors the exponent to perform inversion.
- The inverse of α is computed using Fermat's Little Theorem:

$$\alpha^{-1} = \alpha^{2^{p-1}-2} = (\alpha^{2^{p-2}-1})^2 \text{ in } R$$

- Factor $p - 2 = a \cdot b$
- Decompose
$2^{p-1} - 2 = 2 \cdot (2^{a \cdot b} - 1) = 2 \cdot (2^a - 1)((2^a)^{b-1} + \cdots + (2^a) + 1)$
- Key equation: $\alpha^{2^{p-2}-1} = (\alpha^{2^a-1})^{(2^a)^{b-1}+(2^a)^{b-2}+\cdots+2^a+1}$

CEA Complexity

- Computing $\beta = \alpha^{2^a - 1}$ needs $\lfloor \log(a) \rfloor + \mathrm{wt}(a) - 1$ multiplications.
- Exponentiation β^t where $t = 1 + 2^a + \cdots + (2^a)^{b-1}$ needs $\lfloor \log(b) \rfloor + \mathrm{wt}(b) - 1$ multiplications.
- Final number of multiplications:

$$(\lfloor \log(b) \rfloor + \mathrm{wt}(b) - 1) + (\lfloor \log(a) \rfloor + \mathrm{wt}(a) - 1)$$

TYT Algorithm (Decomposition Method)

- Proposed by Takagi, Yoshiki, and Takagi
- Decomposes the exponent into a product of factors and a remainder as follows: $p - 2 = \prod_{i=1}^{k} r_i + h$
- Decomposition:

$$2^{p-1} - 2 = 2^{p-2} + 2^{p-3} + \cdots + 2^{p-h-1} + 2^{p-h-1} - 2$$

- The inversion performed as follows:

$$\alpha^{-1} = \alpha^{2^{p-1}-2} = \underbrace{\alpha^{2^{p-2}} \cdot \alpha^{2^{p-3}} \cdots \alpha^{2^{p-h-1}} \cdot \alpha^{2^{p-h-1}}}_{h \text{ mults}} \cdot (\alpha^{2^{p-h-2}-1})^2$$

TYT Algorithm

- Let $M(x) = \lfloor \log_2(x) \rfloor + wt(x) - 1$
- $$\alpha^{(2^{\prod_{i=1}^{k} r_i}-1)} = \underbrace{(\cdots \underbrace{(\underbrace{(\underbrace{\alpha^{(2^{r_1}-1)}}_{M(r_1)})^{(2^{r_1})^{r_2-1}+\cdots+1}}_{M(r_2)})^{(2^{r_1 r_2})^{r_3-1}+\cdots+1}}_{M(r_3)} \cdots)^{(2^{\prod_{i=1}^{k-1} r_i})^{r_k-1}+\cdots+1}}_{M(r_k)}$$
- $\#\text{Multiplications}=\sum_{i=1}^{k} M(r_i) + h = \sum_{i=1}^{k}(\lfloor \log_2(r_i) \rfloor + wt(r_i) - 1) + h$

Improved TYT

- Proposed by Y. Li, G. Chen, Y. Chen, and J. Li
- Optimized decomposition:

$$2^{p-1} - 2 = 2^{p-1-h}(2^h - 1) + 2(2^{p-2-h} - 1)$$

- Decompose $p - 2 = \prod_{i=1}^{k} r_i + h$ with $h < r_1$.
- For Optimal decomposition $wt(h) < wt(p-2) - 2$
- Improves TYT algorithm by re-using some intermediate results

Improved TYT

- Let $r_1 \geq h$. $r_1 = \sum_{i=1}^{n} 2^{u_i}$ and $h = \sum_{i=1}^{\ell} 2^{t_i}$ with $u_1 > u_2 > \ldots > u_n$ and $t_1 > t_2 > \ldots > t_\ell$ respectively
- Calculate u_1 intermediate values where $u_1 > t_1$:

$$\{\alpha^{2^{2^0}-1}, \alpha^{2^{2^1}-1}, \ldots, \alpha^{2^{2^{t_1}}-1}, \ldots \alpha^{2^{2^{u_1}}-1}\}$$

- Used to compute $\alpha^{2^{r_1}-1}$ and α^{2^h-1} along with $wt(r_1)$ and $wt(h)$ multiplications.
- Total multiplications:

$$\sum_{i=1}^{k}(\lfloor \log_2(r_i) \rfloor + \mathrm{wt}(r_i) - 1) + \mathrm{wt}(h)$$

Short Addition Chain Method

- The Short Addition Chain (ShAC) of a positive integer r, denoted as C_r, is a sequence of integers with length n where r is obtained by the addition of previous elements within the chain. For example, Let $r = 18$. An addition chain for 18 could be: $1, 2, 4, 8, 9, 18$.
- It is particularly efficient for large values of numbers with higher Hamming weights.
- Provides a significant improvement over traditional IT and TYT algorithms for specific cases.

Addition Chain Algorithm

- Given $p-2=\prod_{i=1}^{k} r_i + h$, a short addition chain of r_1 namely C_{r_1} is constructed with $h \in C_{r_1}$.

- This setup guarantees the computation of α^{2^h-1} while calculating $\alpha^{2^{r_1}-1}$.

- #Multiplications $\leq \sum_{i=1}^{k}(\lfloor \log(r_i) \rfloor + wt(r_i) - 1) + 1$

- The decomposition of $2^{p-1}-2$ can be done as follows:

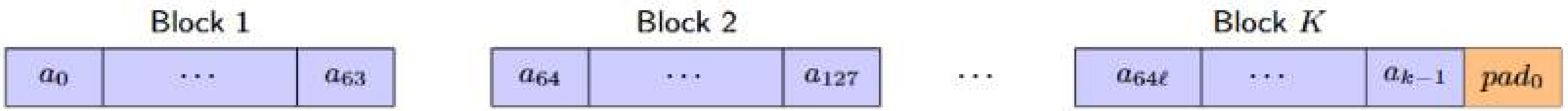
$$
\begin{aligned}
2^{p-1}-2 &= 2(2^{\prod_{i=1}^{k} r_i + h} - 1) = 2(2^h \cdot (2^{\prod_{i=1}^{k} r_i} - 1) + (2^h - 1)) \\
&= 2((2^{r_1}-1) \cdot e \cdot 2^h \cdot (2^h - 1)) \text{ and consequently} \\
\alpha^{-1} &= ((\alpha^{2^{r_1}-1})^{(e)2^h} \cdot (\alpha^{2^h-1}))^2 \\
\text{where } e &= (((2^{r_1})^{r_2-1} + \cdots + 1) \cdots ((2^{r_1 \cdot r_2 \cdots r_{k-1}})^{r_k-1} + \cdots + 1))
\end{aligned}
$$

Comparative Analysis

Table: Comparison of the number of multiplications with different inversion algorithms discussed in this article using primes corresponding to different levels of BIKE implementation.

p	$(p-2)$	CEA Factorization	TYT Decomposition	SAC Decomposition	# Mults (ITI)	# Mults (CEA)	# Mults (TYT)	# Mults (SAC)
10499	4	3×3499	$41 \times 256 + 1$	$41 \times 2^8 + 1$	16	20	16	16
12323	4	$3^2 \times 37^2$	$12289 + 32$	$48 \times 2^8 + 33$	16	19	16	16
24781	7	71×349	$3 \times 8257 + 8$	$193 \times 2^7 + 75$	20	22	18	19
27067	9	5×5413	$67 \times 403 + 64$	$211 \times 2^7 + 57$	22	20	21	20
24659	5	3×8219	$5 \times 4112 + 4097$	$385 \times 2^6 + 17$	18	19	18	18
27581	11	3×9193	$163 \times 169 + 32$	$215 \times 2^7 + 59$	24	22	21	20
40973	5	3×13657	$10 \times 4097 + 1$	$20 \times 2^{1}1 + 11$	19	22	18	18

- A polynomial $\alpha(x) = \sum_{i=0}^{k-1} a_i x^i \in_2 [x]$ (with $a_{k-1} = 1$) is stored in $K = \lceil k/64 \rceil$ blocks.



Benchmarking of Polynomial Inversion Algorithms

p (size64)		BYI	ITI	CEA	TYT	SAC
10499 (165)	x86	4.35	12.85	8.38	15.02	7.66
	arm64	7.37	16.32	13.78	17.35	12.02
	# gf2x_mod_mul	N/A	16	20	16	16
	# gf2x_mod_sqr	N/A	18,688	10,497	20,992	10,538
	# mul64	≈ 1.10M	≈ 0.44M	≈ 0.54M	≈ 0.44M	≈ 0.44M
	# sqr64	-	≈ 3.10M	≈ 1.74M	≈ 3.48M	≈ 1.75M
12323 (193)	x86	5.79	17.02	10.89	20.07	10.93
	arm64	12.23	21.91	18.74	25.25	16.92
	# gf2x_mod_mul	N/A	16	19	16	16
	# gf2x_mod_sqr	N/A	20,152	12,321	24,578	12,369
	# mul64	≈ 1.51M	≈ 0.60M	≈ 0.71M	≈ 0.60M	≈ 0.60M
	# sqr64	-	≈ 3.98M	≈ 2.39M	≈ 4.77M	≈ 2.40M
24659 (386)	x86	21.03	66.25	42.51	64.32	42.61
	arm64	45.73	95.18	77.74	96.50	77.88
	# gf2x_mod_mul	N/A	18	19	18	19
	# gf2x_mod_sqr	N/A	41,040	24,657	41,121	24,739
	# mul64	≈ 6.05M	≈ 2.68M	≈ 2.83M	≈ 2.68M	≈ 2.83M
	# sqr64	-	≈ 15.88M	≈ 9.54M	≈ 15.91M	≈ 9.57M
24781 (388)	x86	22.11	65.64	44.13	76.33	42.18
	arm64	46.30	101.52	86.50	107.21	80.28
	# gf2x_mod_mul	N/A	20	22	18	19
	# gf2x_mod_sqr	N/A	41,162	24,779	49,542	25,091
	# mul64	≈ 6.10M	≈ 3.01M	≈ 3.31M	≈ 2.71M	≈ 2.86M
	# sqr64	-	≈ 16.01M	≈ 9.64M	≈ 19.27M	≈ 9.76M
27067 (423)	x86	24.43	76.65	50.19	91.81	51.94
	arm64	54.42	125.42	98.09	137.71	101.62
	# gf2x_mod_mul	N/A	22	20	21	20
	# gf2x_mod_sqr	N/A	43,448	27,065	54,002	27,337
	# mul64	≈ 7.09M	≈ 3.94M	≈ 3.58M	≈ 3.76M	≈ 3.58M
	# sqr64	-	≈ 18.42M	≈ 11.48M	≈ 22.90M	≈ 11.59M
27581 (431)	x86	25.34	81.5	53.4	94.94	53.4
	arm64	57.34	139.49	111.14	146.93	103.03
	# gf2x_mod_sqr	N/A	43,962	27,579	55,094	27,850
	# gf2x_mod_mul	N/A	24	22	21	20
	# mul64	≈ 7.34M	≈ 4.46M	≈ 4.09M	≈ 3.90M	≈ 3.72M
	# sqr64	-	≈ 18.99M	≈ 11.91M	≈ 23.80M	≈ 12.03M
40973 (641)	x86	58.14	191.2	116.62	208.12	113.43
	arm64	127.29	284.38	246.86	300.02	217.01
	# gf2x_mod_mul	N/A	19	22	18	18
	# gf2x_mod_sqr	N/A	73,738	40,971	81,940	40,983
	# mul64	≈ 17.02M	≈ 7.81M	≈ 9.04M	≈ 7.40M	≈ 7.40M
	# sqr64	-	≈ 47.34M	≈ 26.30M	≈ 52.61M	≈ 26.31M

Our Observation

- IT variants perform fewer polynomial multiplications while computing the inverse with specific choice for the primes. This improvement can be achieved through an optimized decomposition and factoring setup, especially as the Hamming weight of $p - 2$ increases for various primes
- SAC and CEA inversions show a better performance with 1.56x-1.96x on `x86` and 1.24x-1.49x on `arm64` compared to the ITI and TYT methods.
- However, BY inversion has a better performance with 1.76x-3.76x on `x86` and 1.38x-2.56x on `arm64` compared to IT and its variants.
- IT variants seem better than BY inversion in the hardware designs by only comparing the number of polynomial multiplications; because of that, the polynomial squaring can be implemented as nearly "cost-free" performance overhead in the hardware designs through special methods.


THANK YOU... I MUST

Exponentiation Theorem

Lemma 1

Let $\alpha \in GF(2^m)$ *and* $t = 1 + 2^a + (2^a)^2 + \ldots + (2^a)^{b-1}$. *Then there exists an algorithm for computing* α^t *which requires* $M(b) = \lfloor \log_2(b) \rfloor + wt(b) - 1$ *multiplications.*

Multiplications Calculation

- Let $\beta = \alpha^{2^{r_1}-1}$. The exponentiation β^e can be computed with $\sum_{i=2}^{k}(\lfloor \log(r_i) \rfloor + wt(r_i) - 1)$ multiplications.

- Let $\mathcal{C}_{r_1} = \{c_0, c_1, \ldots, c_{n-1}\}$ be an addition chain for r_1 where $c_0 = 1$ and $c_{n-1} = r_1$.

- Let $\mathcal{A}_{r_1} = \{(c_1^1, c_1^2), (c_2^1, c_2^2), \ldots, (c_{n-1}^1, c_{n-1}^2) |\ c_i^1 + c_i^2 = c_i$ and $c_i^j \in \mathcal{C}_{r_1} \forall\ i = 1, 2, \ldots, n-1$ and $j = 1, 2\}$ be the set of addition pairs.

- $\#\text{Multiplications} \leq \sum_{i=1}^{k}(\lfloor \log(r_i) \rfloor + wt(r_i) - 1) + 1$

Multiplications Calculation

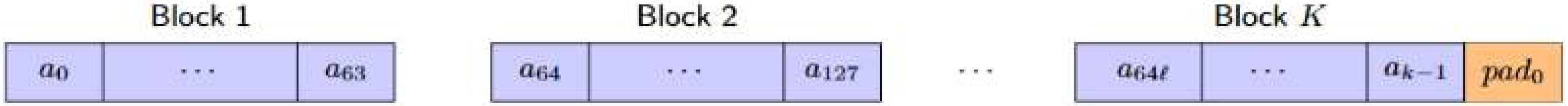
- Let $p = 149$, $p - 2 = 147 = 18 \cdot 8 + 3$ where $\mathcal{C}_{18} = \{1, 2, \mathbf{3}, 6, 12, 18\}$ and $\mathcal{A}_{18} = \{(1,1), (2,1), (3,3), (6,6), (12,6)\}$
- Computation of α^{-1} with $2^{148} - 2 = 2((2^{18 \cdot 8} - 1)2^3 + (2^3 - 1))$

$$\alpha^{2^{18}-1} \to (\alpha^{2^{18}-1})^{(2^{18})^7+\cdots+(2^{18})+1} = \alpha^{2^{18\cdot 8}-1}$$
$$\to (\alpha^{2^{18\cdot 8}-1})^{2^3} \to ((\alpha^{2^{18\cdot 8}-1})^{2^3} \cdot \alpha^{2^3-1})^2 = \alpha^{2^{148}-2}$$

$$(\alpha^{2^1-1})^{2^1} \cdot (\alpha^{2^1-1}) = (\alpha^{2^{1+1}-1}) = (\alpha^{2^2-1}) \quad \text{1 mult}$$
$$(\alpha^{2^2-1})^{2^1} \cdot (\alpha^{2^1-1}) = (\alpha^{2^{2+1}-1}) = (\alpha^{2^3-1}) \quad \text{1 mult}$$
$$(\alpha^{2^3-1})^{2^3} \cdot (\alpha^{2^3-1}) = (\alpha^{2^{3+3}-1}) = (\alpha^{2^6-1}) \quad \text{1 mult}$$
$$(\alpha^{2^6-1})^{2^6} \cdot (\alpha^{2^6-1}) = (\alpha^{2^{6+6}-1}) = (\alpha^{2^{12}-1}) \quad \text{1 mult}$$
$$(\alpha^{2^{12}-1})^{2^6} \cdot (\alpha^{2^6-1}) = (\alpha^{2^{12+6}-1}) = (\alpha^{2^{18}-1}) \quad \text{1 mult}$$

Representation of a polynomial

- A polynomial $\alpha(x) = \sum_{i=0}^{k-1} a_i x^i \in_2 [x]$ (with $a_{k-1} = 1$) is stored in $K = \lceil k/64 \rceil$ blocks.



- `mul64`: Multiplication of two 64-bit blocks
- `sqr64`: Squaring of a 64-bit block
- Implemented in both `x86-64` and `arm64`
- Uses Carry-less multiplication (CLMUL) instructions for 64-bit blocks
- The resulting block is 128-bit

Implemented Functions

- `gf2x_poly_mul`: Regular polynomial multiplication
  - Calls `mul64` as needed
  - Mainly used in BY inversion
- `gf2x_mod_mul`: Modular polynomial multiplication in modulo $x^p - 1$
  - Calls a regular polynomial multiplication and a reduction function
  - Mainly used in FLT-based inversions
- `gf2x_mod_sqr`: Squaring of a polynomial `a` in modulo $x^p - 1$
  - Mostly used in the end of FLT-based inversions
- `gf2x_mod_sqr_k_inplace`: Repeated squaring of a polynomial `c` in-place, k times, with reduction modulo $x^p - 1$ after each squaring
  - Saves from initialization and storing cost
  - Mostly used in FLT-based inversions