Issues in Implementation of Public Key Cryptosystems

by

Jaewook Chung

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2006

©Jaewook Chung 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Jaewook Chung

Abstract

A new class of moduli called the low-weight polynomial form integers (LWPFIs) is introduced. LWPFIs are expressed in a low-weight, monic polynomial form, $p = f(t)$. While the generalized Mersenne numbers (GMNs) proposed by Solinas allow only powers of two for t, LWPFIs allow any positive integers. In our first proposal of LWPFIs, we limit the coefficients of $f(t)$ to be 0 and ± 1, but later we extend LWPFIs to allow any integer of magnitude less than t for the coefficients of $f(t)$. Modular multiplication using LWPFIs is performed in two phases: 1) polynomial multiplication in $\mathbb{Z}[t]/f(t)$ and 2) coefficient reduction. We present an efficient coefficient reduction algorithm based on a division algorithm derived from the Barrett reduction algorithm. We also show a coefficient reduction algorithm based on the Montgomery reduction algorithm. We give analysis and experimental results on modular multiplication using LWPFIs.

New three, four and five-way squaring formulae based on the Toom-Cook multiplication algorithm are presented. All previously known squaring algorithms are symmetric in the sense that the point-wise multiplication step involves only squarings. However, our squaring algorithms are asymmetric and use at least one multiplication in the point-wise multiplication step. Since squaring can be performed faster than multiplication, our asymmetric squaring algorithms are not expected to be faster than other symmetric squaring algorithms for large operand sizes. However, our algorithms have much less overhead and do not require any nontrivial divisions. Hence, for moderately small and medium size operands, our algorithms can potentially be faster than other squaring algorithms. Experimental results confirm that one of our three-way squaring algorithms outperforms the squaring function in GNU multiprecision library (GMP) v4.2.1 for certain range of input size. Moreover, for degree-two squaring in $\mathbb{Z}[x]$, our algorithms are much faster than any other squaring algorithms for small operands.

We present a side channel attack on XTR cryptosystems. We analyze the statistical behavior of simultaneous XTR double exponentiation algorithm and determine what information to gather to reconstruct the two input exponents. Our analysis and experimental results show that it takes $U^{1.25}$ tries, where $U = \max(a, b)$ on average to find the correct exponent pair (a, b). Using this result, we conclude that an adversary is expected to make $U^{0.625}$ tries on average until he/she finds the correct secret key used in XTR single exponentiation algorithm, which is based on the simultaneous XTR double exponentiation algorithm.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor M. Anwar Hasan for his immense support, encouragement and guidance. I am also very grateful to the committee members – Professor Gordon Agnew, Professor Guang Gong and Professor Alfred Menezes – for taking their valuable time to review this thesis.

I would like to thank Jean Claude Bajard of the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) for kindly accepting to be an external examiner. The recent research work [5, 6] by him, Laurent Imbert and Thomas Plantard has inspired me to do the work presented in Chapter **5**.

Contents

1 Introduction **1**

1.1 Motivation and Scope . . . 1

1.1.1 Low-Weight Polynomial Form Integers . . . 1

1.1.2 Coefficient Reduction Based on the Montgomery Reduction Algortihm . 2

1.1.3 Asymmetric Squaring Formulae . . . 2

1.1.4 Side Channel Attack on XTR Cryptosystems . . . 3

1.2 Thesis Outline . . . 4

1.3 Research Contributions . . . 5

2 Review of Multiplication and Modular Reduction Algorithms **6**

2.1 Review of Multiplication Algorithms . . . 6

2.1.1 Zimmermann's 3-Term Toom-Cook Multiplication . . . 9

2.2 Review of Modular Reduction Algorithms . . . 9

2.2.1 Classical Division Algorithm . . . 11

2.2.2 Montgomery Reduction Algorithm . . . 11

2.2.3 Barrett's Reduction Algorithm . . . 13

3 Review of XTR Cryptosystems **17**

3.1 Mathematical Preliminaries on XTR Cryptosystems . . . 17

3.1.1 Basic Ideas . . . 17

3.1.2 Efficient Arithmetic Operations in $GF(p^2)$ . . . 19

3.1.3 Trace Representations and Properties of Sequence c_n . . . 21

3.2 Review of XTR Exponentiation Algorithms . . . 23

3.2.1 XTR Single Exponentiation . . . 23

3.2.2 XTR Double Exponentiation 24
3.2.3 XTR Single Exponentiation Revisited 26

4 Low-Weight Polynomial Form Integers for Efficient Modular Multiplication 28
4.1 Modular Multiplication Using Low-Weight Polynomial Form Integers 28
4.1.1 POLY-MULT-REDC: Multiplication in $\mathbb{Z}[t]/f(t)$ 30
4.1.2 COEFF-REDC: Coefficient Reduction 31
4.2 Optimization of POLY-MULT-REDC Step 33
4.2.1 Case 1: $l = 2$ 34
4.2.2 Case 2: $l = 3$ 35
4.3 Analysis of LWPFI Modular Multiplication 37
4.3.1 POLY-MULT-REDC step 37
4.3.2 COEFF-REDC step 39
4.3.3 Putting It All Together 41
4.3.4 Comments 43
4.4 Implementation Results and Practical Considerations 43
4.4.1 Our Platform and Software Routines 43
4.4.2 Component-wise Breakdown of Timing 44
4.4.3 Overall Timing Results and Comparisons 47
4.4.4 Practical Considerations 47
4.5 Enhancing the LWPFI Modular Multiplication 50
4.5.1 Using Pseudo-Mersenne Numbers for t ($t = 2^n - c$) 50
4.5.2 Using LWPFI for t 51
4.6 Conclusions 52

5 Coefficient Reduction Using Montgomery Reduction Algorithm 54
5.1 Low-Weight Polynomial Form Integers Redefined 54
5.2 Modular Multiplication Using LWPFI moduli 55
5.2.1 COEFF-REDC based on Montgomery Reduction Algorithm 57
5.2.2 Construction of F and Analysis of Algorithm 5.2 60
5.2.3 Conversions to and from the Montgomery Domain 64
5.2.4 Interesting Implementation Options 64
5.3 Modular Multiplication Stability 65

5.3.1 Montgomery Reduction with Final Subtractions 65
5.3.2 Montgomery Reduction without Final Subtractions 66
5.4 Additions and Subtractions 68
5.4.1 Case I: $t/2 + \xi + 1 \leq \psi \leq t - (\xi + 1)$ 70
5.4.2 Case II: $t + 2(\xi + 1) \leq \psi \leq 2t - 2(\xi + 1)$ 72
5.5 Comparisons 74
5.6 Applications of LWPFI Modular Multiplications 75
5.7 Application to Modular Number Systems? 75
5.8 Conclusions 77

6 Asymmetric Squaring Formulae **79**
6.1 Further Details on the Toom-Cook Multiplication Algorithm 79
6.2 New Squaring Formulae 85
6.2.1 Our Approach 86
6.2.2 New 3-way Squaring 89
6.3 Analysis 92
6.4 Implementation Results 95
6.4.1 Application to Large Integer Squaring 95
6.4.2 Application to Polynomial Squaring in $\mathbb{Z}[x]$ 99
6.5 4-way and 5-way Squaring Formulae 100
6.5.1 New 4-Way Squaring 100
6.5.2 New 5-term Squaring Method 101
6.6 Conclusions 104

7 Side Channel Attack on XTR Cryptosystems **105**
7.1 Identifying Elementary XTR Operations 105
7.2 Simple Side Channel Attack 108
7.2.1 Markov Chain Model 109
7.2.2 Statistical Behavior of Algorithm 3.3 115
7.2.3 Determining f_2, f_3 and Sub-step Sequence 117
7.2.4 Determining d at Line 17 of Algorithm 3.3 120
7.2.5 Determining Boundaries Between Steps 125
7.3 Effectiveness of Markov Chain Method 126

7.4 Extension to Single Exponentiation Algorithms 128
7.5 Other Researchers' Results 129
7.6 Conclusions 130

8 Conclusions and Future Work **131**
8.1 Conclusions 131
8.2 Future Work 133

Bibliography **134**

A Source Codes **143**
A.1 Source Code for SQR_3 144
A.2 Source Code for Improved Zimmermann's 3-way Toom-Cook Squaring Algorithm147

List of Tables

3.1 Updating Formulae for Line 10 of Algorithm 3.3 26
3.2 Updating Formulae for Line 13 of Algorithm 3.3 26

4.1 Modular Multiplication and Squaring Cost in $\mathbb{Z}[t]/f(t)$ for All Irreducible $f(t)$'s of degree-2 39
4.2 Functions Used for Implementing LWPFI Modular Multiplications 44
4.3 Detailed Analysis of LWPFI Modular Multiplication ($n = \lceil \log_2(p+1) \rceil$) 45

5.1 Comparison of Algorithm 2.4 and Algorithm 5.2 74

6.1 List of Candidate Vectors 90
6.2 Analysis Results of Various Squaring Algorithms 94
6.3 Conditions for Which SQR_i's Are Faster Than Other 3-way Algorithms 94
6.4 Timing Results of Polynomial Squaring on Pentium IV 3.2GHz (unit= μs.) . . . 99

7.1 Update Rules for Algorithm 3.3 108
7.2 Sub-step Probabilities 116
7.3 Impossible Sub-step Sequences 117
7.4 Expected Number of Choices at a Sub-step 119

List of Figures

4.1 Coefficient Reduction 40
4.2 Timing Results for POLY-MULT-REDC Step on Pentium 4 @ 3.2GHz 46
4.3 Timing Results for COEFF-REDC Step on Pentium 4 @ 3.2GHz 46
4.4 Modular Multiplication Algorithms on Pentium 4 @ 3.2GHz 48
4.5 Modular Squaring Algorithms on Pentium 4 @ 3.2GHz 48

5.1 Range of Transfer Digit t_{i+1} ($\mu = \xi + 1$) 71
5.2 Range of Transfer Digit t_{i+1} ($\mu = 2(\xi + 1)$) 73

6.1 Timing Ratio of `mpn_divexact_by3()` to `mpz_mul()` 84
6.2 Timing Results of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring) . . . 86
6.3 Timing Results of Squaring Algorithms (Pentium IV 3.2GHz) 97
6.4 Timing Ratio of SQR_3 vs. Other Algorithms on Pentium IV 3.2GHz 97
6.5 Timing Ratio of SQR_3 vs. `mpz_mul()` (Pentium II MMX 350MHz) 98
6.6 Timing Ratio of SQR_3 vs. `mpz_mul()` (Pentium III M 1.13GHz) 98
6.7 Timing Ratio of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring) 104

7.1 State Transition Diagram of Sub-steps 118
7.2 The Average Number of Tries Required and the Size of Search Space 120
7.3 Comparison Between Actual Values and (7.19) 124
7.4 Error Between Actual Values and (7.19) 125
7.5 Number of Tries 128

Chapter 1

Introduction

1.1 Motivation and Scope

1.1.1 Low-Weight Polynomial Form Integers

In 1644, Mersenne conjectured that the numbers of the form $p = 2^k - 1$ are prime numbers for a certain set of integers $k \leq 257$. Although his conjecture turned out to be not entirely correct, the numbers of the form $p = 2^k - 1$ are now known as the *Mersenne numbers*. It is very easy to perform modular reduction using these numbers. However, these numbers are not attractive for cryptographic applications since there are very few Mersenne primes (e.g., if k is composite, Mersenne numbers are never primes) that are practically useful.

The moduli of the form $p = 2^k - c$, where c is a small integer, are known as *pseudo-Mersenne numbers*. An efficient modular reduction algorithm using pseudo-Mersenne numbers is patented by Crandall [19]. Modular reduction using a pseudo-Mersenne number is also very efficient. However, because of security threats, these numbers are not recommended for cryptosystems that are based on the difficulty of integer factorization or discrete logarithm problem [49, 50, 75, 73].

In 1999, Solinas proposed generalized Mersenne numbers (GMNs). GMNs are expressed in polynomial form $p = f(t)$, where t is a power of 2 and the coefficients of low-degree polynomial $f(t)$ are very small compared to t. If the modulus is a GMN, the modular reduction requires simple integer additions and subtractions only. It is well-known that all prime-field based elliptic curves recommended by National Institute of Standards and Technology

(NIST) use GMNs [64, 66]. However, two significant shortcomings of GMNs are that there are not many useful GMNs and that each GMN requires dedicated implementation. Hence the use of GMN is currently limited to elliptic and hyperelliptic curve cryptosystems.

We introduce a new family of integers, called the *low-weight polynomial form integers* (LWPFIs). LWPFIs are similar to GMNs. However, for LWPFIs, t does not have to be a power of 2, and the coefficients of $f(t)$ are either 0 or ± 1. Unlike GMNs, LWPFIs do not require a dedicated implementation, since one implementation can be used to perform modular multiplication for many LWPFIs by varying the value of t. We present an efficient modular multiplication method based on LWPFI moduli. Our analysis and implementation results show that modular multiplication based on LWPFIs is asymptotically faster than any reduction algorithms for general moduli. For software implementation, our new modular multiplication based on LWPFI moduli can be implemented without using division instructions of the target processor. This feature is advantageous for processors whose division instruction is much slower than its multiplication instruction.

1.1.2 Coefficient Reduction Based on the Montgomery Reduction Algortihm

In modular multiplication using LWPFI moduli, the coefficient reduction algorithm is based on a division algorithm derived from the Barrett reduction algorithm. In our subsequent work, we generalize LWPFIs by removing the restriction on f_i's and we present a new coefficient reduction algorithm based on the Montgomery reduction algorithm. We show conditions on parameters for which our new coefficient reduction algorithm can perform without final subtractions. We analyze the performance of the new coefficient reduction algorithm using this general framework. As a side result, we present methods for additions and subtractions modulo an LWPFI in its generalized form.

1.1.3 Asymmetric Squaring Formulae

Multiplication is one of the most frequently used arithmetic operations in public key cryptography and the performance of a cryptosystem often depends mostly on the efficiency of a multiplication operation. Squaring is a special case of multiplication when two operands are identical and it is usually faster than multiplication, but not more than a constant factor.

Over the past four decades, many algorithms have been proposed to perform multiplication operation efficiently. Since Karatsuba discovered the first sub-quadratic multiplication algorithm [41], several innovations have been made on multiplication algorithms [79, 17, 87, 74]. Unfortunately, none of these sub-quadratic multiplication algorithms has been considerably specialized for squaring. In this work, we attempt to fill this gap in the literature. It is perhaps not possible to have a squaring algorithm that is asymptotically better than the fastest multiplication algorithm in a ring whose characteristic is greater than 2. However, there are possibilities of some optimization by exploiting the fact that two operands are identical. We present three 3-way squaring formulae that are based on the Toom-Cook multiplication algorithm. Detailed methods for obtaining such formulae are presented. Experimental results show that our algorithms are faster than other 3-way multiplication algorithms for certain range of operand sizes. We also present efficient 4-way and 5-way squaring formulae that are potentially useful in practice.

1.1.4 Side Channel Attack on XTR Cryptosystems

Due to the index calculus method, traditional cryptosystems based on the hardness of discrete logarithm problem must use large representation size, which is usually at least 1024 bits, even though they are based on a subgroup of order only about 2^{160}. Such a long representation size renders traditional cryptosystems disadvantageous both in bandwidth and computational efficiency.

In 2000, XTR cryptosystem was proposed by Lenstra and Verheul [53]. XTR stands for 'ECSTR' which is an abbreviation for 'Efficient and Compact Subgroup Trace Representation'. The security of XTR is based on the hardness of traditional subgroup discrete logarithm problem in prime order q subgroup of $GF(p^6)$, where $q|p^2 - p + 1$ and $p^6 \approx 2^{1024}$. XTR uses trace representation over $GF(p^2)$ which is about $1/3$ of the traditional representation. Such a compact representation allows efficient computations and bandwidth saving. Strong evidence that $1/3$ is the best compression ratio achievable is presented in [12].

XTR is believed to be as fast as elliptic curve cryptosystems (ECC) [35, 44, 58] and significantly faster than RSA [71]. ECC is based on hard mathematics and its parameter selection is not simple. RSA is the most popular cryptosystem and it is easily understood, but it is not so efficient both in terms of bandwidth and computational requirements. XTR does not suffer from such disadvantages found in ECC and RSA, but it uses twice longer public key

size than ECC (with point compression). However, for a properly chosen private key, there is a method to reduce the public key size for XTR [51]. Hence, XTR is considered to be a good compromise between ECC and RSA at a similar security level.

Since the work of Kocher et al. [45, 46], side channel attacks have become the most devastating attack on many implementations of cryptosystems. No matter how hard the underlying mathematical problem is, cryptosystems often succumb to side channel attacks if they are not implemented properly. There have been many articles on the side channel attacks on various cryptosystems on various hardware [45, 46, 56, 57]. There also have been many countermeasures to side channel attacks [18, 33, 40, 68, 81, 37]. We present a side channel attack on XTR cryptosystem.

1.2 Thesis Outline

This thesis is organized as follows. In Chapter 2, we review well-known multiplication and modular reduction algorithms. Then we review some background materials on XTR cryptosystems in Chapter 3. The main research contributions of this thesis are presented in Chapters 4, 5, 6, and 7.

In Chapter 4, we introduce a new class of moduli called the low-weight polynomial form integers. We present an efficient modular reduction scheme using LWPFI moduli. We show analysis results of our modular multiplication scheme and present experimental results. We also present ideas to enhance modular multiplication algorithm based on LWPFIs and discuss practical issues.

In Chapter 5, we present an improved coefficient reduction algorithm for use in modular multiplication using LWPFI moduli. In Chapter 4, our coefficient reduction algorithm is based on a division algorithm derived from the Barrett reduction algorithm. The new coefficient reduction algorithm in Chapter 5 is based on the Montgomery reduction algorithm. Since the Montgomery reduction algorithm usually performs better than any other modular reduction algorithms for general moduli, our new coefficient reduction algorithm is likely to be better than the previous one.

In Chapter 6, we present new 3, 4, 5-way squaring algorithms based on the Toom-Cook multiplication algorithm. We present how our squaring algorithms have been derived by explicitly showing details on our approach. We show experimental results of our 3-way squaring formulae.

In Chapter 7, we present a side channel attack on XTR cryptosystems. First, we show how one can identify individual XTR operations under a simple assumption that multiplication and modular reduction can be distinguished by looking at the power trace. The statistical behavior of XTR double exponentiation algorithm is analyzed. Then we show analysis and experimental results of our attack.

Conclusions and future research directions are given in Chapter 8.

1.3 Research Contributions

The main contributions of this thesis are listed below:

- Low-weight polynomial form integers (LWPFIs) are introduced and efficient modular multiplication algorithms based on LWPFI moduli are developed.

- Modular multiplication using LWPFI moduli are improved by using the Montgomery reduction algorithm for coefficient reduction.

- The first 3, 4, 5-way asymmetric squaring algorithms are developed. The new squaring algorithms can be used to improve the polynomial squaring used in modular multiplication algorithm using LWPFI moduli.

- Attempted a side channel attack on XTR cryptosystems.

Chapter 2

Review of Multiplication and Modular Reduction Algorithms

2.1 Review of Multiplication Algorithms

Multiplication is one of the most important basic arithmetic operations in popular public key cryptosystems. In this section, we briefly review some well-known multiplication algorithms. Since cryptographic computations must be exact and efficient, we focus only on the algorithms that compute such results using only integer arithmetic. Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ be in $\mathbb{Z}[x]$. The product of $A(x)$ and $B(x)$ is computed as follows:

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i = A(x) \cdot B(x), \tag{2.1}$$

where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$ for $0 \leq i \leq 2n-2$ and $a_j = 0$ and $b_j = 0$ for $j \geq n$ and $j < 0$. Let $L(\cdot)$ denote the set of all integral combinations of the coefficients of a polynomial. We call a computation of form "$a \cdot b$", where $a \in L(A)$ and $b \in L(B)$, a *coefficient multiplication*. The performance of multiplication algorithms are often analyzed in terms of the number of coefficient multiplications required to compute (2.1). The rest of the computational cost including the cost for computing the linear combinations $a \in L(A)$ and $b \in L(B)$ necessary to compute (2.1) is referred to as *overhead*. The multiplication $a \cdot b$ can be slower than computing $a_i \cdot b_j$, due to the carries occurring when computing the linear combinations a

and b. We count the cost difference of two computations ($a \cdot b$ and $a_i \cdot b_j$) toward the overhead.

In order to compute (2.1) using paper and pencil, n^2 coefficient multiplications are required. Such a method is called the schoolbook multiplication method. When $A(x) = B(x)$, only $n(n+1)/2$ coefficient multiplications are required, since off-diagonal products (i.e., $a_i b_j$ where $i \neq j$) always occur twice and need to be computed only once. We call this squaring method the schoolbook squaring method.

The first multiplication algorithm that has sub-quadratic complexity was developed by Karatsuba in 1963. The Karatsuba algorithm (KA) performs the multiplication of two 2-term polynomials using only three coefficient multiplications as follows [41]:

$$C(x) = a_1 b_1 x^2 + ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1)x + a_0 b_0. \tag{2.2}$$

The time complexity of $O(n^{\log_2 3})$ can be achieved by recursively applying (2.2). The KA is asymptotically better than the schoolbook method since $\log_2 3 \approx 1.58 < 2$. However, in real world applications, KA is faster than the schoolbook method only when n is sufficiently large, due to the fact that a larger amount of overhead is required in the KA than in the schoolbook method. The crossover between KA and the schoolbook method is highly dependent on the machine characteristic, programmer's skill, programming language and compilers, etc. In practice, for integer multiplications, KA is faster than the schoolbook method if the operand is about 500–1000 bits long.

There is a well-known 3-term multiplication method which is shown below [4].

$$\begin{aligned} (a_2 x^2 + a_1 x + a_0)&(b_2 x^2 + b_1 x + b_0) \\ &= D_2 x^4 + (D_5 - D_2 - D_1)x^3 \\ &+ (D_4 - D_0 - D_2 + D_1)x^2 \\ &+ (D_3 - D_0 - D_1)x + D_0, \end{aligned} \tag{2.3}$$

where

$$\begin{aligned} D_0 &= a_0 b_0, & D_3 &= (a_0 - a_1)(b_0 - b_1), \\ D_1 &= a_1 b_1, & D_4 &= (a_0 - a_2)(b_0 - b_2), \\ D_2 &= a_2 b_2, & D_5 &= (a_1 - a_2)(b_1 - b_2). \end{aligned}$$

This formula requires 6 coefficient multiplications. In [61], Montgomery shows a family of 3-way multiplication algorithms requiring 6 coefficient multiplications. The method in (2.3)

Algorithm 2.1 Toom-Cook Multiplication Algorithm

Require: Degree $n-1$ polynomials $A(x)$ and $B(x)$.
Ensure: $C(x) = A(x) \cdot B(x)$.
1: (Evaluation) $u_i = A(x_i)$ and $v_i = B(x_i)$ for $i = 1, \ldots, 2n-1$, where x_i's are all distinct.
2: (Point-Wise Multiplication) $C(x_i) = u_i v_i$ for $i = 1, \ldots 2n-1$.
3: (Interpolation) given $C(x_i)$'s, uniquely determine c_j's for $j = 0, \ldots, 2n-2$, where $C(x) = \sum_{j=0}^{2n-2} c_j x^j$.

is a special case. Recursive use of (2.3) results in $O(n^{\log_3 6})$ time complexity. This method is less efficient than KA in an asymptotic sense, since $\log_3 6 \approx 1.63 < \log_2 3$. However, (2.3) is efficient when the input size is small and the input can be equally separated into three parts. We call (2.3) as 3-way KA-like formula.

In 1963, Toom developed an elegant idea to perform multiplication of two degree-$(n-1)$ polynomials using only $(2n-1)$ coefficient multiplications [79]. He showed that it is possible to construct a multiplication scheme that has $O(nc^{\sqrt{\log n}})$ operations and $O(c^{\sqrt{\log n}})$ delay. In 1966, Cook improved Toom's idea [17]. The multiplication method they developed is now called the Toom-Cook algorithm. The latter is based on a well-known result from linear algebra: any degree-n polynomial can be uniquely determined by its evaluation at $(n+1)$ distinct points. Algorithm 2.1 shows a general idea how the Toom-Cook multiplication algorithm works.

Interestingly, many fast multiplication algorithms having sub-quadratic complexity are related to the Toom-Cook multiplication algorithm. In particular, KA can be considered as a special case of the Toom-Cook multiplication algorithm for the evaluation points $\{0, 1, \infty\}$, where evaluation at ∞ means computing $\lim_{x\to\infty} A(x)/x^{n-1}$ [87]. The Winograd algorithm [87] is very similar to Algorithm 2.1. The difference is that the Winograd algorithm considers not only integers, but also imaginary numbers for evaluation points. Multiplication methods based on number theoretic transform (NTT) can be viewed also as special cases of the Toom-Cook algorithm. NTT based multiplication algorithms [74] use $x_i = \gamma^i \bmod p$ for $1 \le i \le N$, where γ is a primitive N-th root of unity modulo some prime $p \ge N$, where p is greater than or equal to the largest coefficient of the resulting polynomial, $N \ge 2n-1$ and $N | (p-1)$. In this case, some changes are required in Algorithm 2.1. Steps 1 and 2 must run through $i = 1$ to N, which may be greater than $2n-1$. Moreover, the computations must be performed in $\mathbb{Z}_p$. NTT based algorithms are asymptotically faster, since steps 1 and 3 can enjoy fast al-

gorithms that requires $O(N \log N)$ operations in $\mathbb{Z}_p$ by choosing N having only small prime factors, or ideally a power of 2.

There are other efficient multiplication algorithms that cannot be derived from Algorithm 2.1. The 3-way KA-like formula shown in (2.3) and Montgomery's Karatsuba-like formulae [61] do not appear to be a special case of the Toom-Cook algorithm. Montgomery's formulae use 13, 17 and 22 coefficient multiplications for 5, 6 and 7-way polynomial multiplications, respectively.

For more comprehensive survey on multiplication algorithm, we refer the readers to Daniel Bernstein's paper [8].

2.1.1 Zimmermann's 3-Term Toom-Cook Multiplication

This method has been developed by Zimmermann and implemented in GMP library as subroutines of `mpz_mul()`. Zimmermann uses $\{0, 1, -1, 2, \infty\}$ for the set of evaluation points.

Let $A(x) = a_2x^2+a_1x+a_0$, $B(x) = b_2x^2+b_1x+b_0$ and $C(x) = A(x)B(x) = c_4x^4+c_3x^3+c_2x^2+c_1x+c_0$. Evaluation of $A(x)$ and $B(x)$ at $x_i \in \{0, 1, -1, 2, \infty\}$ and point-wise multiplication of $A(x_i)$'s and $B(x_i)$'s result in the following system of equations:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} = \begin{bmatrix} a_0b_0 \\ (a_2+a_1+a_0)(b_2+b_1+b_0) \\ (4a_2+2a_1+a_0)(4b_2+2b_1+b_0) \\ (a_2-a_1+a_0)(b_2-b_1+b_0) \\ a_2b_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}. \tag{2.4}$$

Then the above linear system can be solved very efficiently using row operations as shown in Algorithm 2.2. The latter requires 8 additions/subtractions, 4 shifts, 1 division by 3. To the best of our knowledge this is by far the best method for performing the 3-term Toom-Cook multiplication.

2.2 Review of Modular Reduction Algorithms

Many algorithms have been proposed for implementing efficient modular multiplication. These algorithms can be classified into the following three categories:

Algorithm 2.2 Zimmermann's 3-Way Interpolation

Require: $(S_1, S_2, S_3, S_4, S_5)$ as in (2.4).
Ensure: $C(x) = A(x) \cdot B(x)$.
1: $T_1 \leftarrow 2S_4 + S_3$. $(= 18c_4 + 6c_3 + 6c_2 + 3c_0)$
2: $T_1 \leftarrow T_1/3$. $(= 6c_4 + 2c_3 + 2c_2 + c_0)$
3: $T_1 \leftarrow S_1 + T_1$. $(= 6c_4 + 2c_3 + 2c_2 + 2c_0)$
4: $T_1 \leftarrow T_1/2$. $(= 3c_4 + c_3 + c_2 + c_0)$
5: $T_1 \leftarrow T_1 - 2S_5$. $(= c_4 + c_3 + c_2 + c_0)$
6: $T_2 \leftarrow (S_2 + S_4)/2$. $(= c_4 + c_2 + c_0)$
7: $S_2 \leftarrow S_2 - T_1$. $(= c_1)$
8: $S_3 \leftarrow T_2 - S_1 - S_5$. $(= c_2)$
9: $S_4 \leftarrow T_1 - T_2$. $(= c_3)$
10: **return** $C(x) = S_5x^4 + S_4x^3 + S_32x^2 + S_2x + S_1$.

1. Algorithms for general moduli: the classical algorithm [43], the Barrett algorithm [7] and the Montgomery algorithm [60].

2. Algorithms for special moduli: modular reduction methods based on pseudo-Mersenne numbers [19] and generalized Mersenne numbers [76].

3. Look-up table methods: Kawamura, Takabayashi and Shimbo's method [42]; Hong, Oh and Yoon's method [34]; and Lim, Hwang and Lee's method [54].

Look-up table methods are normally faster than the generalized ones, but require a large amount of memory. The Barrett algorithm and the Montgomery algorithm requires small amount of pre-computation. The algorithms using pre-computation are only suitable when some parameters are fixed. In this section, we briefly review modular reduction algorithms for general moduli: the classical division algorithm, and the Montgomery algorithm. Then we present generalization of the Barrett algorithm for modular reduction. Unlike the original Barrett algorithm, the generalized one does not have a limitation on the input size and can perform a multiple precision division for a fixed divisor. In describing the above mentioned algorithms, we use the following notations:

- $b \geq 2$ is a radix for integer representation. In software implementation, $b = 2^w$, where w is the word length in bits of the processor used.

- $(x_{n-1} \cdots x_1 x_0)_b = x_{n-1}b^{n-1} + \cdots + x_1 b + x_0$. In general, x_i's can be signed digits in this notation. However, in this section, we use only radix-b representation with unsigned digits, i.e., $0 \le x_i < b$ for $0 \le i < n$.

2.2.1 Classical Division Algorithm

One straightforward method to perform modular reduction is to use the classical division algorithm, which gives both remainder and quotient as output. A good description and analysis of the classical algorithm for integer division (CAID) can be found in [55]; we have slightly modified this algorithm so that it accepts only normalized input, i.e., the most significant digit m_{k-1} of the divisor satisfies $m_{k-1} \ge \lfloor b/2 \rfloor$. The resulting pseudo code is given in Algorithm 2.3.

The input condition $m_{k-1} \ge \lfloor b/2 \rfloor$ guarantees that line 16 is repeated at most twice [43]. This condition can be met by left shifting x and m by a suitable number of bits. To obtain a correct result, we only need to shift the remainder r to the right by the same number of bits. After the while loop in lines 15-17, q_{i-k} is at most one larger than the true value of quotient digit. The probability of $r < 0$ at line 19 is approximately $2/b$. Note that the values $q_{i-k}m_{k-1}$ and $q_{i-k}m_{k-2}$ in line 15 can be reused in line 18. Hence Algorithm 2.3 requires $k(n-k)$ single-precision multiplications and at most $(n-k)$ single-precision divisions.

2.2.2 Montgomery Reduction Algorithm

The Montgomery algorithm performs modular reduction without using any division instruction of the underlying processor [60]. Let m be a modulus, and T be a positive integer which is to be reduced. We choose an integer R such that $R > m$, $\gcd(m, R) = 1$ and $0 \le T < mR$.

Algorithm 2.4 computes $T \cdot b^{-q} \bmod m$, given an integer $0 \le T < mR$, where $R = b^q$. In each iteration of Algorithm 2.4, a multiple of the modulus M is added to T_i such that the least significant digit becomes zero. Then, the division of T_{i+1} by b can be performed simply by shifting all digits of T by one place to the right. If q is chosen to be the digit length of T, then it can be easily shown that $T_q \in [0, 2m)$. Therefore, one final subtraction by m may be required to output an integer within $[0, m)$. Some researchers have proposed ways to eliminate this final subtraction to avoid timing attacks [45, 72, 85]. Walter proposed using q such that $2m < b^{q-1}$ [83]. Hachez and Quisquater improved this condition to $m < b^{q-1}$ for $b = 2$ [29]. Walter improved this condition again to $4m < b^q$ [84]. Line 3 of MAIR requires

Algorithm 2.3 Classical Algorithm for Integer Division (CAID)

Require: Integers $x = (x_{n-1} \cdots x_1 x_0)_b$ and $m = (m_{k-1} \cdots m_1 m_0)_b$ with $n \geq k \geq 1$ and $m_{k-1} \geq \lfloor b/2 \rfloor$.

Ensure: The quotient $q = (q_{n-k} \cdots q_1 q_0)_b$ and the remainder $r = (r_{k-1} \cdots r_1 r_0)_b$ such that $x = qm + r$, $0 \leq r < m$.

1: **for** j from 0 to $(n-k)$ **do**
2: $q_j \leftarrow 0$.
3: **end for**
4: **if** $x > mb^{n-k}$ **then**
5: $q_{n-k} \leftarrow q_{n-k} + 1$, $r \leftarrow x - mb^{n-k}$;
6: **else**
7: $r \leftarrow x$.
8: **end if**
9: **for** i from $n-1$ down to k **do**
10: **if** $r_i = m_{k-1}$ **then**
11: $q_{i-k} \leftarrow b - 1$;
12: **else**
13: $q_{i-k} \leftarrow \lfloor (r_i b + r_{i-1})/m_{k-1} \rfloor$.
14: **end if**
15: **while** $q_{i-k} m_{k-2} > (r_i b + r_{i-1} - q_{i-k} m_{k-1}) b + r_{i-2}$ **do**
16: $q_{i-k} \leftarrow q_{i-k} - 1$.
17: **end while**
18: $r \leftarrow r - q_{i-k} m b^{i-k}$.
19: **if** $r < 0$ **then**
20: $r \leftarrow r + mb^{i-k}$ and $q_{i-k} \leftarrow q_{i-k} - 1$.
21: **end if**
22: **end for**
23: **return** q and r.

Algorithm 2.4 Montgomery Algorithm for Integers Reduction (MAIR)

Require: integers T and $m = (m_{k-1} \cdots m_1 m_0)_b$, such that $R = b^q$, $0 \leq T < mR$ and $\gcd(b, m) = 1$.

Ensure: $T \cdot b^{-q} \bmod m$.

1: $T_0 \leftarrow T$.
2: **for** i from 0 to $q-1$ **do**
3: $u_i \leftarrow -m^{-1} \cdot T_i \bmod b$.
4: $T_{i+1} \leftarrow (T_i + u_i \cdot m)/b$.
5: **end for**
6: **if** $T_q \geq m$ **then**
7: $T_q \leftarrow T_q - m$.
8: **end if**
9: **return** T_q.

one single-precision multiplication and line 4 requires k single-precision multiplications, where k is the digit length of m. Therefore, MAIR requires a total of $q(k+1)$ single-precision multiplications.

2.2.3 Barrett's Reduction Algorithm

The Barrett algorithm [7, 21] is advantageous for applications in which a fixed modulus is used. It does not use any division instructions of the underlying processor, but it uses a small amount of pre-computation of size similar to that of the modulus. The description given in Algorithm 2.5 is a generalized version of the Barrett algorithm. We refer to it as GBAID (the generalized Barrett algorithm for integer division) since it has been modified such that a quotient is also computed. The original Barrett algorithm can reduce integers that are at most twice as long as a modulus. However, GBAID does not have such a limitation. Note that, Algorithm 2.5 becomes the original Barrett algorithm for integer reduction if we let $u = 2v$, remove "$q \leftarrow q+1$" in line 7, and change line 9 to "**return** r".

The computation of q in line 1 of Algorithm 2.5 is not exact, but it is quite accurate. Let $q' = \left\lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \right\rfloor$. Then q computed in line 1 is an approximation of q'. The approximation error $q' - q$ is at most 1 when $u - v \leq b$ [55, 63].

Proposition 1. *Computation of q in line 1 of Algorithm 2.5 is not exact. The error in q is at most 1, if $b \geq u - v$.*

Algorithm 2.5 The Generalized Barrett Algorithm for Integer Division (GBAID)

Require: Positive integers $x = (x_{u-1} \cdots x_1 x_0)_b$, $m = (m_{v-1} \cdots m_1 m_0)_b$ with $m_{v-1} \neq 0$, $u \geq v$ and a pre-computed value $\mu = (\mu_{u-v} \cdots \mu_1 \mu_0)_b = \lfloor b^u/m \rfloor$.

Ensure: Integers q and r such that $x = qm + r$, where $r < m$.

1: $q \leftarrow \left\lfloor \sum_{u-v-1 \leq i+j} x_{i+v-1} \mu_j b^{i+j-u+v+1} / b^2 \right\rfloor \left(\approx \left\lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \right\rfloor\right)$.

2: $r_1 \leftarrow x \bmod b^{v+1}$, $r_2 \leftarrow \sum_{i+j<v+1} q_i m_j b^{i+j} \bmod b^{v+1} (= q \cdot m \bmod b^{v+1})$, $r \leftarrow r_1 - r_2$.

3: **if** $r < 0$ **then**

4: $\quad r \leftarrow r + b^{v+1}$.

5: **end if**

6: **while** $r \geq m$ **do**

7: $\quad r \leftarrow r - m$ and $q \leftarrow q + 1$.

8: **end while**

9: **return** q and r.

Proof. For simplicity, we let $k = u - v$, $\gamma = (\gamma_k \cdots \gamma_1 \gamma_0)_b = (x_{u-1} \cdots x_v x_{v-1})_b = \lfloor x/b^{v-1} \rfloor$. Note that μ is also at most $k+1 = u-v+1$ words long, since $\mu = \lfloor b^u/m \rfloor$ where m is v words long. Let $q' = \lfloor \gamma\mu/b^{u-v+1} \rfloor$ be the value computed by the following full multiplication:

$$q' = \left\lfloor \frac{\gamma \cdot \mu}{b^{k+1}} \right\rfloor = \left\lfloor \frac{\sum_{i+j} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor.$$

The computation of q in line 1 of Algorithm 2.5 is done by the following partial multiplication:

$$q = \left\lfloor \frac{\sum_{k-1 \leq i+j} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor.$$

Observe that q' and q have a common part:

$$
\begin{aligned}
q' &= \left\lfloor \frac{\sum_{i+j\leq k}\gamma_i\mu_j b^{i+j} + \sum_{k+1\leq i+j}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor \\
&= \sum_{k+1\leq i+j}\gamma_i\mu_j b^{i+j-k-1} + \left\lfloor \frac{\sum_{i+j\leq k}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor, \\
q &= \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1} + \sum_{k+1\leq i+j}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} \right\rfloor \\
&= \sum_{k+1\leq i+j}\gamma_i\mu_j b^{i+j-k-1} + \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} \right\rfloor.
\end{aligned}
$$

Let ϵ be the difference between q' and q, i.e., $\epsilon = q' - q$.

$$
\begin{aligned}
\epsilon &= \left\lfloor \frac{\sum_{i+j\leq k}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor - \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} \right\rfloor \\
&= \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j} + \sum_{i+j\leq k-2}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor - \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} \right\rfloor \\
&= \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} + \frac{\sum_{i+j\leq k-2}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor - \left\lfloor \frac{\sum_{k-1\leq i+j\leq k}\gamma_i\mu_j b^{i+j-k+1}}{b^{2}} \right\rfloor \\
&\leq \left\lfloor \frac{\sum_{i+j\leq k-2}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor + 1. \\
&\quad (\because \lfloor A+B \rfloor \leq \lfloor A \rfloor + \lfloor B \rfloor + 1.)
\end{aligned}
$$

Since $\gamma_i,\ \mu_j < b$,

$$
\left\lfloor \frac{\sum_{i+j\leq k-2}\gamma_i\mu_j b^{i+j}}{b^{k+1}} \right\rfloor < \left\lfloor \frac{\sum_{i+j\leq k-2} b^{i+j}}{b^{k-1}} \right\rfloor.
$$

Then we can see that for $b \geq k$,

$$
\sum_{i+j\leq k-2} b^{i+j} = (k-1)b^{k-2} + (k-2)b^{k-3} + \cdots + 2b + 1 < b^{k-1}.
$$

Therefore,

$$\left\lfloor \frac{\sum_{i+j \le k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor = 0,$$

and

$$\epsilon \le 1.$$

□

Let Q denote $\lfloor x/m \rfloor$. Then it can be easily shown that $Q - 2 \le q' = \lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \rfloor \le Q$.

$$\begin{aligned} q' &> \frac{1}{b^{u-v+1}} \cdot \left(\frac{b^u}{m} - 1 \right) \cdot \left(\frac{x}{b^{v-1}} - 1 \right) - 1 \\ &= \frac{x}{m} - \frac{b^{v-1}}{m} - \frac{x}{b^u} + \frac{1}{b^{u-v+1}} - 1 \\ &> Q - 3. \end{aligned}$$

Trivially, $q' \le Q$.

Let $k = u - v$ for simplicity of description; then both μ and $\lfloor x/b^{v-1} \rfloor$ are at most $(k+1)$ words long. It can be easily seen that line 1 requires at most $(k^2 + 5k + 2)/2$ single-precision multiplications. Note that q computed in step 1 is also at most $(k+1)$ words long. One can easily verify that the number of single-precision multiplications required in line 2 is at most $(k+1)v - k(k-1)/2$ if $k \le v$, and $(v^2 + 3v)/2$ otherwise; therefore, the total number of single-precision multiplications required in Algorithm 2.5 is $uv + 3u - v^2 - 2v + 1$ if $u \le 2v$, or $(u^2 + 5u)/2 - uv + v^2 - v + 1$ if $u > 2v$.

Chapter 3

Review of XTR Cryptosystems

3.1 Mathematical Preliminaries on XTR Cryptosystems

3.1.1 Basic Ideas

Due to the index calculus method, traditional cryptosystems based on the hardness of discrete logarithm problem must use large representation size, which is usually at least 1024 bits, even though they are based on a subfield of order only about 2^{160}. Such a long representation size renders traditional cryptosystems disadvantageous both in bandwidth and computational efficiency.

The prime subgroup of size q must be chosen carefully in $GF(p^t)$ case. If q is chosen carelessly, the DLP could be easily broken even though it is a prime number large enough to resist square-root type attacks. If q divides $p^s - 1$ where $s < t$ and $s|t$, the index calculus method can be applied in some proper subgroup of $GF(p^t)$. The running time of the index calculus method depends on the size of group representation. Therefore, if s is small enough, the DLP can be easily broken. Hence, Lenstra proposed the method to select q which does not divide the group order of proper subgroup of $GF(p^t)^*$ [48].

Theorem 1 ([48])**.** *Let $\Phi_t(x)$ be a t-th cyclotomic polynomial. Suppose that $q > t$ be a prime factor of $\Phi_t(p)$. Then q does not divide $\Phi_d(p)$ for all d such that $d < t$ and $d|t$.*

This idea of choosing q is also used in XTR cryptosystem. In XTR cryptosystem, t is fixed to 6 and q is chosen such that $q|\Phi_6(p) = p^2 - p + 1$ and $q > 6$. Therefore, no subgroup of order q can be embedded in any of $GF(p)^*$, $GF(p^2)^*$ and $GF(p^3)^*$.

Definition 1 (XTR-supergroup and XTR-subgroup [52]). *Suppose* $p \equiv 2 \pmod 3$ *is an odd prime number and there exists a prime* $q > 6$ *that divides* $p^2 - p + 1$.

- ***XTR-supergroup***: *order* $p^2 - p + 1$ *subgroup of* $GF(p^6)^*$.
- ***XTR-(sub)group***: *order* $q > 6$ *subgroup of XTR-supergroup.*

Theorem 2. *XTR-subgroup is not contained in any of* $GF(p)^*$, $GF(p^2)^*$ *and* $GF(p^3)^*$.

Proof. Due to Theorem 1, the order q of XTR-subgroup does not divide any of $\Phi_1(p) = p - 1$, $\Phi_2(p) = p + 1$ and $\Phi_3(p) = p^2 + p + 1$. Therefore, XTR-subgroup $\not\subseteq GF(p)^*$, $GF(p^2)^*$, $GF(p^3)^*$. □

It follows that, to solve the DLP in XTR-group with index calculus method, we have to apply it to the full group $GF(p^6)^*$. The Pohlig-Hellman attack does not work if the group order q is a large enough prime number [70]. In fact, it is shown that the DLP in XTR-subgroup is polynomial time equivalent to that in $GF(p^6)^*$ [53]. Moreover, it is proven that the DLP in XTR-subgroup is at least harder than the DLP in supersingular elliptic curves of the same order [80].

The use of the XTR-supergroup allows a very compact representation of its elements using elements of $GF(p^2)$. This allows us to represent XTR-supergroup elements with only one third of the bits that are required in traditional representation. Moreover, computations take place in $GF(p^2)$ instead of the $GF(p^6)$. This results in much less storage and bandwidth requirement as well as much faster performance than was previously possible with traditional representations.

For example, a prime p must be chosen such that p^6 is large enough to resist the index calculus method. Hence, $p^6 \approx 2^{1024}$ or $p \approx 2^{170}$ to achieve the security level equivalent to 1024-bit RSA and approximately equivalent to 170-bit ECC [77]. Since XTR operations are done in $GF(p^2)$, arithmetic operations deal with numbers of only $2 \times 170 = 340$ bits long. The group size q only needs to be large enough to withstand square-root type attacks on DLP, i.e., $q \approx 2^{160}$. The details on how elements in XTR-supergroup (thereby XTR-subgroup also) are represented with the elements of $GF(p^2)$ are explained in the following.

3.1.2 Efficient Arithmetic Operations in $GF(p^2)$

Before we review the trace representations used in XTR cryptosystems, we analyze the cost required for arithmetic operations in $GF(p^2)$. Let $p > 3$ be a prime number. If $p \bmod 3 = 2$, then 2 is primitive in $\mathbb{Z}_3$. Therefore, if $p \equiv 2 \pmod 3$, it follows that $(x^3-1)/(x-1) = x^2+x+1$ is an irreducible polynomial over $GF(p)$ and its roots α and α^p $(= \alpha^2)$ form an optimal normal basis for $GF(p^2)$ over $GF(p)$.

$$GF(p^2) \cong \{x_1\alpha + x_2\alpha^2 : \alpha^2 + \alpha + 1 = 0 \text{ and } x_1,\, x_2 \in GF(p)\}.$$

We now discuss the computational costs for some arithmetic operations that are used in XTR cryptosystems. The following are the methods for performing arithmetic operations in $GF(p^2)$ required in XTR cryptosystems [77].

1. x^p: free!
   $x^p = (x_1\alpha + x_2\alpha^2)^p = (x_2\alpha + x_1\alpha^2)$.

2. x^2: two multiplications in $GF(p)$

$$\begin{aligned} x^2 &= (x_1\alpha + x_2\alpha^2)^2 \\ &= (x_1^2\alpha^2 + 2x_1x_2\alpha^3 + x_2^2\alpha^4) \\ &= (x_1^2\alpha^2 + 2x_1x_2(-\alpha - \alpha^2) + x_2^2\alpha) \\ &= (x_2^2 - 2x_1x_2)\alpha + (x_1^2 - 2x_1x_2)\alpha^2 \\ &= x_2(x_2 - 2x_1)\alpha + x_1(x_1 - 2x_2)\alpha^2. \end{aligned}$$

3. $x \cdot y$: three multiplications in $GF(p)$
   By using the KA,

$$A = x_1y_1,\ B = x_2y_2,\ C = (x_1 + x_2)(y_1 + y_2) - A - B,$$

$$x \cdot y = (B - C)\alpha + (A - C)\alpha.$$

4. $x \cdot z - y \cdot z^p$: four multiplications in $GF(p)$

$$x \cdot z - y \cdot z^p = (z_1(y_1 - x_2 - y_2) + z_2(x_2 - x_1 + y_2))\alpha + (z_1(x_1 - x_2 + y_1) + z_2(y_2 - x_1 - y_1))\alpha^2.$$

Note that one multiplication in $GF(p)$ is composed of 1 multiplication in $\mathbb{Z}$ and one reduction modulo p. However, as noted in [77], we can do better by separating the "multiplication in $\mathbb{Z}$" step and "reduction modulo p" step in computing the multiplication in $GF(p)$. Speeding up the arithmetic operations in $GF(p)$ is achieved by delaying the reduction steps. In addition, the use of Montgomery algorithm [60] is also proposed in [77]. On the assumption that a multiplication and a Montgomery reduction have similar computational costs, the following results can be obtained [77]. Note that for the typical size of operands used in XTR cryptosystems, the classical multiplication algorithm is the most efficient one. The timing difference between the classical multiplication and the Montgomery reduction is quite small for operands of small sizes.

Lemma 1. *Let x, y, $z \in GF(p^2)$ with $p \equiv 2 \pmod 3$. Let us define,*

- T_1 : *time required to compute multiplication in $\mathbb{Z}$*
- T_2 : *time required to compute reduction using Montgomery's algorithm*
- T_3 : *time required to compute the multiplication in $GF(p)$ using Montgomery's algorithm*

Then $T_3 = T_1 + T_2$ where $T_1 \approx T_2$.

1. x^p : *free*
2. x^2 : *2 multiplications in $GF(p)$ $(2T_1 + 2 \cdot T_2 = 2T_3)$*
3. $x \cdot y$: *2.5 multiplications in $GF(p)$ $(3T_1 + 2 \cdot T_2 \approx 2.5T_3)$*
4. $x \cdot z - y \cdot z^p$: *3 multiplications in $GF(p)$ $(4T_1 + 2T_2 \approx 3T_3)$*

3.1.3 Trace Representations and Properties of Sequence c_n

Definition 2 (Conjugate and Trace over $GF(p^2)$)**.** *Let $h \in GF(p^6)$. Then* conjugates *of h over $GF(p^2)$ are h, h^{p^2} and h^{p^4}. The* trace *over $GF(p^2)$, denoted by $Tr(\cdot)$, of h is the sum of conjugates of h over $GF(p^2)$, i.e., $Tr(h) = h + h^{p^2} + h^{p^4} \in GF(p^2)$.*

Let g be an element in XTR-supergroup. The heart of XTR cryptosystem, which allows efficient and compact representation, is that g and its conjugates g^{p^2} and g^{p^4} are completely characterized by a single element $Tr(g) \in GF(p^2)$. Let us observe that

Theorem 3. *Let $g \in$ XTR-supergroup. The conjugates of g are completely determined by its trace, $Tr(g)$.*

Proof. Clearly, $\mathrm{ord}(g)|(p^2 - p + 1)$. The roots of $X^3 + Tr(g)X^2 + Tr(g)^p X - 1 = 0$ are the conjugates of g, since

$$\begin{aligned}
&(X - g)(X - g^{p-1})(X - g^{-p}) \\
&\quad = X^3 - (g + g^{p-1} + g^{-p})X^2 + (gg^{p-1} + gg^{-p} + g^{p-1}g^{-p})X - gg^{p-1}g^{-p} \\
&\quad = X^3 - (g + g^{p-1} + g^{-p})X^2 + (g^p + g^{1-p} + g^{-1})X - 1 \\
&\quad = X^3 - Tr(g)X^2 + Tr(g)^p X - 1.
\end{aligned} \tag{3.1}$$

□

By Theorem 3, we have a compact representation of XTR-supergroup. We can represent any element of XTR-supergroup with its trace over $GF(p^2)$ which requires only one third of bits that are originally required. However, since $Tr(\cdot)$ is not an injection, we lose the distinction among g and its conjugates.

Similar discussion can be made for $Tr(g^n)$: $Tr(g^n)$ fully characterizes the conjugates of g^n. Therefore, $Tr(g^n)$ is a compact representation of g^n (and its conjugates). This is proven in Lemma 3.*2*. In traditional DLP based cryptosystems, exponentiation operation, i.e., computing g^n from g, is well-studied and there are many efficient algorithms available. However, with the compact representation using traces, computing $Tr(g^n)$ from $Tr(g)$ (without using representation in $GF(p^6)$) is not straightforward, since $Tr(g^n) \neq Tr(g)^n$ in general. In addition, the fact that it is not straightforward to compute $Tr(g^{a+b})$ from $Tr(g^a)$ and $Tr(g^b)$, makes things more complicated. We discuss this issue in Section 3.2.

Definition 3 ($F(c,X)$ and c_n)**.** *$F(c,X) = X^3 - cX^2 + c^pX - 1 \in GF(p^2)[X]$, where $c \in GF(p^2)$. Let us denote the three roots of $F(c,X)$ as h_0, h_1, $h_2 \in GF(p^6)$. Then c_n is defined as $c_n = h_0^n + h_1^n + h_2^n$.*

Let c_n denote $Tr(g^n)$. The notation c_n makes more sense than $Tr(g^n)$, since, in XTR cryptosystems, the explicit value of g or g^n, which are in $GF(p^6)$, are not used at all.

Lemma 2 ([53])**.** *Let $c \in GF(p^2)$ and let h_1, h_2 and h_3 be the three roots of $F(c,X)$.*

1. $c = c_1$.
2. $h_0 \cdot h_1 \cdot h_2 = 1$.
3. $c_{-n} = h_0^n \cdot h_1^n + h_0^n \cdot h_2^n + h_1^n \cdot h_2^n$, $\forall n \in \mathbb{Z}$.
4. *If $h \in GF(p^6)$ is a root of $F(c,X)$ then h^{-p} is also a root.*
5. $c_{-n} = c_{np} = c_n^p$ *for* $n \in \mathbb{Z}$.
6. *Either ord$(h_j)|(p^2 - p + 1)$ and ord$(h_j) > 3$ for $j = 0, 1, 2$ or $h_j \in GF(p^2)$ for $j = 0, 1, 2$.*
7. $c_n \in GF(p^2)$ *for* $n \in \mathbb{Z}$.

Due to Lemma 2.6, we have the following theorem.

Theorem 4 ([53])**.** *$F(c,X) \in GF(p^2)[X]$ is irreducible if and only if its roots have order dividing $p^2 - p + 1$ and > 3.*

Proof. It is easy to see that if $F(c,X) \in GF(p^2)[X]$ is reducible, it must have at least one root in $GF(p^2)$. By Lemma 2.6, we see that all the roots must be in $GF(p^2)$.

The reverse direction is straightforward. □

Lemma 3 ([53])**.** *Let h_0, h_1 and h_2 be the roots of $F(c,X)$.*

1. $c_{u+v} = c_u \cdot c_v - c_v^p \cdot c_{u-v} + c_{u-2v}$ *for* $u, v \in \mathbb{Z}$.
2. $F(c_n, h_j^n) = 0$ *for* $j = 0, 1, 2$ *and* $n \in \mathbb{Z}$.
3. *$F(c,X)$ is reducible over $GF(p^2)$ if and only if $c_{p+1} \in GF(p)$.*

Now we have the following formulae that are useful in computing c_n from c.

Lemma 4. *Let c, c_{n-1}, c_n and c_{n+1} are available. Then we can compute the followings:*

1. $c_{2n} = c_n^2 - 2c_n^p$: *2 multiplications in $GF(p)$.*
2. $c_{3n} = c_n^3 - 3c_n^{p+1} + 3$: *4.5 multiplications in $GF(p)$.*
3. $c_{n+2} = c \cdot c_{n+1} - c^p \cdot c_n + c_{n-1}$: *3 multiplications in $GF(p)$.*
4. $c_{2n-1} = c_{n-1} \cdot c_n - c^p \cdot c_n^p + c_{n+1}^p$: *3 multiplications in $GF(p)$.*
5. $c_{2n+1} = c_{n+1} \cdot c_n - c \cdot c_n^p + c_{n-1}^p$: *3 multiplications in $GF(p)$.*
6. *If $\tilde{c}_1 = c_n$, then $\tilde{c}_v = c_{nv}$ (re-indexing).*

3.2 Review of XTR Exponentiation Algorithms

In this section, we review various exponentiation algorithms for XTR cryptosystem. The computation of c_n given $c = c_1$ is referred to as XTR single exponentiation. The computation of c_{au+bv} given c_u, c_v (u and v not necessarily known) and two exponents a and b is referred to as XTR double exponentiation. XTR Double exponentiations are required for signature verification in XTR versions [52] of well-known digital signature schemes based on Digital Signature Algorithm[66], ElGamal signature scheme [23] and Nyberg-Rueppel signature scheme [65]. All exponentiation algorithms reviewed in this section have been proposed in [53, 77].

3.2.1 XTR Single Exponentiation

Let $S_n(c) = (c_{n-1}, c_n, c_{n+1}) \in GF(p^2)^3$. Algorithm 3.1 presented below computes $S_{2v+1}(c)$ given c, $S_1(c) = (3, c, c^2 - 2c^p)$ and v. Note that the values y and e are not needed in actual implementations. In each iteration of the algorithm, $S_{2n-1}(c)$ and $S_{2n+1}(c)$ are computed from S_n depending on the exponent bit.

Lines 5 and 9 in Algorithm 3.1 require seven multiplications each. Hence, Algorithm 3.1 requires $7 \log_2 v$ multiplications in $GF(p)$. Note that the required computational cost and sequence of computation in each iteration does not depend on the exponent bits. This makes the timing attacks and simple power analysis attacks difficult.

Algorithm 3.1 XTR Single Exponentiation (helper function)

Require: $c \in GF(p^2)$, $S_1(c)$ and $v = \sum_{i=0}^{r-1} v_i 2^i \in \mathbb{Z}_{\geq 0}$, where $v_{r-1} = 1$.
Ensure: $S_{2v+1}(c)$.
1: $y \leftarrow 1$, $e \leftarrow 0$ $(y = 2e + 1)$.
2: **for** $i = r - 1$ down to 0 **do**
3: (Loop Invariant: $y = 2e + 1$)
4: **if** $v_i = 0$ **then**
5: $S_y(c) \leftarrow S_{2y-1}(c)$. (7 muls)
6: $y \leftarrow 2y - 1$, $e \leftarrow 2e$.
7: **end if**
8: **if** $v_i = 1$ **then**
9: $S_y(c) \leftarrow S_{2y+1}(c)$. (7 muls)
10: $y \leftarrow 2y + 1$, $e \leftarrow 2e + 1$.
11: **end if**
12: **end for**
13: **return** $S_y(c) = S_{2v+1}(c)$.

Note that Algorithm 3.1 does not compute $S_n(c)$ given an arbitrary n and $S_1(c)$, but it can only compute $S_{2n+1}(c)$. Algorithm 3.2 computes $S_n(c)$, given $S_1(c)$ and n, with the help of Algorithm 3.1.

Note that, in line 4 of Algorithm 3.2, $S_n(c)$ can be computed given $S_{n-1}(c)$ and c using three multiplications in $GF(p)$. Line 1 and 5 of Algorithm 3.2 each requires one execution of Algorithm 3.1 which takes $7 \log_2 v$ multiplications in $GF(p)$. Therefore, on average, Algorithm 3.2 requires $7 \log_2 v + 1.5$ multiplications in $GF(p)$.

3.2.2 XTR Double Exponentiation

In [77], Stam and Lenstra proposed an XTR double exponentiation algorithm based on Lucas sequence computation using the continued fraction method [59] and it is presented in Algorithm 3.3. Note that Algorithm 3.3 is a much improved version over the one presented in [53]. The algorithm computes c_{bk+al}, given c_k, c_l, c_{k-l}, c_{k-2l}, a and b, where k and l are not necessarily known. Initially, we let $u = k$, $v = l$, $d = b$ and $e = a$, so $ud + ve = bk + al$. Then we update u and v while decreasing d and e. At some point of algorithm $d = e = \gcd(u, v)$. Toward the end of algorithm, c_{u+v} and d such that $d(u + v) = bk + al$ are computed. Given c_{u+v} and d, we can compute $c_{d(u+v)}$ using Algorithm 3.2.

Algorithm 3.2 XTR Single Exponentiation I

Require: $c \in GF(p^2)$, $S_1(c)$ and $n = \sum_{i=0}^{r} n_i 2^i \in \mathbb{Z}_{\geq 0}$.
Ensure: $S_n(c)$.
1: **if** $n_0 = 0$ **then**
2: $v \leftarrow n/2 - 1$,
3: Compute S_{2v+1} $(= S_{n-1})$ using Algorithm 3.1.
4: Compute S_n from S_{n-1}. (3 muls)
5: **else**
6: $v = (n-1)/2$.
7: Compute S_{2v+1} $(= S_n)$ using Algorithm 3.1.
8: **end if**
9: **return** S_n.

Algorithm 3.3 XTR Double Exponentiation I

Require: a, b, c_k, c_l, c_{k-l} and c_{k-2l}, where $0 < a,\ b < q$.
Ensure: c_{bk+al}.
1: $d \leftarrow b$, $e \leftarrow a$, $c_u \leftarrow c_k$, $c_v \leftarrow c_l$, $c_{u-v} \leftarrow c_{k-l}$, $c_{u-2v} \leftarrow c_{k-2l}$, $f_2 = 0$ and $f_3 = 0$.
2: **if** both d and e are even **then**
3: $(d, e) \leftarrow (d/2, e/2)$, $f_2 \leftarrow f_2 + 1$.
4: **end if**
5: **if** both d and e are divisible by 3 **then**
6: $(d, e) \leftarrow (d/3, e/3)$, $f_3 \leftarrow f_3 + 1$.
7: **end if**
8: **while** $d \neq e$ **do**
9: **if** $d > e$ **then**
10: Update $(d, e, c_u, c_v, c_{u-v}, c_{u-2v})$ according to Table 3.1.
11: **end if**
12: **if** $e > d$ **then**
13: Update $(d, e, c_u, c_v, c_{u-v}, c_{u-2v})$ according to Table 3.2.
14: **end if**
15: **end while**
16: Compute c_{u+v}, given c_u, c_v, c_{u-v} and c_{u-2v}. Let $\tilde{c}_1 \leftarrow c_{u+v}$. (3 muls)
17: Using Algorithm 3.2 with $\tilde{S}_1 = (3, \tilde{c}_1, \tilde{c}_1 + 2\tilde{c}_1^p)$, $\tilde{c}_1$ and d, compute $\tilde{c}_d = c_{d(u+v)}$. Or use Algorithm 3.4 to compute $\tilde{c}_d = c_{d(u+v)}$ based on $\tilde{c}_1$ (Note that this leads to recursive calls to this algorithm).
18: Compute $c_{2^{f_2} d(u+v)}$ based on $c_{d(u+v)}$ by applying Lemma 4.*1*.
19: Compute $c_{3^{f_3} 2^{f_2} d(u+v)}$ based on $c_{2^{f_2} d(u+v)}$ by applying Lemma 4.*2*.

Table 3.1: Updating Formulae for Line 10 of Algorithm 3.3

Condition	Update $(d, e, c_u, c_v, c_{u-v}, c_{u-2v})$	Costs
i. If $d \leq 4e$	$(e, d-e, c_{u+v}, c_u, c_v, c_{v-u})$	3 muls
ii. Else if d is even	$(\frac{d}{2}, e, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$	7 muls
iii. Else if e is odd	$(\frac{d-e}{2}, e, c_{2u}, c_{u+v}, c_{u-v}, c_{-2v})$	7 muls
iv. Else (e is even)	$(\frac{e}{2}, d, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$	4 muls

Table 3.2: Updating Formulae for Line 13 of Algorithm 3.3

Condition	Update $(d, e, c_u, c_v, c_{u-v}, c_{u-2v})$	Costs
i. If $e \leq 4d$	$(d, e-d, c_{u+v}, c_v, c_u, c_{u-v})$	3 muls
ii. Else if e is even	$(\frac{e}{2}, d, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$	4 muls
iii. Else if d is odd	$(\frac{e-d}{2}, d, c_{2v}, c_{u+v}, c_{v-u}, c_{-2u})$	7 muls
iv. Else (d is even)	$(\frac{d}{2}, e, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$	7 muls

The exact analysis of Algorithm 3.3 seems difficult. The statistical behavior based on actual experiments of the algorithm is presented in [77].

Conjecture 1 ([77]). *Given a and b such that $0 < a, b < q$, and trace values c_k, c_l, c_{k-l} and c_{k-2l}, where k and l are not known explicitly, the trace value c_{bk+al} can be computed on average in about $6\log_2(\max(a, b))$ multiplications in $GF(p)$ using Algorithm 3.3.*

3.2.3 XTR Single Exponentiation Revisited

In [77], an efficient single exponentiation algorithm based on Algorithm 3.3 is proposed. It is easily seen that running Algorithm 3.3 with any integers a, b such that $u = a+b$, and $k = 1$ and $l = 1$, will result in a desired output, $c_u = c_{a+b}$. Note that we have much freedom of choice of a and b. They observed that a and b can be chosen in such a way that Algorithm 3.3 favors the 'cheap' step, while quickly decreasing d and e.

The least expensive steps in Algorithm 3.3 are steps 10.i and 13.i. It is possible to choose a and b such that step 10.i is favored and $u = a + b$. A good way to split u into the sum of a and b,

$$a = \left\lfloor \frac{3-\sqrt{5}}{2} u \right\rfloor,$$

$$b = u - a.$$

Algorithm 3.4 XTR Single Exponentiation
Require: $0 < u < q$ and c_1.
Ensure: c_u.
1: $a \leftarrow \lfloor \frac{3-\sqrt{5}}{2} u \rfloor$ and $b \leftarrow u - a$.
2: Run Algorithm 3.3 with input, $c_k = c_l = c_1$, $c_{k-l} = c_0 = 3$ and $c_{k-2l} = c_{-1} = c_1^p$, resulting in $c_{a+b} = c_u$.

In such a case, a and b are chosen such that b/a is close to the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$. Algorithm 3.3 sets $d = b$ and $e = a$ initially in line 1. Lines 3 and 6 do not affect the ratio of $d/e = \phi$. The step 10.i do not break the golden ratio either. For example, suppose that $d/e = \phi$, so $d = \frac{1+\sqrt{5}}{2} e$. After step 10.$i$, the ratio between new d and e is also ϕ. This feature is called the 'Fibonacci step back' behavior. Furthermore, the sum of d and e is reduced by the factor $\phi \approx 1.62$ after step 10.i is applied.

Combining all those observations leads to Algorithm 3.4.

To analyze Algorithm 3.4, we divide iterations in Algorithm 3.3 into two phases:

1. Phase 1: for the first a few iterations, only step 10.i will be executed.
2. Phase 2: due to the small error resulting from the rounding ($a = \lfloor \frac{3-\sqrt{5}}{2} u \rfloor$), the Fibonacci behavior will be lost at some point of the algorithm.

Proposition 2 ([77]). *In Algorithm 3.4, Phase 1 takes about $\log_\phi \sqrt{u}$ iterations. Furthermore, after Phase 1, d and e are reduced to half their original sizes.*

Corollary 1. *Given an integer u with $0 < u < q$ and a trace value c_1, the trace value c_u can on average be computed in about $5.2 \log_2 u$ multiplications in $GF(p)$ using Algorithm 3.4.*

Proof. We divide the running of Algorithm 3.3 in Algorithm 3.4, into two phases.

1. Phase 1: step 10.i requires 3 multiplications and there are $\log_\phi \sqrt{u}$ iterations in Phase 1, $3 \log_\phi \sqrt{u} \approx 2.2 \log_2 u$ multiplications in $GF(p)$ are required for Phase 1.
2. Phase 2: Fibonacci behavior is lost. The remaining d and e are assumed to be random integers of about the same order of magnitude as $\sqrt{u}$. So it takes $6 \log_2 \sqrt{u} = 3 \log_2 u$ multiplications in $GF(p)$.

Therefore Algorithm 3.4 is expected to require about $5.2 \log_2 u$ multiplications in $GF(p)$. □

Chapter 4

Low-Weight Polynomial Form Integers for Efficient Modular Multiplication

In this chapter, we introduce a new class of moduli called the low-weight polynomial form integers (LWPFIs). LWPFIs are expressed in a monic, low-weight polynomial form, $f(t) = t^l - f_{l-1}t^{l-1} - \cdots - f_0$, where t is a positive integer and $f_i \in \{0, \pm 1\}$. We present a modular multiplication scheme based on LWPFI moduli and show analysis and experimental results. The work presented in this chapter appeared in [13] and in our forthcoming paper in IEEE Transactions on Computers [15].

4.1 Modular Multiplication Using Low-Weight Polynomial Form Integers

In [76], Solinas has proposed generalized Mersenne numbers (GMNs) for efficient modular multiplication. A GMN is expressed as a low-weight polynomial $f(t)$, where t is a power of 2 and $f(t)$ is a small-degree polynomial. An LWPFI is also expressed as a polynomial $f(t)$, but t is not necessarily restricted to a power of 2, and the coefficients of $f(t)$ are limited to 0 and ± 1. Even though allowing only 0 and ± 1 for the coefficients of $f(t)$ leaves only 3^l possible choices for $f(t)$, allowing any integer for t gives far more choices of integers than does GMN.

Definition 4 (LWPFI). *For a positive integer t, let $f(t) = t^l - f_{l-1}t^{l-1} - f_{l-2}t^{l-2} - \cdots - f_1 t - f_0$ be a monic polynomial of degree l. We call a positive integer $p = f(t)$ a* **low-weight polynomial form integer (LWPFI)** *if $f_i \in \{-1, 0, 1\}$, $l \geq 2$ and $t > 2(2^{2l+1} - 1)(2^l - 1) \approx 2^{3l+2}$.*

In Definition 4, the value $l = 1$ is excluded so that LWPFIs are different from the usual form of integers. The reason for having the condition $t > 2(2^{2l+1} - 1)(2^l - 1)$ is explained in Section 4.1.2. In practice, the condition $t > 2(2^{2l+1} - 1)(2^l - 1)$ is easily satisfied. For cryptographically useful values of $p = f(t)$, the degree l of $f(t)$ is a very small integer ($l = 2, 3, 4, \ldots$) and t is a large integer (at least $t > 2^w$, where w is the processor's word length in bits). For an n-bit integer t, it can be proven that an LWPFI is at least $((n-1)l+1)$ bits long and at most nl bits long.

When computing modular arithmetic using LWPFI moduli, operands are to be expressed in a signed-digit representation. For an integer $x \in \mathbb{Z}_{p=f(t)}$, we use the following redundant signed-digit representation,

$$x \equiv x_{l-1}t^{l-1} + \cdots + x_1 t + x_0 \pmod{p = f(t)}, \tag{4.1}$$

such that $|x_i| \leq \psi = (t + 2^{l+1} - 2)$.

Equation (4.1) can be written as $x(t) = (x_{l-1} \cdots x_1 x_0)_t$, since the former can be viewed as a degree-$(l-1)$ polynomial in $\mathbb{Z}[t]$. For simplicity, we say a representation $x(t)$ of an integer x is in ***SD-(t, ψ) form***, and write it as $(x_{l-1} \cdots x_1 x_0)_{SD-(t,\psi)}$ if it satisfies the above conditions. Such a representation exists for any $x \in \mathbb{Z}_p$, if $\psi > (t^{l+1} - 1)/(2t^l - 2)$ and this condition is easily satisfied with $\psi = t + 2^{l+1} - 2$. Note that we have chosen a slightly wider range $|x_i| \leq (t + 2^{l+1} - 2)$ than $|x_i| < t$, which is used in traditional redundant signed-digit representation [3]. The use of this wider range makes it possible to simplify our modular multiplication method using LWPFI moduli described later in this section. Given two input values in SD-(t, ψ) form, our modular multiplication method computes an output also in SD-(t, ψ) form.

Converting an integer in $\mathbb{Z}_{f(t)}$ into an SD-(t, ψ) form requires no more than $(l-1)$ integer divisions by t, where $l = \deg f(t)$. Usually this requirement is not an issue in cryptographic applications since, when the modular multiplication algorithm based on LWPFI moduli is used in exponentiation, conversions between the usual representation and an SD-(t, ψ) form is not significant compared to the entire exponentiation. To convert an integer in usual form to SD-(t, ψ) form, one needs to perform at most $(l-1)$ divisions by t, where each time the bit

Algorithm 4.1 Polynomial Multiplication & Reduction (POLY-MULT-REDC)

Require: $\hat{z}(t) = x(t) \cdot y(t) \bmod f(t)$.
Ensure: $x(t)$ and $y(t)$ in SD-(t, ψ) form.
1: $\hat{z}(t) = \hat{z}_{2l-2}t^{2l-2} + \cdots \hat{z}_1 t + \hat{z}_0 \leftarrow x(t) \cdot y(t)$.
2: **for** For i from $2l-2$ down to l **do**
3: $\quad \hat{z}(t) \leftarrow \hat{z}(t) - \hat{z}_i \cdot f(t) \cdot t^{i-l}$.
4: **end for**
5: **return** $\hat{z}(t)$.

length of the dividend is decreased by approximately the bit length of t. Conversion from SD-(t, ψ) form can be performed by Horner's rule and it requires at most $(l-1)$ multiplications, where each time the bit length of the multiplicand increases approximately by that of t.

Let x, $y \in \mathbb{Z}_{p=f(t)}$ be represented in SD-(t, ψ) form as follows:

$$\begin{aligned} x(t) &= (x_{l-1} \cdots x_1 x_0)_{SD-(t,\psi)}, \\ y(t) &= (y_{l-1} \cdots y_1 y_0)_{SD-(t,\psi)}. \end{aligned}$$

In the following, we show an efficient way to perform modular multiplication of these two integers modulo an LWPFI $p = f(t)$. We call the proposed scheme the *LWPFI modular multiplication*.

The LWPFI modular multiplication is performed in two steps:

1. POLY-MULT-REDC: compute $\hat{z}(t) = x(t) \cdot y(t) \bmod f(t)$ in $\mathbb{Z}[t]/f(t)$.

2. COEFF-REDC: reduce coefficients of $\hat{z}(t)$, such that the resulting polynomial has coefficients that are at most ψ in magnitude.

4.1.1 POLY-MULT-REDC: Multiplication in $\mathbb{Z}[t]/f(t)$

Algorithm 4.1 is a simple and general way to perform the POLY-MULT-REDC step. Line 1 is a multiplication of two l-term polynomials and can be computed in different ways, requiring different amounts of computation as discussed in Section 4.3.1. Lines 2-4 perform a polynomial reduction of a degree-$(2l-2)$ polynomial by $f(t)$. Note that it is only a general polynomial reduction method that works for any $f(t)$. For specific $f(t)$'s, one may find better ways to do this step.

Algorithm 4.2 Coefficient Reduction (COEFF-REDC)

Require: $\hat{z}(t) = (\hat{z}_{l-1} \cdots \hat{z}_1 \hat{z}_0)_t$, where $|\hat{z}_i| \leq (2^l - 1)\psi^2$ for all $i = 0, \ldots, l-1$.
Ensure: $z'(t) = (z'_{l-1} \cdots z'_1 z'_0)_{SD-(t,\psi)}$.

1: $z'(t) = (z'_l \cdots z'_1 z'_0)_t \leftarrow \hat{z}(t)$. (note: $z'_l = 0$)
2: $z'_l \leftarrow \lfloor z'_{l-1}/t \rceil$, $z'_{l-1} \leftarrow z'_{l-1}$ `rem` t.
3: $z'(t) \leftarrow z'(t) - z'_l \cdot f(t)$.
4: **for** i from 0 to $l-1$ **do**
5: $\quad$ $\mathtt{C_i} \leftarrow \lfloor z'_i/t \rceil$ and $z'_i \leftarrow z'_i$ `rem` t.
6: $\quad$ $z'_{i+1} \leftarrow z'_{i+1} + \mathtt{C_i}$.
7: **end for**
8: $z'(t) \leftarrow z'(t) - z'_l \cdot f(t)$.
9: **return** $z'(t)$.

Even though polynomial multiplication and polynomial reduction are separated in Algorithm 4.1, one can choose to combine them for better performance. In Section 4.2, we show how Algorithm 4.1 can be optimized by combining polynomial multiplication and polynomial reduction for $l = 2$ and 3.

Proposition 3. *Suppose that the magnitudes of the coefficients in $x(t)$ and $y(t)$ are bounded by a positive integer ψ. Then the coefficients of $\hat{z}(t)$ computed by Algorithm 4.1 are at most $(2^l - 1)\psi^2$ in magnitude.*

Proof. Let $z(t) = x(t) \cdot y(t)$. It is easily seen that $|z_i| \leq (i+1)\psi^2$ for $i = 0, \ldots, l-1$ and $|z_i| \leq (2l - 1 - i)\psi^2$ for $i = l, \ldots, 2l-2$. The magnitudes of the coefficients in $\hat{z}(t) = z(t) \bmod f(t)$ are maximum when all the coefficients f_i's of $f(t)$ are either 1 or -1. In both cases, the maximum value of $|\hat{z}_i|$ is computed as $(2^l - 2^{l-i-1})\psi^2$. Therefore, $|\hat{z}_i| \leq (2^l - 1)\psi^2$ for all $i = 0, \ldots, l-1$. □

4.1.2 COEFF-REDC: Coefficient Reduction

After POLY-MULT-REDC is completed, we obtain a degree-$(l-1)$ polynomial $\hat{z}(t)$. As shown in Proposition 3, the bit lengths of $\hat{z}_i$'s could be more than twice as long as that of t. The coefficients $\hat{z}_i$'s must be reduced so that the result can be used as input to subsequent modular multiplications. Algorithm 4.2 shows an efficient way to reduce the coefficients of $\hat{z}(t)$, where we used $\lfloor \cdot \rceil$ to denote truncation toward zero, and u `rem` v to denote $u - v \cdot \lfloor u/v \rceil$.

Below we show that Algorithm 4.2 results in SD-(t, ψ) form output.

Proposition 4. *Suppose that the coefficients of $\hat{z}(t)$ satisfy $|\hat{z}_i| \leq (2^l - 1)\psi^2$, where $\psi = t + 2^{l+1} - 2$. Given this input $\hat{z}(t)$, Algorithm 4.2 outputs $z'(t)$, whose coefficients are no greater than ψ in magnitude.*

Proof. Let $\theta = 2^{l+1} - 2$. Then it follows that

$$(2^l - 1)(\theta^2 + 4\theta + 2) = 2(2^l - 1)(2^{2l+1} - 1) < t, \tag{4.2}$$

due to Definition 4. We use (4.2) throughout this proof.

In line 2, since $|z'_{l-1}| \leq (2^l - 1)(t + \theta)^2$, it is easy to see that

$$|z'_l| \leq \lfloor (2^l - 1)(t + \theta)^2 / t \rceil = (2^l - 1)(t + 2\theta), \qquad (\because (2^l - 1)\theta^2 < t)$$

where $\lfloor \cdot \rceil$ is a truncation toward zero. After line 3,

$$\begin{aligned} |z'_i| &\leq (2^l - 1)[(t + \theta)^2 + t + 2\theta] \text{ for } i = 0, \ldots, l - 2, \\ |z'_{l-1}| &\leq (2^l - 1)(t + 2\theta) + t - 1. \end{aligned}$$

In the first iteration of lines 4-6,

$$|\mathtt{C}_0| \leq \left| \left\lfloor \frac{(2^l - 1)[(t + \theta)^2 + t + 2\theta]}{t} \right\rceil \right| \leq (2^l - 1)(t + 2\theta + 1),$$

since $(2^l - 1)(\theta^2 + 2\theta) < t$. To determine the maximum value of $|\mathtt{C}_{1-1}|$, we consider three cases where $l = 2$, $l = 3$ and $l > 3$.

1. Case 1: if $l = 2$,

$$\begin{aligned} |\mathtt{C}_1| &= \left| \left\lfloor \frac{z'_1 + \mathtt{C}_0}{t} \right\rceil \right| \\ &\leq \left\lfloor \frac{(2^l - 1)(t + 2\theta + t + 2\theta + 1) + t - 1}{t} \right\rceil. \\ &\quad (\because (2^l - 1)(4\theta + 1) - 1 < t) \end{aligned}$$

2. Case 2: if $l = 3$,

$$
\begin{aligned}
|\mathtt{C}_1| &\leq \left|\left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+1]}{t} \right\rceil\right| \leq (2^l-1)(t+2\theta+2), \\
|\mathtt{C}_2| &\leq \left|\left\lfloor \frac{(2^l-1)(t+2\theta+t+2\theta+2)+t-1}{t} \right\rceil\right| \leq (2^{l+1}-1).
\end{aligned}
$$

3. Case 3: if $l > 3$,

$$
\begin{aligned}
|\mathtt{C}_1| &\leq \left|\left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+1]}{t} \right\rceil\right| \leq (2^l-1)(t+2\theta+2), \\
|\mathtt{C}_2| &\leq \left|\left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+2]}{t} \right\rceil\right| \leq (2^l-1)(t+2\theta+2), \\
&\vdots \\
|\mathtt{C}_{\mathtt{l-1}}| &\leq \left|\left\lfloor \frac{(2^l-1)(t+2\theta+t+2\theta+2)+t-1}{t} \right\rceil\right| \leq (2^{l+1}-1).
\end{aligned}
$$

Since $z'_l = 0$ after line 3, it follows that $z'_l = \mathtt{C}_{\mathtt{l-1}}$ after line 7. Hence, $|z'_l| \leq (2^{l+1}-1)$ after line 7 for all $l \geq 2$. Since $|z'_i| \leq t-1$ for $i = 0, \ldots, l-1$ after line 7, the magnitudes of $|z'_i|$'s for $0 \leq i < l$ will be no greater than $\psi = t + 2^{l+1} - 2$ after the execution of line 8. Therefore, the output of Algorithm 4.2 is in SD-(t, ψ) form. □

Algorithm 4.2 is much like the modular reduction algorithm using pseudo-Mersenne numbers [55]. However, Algorithm 4.2 is quite different from it, since Algorithm 4.2 does not require a "while" loop, the reason being that the output of Algorithm 4.2 is reduced only to the point where the output meets the conditions for SD-(t, ψ) form. This feature makes Algorithm 4.2 behave in a completely deterministic way.

4.2 Optimization of POLY-MULT-REDC Step

In this section, we show that the POLY-MULT-REDC step can be implemented efficiently for some specific $f(t)$'s by combining polynomial multiplication and polynomial reduction by $f(t)$. We provide optimal $f(t)$'s for implementing the POLY-MULT-REDC step for $l = 2$ and 3. It will be shown in Section 4.3 that larger values of l lead to a better asymptotic bound; however, they introduce more overheads. We consider only small-degree $f(t)$'s that

are useful in practice. It is straightforward however to extend this idea to larger degrees of $f(t)$.

The combining methods shown in this section are more efficient than the multiply-then-reduce method described in Algorithm 4.1. For $l = 2$, the combining method's performance is almost as good as that of polynomial multiplication only. Moreover, polynomial squaring in $\mathbb{Z}[t]/f(t)$ for $l = 2$ is asymptotically faster than polynomial multiplication for some $f(t)$'s. For $l = 3$, some $f(t)$'s make it possible that combined polynomial multiplication and polynomial reduction can be performed using the same number of operations as for polynomial multiplication only.

The methods shown in this section are to optimize Algorithm 4.1 for $l = 2$ and 3. The resulting output of the following methods will be identical to that of Algorithm 4.1 for the same input. Thus, the polynomials computed by the following methods will meet the input condition of Algorithm 4.2 too, provided that the input $x(t)$ and $y(t)$ are also in SD-(t, ψ) form. However, computations in Algorithm 4.1 and the methods in this section do not depend on the fact that the input is in SD-(t, ψ).

We only consider irreducible $f(t)$'s. Reducible $f(t)$'s are guaranteed to generate composite numbers that are, in most cases, not useful for cryptography. When $f(t)$ is reducible there are better ways to perform polynomial multiplications in $\mathbb{Z}[t]/f(t)$. In particular, for $f(t) = \prod_{i=1}^{k} f_i(t)$, where $f_i(t)$'s are irreducible factors of $f(t)$, the minimum number of multiplications required to compute a polynomial multiplication in $\mathbb{Z}[t]/f(t)$ is $2 \cdot \deg f(t) - k$ [87].

4.2.1 Case 1: $l = 2$

We use the Karatsuba algorithm [41] (KA) for 2-term polynomial multiplication. For two degree-2 polynomials $x(t)$ and $y(t)$, KA computes $x(t) \cdot y(t)$ using only three multiplications.

$$x(t) \cdot y(t) = x_1 y_1 t^2 + ((x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0)t + x_0 y_0. \tag{4.3}$$

After polynomial reduction by $f(t)$, we have the following formula for polynomial multiplication and squaring in $\mathbb{Z}[t]/f(t)$.

$$
\begin{aligned}
x(t) \cdot y(t) &\equiv ((x_0 + x_1)(y_0 + y_1) + (f_1 - 1)x_1 y_1 - x_0 y_0)t \\
&\quad + f_0 x_1 y_1 + x_0 y_0 \pmod{f(t)}. &(4.4) \\
x(t)^2 &\equiv ((x_0 + x_1)^2 + (f_1 - 1)x_1^2 - x_0^2)t + f_0 x_1^2 + x_0^2 \pmod{f(t)}. &(4.5)
\end{aligned}
$$

Or, we can obtain alternative formulae by using the following version of KA due to Knuth [43]:

$$x(t) \cdot y(t) = x_1 y_1 t^2 + (x_1 y_1 + x_0 y_0 - (x_0 - x_1)(y_0 - y_1))t + x_0 y_0. \tag{4.6}$$

The following formulae are obtained by taking modulo $f(t)$ of (4.6).

$$
\begin{aligned}
x(t) \cdot y(t) &\equiv ((f_1 + 1)x_1 y_1 + x_0 y_0 - (x_0 - x_1)(y_0 - y_1))t \\
&\quad + f_0 x_1 y_1 + x_0 y_0 \pmod{f(t)}. &(4.7) \\
x(t)^2 &\equiv ((f_1 + 1)x_1^2 + x_0^2 - (x_0 - x_1)^2)t + f_0 x_1^2 + x_0^2 \pmod{f(t)}. &(4.8)
\end{aligned}
$$

Note that (4.4) and (4.5) are good when $f_1 = 1$ and (4.7) and (4.8) are good when $f_1 = -1$. Interestingly, when $f_0 = -1$, we can simplify (4.5) and (4.8) as follows:

$$x(t)^2 \equiv x_1(f_1 x_1 + 2x_0)t + (x_0 - x_1)(x_0 + x_1) \pmod{f(t)}. \tag{4.9}$$

Formula (4.9) needs only two multiplications. Long integer squaring is usually faster than long integer multiplication. As long as integer squaring takes approximately no less than $2/3$ of multiplication time, (4.9) is faster than (4.5) and (4.8), since (4.9) requires two multiplications and (4.5) and (4.8) both require three squarings.

We find that $f(t) = t^2 \pm t + 1$ and $f(t) = t^2 + 1$ are the most attractive choices for $l = 2$. We present detailed analysis results in Section 4.3.1.

4.2.2 Case 2: $l = 3$

For 3-term polynomials, the following 3-way method requires six multiplications [86]:

$$\begin{aligned} x(t) =& D_2 \cdot t^4 \\ &+ (D_2 + D_1 - D_5) \cdot t^3 \\ &+ (D_2 + D_1 + D_0 - D_4) \cdot t^2 \\ &+ (D_1 + D_0 - D_3) \cdot t \\ &+ D_0, \end{aligned} \tag{4.10}$$

where

$$\begin{array}{lllllll} D_0 &=& x_0 y_0, & \quad & D_3 &=& (x_0 - x_1)(y_0 - y_1), \\ D_1 &=& x_1 y_1, & & D_4 &=& (x_0 - x_2)(y_0 - y_2), \\ D_2 &=& x_2 y_2, & & D_5 &=& (x_1 - x_2)(y_1 - y_2). \end{array}$$

After polynomial reduction by $f(t)$, we have the following result:

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv ([f_2(f_2+1) + (f_1+1)] \cdot D_2 + (f_2+1) \cdot D_1 + D_0 - f_2 \cdot D_5 - D_4) \cdot t^2 \\ & \qquad + ([f_1(f_2+1) + f_0] \cdot D_2 + (f_1+1) \cdot D_1 + D_0 - f_1 \cdot D_5 - D_3) \cdot t \\ & \qquad + (f_0(f_2+1) \cdot D_2 + f_0 \cdot D_1 + D_0 - f_0 \cdot D_5). \end{aligned} \tag{4.11}$$

Among all combinations of (f_2, f_1, f_0) that make $f(t)$ irreducible, $(f_2, f_1, f_0) = (-1, -1, 1)$ and $(0, -1, 1)$ put (4.11) into the simplest form.

For $f(t) = t^3 + t^2 + t - 1$,

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv (D_0 - D_4 + D_5) \cdot t^2 + (D_0 + D_2 - D_3 + D_5) \cdot t + (D_0 + D_1 - D_5). \end{aligned} \tag{4.12}$$

For $f(t) = t^3 + t - 1$,

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv (D_0 + D_1 - D_4) \cdot t^2 + (D_0 - D_3 + D_5) \cdot t + (D_0 + D_1 + D_2 - D_5). \end{aligned} \tag{4.13}$$

It is interesting to observe that the computational cost of each of (4.12) and (4.13) is

almost the same as that of (4.10).

4.3 Analysis of LWPFI Modular Multiplication

In this section, the performance of LWPFI modular multiplication described in Section 4.1 is analyzed. In our analysis, we use n to denote the bit length used for $t + 2^{l+2} - 2$; that is, $t + 2^{l+2} - 2 < 2^n$. In practice, t will be quite larger than $2^{l+2} - 2$, hence both t and $\psi = t + 2^{l+1} - 2$ are almost always n-bit integers. We use τ to denote the number of non-zero f_i's in $f(t)$. The following notations are used in our analysis of Algorithms 4.1 and 4.2.

- $T_m(u)$: time needed for multiplying two u-bit integers.
- $T_a(u)$: time needed for adding/subtracting u-bit integers.
- $T_d(u, v)$: time needed for dividing a u-bit integer by a v-bit integer.

We will use an assumption that adding a u-bit integer to a v-bit integer takes $T_a(\min(u, v))$ time. This is a reasonable assumption for most software implementations. A carry at the top most bit position of the shorter integer may occur when adding two integers and it may increase the computation time slightly. However, the carry occurs with probability $\approx 1/2$ when adding two random integers and the probability that the carry will propagate more than one word is only $1/2^w$, where w is the bit size of a computer word.

4.3.1 POLY-MULT-REDC step

POLY-MULT-REDC takes two polynomials in SD-(t, ψ) form as input; that is, the coefficients of the two input polynomials are at most ψ ($< t + 2^{l+2} - 2 < 2^n$) in magnitude.

There are different ways to perform POLY-MULT-REDC step. Algorithm 4.1 is the most straightforward and general approach. If the schoolbook method is used for polynomial multiplication in line 1, l^2 multiplications and $(l-1)^2$ additions are required. The polynomial reduction, lines 2-4, requires $\tau(l - 1)$ additions.

Clearly, the multiplications among coefficients are all n-bit wide. For integer additions, the bit lengths of operands are not the same. However, regardless of the method used for polynomial multiplication and polynomial reduction, the output $\hat{z}(t)$ of Algorithm 4.1 will have coefficients that are at most $(2^l - 1)\psi^2$ in magnitude, which is at most $(2n + l)$ bits long.

Hence, for simplicity, we assume that all the integer additions are $(2n + l)$ bits wide. As a result, we have the upper bound for the running time of Algorithm 4.1 as follows:

$$T(\text{POLY-MULT-REDC}) \leq l^2 \cdot T_m(n) + (l + \tau - 1)(l - 1) \cdot T_a(2n + l). \quad \textbf{(4.14)}$$

Instead of the schoolbook method, other methods can be used for the multiplication of two degree-l polynomials in POLY-MULT-REDC step. For example, at the expense of some overheads, KA [41] or KA-like formulae [61] can reduce the factor l^2 associated with $T_m(n)$ in (4.14) to $M(l)$, where $M(l)$'s for some small l's are given as follows:

$$M(2) = 3, \quad M(3) = 6, \quad M(5) = 13, \quad M(6) = 17, \quad M(7) = 22.$$

Alternatively, one can use the Toom-Cook multiplication method [79, 17, 90] which requires only $(2l+1)$ multiplications at the expense of much higher overheads, including exact divisions by fixed integers.

The number of additions and subtractions in (4.14) can be reduced by combining polynomial multiplication and polynomial reduction as shown in Section 4.2. There are only five irreducible $f(t)$'s for $l = 2$ and we list all of them in Table 4.1. The table also shows required cost for polynomial multiplication and squaring in $\mathbb{Z}[t]/f(t)$, where the notations $M_{u,v}$, S_u, A, a and h respectively mean u-bit $\times$ v-bit multiplication, u-bit squaring, $(2n + l)$-bit addition, n-bit addition and bit shift. We have assumed that $2x_1x_0$, which occurs when squaring $(x_1t + x_0)^2 \bmod (t^2 + 1)$, is computed by computing x_1x_0 ($M_{n,n}$) first and then shifting $(2n)$-bit result to the left by one bit ($2h$). For $l = 3$, there are twelve irreducible $f(t)$'s. The best performance is obtained when $f(t) = t^3 + t^2 + t - 1$ or $f(t) = t^3 + t - 1$ is used. In these cases, the running time of the POLY-MULT-REDC step is $6T_m(n) + 6T_a(n) + 6T_a(2n + l)$. This computational cost is almost the same as that for performing one 3-way multiplication as shown in (4.10).

In terms of the number of single-precision multiplications, there is little difference between multiplying two ln-bit long integers and multiplying two degree-$(l - 1)$ polynomials whose coefficients are n bits long. In fact, polynomial multiplication has a little less overhead since coefficients do not have to overlap, unlike the long integer multiplication. However, in software implementations, polynomial multiplication could be slower because microprocessors can deal only with units of data called words. For example, a 160-bit integer needs five words on a 32-bit architecture, while the same integer in SD-(t, ψ) form with

Table 4.1: Modular Multiplication and Squaring Cost in $\mathbb{Z}[t]/f(t)$ for All Irreducible $f(t)$'s of degree-2

$f(t)$	Multiplication modulo $f(t)$	Squaring modulo $f(t)$
t^2+1	$3M_{n,n}+3A+2a$	$M_{n,n}+M_{n,n+1}+2a+2h$
t^2+t+1	$3M_{n,n}+2A+2a$	$2M_{n,n+1}+3a+h$
t^2-t+1	$2M_{n,n}+M_{n+1,n+1}+2A+2a$	$M_{n,n+2}+M_{n,n+1}+3a+h$
t^2+t-1	$3M_{n,n}+2A+2a$	$3S_n+2A+a$
t^2-t-1	$2M_{n,n}+M_{n+1,n+1}+2A+2a$	$2S_n+S_{n+1}+2A+a$

$l=2$ needs three 2-word coefficients, each filled with 80 bits, assuming $t+2^{l+1}-2<2^{80}$. Multiplying two 160-bit integers require only 15 multiplications using 2-way and 3-way KA. However, multiplying two integers in SD-(t,ψ) form requires 18 multiplications using the same KA methods.

4.3.2 COEFF-REDC step

Figure 4.1 shows how Algorithm 4.2, i.e. COEFF-REDC step, is performed; some input and intermediate values are labeled with circled numbers. We first determine the maximum possible bit lengths of these values. Note that Algorithm 4.1 results in a degree-$(l-1)$ polynomial $\hat{z}(t)=(\hat{z}_{l-1}\cdots\hat{z}_0)_t$ such that $|\hat{z}_i|\le(2^l-1)\psi^2$ for $i=0,\ldots,l-1$.

① It is clear that $\hat{z}_i$'s are at most $(2n+l)$ bits long.

② $|z'_l|\le(2^l-1)(t+2^{l+2}-4)$ is at most $(n+l)$ bits long.

③ $|z'_{l-1}|<(2^l-1)(t+2^{l+2}-4)+t-1$ is at most $(n+l)$ bits long, since $|z'_{l-1}|<(2^l-1)(2^n-1)+2^n-1<2^{n+l}$.

④ $|z'_i|\le(2^l-1)((t+2^{l+1}-2)^2+t+2^{l+2}-4)$, for $i<l-1$, are at most $(2n+l)$ bits long. Note that $t+2^{l+1}-2<2^n$ and $t+2^{l+2}-4<2^n$. It follows that $|z'_i|<(2^l-1)((2^n-1)^2+2^n-1)=(2^l-1)(2^{2n}-2^n)<2^{2n+l}$.

⑤ $|\mathtt{C_i}|\le(2^l-1)(t+2^{l+2}-2)$, for $0\le i<l-1$, is at most $(n+l)$ bits long.

⑥ $|z'_l|\le(2^{l+1}-1)$ is at most $(l+1)$ bits long.

⑦ $|z'_i|\le(t+2^{l+1}-2)$, for $0\le i<l$, is at most n bits long.

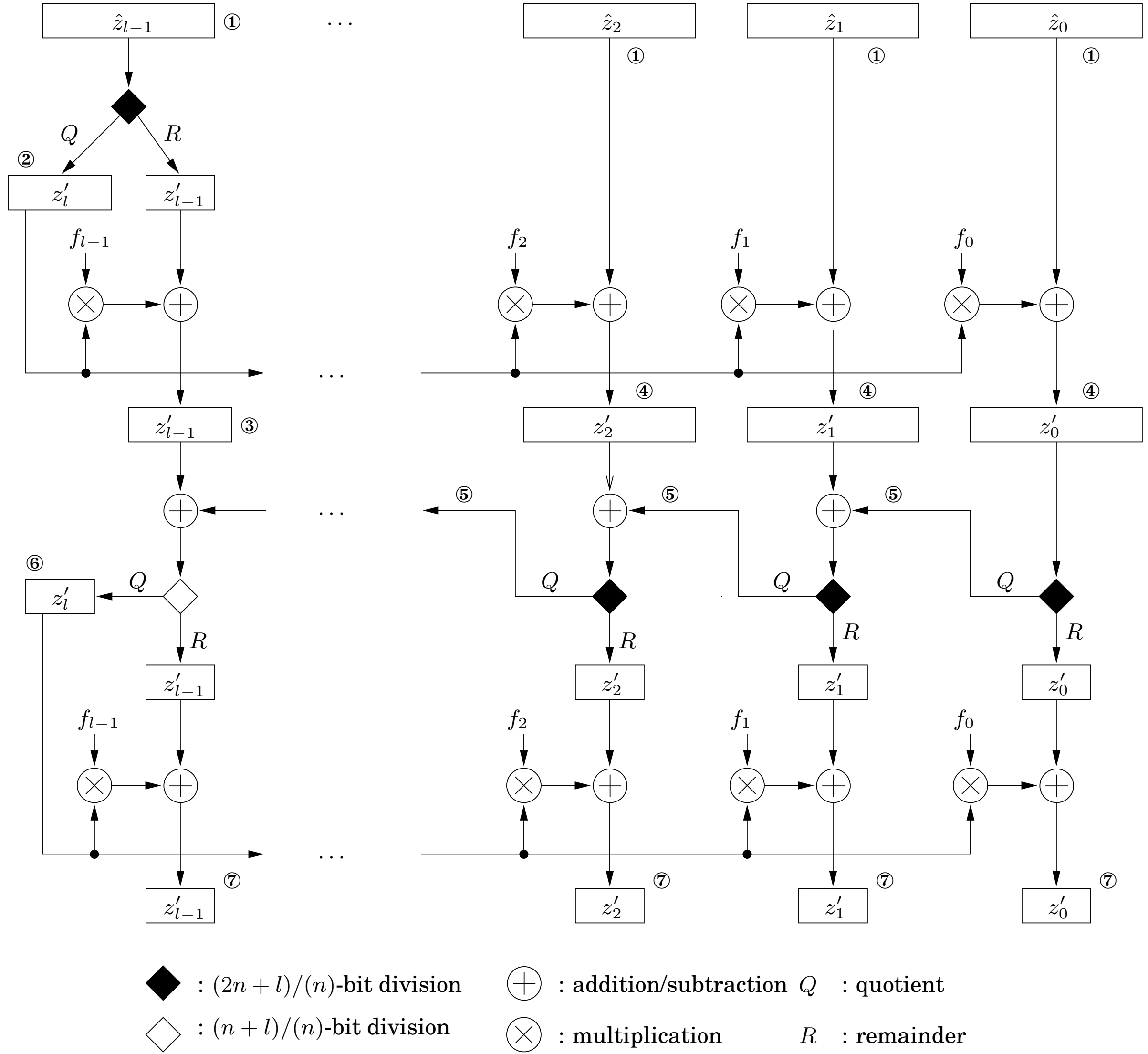

Figure 4.1: Coefficient Reduction

Note that we used the detailed calculations that have been already done in the proof of Proposition 4. Now it is easy to analyze Algorithm 4.2 using the above results.

- Line 2: one integer division for dividing a $(2n+l)$-bit integer by an n-bit integer is needed; that is, $T_d(2n+l,n)$.
- Line 3: τ additions of $(2n+l)$-bit integers and $(n+l)$-bit integers; that is, $\tau \cdot T_a(n+l)$.
- Line 5: for $i=0,\ldots,l-2$, a total of $(l-1)$ integer divisions for dividing $(2n+l)$-bit integer by an n-bit integer are required. For $i=l-1$, The division can be done by at most $(l+1)$ subtractions of up to $(n+l)$-bit integers from an $(n+l)$-bit integer. Thus, for line 5, the required cost is $(l-1)\cdot T_d(2n+l,n)+(l+1)\cdot T_a(n+l)$.
- Line 6: for $i=0,\ldots l-3$, a total of $(l-2)$ additions of $(2n+l)$-bit and $(n+l)$-bit integers are required. For $i=l-2$, an addition of two $(n+l)$-bit integers is performed. For $i=l-1$, no computation is required since $z'_l=0$. Thus, the cost of step 6 is $(l-1)\cdot T_a(n+l)$.
- Line 8: τ additions of an $(l+1)$-bit integer to n-bit integers are performed; that is, $\tau \cdot T_a(l+1)$.

In total, Algorithm 4.2 requires the following amount of time for reducing coefficients:

$$T(\text{COEFF-REDC}) = l\cdot T_d(2n+l,n)+(2l+\tau)\cdot T_a(n+l)+\tau\cdot T_a(l+1). \tag{4.15}$$

With regard to the time complexity related to the long integer division in (4.15), i.e., $l\cdot T_d(2n+l,n)$, note that the division algorithms, CAID and GBAID, shown in Section 2.2 take $O(n^2)$ time for one division, where n is the modulus size in bits. Hence, l executions of such algorithms for n-bit modulus take $O(ln^2)$ time. The overhead terms in (4.15), $(2l+\tau)\cdot T_a(n+l)+\tau\cdot T_a(l+1)$, take $O(ln)$ time.

4.3.3 Putting It All Together

The main computational cost of an LWPFI modular multiplication is due to the following three, where L1 and L2 are performed in POLY-MULT-REDC step and L3 is in COEFF-REDC step.

- L1: polynomial multiplication (e.g., using KA)
- L2: polynomial reduction
- L3: coefficient reduction (e.g., using GBAID)

On the other hand, the main computational cost of a usual modular multiplication is due to the following two:

- U1: integer multiplication (e.g., using KA)
- U2: modular reduction (e.g., using MAIR)

If L1 and U1 use the same algorithm (e.g., KA), then they incur a similar amount of computation.

For an (ln)-bit modulus, assuming that GBAID is used for L3, the combined cost of L2 and L3 is

$$\tau(l-1) \cdot T_a(2n+l) + l \cdot T_d(2n+l, n) + (2l+\tau) \cdot T_a(n+l) + \tau \cdot T_a(l+1),$$

which is $O(ln^2)$ time. On the other hand, U2 using MAIR requires $O(l^2n^2)$ time. Therefore, the LWPFI modular multiplication has better asymptotic behavior than the usual modular multiplication. For example, the number of multiplication instructions required in GBAID for $(2n+l)$-bit dividend and n-bit divisor is expressed as follows:

$$\#\mathcal{M}_{GBAID} = \begin{cases} uv + 3u - v^2 - 2v + 1 & \text{if } u \leq 2v, \\ (u^2 + 5u)/2 - uv + v^2 - v + 1 & \text{if } u > 2v, \end{cases}$$

where $u = \lceil (2n+l)/w \rceil$, $v = \lceil n/w \rceil$, and w is the word length of a target architecture in bits. For $n = 512$ and $l = 2$, the COEFF-REDC step requires only $680 = 2 \cdot 340$ multiplication instructions, whereas MAIR for a similar size (i.e., 1024-bit) modulus requires 1056 multiplications.

Based on the above discussion, we see that the main advantage of LWPFI modular multiplication compared to usual modular multiplication is not due to the POLY-MULT-REDC step. Rather, the main performance gain for using LWPFI modular multiplication comes from the reduced complexity in the COEFF-REDC step.

4.3.4 Comments

The reduced complexity of LWPFI modular multiplication does not come for free. In fact, LWPFI modular multiplication introduces overhead mainly resulting from additions and subtractions. Such overhead due to additions and subtractions needs to be carefully considered. On some microprocessors, the time difference between multiplication and addition/subtraction is relatively not that significant. For example, on Pentium 4 3.2GHz processor (Family 7, Model 4), the latency of multiplication instruction `mul` is 11 clock cycles, and that of add-with-carry `adc` and subtract-with-borrow `sbb` instructions, the most frequently used ones for long integer additions and subtractions, is 10 clock cycles [28]. On the other hand, on Freescale ColdFire 5307, timing ratio of multiplication to addition is 5 when operands are in registers, and the ratio is only 2 when the operands are in memory [25].

In addition, overheads may result from factors pertaining to the implementation environment, and can potentially affect the performance of the modular multiplication algorithms. For example, for software implementation using general purpose processors, these factors would include the size and the number of the registers, cache size and speed, features of the data-path including pipe-lining, multiple execution units, etc. A detailed analysis of the effect of such factors on the performance of the modular multiplication algorithms is not simple. However, to give a good indication on how the LWPFI based algorithm compares with its counterparts we will consider timing results based on actual implementations. This is presented in the following section.

4.4 Implementation Results and Practical Considerations

In this section, first we present timing results of modular multiplications. Then we discuss some general practical considerations for LWPFIs.

4.4.1 Our Platform and Software Routines

We have implemented LWPFI modular multiplications based on $f(t) = t^2 + 1$ and $f(t) = t^3 + t - 1$, and an LWPFI modular squaring based on $f(t) = t^2 + 1$. Our implementation uses GNU multiple precision (GMP) library v$4.1.4$ (`http://www.swox.com/gmp`). We implemented GBAID, which is not provided in GMP, using the C programming language. Since our implementation of GBAID uses only the C programming language, we have disabled all

Table 4.2: Functions Used for Implementing LWPFI Modular Multiplications

Long integer operation	GMP
Multiplication	`mpz_mul()`
Addition	`mpz_add()`
Subtraction	`mpz_sub()`
Bit Shift	`mpz_mul_2exp()`

assembly routines in GMP library. We used Microsoft Visual Studio 2005 to compile all programs, and performed timing measurements on Intel Pentium 4 3.20GHz (Family 7, Model 4). To compile GMP with Visual Studio, we used Visual Studio project file for GMP v$4.1.4$ downloaded from `http://fp.gladman.plus.com/computing/gmp4win.htm`.

Our implementation of LWPFI modular multiplication is based on high level functions of GMP library. Table 4.2 lists GMP functions that we used for implementing LWPFI modular multiplications. We used our GBAID routine for divisions in COEFF-REDC step, since our GBAID routine is much faster than the division function in GMP (`mpz_tdiv_r()`). The timing results shown in this section could be improved by using low level functions (`mpn_*()` functions) that have less redundancy than high level functions.

Our GBAID routine turned out to be faster than MAIR routines in GMP. Thus, we have written our own MAIR routine using the same coding style and optimization that we used when writing GBAID. Our MAIR performs better than our GBAID for all input lengths. The timing results in the following subsection are based on our own Montgomery reduction routine, not on `redc()` in GMP library.

4.4.2 Component-wise Breakdown of Timing

Table 4.3 shows detailed analyses of LWPFI modular multiplication methods for the two $f(t)$'s that we used in our implementation. The notations $T_m(n)$, $T_a(n)$ and $T_B(u, v)$ respectively refer to the running time for long integer multiplication of two n-bit integers, long integer addition of two n-bit integers and GBAID for u-bit dividend and v-bit divisor. $T($POLY-MULT-REDC$)$ and $T($COEFF-REDC$)$ refer to the time required for POLY-MULT-REDC and COEFF-REDC steps, respectively.

We experimentally measured T_1, T_2, T_3 and T_4, as defined in Table 4.3, for varying bit sizes of p and plotted the results in Figures 4.2 and 4.3. In the figures, we use $T_i(l)$ to

Table 4.3: Detailed Analysis of LWPFI Modular Multiplication ($n = \lceil \log_2 (p+1) \rceil$)

	T(POLY-MULT-REDC) $= T_1 + T_2$	
$f(t)$	T_1	T_2
$f(t) = t^2 + 1$	$3 \cdot T_m(n/2)$	$2 \cdot T_a(n+2) + 2 \cdot T_a(n/2)$
$f(t) = t^3 + t - 1$	$6 \cdot T_m(n/3)$	$6 \cdot T_a(2n/3+2) + 6 \cdot T_a(n/3)$
	T(COEFF-REDC) $= T_3 + T_4$	
$f(t)$	T_3	T_4
$f(t) = t^2 + 1$	$2 \cdot T_B(n+2, n)$	$5 \cdot T_a(n/2+2) + T_a(3)$
$f(t) = t^3 + t - 1$	$3 \cdot T_B(2n/3+3, n)$	$8 \cdot T_a(n/3+3) + 2 \cdot T_a(4)$

denote T_i for the l-th degree $f(t)$ in Table 4.3. In Figure 4.3, $T_M(u,v)$ denotes the timing for Montgomery reduction when the input integer is u bits long and the modulus is v bits long. In Figure 4.3, we present $T_B(2n,n)$ to show how much time COEFF-REDC saves by breaking up a full $(2ln)$-bit by (ln)-bit division into l short divisions for $(2n+l)$-bit dividend and n-bit divisor plus some overheads. The $T_M(u,v)$ is shown as a reference timing of the best modular reduction algorithm considered in this chapter.

In Figures 4.2 and 4.3, we see that the overheads resulting from additions/subtractions (T_2's and T_4's) are not significant in both POLY-MULT-REDC and COEFF-REDC steps. Especially in Figure 4.3, the overhead timings, $T_4(i)$ for $i = 2$ and 3, are very small compared to the reduction timings and they both are plotted close to the x-axis of the graph. The figures confirm our analytical conclusion in Section 4.3 that the efficient modular multiplication using LWPFI moduli is not due to the POLY-MULT-REDC step where KA has been used, but due to the COEFF-REDC step.

Note that GMP's `mpz_mul()` routine switches from the classical algorithm to KA when the operand size is more than 32 words long (i.e., 1024 bits for $w = 32$). This explains the sudden changes in $T_m(n)$ whenever $\lceil \log_2 (p+1) \rceil = 1024 \cdot 2^i$ for $i \geq 0$. The performance of POLY-MULT-REDC for $l = 2$ is very close to that of `mpz_mul()` ($T_m(n)$ in Figure 4.2) for $\lceil \log_2 (p+1) \rceil \geq 1024$, since they both have the same asymptotic speed-up due to KA. However, the timing results of POLY-MULT-REDC for $l = 3$ shows sudden changes in timing whenever $\lceil \log_2 (p+1) \rceil = 3 \cdot 1024 \cdot 2^i$ for $i \geq 0$, but it does not become similar to $T_m(n)$, since the 3-way KA presented in (4.10) does not lead to the same asymptotic speed-up as the original 2-way KA.

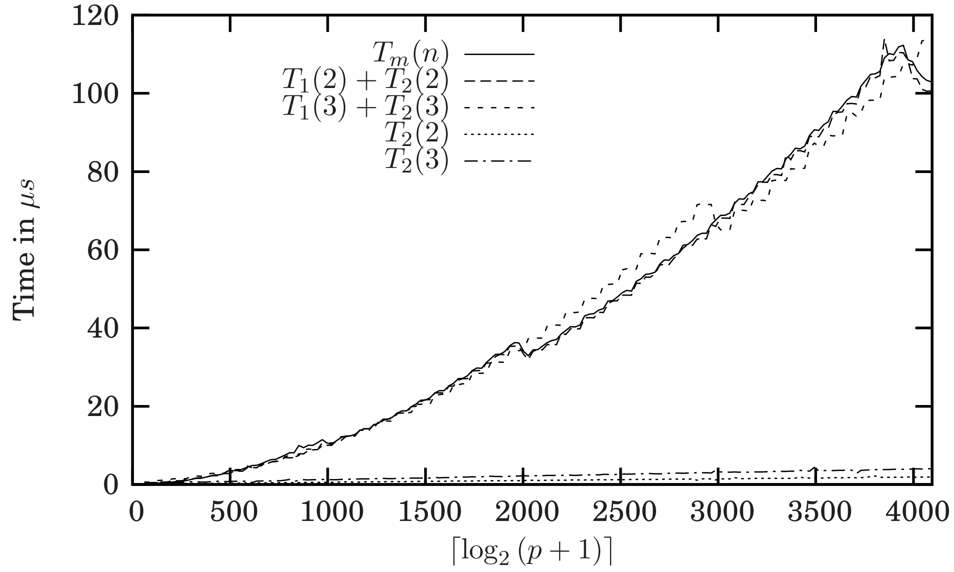

Figure 4.2: Timing Results for POLY-MULT-REDC Step on Pentium 4 @ 3.2GHz

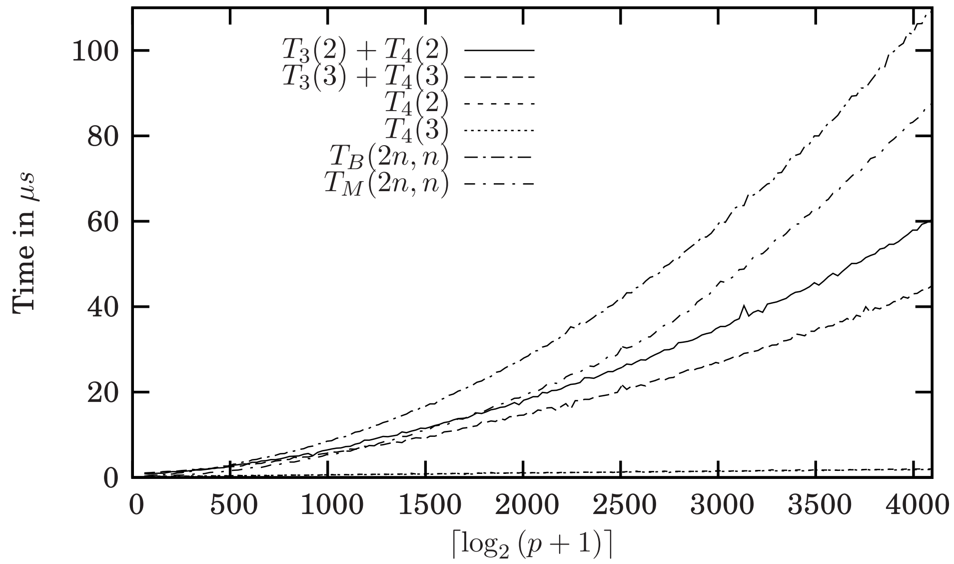

Figure 4.3: Timing Results for COEFF-REDC Step on Pentium 4 @ 3.2GHz

4.4.3 Overall Timing Results and Comparisons

Figure 4.4 shows timing results of our implementations of the following three modular multiplication methods:

1. "LWPFI mul." where KA is used for polynomial multiplication and GBAID is used for COEFF-REDC (as discussed in this chapter). For this method, we show three plots corresponding to $f(t) = t^2 + 1$, $f(t) = t^3 + t - 1$ and $f(t) = t^4 - t^2 - 1$.

2. "Mul. + MAIR" where KA is used for long integer multiplications and MAIR is for modular reduction.

3. "Mul. + GBAID" where KA is used for long integer multiplications and GBAID is for modular reduction.

In method 3), instead of GBAID, one can use the original Barrett reduction algorithm, which does not generate a quotient as output, However, the difference between the computational costs of these two schemes is negligible. Also note that in method 3), the divisor of GBAID is the modulus (say (ln) bits long). On the other hand, for the same size moduli, the size of the divisor in the GBAID used in method 1) is n bits only. However, in method 1), the GBAID routine is used l times, whereas in method 3), the GBAID is used only once for each modular multiplication.

Figure 4.5 shows timing results for modular squaring operations using the same three methods shown above. In the case of LWPFI modular squaring, only the timing result for $l = 2$ is shown.

We clearly observe in the figures that LWPFI modular multiplications become more efficient than GBAID and MAIR based modular multiplications as the modulus size increases. We also observe that the asymptotic behavior of LWPFI modular multiplication improves as l increases, and that the LWPFI modular squaring for $l = 2$ indeed performs better than modular squaring methods using GBAID and MAIR.

4.4.4 Practical Considerations

- General implementation is possible using LWPFI proposed in this work. For a given bit length, we can find many useful moduli by varying the value of t, even for a fixed $f(t)$. On the other hand, the most limiting part of GMNs proposed in [76] is that there can be

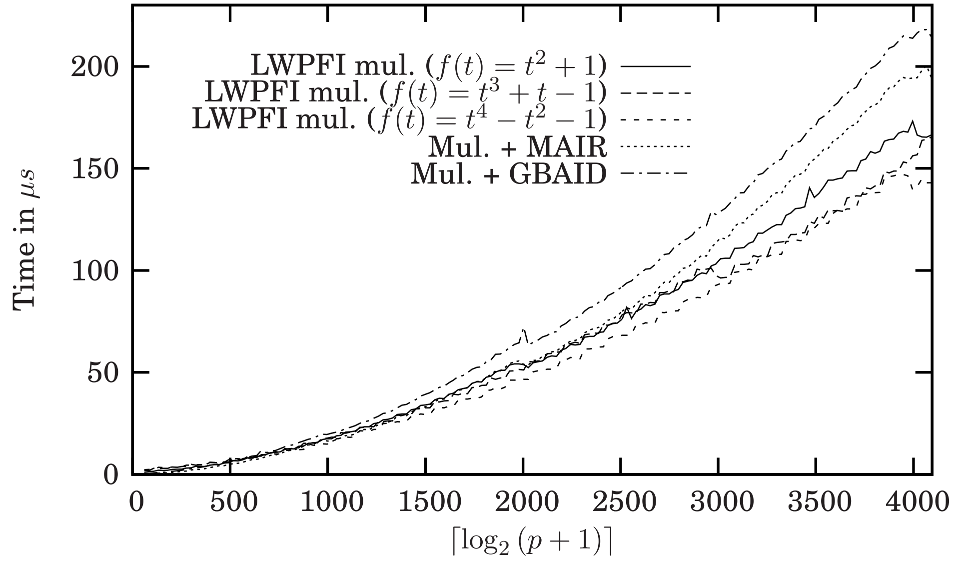

Figure 4.4: Modular Multiplication Algorithms on Pentium 4 @ 3.2GHz

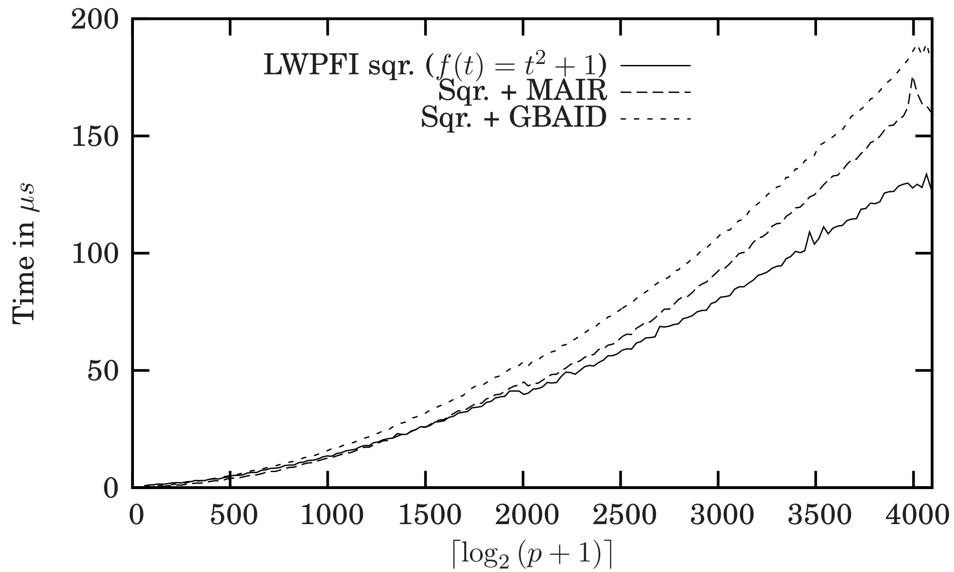

Figure 4.5: Modular Squaring Algorithms on Pentium 4 @ 3.2GHz

only one GMN for a given bit length and a polynomial $f(t)$. This fact makes generalized implementation infeasible, and each GMN requires a dedicated implementation. This is not a problem in ECC and HECC, since it is the usual practice to set up domain parameters for use by many users in such cryptosystems. There are even pre-defined sets of recommended parameters for ECC [64, 66]. However, in RSA cryptosystems, every user has to generate his or her own parameters, and in XTR cryptosystems [53], every user is advised to do so. Hence, these cryptosystems do not benefit from the fast modular multiplication that GMNs provide.

- In RSA, private key operations (e.g., decryption and signature generation) can be performed faster by using LWPFI for both prime factors. In such a case, the degree of $f(t)$ must be sufficiently small for a fixed modulus size so that it is infeasible to find a prime factor using exhaustive search.

- LWPFI can be used in cryptosystems based on the hardness of integer discrete logarithm problem. In such cryptosystems, one has to generate a large prime modulus p such that $p-1$ has a prime factor q. Such an integer is easily constructed by first choosing a large enough prime q and then randomly trying k until $p = qk+1$ is also a prime number. Using LWPFI, one can choose a prime integer t and $f(t) = t(t^{l-1} + f_{l-1}t^{l-2} + \cdots + f_1) + 1$ such that they are both prime integers. For instance, to generate a 1020-bit prime modulus, one needs to try randomly 170-bit prime t until $f(t) = t^6 + f_5t^5 + \cdots + f_1t + 1$ is also a prime integer.

- LWPFI modular multiplication makes it easy for parallel implementation. In POLY-MULT-REDC, KA involves several multiple-precision multiplications. These multiplications are independent of each other and they can be computed in a parallel manner. For example, if $l = \deg(f(t))$ is 3, then six multiplications (D_0 through D_5 in (4.10)) can be computed by two, three or six processors or multipliers simultaneously. Even though it is not explicitly shown here, it is not hard to modify Algorithm 4.2 so that l divisions by t are parallelized.

- Cryptographic computations usually require operations using large operands. However, implementing operations which deal with very long operand sizes is challenging in restricted environments, such as smart cards and certain embedded systems. LWPFI modular multiplication makes it possible to reduce the operand sizes by about $1/l$,

where $l = \deg(f(t))$.

- There are security concerns when moduli take a special form. Mersenne numbers are avoided in RSA cryptosystems, since they are easier to be factored using the special number field sieve (SNFS) [49, 50]. A similar technique known as special function field sieve (SFFS) can be used for solving discrete logarithm problems based on special form of moduli [75, 73]. However, SNFS and SFFS are applicable only to integers having very low Hamming weight (e.g., Fermat's numbers, Mersenne numbers, *etc*.). SNFS and SFFS are not applicable to LWPFI, since LWPFI moduli are not in such a form i.e., $p = t^l - (f_{l-1}t^{l-1} + \cdots + f_0)$ in which t and $(f_{l-1}t^{l-1} + \cdots + f_0)$ are both very large. However, currently there is no guarantee that cryptographic applications are secure when LWPFI is used. It is an open question whether LWPFI makes factoring or discrete logarithm easier.

4.5 Enhancing the LWPFI Modular Multiplication

In this section, we show methods for enhancing LWPFI modular multiplication.

4.5.1 Using Pseudo-Mersenne Numbers for t ($t = 2^n - c$)

If t is chosen to be a pseudo-Mersenne number, such that $t = 2^n - c$ for some n and small c, the performance of the LWPFI modular multiplication can be further improved. For a pseudo-Mersenne number t, there exists an efficient modular reduction algorithm due to Crandall [19]. The original Crandall algorithm is used only for computing the remainder. Algorithm 4.3 shows the modified Crandall algorithm which also computes the quotient.

The correctness of Algorithm 4.3 can be easily derived from the one shown in [55] for the original Crandall algorithm. The main difference between the original Crandall algorithm and Algorithm 4.3 is that the former accumulates only r_i's, while the latter accumulates q_i's also.

In LWPFI modular multiplication, divisions of $(2n + l)$-bit integers by an n-bit integer t are required. By observing the fact that q_{i+1} is always at least $(n - g)$ bits shorter than q_i for a g-bit c, we can see that, for $(2n + l)$-bit input, Algorithm 4.3 requires at most s iterations of lines 3-7 if $c < 2^{((s-1)n-l)/s}$.

Algorithm 4.3 Modified Crandall Algorithm

Require: positive integers $x \geq t$ and $t = 2^n - c$.
Ensure: q and r, such that $x = q \cdot t + r$ and $0 \leq r < t$.
1: $q_0 \leftarrow \lfloor x/2^n \rfloor$, $r_0 \leftarrow x \bmod 2^n$.
2: $q \leftarrow q_0$, $r \leftarrow r_0$, $i \leftarrow 0$.
3: **while** $q_i > 0$ **do**
4: (Loop invariant: $x = qt + r + q_i c$.)
5: $q_{i+1} \leftarrow \lfloor q_i \cdot c/2^n \rfloor$, $r_{i+1} \leftarrow q_i \cdot c \bmod 2^n$.
6: $q \leftarrow q + q_{i+1}$, $r \leftarrow r + r_{i+1}$, $i \leftarrow i + 1$.
7: **end while**
8: **while** $r \geq t$ **do**
9: $r \leftarrow r - t$, $q \leftarrow q + 1$.
10: **end while**
11: **return** q and r.

4.5.2 Using LWPFI for t

When LWPFIs are used for t, coefficient reduction could be done trivially. We show that dividing an integer in SD-(t, ψ) form by an LWPFI can be done very efficiently. Suppose $f(t)$ is a monic polynomial of degree l:

$$f(t) = t^l - f_{l-1}t^{l-1} - \cdots - f_1 t - f_0.$$

Let $x(t)$ be a degree-$(2l-2)$ polynomial:

$$x(t) = x_{2l-2}t^{2l-2} + \cdots + x_1 t + x_0.$$

Define a polynomial $q(t)$ of degree-$(l-2)$ such that

$$\begin{aligned} q(t) &= q_{l-2}t^{l-2} + \cdots + q_1 t + q_0, \\ q_i &= x_{l+i} + \sum_{j=i+1}^{l-2} q_j f_{j+1}. \end{aligned}$$

Then it follows that $q(t)$ satisfies the following:

$$x(t) = q(t)f(t) + r(t) \qquad (\deg(r(t)) < \deg(f(t))).$$

Therefore, we have a formula for the quotient polynomial $q(t)$. Since the quotient polynomial $q(t)$ can be obtained while computing the remainder polynomial $r(t)$, the above method requires at most $\tau(l-1)$ additions/subtractions, where τ is the number of non-zero f_i's in $f(t)$.

4.6 Conclusions

In this chapter, a new family of integers called the low-weight polynomial form integers (LWPFIs) have been introduced. LWPFIs are expressed in polynomial form $f(t)$, and they further extend GMNs by allowing any positive integer integer for t. However, LWPFIs allow only 0 and ± 1 for the coefficients of $f(t)$. A modular multiplication scheme using LWPFIs have been presented. Detailed analysis and experimental results on our modular multiplication scheme have been presented. Our analysis shows that LWPFI modular multiplication has better asymptotic behavior than other general modular reduction methods. Our implementation results show that LWPFI modular multiplication is faster than Montgomery reduction for moduli of large sizes. GMN or pseudo-Mersenne number based modular multiplication would be faster than LWPFI based one, however there are not that many GMNs and pseudo-Mersenne numbers. LWPFI has its advantage that the implementation does not have to be specific to a single modulus and that LWPFI provides a considerably larger choice of moduli than GMN.

Since the publication of a preliminary version of this work at SAC 2003 [13], Bajard, Imbert and Plantard have proposed two number systems called the adaptive modular number system (AMNS) [5] and the polynomial modular number systems (PMNS) [6]. These modular number systems have some similarities with LWPFIs in the sense that they use low-weight polynomial form moduli for efficient arithmetic and that numbers are represented in polynomial form. However, the representation of numbers and modular arithmetic in modular number systems are quite different from those in our modular multiplication using LWPFI moduli. In the modular number systems, an integer $x \in \mathbb{Z}_p$ is represented as a vector $(x_0, x_1, \ldots, x_{n-1})$, where $x = \sum_{i=0}^{n-1} x_i \gamma^i \bmod p$, $1 < \gamma < p$ and $x_i \in \{0, \ldots, \rho - 1\}$. The arithmetic operations in the modular number systems are efficient, if the parameters γ, ρ

and p are carefully chosen. Analysis in [5] shows that the modular multiplication in AMNS is more efficient than the usual modular multiplication of integers using the Montgomery reduction algorithm. However, the drawbacks of modular number systems are that the number of moduli for AMNS of practical use appears to be quite limited and that modular multiplications in PMNS require a large look-up table.

Chapter 5

Coefficient Reduction Using Montgomery Reduction Algorithm

In Chapter 4, we have presented a coefficient reduction algorithm based on the Barrett reduction algorithm. In this chapter, we present an improved coefficient reduction algorithm based on the Montgomery reduction algorithm. We show detailed analysis and discuss conditions on parameters to perform the new coefficient reduction method without final subtractions. As a side result, we present methods for modular additions and subtractions modulo an LWPFI.

5.1 Low-Weight Polynomial Form Integers Redefined

As in Chapter 4, LWPFIs are defined as integers expressed in low-weight, monic polynomial form: $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$, where $l \geq 2$, $f_i \in \{0, \pm 1\}$ and $t > 2(2^{2l+1} - 1)(2^l - 1)$.

Here we loosen the restriction on f_i's so that $|f_i| \leq \xi$ for some small positive integer $\xi < t$. The condition $t > 2(2^{2l+1} - 1)(2^l - 1) \approx 2^{3l+2}$ is applied in Chapter 4 due to the use of coefficient reduction based on a division algorithm. However, such a condition is not needed in our improved coefficient reduction presented here. In this chapter, we work in this general framework and narrow down conditions on parameters that allow efficient implementation of modular arithmetic modulo an LWPFI.

Definition 5 (LWPFI Redefined)**.** *For a degree-l, monic polynomial* $f(t) = t^l + f_{l-1}t^{l-1} + \cdots +$

$f_1 t + f_0$, *where* t *is a positive integer and* $|f_i| \leq \xi$ *for some small positive integer* $\xi < t$, $p = f(t)$ *is a* **low-weight polynomial form integer**.

In modular arithmetic based on LWPFI moduli, we express elements of $\mathbb{Z}_p$ as polynomials in $\mathbb{Z}[t]/f(t)$. Such a representation always exists for any element in $\mathbb{Z}_p$ using coefficients at most $(t + \xi)/2$ in magnitude.

Proposition 5. *For any integer* $x \in \mathbb{Z}_p$, *there exists a degree-*$(l-1)$ *polynomial* $x(t) = \sum_{i=0}^{l-1} x_i t^i$ *such that* $x \equiv x(t) \pmod{p}$ *and* $|x_i| \leq \psi$, *if* $\psi \geq (t + \xi)/2$.

Proof. Let $p_{\max} = t^l + \xi t^{l-1} + \cdots + \xi t + \xi$. Then $p_{\max}$ is the maximum possible LWPFI of the form $f(t) = t^l + \sum_{i=0}^{l-1} f_i t^i$, where $|f_i| \leq \xi$. Let $x(t) = \sum_{i=0}^{l-1} x_i t^i$. If $\max(x(t)) - \min(x(t)) \geq p_{\max}$ holds, then $x(t)$ can represent any element in $\mathbb{Z}_{f(t)}$. It is straightforward that

$$\max(x(t)) = \sum_{i=0}^{l-1} \psi t^i = -\min(x(t)). \tag{5.1}$$

It follows that

$$\begin{aligned} \max(x(t)) - \min(x(t)) &\geq p_{\max} \\ \iff (2\psi - \xi) \cdot \frac{t^l - 1}{t - 1} &\geq t^l. \end{aligned} \tag{5.2}$$

It is easy to see that $2\psi - \xi = t - 1$ does not satisfy the above inequality, but $2\psi - \xi \geq t$ does. Therefore $\psi \geq (t + \xi)/2$. □

We let $\psi_{\min} = (t + \xi)/2$. However, in practice, the magnitudes of the coefficients do not have to be limited to ψ_{min}. To find a polynomial that corresponds to a given integer, Algorithm 5.1 can be used. The resulting polynomial has coefficients that are at most $(t/2 + \xi)$ in magnitude. Since $t/2 + \xi > \psi_{\min}$, Algorithm 5.1 results in a slightly redundant representation.

5.2 Modular Multiplication Using LWPFI moduli

In this section, we present an efficient modular multiplication scheme using LWPFI moduli. The modular multiplication using LWPFI moduli is performed in the following steps.

Algorithm 5.1 Conversion to Polynomial Form

Require: an integer $0 \leq x < p$, where $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$.
Ensure: a polynomial $x(t) = \sum_{i=0}^{l-1} x_i t^i$, such that $x \equiv x(t) \pmod{p}$, where $|x_i| \leq t/2 + \xi$.

1: $c_{-1} \leftarrow x$.
2: **for** i from 0 to $l-1$ **do**
3: Find c_i and x_i such that $c_{i-1} = c_i t + x_i$, where $-t/2 \leq x_i < t/2$.
4: **end for**
5: **for** i from 0 to $l-1$ **do**
6: $x_i \leftarrow x_i - f_i \cdot c_{l-1}$. (Note: $|c_{l-1}| \leq 1$)
7: **end for**
8: **return** $x(t) = \sum_{i=0}^{l-1} x_i t^i$.

1. POLY-MULT: $\hat{z}(t) = x(t) \cdot y(t)$.
2. POLY-REDC: $z'(t) = \hat{z}(t) \bmod f(t)$.
3. COEFF-REDC: coefficient reduction of $z'(t)$.

The above modular multiplication scheme is called the *LWPFI modular multiplication*. POLY-MULT step can be performed by at most l^2 multiplications of coefficients using the schoolbook method. Sub-quadratic multiplication algorithms may be applied to achieve better performance [41, 79, 17, 61]. POLY-REDC step requires at most $(l-1)\tau$ constant multiplications by integers at most ξ in magnitude, where τ is the number of non-zero f_i's. The range of f_i we use here is larger than that in Chapter 4. Note that, due to this extended range for f_i's, our POLY-REDC step is potentially slower than that in Chapter 4. However, we will not go over the details on POLY-REDC and focus only on the establishment of a new coefficient reduction algorithm based on the Montgomery reduction algorithm. For fixed $f(t)$, one may consider combining POLY-MULT and POLY-REDC steps for better performance as we propose in Chapter 4.

Suppose that the coefficients of $x(t)$ and $y(t)$ are at most ψ in magnitude. It easily follows that the result of POLY-REDC has coefficients that are at most $\psi^2((\xi+1)^l - 1)/\xi$ in magnitude as shown in Proposition 6. Throughout this chapter, we will use λ to denote $((\xi+1)^l - 1)/\xi$.

Proposition 6. $|z'_i| \leq \lambda\psi^2$.

Proof. Let $x(t)$ and $y(t)$ be the polynomials whose coefficients are at most ψ in magnitude. Let $\hat{z}(t) = (\hat{z}_{2l-2}, \ldots, \hat{z}_1, \hat{z}_0) = x(t) \cdot y(t)$. It follows that $|\hat{z}_i| \leq (i+1)\psi^2$ for $i = 0, \ldots, l-1$ and

$|\hat{z}_i| \le (2l-1-i)\psi^2$ for $i = l, \ldots, 2l-2$. The magnitudes of coefficients in $z'(t) = \hat{z}(t) \bmod f(t)$ are maximum when $f(t) = t^l \pm \xi \sum_{i=0}^{l-1} t^i$. In both cases, $\max(|z'_i|) = ((\xi+1)^{l-1} + (\xi+1)^{l-2} + \cdots + 1) \cdot \psi^2$. Therefore $|z'_i| \le ((\xi+1)^l - 1)/\xi \cdot \psi^2$. □

In Section 5.3, we discuss how the value ψ is related to other parameters, t, ξ and l. In Chapter 4, $\psi = t + 2^{l+2} - 2$ is fixed and a division algorithm derived from the Barrett reduction algorithm is used to perform COEFF-REDC step. In this work, we apply the Montgomery reduction algorithm to perform COEFF-REDC step and determine appropriate value ψ.

Note that the output of our COEFF-REDC based on the Montgomery reduction algorithm (MONT-COEFF-REDC) is different from the output from Algorithm 4.2 in Chapter 4. In Chapter 4, Algorithm 4.2 computes $z(t)$ such that $z(t) \equiv x(t) \cdot y(t) \pmod{p}$. However, the MONT-COEFF-REDC presented here outputs $z(t) \equiv x(t) \cdot y(t) \cdot b^{-q} \pmod{p}$, where b is the radix used to represent coefficients of polynomials in $\mathbb{Z}[t]/f(t)$ and q is a positive integer. Consider two integers $\bar{x}(t) \equiv x(t) \cdot b^q \pmod{p}$ and $\bar{y}(t) \equiv y(t) \cdot b^q \pmod{p}$. These are the transformation of $x(t)$ and $y(t)$ to the so-called *the Montgomery domain*. The direct product of $\bar{x}(t)$ and $\bar{y}(t)$ in $\mathbb{Z}[t]/f(t)$ results in $\bar{x}(t)\bar{y}(t) \equiv x(t)y(t) \cdot b^{2q} \pmod{p}$. Applying our new coefficient reduction algorithm results in $\bar{z}(t) \equiv x(t)y(t) \cdot b^q \pmod{p}$, whose coefficients are at most ψ. Note that the result is the transformation of $x(t)y(t)$ to the Montgomery domain. We discuss the relationship between the value q and other parameters of LWPFI in Section 5.3.

5.2.1 COEFF-REDC based on Montgomery Reduction Algorithm

Here, we construct a new coefficient reduction algorithm which is similar to Algorithm 2.4. Given an input polynomial $z'(t)$ of degree $(l-1)$, our new algorithm computes a polynomial whose evaluation at t is congruent to $z'(t) \cdot b^{-q} \bmod p$.

Before, we begin the description of a new coefficient reduction algorithm, we clarify notations that we use in this chapter. Let $\vec{u}$ and $\vec{v}$ be the column vectors in $\mathbb{Z}^l$ such that the following condition is satisfied:

$$[t^{l-1}, \ldots, t, 1] \cdot \vec{u} \equiv [t^{l-1}, \ldots, t, 1] \cdot \vec{v} \pmod{p}. \tag{5.3}$$

Then we say $\vec{u}$ *is congruent to* $\vec{v}$ *modulo* p and write as $\vec{u} \cong_p \vec{v}$. We slightly abuse this notation and write as $\vec{u} \cong_b v$ for some integer v satisfying $[t^{l-1}, \ldots, t, 1] \cdot \vec{u} \equiv v \pmod{b}$. We also say $\vec{u}$ *is congruent to* v *modulo* b, if $\vec{u} \cong_b v$. We use '$\equiv$', to express element-wise congruence

relation, i.e., $\vec{u} \equiv \vec{v} \pmod{b}$. In "$\vec{u} \bmod b$", modulo operation applies to each element of $\vec{u}$.

Let $x(t) = (x_{l-1}, \ldots, x_1, x_0)_t$ be the result of POLY-REDC step and b be the radix used for representing x_i's. When performing multiplication in $GF(p)[t]/f(t)$, we can apply Algorithm 2.4 individually to each coefficient to reduce them modulo p. However, individual reduction of coefficients is not possible with arithmetic in $\mathbb{Z}[t]/f(t)$. To reduce coefficients in $\mathbb{Z}[t]/f(t)$, we must apply the Montgomery reduction algorithm to all coefficients simultaneously.

The coefficient reduction is closely related to the *closest vector problem* from lattice theory. A lattice $\mathcal{L}$ is a discrete subgroup of $\mathbb{R}^l$. Let $\vec{V} = \{\vec{v}_1, \ldots, \vec{v}_{d-1}, \vec{v}_d\}$ be a set of linearly independent vectors in $\mathbb{R}^l$. The lattice $\mathcal{L} = \mathcal{L}(\vec{V})$ is a set of all integral combination of $\vec{v}_i$'s. The set $\vec{V}$ is called the basis of the lattice $\mathcal{L}(\vec{V})$. If $d = l$, $\mathcal{L}$ is called a full-rank lattice. If $\vec{v}_i \in \mathbb{Z}^l$ for all i, then $\mathcal{L}$ is called an integral lattice. For our purpose, we assume that $\mathcal{L}$ is a full-rank, integral lattice.

Suppose $\vec{v}_i \cong_p 0 \pmod{p}$ for all $i = 1, \ldots, l$. Then all the lattice points in $\mathcal{L}$ represent 0 modulo p. Let $\vec{x}$ be a vector whose elements are the coefficients of $x(t)$. Suppose $\vec{y} \in \mathcal{L}(\vec{V})$ is the closest lattice point (with respect to L_∞ norm) to $\vec{x}$, then $\vec{z} = \vec{x} - \vec{y}$ belongs to the fundamental domain of $\mathcal{L}$. The coordinate values of $\vec{z}$ forms a polynomial $z(t)$ such that $z(t) \equiv x(t) \pmod{p}$ and it has only reasonably small coefficients. However, closest vector problem is believed to be an NP-hard problem. There are polynomial time algorithms that give approximate solutions [1], but they require arithmetic using floating point or rational numbers and are too cumbersome to use for our purposes.

Rather than solving the closest vector problem, we search for $\vec{z'}$ such that $\vec{x} \cong_p \vec{z'} \cdot b^q \pmod{p}$ and the elements of $\vec{z'}$ are reasonably small. Below we show how to find such a vector $\vec{z'}$ using a method similar to the Montgomery reduction algorithm. This approach requires only simple integer arithmetic and enjoys good features of the Montgomery reduction algorithm for integers.

Algorithm 5.2 shows our Montgomery reduction algorithm adapted to perform COEFF-REDC step. Note that we have used $\vec{x}_q^{(i)}$ to denote the element of $\vec{x}_q$ at the i-th row in Algorithm 5.2. Moreover, F is an $l \times l$ integral matrix such that the following holds for any column vectors $\vec{x}$ and $\vec{u} \in \mathbb{Z}^l$:

$$\vec{x} + F \cdot \vec{u} \cong_p \vec{x}. \tag{5.4}$$

A non-trivial matrix F that satisfies (5.4) can be constructed by collecting l column vec-

Algorithm 5.2 MONT-COEFF-REDC

Require: $x(t) = (x_{l-1}, \ldots, x_1, x_0)_t$, a matrix F and $F' = -F^{-1} \bmod b$, where $\det F \neq 0$ and $\gcd(\det F, b) = 1$.

Ensure: $z(t) \equiv x(t) \cdot b^{-q} \pmod{p}$.

1: $\vec{x}_0 \leftarrow [x_{l-1}, x_{l-2}, \ldots, x_0]^T$.
2: **for** i from 0 to $q-1$ **do**
3: $\quad \vec{u}_i \leftarrow F' \cdot \vec{x}_i \bmod b$.
4: $\quad \vec{x}_{i+1} \leftarrow (\vec{x}_i + F \cdot \vec{u}_i)/b$.
5: **end for**
6: Perform final subtractions if necessary.
7: **return** $z(t) = \sum_{i=0}^{l-1} z_i t^i$, where $z_i = \vec{x}_q^{(i)}$.

tors that are congruent to 0 modulo p. Such a matrix F must be invertible modulo b, since we need $F' = -F^{-1} \bmod b$ in line 3 of Algorithm 5.2. The invertibility of F modulo b can be verified by checking if $\det F \neq 0$ and the determinant has no common factor with b, i.e., $\gcd(\det F, b) = 1$.

Theorem 5. *Algorithm 5.2 returns* $z(t) \equiv x(t) \cdot b^{-q} \pmod{p}$.

Proof. It is easily seen that each iteration of Algorithm 5.2 computes the following:

$$\vec{x}_{i+1} \leftarrow \frac{\vec{x}_i + F \cdot (-F^{-1} \cdot \vec{x}_i \bmod b)}{b}. \tag{5.5}$$

Since F is a collection of column vectors that are congruent to 0 modulo p, adding any integral linear combination of the column vectors in F to $\vec{x}_i$ does not change its value in $\mathbb{Z}_p$. Hence, $\vec{x}_{i+1} \cong_p (\vec{x}_i + F \cdot (-F^{-1} \cdot \vec{x}_i \bmod b)) \cdot b^{-1}$. The division by b in (5.5) is exact and requires no division, since

$$\begin{aligned} \vec{x} + F \cdot \vec{u} &= \vec{x} + F \cdot (-F^{-1} \cdot \vec{x} \bmod b) \\ &\equiv [0, \ldots, 0, 0]^T \pmod{b}. \end{aligned} \tag{5.6}$$

Therefore, $\vec{x}_{i+1} \cong_p \vec{x}_i \cdot b^{-1}$. In Algorithm 5.2, the process (5.5) is performed iteratively q times starting with $\vec{x}_0 = \vec{x}$ resulting in $\vec{x}_q \equiv \vec{x} \cdot b^{-q} \pmod{p}$. This is quite similar to the original Montgomery reduction algorithm. The only difference is that Algorithm 5.2 uses vectors and matrix, while the original Montgomery reduction algorithm deals with integers. □

At this point, a number of questions arise: what are the conditions for q such that $\vec{x}_q$ are sufficiently reduced, so that the result can be used as input to the subsequent LWPFI modular multiplications? How do we construct the matrix F? Is Algorithm 5.2 efficient? We answer these questions in the following.

5.2.2 Construction of F and Analysis of Algorithm 5.2

For $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots f_1 t + f_0$, where $|f_i| \leq \xi$, consider the following $l \times l$ matrix F:

$$F = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -t - f_{l-1} \\ -t & 1 & \cdots & 0 & 0 & -f_{l-2} \\ 0 & -t & \cdots & 0 & 0 & -f_{l-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -t & 1 & -f_1 \\ 0 & 0 & \cdots & 0 & -t & -f_0 \end{bmatrix}. \quad (5.7)$$

We have constructed the matrix F such that the column vectors of F are congruent to 0 modulo p, i.e., $F \cong_p [0, \ldots, 0, 0]$. It remains to verify whether F has its inverse modulo b. The invertibility of F modulo b can be easily checked as shown in Proposition 7.

Proposition 7. *The $l \times l$ matrix F as shown in* (5.7) *is invertible modulo b if and only if* $\gcd(p = f(t), b) = 1$ *and* $f(t) \neq 0$.

Proof. We perform some elementary row operations on both sides of $I_l \cdot F = F$, where I_l is an $l \times l$ identity matrix, to obtain

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ t & 1 & \cdots & 0 & 0 & 0 \\ t^2 & t & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t^{l-2} & t^{l-3} & \cdots & t & 1 & 0 \\ t^{l-1} & t^{l-2} & \cdots & t^2 & t & 1 \end{bmatrix} \cdot F = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -C_{l-1} \\ 0 & 1 & \cdots & 0 & 0 & -C_{l-2} \\ 0 & 0 & \cdots & 0 & 0 & -C_{l-3} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 1 & -C_1 \\ 0 & 0 & \cdots & 0 & 0 & -C_0 \end{bmatrix}, \quad (5.8)$$

where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$. Using the fact that the determinant of a triangular matrix is the product of all diagonal entries, we easily obtain that $\det(F) = -C_0 = -f(t)$ and F

is invertible modulo b if and only if $\gcd(f(t), b) = 1$ and $f(t) \neq 0$. We remark that this invertibility condition of F modulo b is always satisfied when $p = f(t)$ is an odd number, for an even radix b. $\square$

We analyze the performance of Algorithm 5.2 in terms of the number of single-precision multiplications and single-precision additions/subtractions. The overhead caused by additions and subtractions shall not be ignored. Additions and subtractions are ignored in many literature, however, the difference between addition/subtraction and multiplication is not significant in many modern microprocessors. The latency of `add` and `sub` instructions is only one clock cycle on Intel Pentium 4 Family 4 processors. However, when long integer addition operation is performed, they are used only when adding or subtracting the least significant digits. The rest of the digits are added or subtracted with slow `adc` (add with carry) and `sbb` (subtract with borrow) instructions, whose latency is 10 clock cycles. These instructions are only 9% faster than `mul` instruction, whose latency is 11 clock cycles [28].

For convenience, we use Intel x86 instructions `mul`, `add` and `adc` to denote the following operations:

- `mul`: single-precision multiplication,
- `add`: addition/subtraction without carry/borrow,
- `adc`: addition/subtraction with carry/borrow.

When multiplying n-digit integer with a single-digit integer, it is clear that n `mul` instructions are required. The numbers of required `add` and `adc` instructions are 1 and $(n-1)$, respectively. When adding i-digit and j-digit integers, the required number of `add` and `adc` instructions are one and $\min(i, j)$, respectively, assuming that carry does not propagate more than one digit place above the most significant digit of the shorter operand. The probability of having carry above the most significant digit place of the shorter integer is $1/2$. The probability that the carry will propagate one more digit place is only $1/b$. Similar argument holds for subtracting two long integers.

Straightforward computation of $\vec{u}_i = -F^{-1} \cdot \vec{x}_i \bmod b$ requires l^2 `mul` and $(l^2 - l)$ `add` instructions. However, exploiting the special structure of F, we can compute $\vec{u}_i$ using only $(2l-1)$ `mul` and $2(l-1)$ `add` instructions, provided that we are allowed to have l-digit precomputed values that depend on the coefficients of $f(t)$ and the value t.

Theorem 6. *The computation $\vec{u}_i = -F^{-1} \cdot \vec{x}_i \bmod b$ can be performed using only $(2l-1)$* `mul` *and $2(l-1)$* `add` *instructions, using l-digit pre-computed values that depend only on the coefficients of $f(t)$ and the value t.*

Proof. Further row operations from (5.8) easily reveals the exact form of $F' = -F^{-1}$ as follows:

$$F' = \frac{-1}{C_0}\begin{bmatrix} C_0 - C_{l-1}t^{l-1} & -C_{l-1}t^{l-2} & \cdots & -C_{l-1}t^2 & -C_{l-1}t & -C_{l-1} \\ tC_0 - C_{l-2}t^{l-1} & C_0 - C_{l-2}t^{l-2} & \cdots & -C_{l-2}t^2 & -C_{l-2}t^1 & -C_{l-2} \\ t^2C_0 - C_{l-3}t^{l-1} & tC_0 - C_{l-3}t^{l-2} & \cdots & -C_{l-3}t^2 & -C_{l-3}t^1 & -C_{l-3} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ t^{l-2}C_0 - C_1t^{l-1} & t^{l-3}C_0 - C_1t^{l-2} & \cdots & tC_0 - C_1t^2 & C_0 - C_1t & -C_1 \\ -t^{l-1} & -t^{l-2} & \cdots & -t^2 & -t & -1 \end{bmatrix}, \tag{5.9}$$

where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$. Now, we can express F' as follows,

$$F' = F'_1 - F'_2, \tag{5.10}$$

where,

$$F'_1 = \begin{bmatrix} \frac{C_{l-1}}{C_0}\vec{v} \\ \frac{C_{l-2}}{C_0}\vec{v} \\ \frac{C_{l-3}}{C_0}\vec{v} \\ \vdots \\ \frac{C_1}{C_0}\vec{v} \\ \frac{1}{C_0}\vec{v} \end{bmatrix}, \qquad F'_2 = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ t & 1 & \cdots & 0 & 0 & 0 \\ t^2 & t & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ t^{l-2} & t^{l-3} & \cdots & t & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}, \qquad \vec{v} = [t^{l-1}, \ldots, t^1, t, 1]. \tag{5.11}$$

The matrix-vector product $F'_2 \cdot \vec{x}_i \bmod b$ can be computed using Horner's rule, and it requires only $(l-2)$ single-precision multiplications and $(l-2)$ single-precision additions. The vector product $\vec{v} \cdot \vec{x}_i$ can be computed by multiplying $(t \bmod b)$ to the $(l-1)$-th entry of $F'_2 \cdot \vec{x}_i \bmod b$, and then adding $(f_0 \bmod b)$ to the result. Assuming that $1/C_0 \bmod b$ and $C_i/C_0 \bmod b$ for $i = 1, \ldots, l-1$ are precomputed, computing $F'_1 \cdot \vec{x}_i \bmod b$ requires only l single-precision multiplications. It only remains to compute $F' \cdot \vec{x}_i = F'_1 \cdot \vec{x}_i - F'_2 \cdot \vec{x}_i$ using l single-precision subtractions. □

Algorithm 5.3 Computing $F' \cdot \vec{x} \bmod b$

Require: $f(t) = t^l + f_{l-1}t^{l-1} + \cdots f_1 t + f_0$, $\vec{x} = [x_{l-1}, \ldots, x_1, x_0]$ and pre-computed values $C_i/C_0 \bmod b$ for $i = 1, \ldots, l-1$ and $1/C_0 \bmod b$, where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$.

Ensure: $F' \cdot \vec{x}^T = [u_{l-1}, \ldots, u_1, u_0]^T$.

1: $v_l \leftarrow 0$.
2: **for** i from 0 to $l-1$ **do**
3: $\quad v_{l-1-i} \leftarrow v_{l-i} \cdot t + x_{l-1-i} \bmod b$ ($l-1$ `mul`, $l-1$ `add`)
4: **end for**
5: **for** i from 1 to $l-1$ **do**
6: $\quad u_i \leftarrow v_0 \cdot C_i/C_0 \bmod b$. ($l-1$ `mul`)
7: **end for**
8: $u_0 \leftarrow v_0/C_0 \bmod b$. ($1$ `mul`)
9: **for** i from 1 to $l-1$ **do**
10: $\quad u_i \leftarrow u_i - v_i \bmod b$. ($l-1$ `add`)
11: **end for**
12: **return** $[u_{l-1}, \ldots, u_1, u_0]^T$.

Algorithm 5.3 explicitly shows how $\vec{u}_i$ is computed using $(2l-1)$ `mul` and $2(l-1)$ `add` instructions. Since $l \geq 2$, Algorithm 5.3 always performs better than the straightforward matrix-vector product, which requires l^2 `mul` and $(l^2 - l)$ `add` instructions.

We analyze the line 4 of Algorithm 5.2. Since each row of F contains only one t and f_i, the matrix-vector product $F \cdot \vec{u}_i$ requires l multiplications of t and f_i's by a 1-digit integer, and some additions/subtractions. Let n and k ($\leq n$) be the digit length of t and f_i, respectively and let τ be the number of non-zero f_i's in $f(t)$. Then the number of `mul` required in line 4 is $(ln + \tau k)$. If f_i's are small powers of 2 or integers with very small Hamming weight, multiplications by f_i's can be efficiently computed, replacing τk multiplications with τ bit shifts.

We now count the numbers of `add` and `adc` instructions in line 4. There are l multiplications of t with single-digit integers from $\vec{u}_i$, and the total numbers of `add` and `adc` instructions required in this computation are l and $l(n-1)$, respectively. There are τ multiplications of f_i with one digit integer from $\vec{u}_i$, and the total numbers of `add` and `adc` instructions are τ and $\tau(k-1)$, respectively. The matrix vector product $F \cdot \vec{u}_i$ involves $(l-1)$ additions/subtractions of $(n+1)$-digit integer and a single digit integer. This can be computed with $(l-1)$ `add` and `adc` instructions. There are τ additions/subtractions of an $(n+1)$-digit integer with a $(k+1)$-digit integer. Since $k \leq n$ by definition, this computation requires τ

add and $\tau(k+1)$ adc instructions. So far, the numbers of add and adc instructions in $F \cdot \vec{u}_i$ have been counted. It only remains to add $F \cdot \vec{u}_i$ to $\vec{x}_i$. This computation requires l add and $l(n+1)$ adc instructions. In total, the numbers of add and adc instructions required in line 4 are $3l + 2\tau - 1$ and $l(2n+1) + 2\tau k - 1$, respectively.

The total number of mul, add and adc instructions required in Algorithm 5.2, not considering the final subtraction step, is summarized as follows:

$$\begin{aligned}
\#\texttt{mul} &= q(l(n+2) + \tau k - 1), \\
\#\texttt{add} &= q(5l + 2\tau - 3), \\
\#\texttt{adc} &= q(l(2n+1) + 2\tau k - 1).
\end{aligned}$$

5.2.3 Conversions to and from the Montgomery Domain

To perform modular multiplication using Algorithm 5.2 as a coefficient reduction algorithm, we must transform operands to the Montgomery domain. For $x(t) \in \mathbb{Z}[t]/f(t)$, we compute $\overline{x}(t) \equiv x(t) \cdot b^q \pmod{p}$. This computation can be easily achieved by multiplying two polynomials $x(t)$ and $y(t) \equiv b^{2q} \bmod p$, and then reduce coefficients using Algorithm 5.2. The result will be $\overline{x}(t) \equiv x(t) \cdot b^q \bmod p$, as desired. It is convenient to have $y(t) \equiv b^{2q} \bmod p$ pre-computed for each p. Conversion from the Montgomery domain can be performed by directly applying Algorithm 5.2 on $\overline{x}(t)$. The result is $\overline{x}(t) \cdot b^{-q} \equiv x(t) \cdot b^q \cdot b^{-q} \equiv x(t) \pmod{p}$, as desired.

5.2.4 Interesting Implementation Options

We consider some special cases for which Algorithm 5.2 can speed up.

- Special Case I: $t \equiv 0 \pmod{b}$.
  In such a case, it can be shown that

$$F' = -F^{-1} = \begin{bmatrix} -1 & 0 & 0 & \cdots & 0 & f_{l-1}/f_0 \\ 0 & -1 & 0 & \cdots & 0 & f_{l-2}/f_0 \\ 0 & 0 & -1 & \cdots & 0 & f_{l-3}/f_0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & f_1/f_0 \\ 0 & 0 & 0 & \cdots & 0 & 1/f_0 \end{bmatrix} \bmod b.$$

Note that F is invertible if and only if $f_0 \neq 0$ and $\gcd(f_0, b) = 1$. Then, $\vec{u}_i = F' \cdot \vec{x}_i \bmod b$ can be computed with only τ `mul` and $(l-1)$ `add` instructions, provided that $f_i/f_0 \bmod b$ for $i = 1, \ldots, l-1$ and $1/f_0 \bmod b$ are pre-computed. Hence, compared to the general case, we save $(2l - \tau - 1)$ `mul` and $(l-1)$ `add` instructions in line 3 of Algorithm 5.2.

When computing $F \cdot \vec{u}_i$, we can save l `mul` instructions and one `adc` instruction, since the least significant digit of t is zero. The total number of saved instructions is given as follows:

$$\#\texttt{mul}_{save} = q(3l - \tau - 1),$$
$$\#\texttt{add}_{save} = q(l - 1),$$
$$\#\texttt{adc}_{save} = q.$$

- Special Case II: f_i's are powers of 2.
  In such a case, multiplication by f_i's can be simply performed by bit shifts. In line 3, there is no speed up. The number of instructions we can save in line 4 is given below.

$$\#\texttt{mul}_{save} = q\tau k,$$
$$\#\texttt{add}_{save} = q\tau,$$
$$\#\texttt{adc}_{save} = q\tau(k - 1).$$

Note that Algorithm 5.2 requires ql bit shift instructions in exchange for the above saved instructions.

5.3 Modular Multiplication Stability

In this section, we carefully analyze the bounds on the input and output of Algorithm 5.2. The conditions on the parameters for which we can eliminate the final subtractions in Algorithm 5.2 are determined.

5.3.1 Montgomery Reduction with Final Subtractions

Suppose that the number of iterations q in Algorithm 5.2 is the same as the digit size of t, i.e., $q = n$. Suppose that $||\vec{x}_0||_\infty \leq \lambda\psi^2$, where $||\cdot||_\infty$ is the maximum norm of $\vec{x}$ defined as

$||[x_l - 1, \ldots, x_1, x_0]||_\infty = \max(|x_{l-1}|, \ldots, |x_1|, |x_0|)$. This is the case when the coefficients of input polynomial to the LWPFI modular multiplication are at most ψ in magnitude. Since $\psi = t - 1$ leads to a maximally redundant signed-digit representation, we assume that $\psi < t$. The bound on the output of Algorithm 5.2 is determined as follows:

$$\begin{aligned} ||\vec{x}_n||_\infty &= \left|\left| \frac{\vec{x}_0 + F \cdot (\sum_{i=0}^{n-1} \vec{u}_i b^i)}{b^n} \right|\right|_\infty \\ &< \frac{\lambda t^2 + b^n \cdot (t + \xi)}{b^n} \\ &< (\lambda + 1)t + \xi. \end{aligned}$$

We used the fact that the magnitude of the sum of elements in each row of F is maximum at the l-th row and it is $(t + \xi)$. Since the magnitude of the output coefficients may be greater than ψ, we must perform final subtractions to reduce the coefficients. To perform final subtractions we find integers h_i and s_i such that $x_i = h_i t + s_i$, where $-t/2 \leq s_i < t/2$, for each coefficient x_i. Then compute $x_i \leftarrow s_i + h_{i-1} - h_l \cdot f_i$ for $i = 0, \ldots, l-1$. Then the new coefficients are at most $t/2 + (\lambda + 2)(\xi + 1)$ in magnitude. This method is fast when λ is very small. In such a case, h_i and s_i can be obtained by additions and subtractions. However it may require long integer division if λ is large.

5.3.2 Montgomery Reduction without Final Subtractions

Under some conditions with a $q \geq n$, we show that it is possible to avoid the final subtractions. We use an approach similar to [84]. Suppose that $||\vec{x}_0||_\infty \leq \lambda\Psi^2$ in magnitude, where Ψ is a positive integer. This is the case when the coefficients of the input polynomials to the LWPFI modular multiplication are at most Ψ in magnitude. To maintain input/output stability, we have to ensure that

$$||\vec{x}_q||_\infty = \left|\left| \frac{\vec{x}_0 + F \cdot (\sum_{i=0}^{q-1} \vec{u}_i b^i)}{b^q} \right|\right|_\infty < \frac{\lambda\Psi^2}{b^q} + t + \xi \leq \Psi. \tag{5.12}$$

Solving (5.12) for Ψ, we obtain

$$\frac{b^q - \sqrt{b^{2q} - 4\lambda(t+\xi)b^q}}{2\lambda} \leq \Psi \leq \frac{b^q + \sqrt{b^{2q} - 4\lambda(t+\xi)b^q}}{2\lambda}. \tag{5.13}$$

The above solution has real roots when $t \leq b^q/(4\lambda) - \xi$. However, this condition only assures that there exist a real solution for Ψ, whereas Ψ must be an integral value. A sufficient condition for the existence of at least one integral value of Ψ is given as follows:

$$\sqrt{b^{2q} - 4\lambda(t+\xi)b^q} \geq \lambda. \tag{5.14}$$

It follows that

$$t \leq \frac{b^{2q} - \lambda^2}{4\lambda b^q} - \xi = \frac{b^q}{4\lambda} - \frac{\lambda}{4b^q} - \xi. \tag{5.15}$$

For t satisfying (5.15), there exists at least one integral value for Ψ. Since the width of the interval (5.13) is at least one, the choice $\Psi = \lfloor b^q/(4\lambda) \rceil$, where $\lfloor x \rceil$ denotes the nearest integer from x, is always within the interval. For this value of Ψ, it always holds that $t + \xi \leq \Psi$.

Suppose that $t + \xi$ is an n'-bit integer, i.e., $2^{n'-1} \leq t + \xi < 2^{n'}$, It can be easily verified that an n'-bit integer $t + \xi$ satisfies (5.15) if

$$\lambda < \frac{b^q}{4 \cdot 2^{n'}}. \tag{5.16}$$

Note that $\lambda = ((\xi+1)^l - 1)/\xi \geq 3$, since $\xi \geq 1$ and $l \geq 2$. Let w be the bit length of a digit, i.e., $b = 2^w$. It follows that ξ and q must be chosen such that the following holds:

$$3 \leq ((\xi+1)^l - 1)/\xi < 2^{qw-n'-2}. \tag{5.17}$$

For efficient implementation, the number of iterations q must be as small as possible, but not smaller than n. For instance, $q = n$ or $q = n+1$, where n is the digit length of $t + \xi$, would be the most interesting cases for implementation. If $n' = nw - \rho$, where $4 \leq \rho < w$, and ξ is chosen such that $3 \leq \lambda < 2^{\rho-2}$, then $q = n$ satisfies (5.16). If $n' = nw - \rho$, where $0 \leq \rho < w$, and ξ is chosen such that $3 \leq \lambda < 2^{w+\rho-2}$, then $q = n+1$ satisfies (5.16).

One may consider using $f(t)$ such that only f_0 and f_1 are non-zero, but $f_i = 0$ for $2 \leq i \leq l-1$. In such a case, it can be proven that the coefficients of the output of POLY-REDC step are at most $(l-1)(|f_0| + |f_1|)\Psi^2$. Then we can choose l, f_0 and f_1 such that $(l-1)(|f_0| + |f_1|) < 2^{qw-n'-2}$. Using this method, we can choose larger coefficients f_0 and f_1 than we can with the condition (5.17). Note that such an $f(t)$ is also used in [6].

To convert $\overline{x}(t) = x(t) \cdot b^q$ from Montgomery domain to the usual domain, we can simply apply Algorithm 5.2 on $\overline{x}(t)$. In such a case, coefficients of the input are bounded by Ψ. The

output $\vec{x_q}$ of Algorithm 5.2 is bounded as follows:

$$||\vec{x_q}||_\infty \leq \frac{\Psi + (b^q - 1)(t + \xi)}{b^q} \leq \frac{\Psi + (b^q - 1)\Psi}{b^q} = \Psi. \tag{5.18}$$

Therefore, the final subtractions are not required even after the final conversion.

Another interesting stability condition is to make the output bounded by b^{q-1}. This may be useful to prevent any potential side channel threat that may exploit the probability on the digit length of the output. Considering that the magnitude of input to Algorithm 5.2 is bounded by λb^{2q-2}, we obtain the following stability condition:

$$||X_q||_\infty < \frac{\lambda b^{2q-2}}{b^q} + t + \xi \leq b^{q-1}. \tag{5.19}$$

It follows that $t \leq b^{q-2}(b - \lambda) - \xi$. Of course, we must ensure that $b > \lambda$, since otherwise we will have a negative t. It is easily seen that any $t < b^{q-1}$. If t is chosen as above, the output of Algorithm 5.2 will be strictly within b^{q-1} bound. When converting back to usual domain, we have that

$$||X_q||_\infty < \frac{b^{q-1} + (b^q - 1)(t + \xi)}{b^q} \leq t + \xi. \tag{5.20}$$

5.4 Additions and Subtractions

In this section, we study additions and subtractions modulo an LWPFI. It is well-known that redundant signed-digit representation allows carry/borrow free additions/subtractions [3]. Here we derive similar methods for additions and subtractions modulo an LWPFI.

To perform an addition or a subtraction of two numbers $x(t)$ and $y(t)$, we first compute coefficient-wise additions and subtractions as follows:

$$z(t) = x(t) \pm y(t) = \sum_{i=0}^{l-1} (x_i \pm y_i) t^i,$$

where $x(t)$ and $y(t)$ are polynomials of degree $(l - 1)$ and the coefficients are at most ψ in magnitude. The coefficients of $z(t)$ will be at most 2ψ in magnitude. Algorithm 5.4 efficiently reduces the coefficients of $z(t)$ so that the coefficients of the resulting polynomial are at most ψ in magnitude. We call it the *short coefficient reduction*.

Algorithm 5.4 Short Coefficient Reduction

Require: $a(t) = (a_{l-1}, \ldots, a_1, a_0)_t$.
Ensure: $c = (c_{l-1}, \ldots, c_1, c_0)_t \equiv a(t) \pmod p$, where $c_i \leq \psi$.
1: $t_0 \leftarrow 0$.
2: **for** i from 0 to $l-1$ **do**
3: (Transfer Digit) $t_{i+1} \leftarrow C(a_i)$.
4: (Reduction) $w_i \leftarrow a_i - t_{i+1} \cdot t$.
5: (Sum) $s_i \leftarrow w_i + t_i$.
6: **end for**
7: **for** i from 0 to $l-1$ **do**
8: (Reduction modulo $f(t)$) $c_i = s_i - t_l \cdot f_i$.
9: **end for**
10: **return** $c = (c_{l-1}, \ldots, c_1, c_0)_t$.

Algorithm 5.4 is based mostly on the carry-free addition or borrow-free subtraction algorithm widely used in redundant signed-digit arithmetic [3]. The only difference is the inclusion of reduction modulo $f(t)$. The reduction modulo $f(t)$ is necessary, since the result must be represented with $(l-1)$ coefficients. The function $C(\cdot)$, depending on the value of its input, outputs an integer in $[-t_{\max}, t_{\max}]$ for some positive integer $t_{\max}$, We will determine the exact behavior of $C(\cdot)$ in the following.

Note that the c_i's computed in Algorithm 5.4 must satisfy the condition, $-\psi \leq c_i \leq \psi$. Since $-t_{\max} \leq C(a_i) \leq t_{\max}$ for some positive integer $t_{\max}$ and $|f_i| \leq \xi$, $w_i + t_i$'s at line 5 must satisfy

$$-\psi + t_{\max} \cdot \xi \leq w_i + t_i \leq \psi - t_{\max} \cdot \xi. \tag{5.21}$$

Substituting $w_i = a_i - t_{i+1} \cdot t$ into (5.21) results in

$$\frac{a_i - (\psi - t_{\max} \cdot \xi - t_i)}{t} \leq t_{i+1} \leq \frac{a_i + (\psi - t_{\max} \cdot \xi + t_i)}{t}.$$

Suppose that $-\mu + t_{\max} \cdot \xi \leq t_i \leq \mu - t_{\max} \cdot \xi$, where $\mu = (\xi + 1) \cdot t_{\max}$. In the worst case, the range of t_{i+1} is restricted by the following inequality:

$$\frac{a_i - (\psi - \mu)}{t} \leq t_{i+1} \leq \frac{a_i + (\psi - \mu)}{t}.$$

We consider two interesting cases.

- Case I: $t/2+\xi+1 \le \psi \le t-(\xi+1)$.
- Case II: $t+2(\xi+1) \le \psi \le 2t-2(\xi+1)$.

5.4.1 Case I: $t/2+\xi+1 \le \psi \le t-(\xi+1)$

We assume that ψ satisfies the following:

$$\frac{t}{2}+\xi+1 \le \psi \le t-(\xi+1). \tag{5.22}$$

We plot a graph of t_{i+1} versus a_i in Figure 5.1. Note that $t_{i+1} \in \mathbb{Z}$ must be chosen within the parallelogram ABCD. It is easy to show that there always exists an integral value of a transfer digit t_{i+1} for all a_i in $[-2\psi, 2\psi]$, since the following inequalities are immediate from (5.22).

1. $1 \ge (\psi+\mu)/t$: this is the reason why we can choose $t_{\max}=1$.
2. $\psi-\mu \ge t-(\psi-\mu)$.

Now we can define $C(\cdot)$ as follows:

$$C(x) = \begin{cases} -1 & (\text{if } \le x < -\mathcal{C}), \\ 0 & (\text{if } -\mathcal{C} \le x < \mathcal{C}), \\ 1 & (\text{if } \mathcal{C} \le x), \end{cases} \tag{5.23}$$

where $\mathcal{C}$ is any positive integer in the range $[t-(\psi-\mu), \psi-\mu]$.

Even though any $\psi > t/2+\xi$ can be used to represent any element in $\mathbb{Z}_p$ as shown in Proposition 5, only ψ that satisfies (5.22) allows carry/borrow free addition/subtraction. If ψ is chosen in the range (5.22), one can use Algorithm 5.4 to perform final subtractions required at the end of Algorithm 5.2. In Section 5.3.1, we have observed that $||\vec{x_n}||_\infty < (\lambda+1)t+\xi$. Therefore, at most $\lfloor((\lambda+1)t+\xi)/\psi\rfloor$ executions of Algorithm 5.4 are required to reduce the result to $[-\psi,\psi]$. It is advantageous to choose $\psi = t-(\xi+1)$, to reduce the number of executions of Algorithm 5.4.

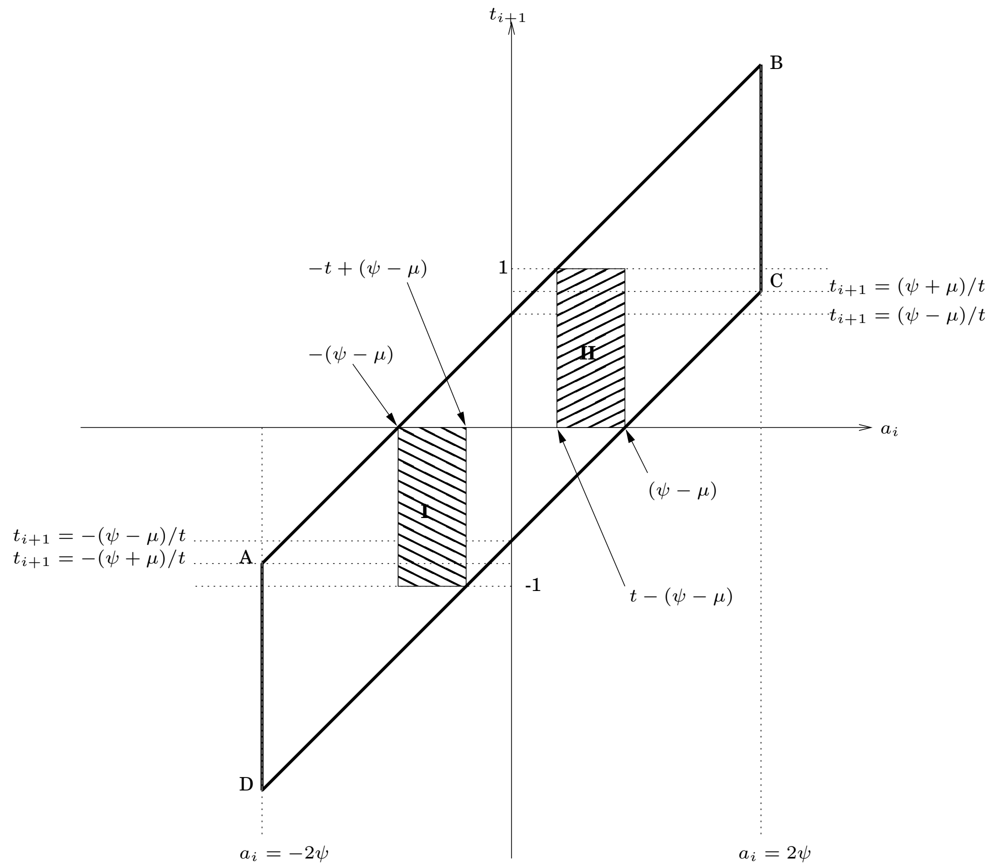

Figure 5.1: Range of Transfer Digit t_{i+1} ($\mu = \xi + 1$)

5.4.2 Case II: $t + 2(\xi + 1) \leq \psi \leq 2t - 2(\xi + 1)$

The conditions required to perform Algorithm 5.2 without final subtractions that we have discussed in Section 5.3.2 result in $t + \xi \leq \Psi$. Hence, this condition results in an overly redundant signed-digit representation. For completeness, we study the conditions to perform carry/borrow free additions/subtractions in overly redundant signed-digit representation.

Suppose $\psi = \Psi$ is an integer such that the following holds:

$$t + 2(\xi + 1) \leq \Psi \leq 2t - 2(\xi + 1). \tag{5.24}$$

We plot a graph of t_{i+1} versus a_i in Figure 5.2. Note that $t_{i+1} \in \mathbb{Z}$ must be chosen within the parallelogram ABCD. It is easy to show that there always exists an integral value of a transfer digit t_{i+1} for all a_i in $[-2\Psi, 2\Psi]$, since the following inequalities are immediate from (5.24).

1. $(\Psi + \mu)/t \leq 2$.
2. $1 \leq (\Psi - \mu)/t$: due to this, $t_{\max} = 1$ cannot be chosen.
3. $\Psi - \mu \geq t$.
4. $2t - (\Psi - \mu) \leq t$.
5. $-t + (\Psi - \mu) \geq 0$.

Since it is better to have smaller set of digits for $C(\cdot)$, we have chosen $t_{\max} = 2$. Observe that t is always within the range $2t - (\Psi - \mu) \leq t \leq \Psi - \mu$. Therefore, $C(\cdot)$ can be defined as follows:

$$C(x) = \begin{cases} -2 & (\text{if } x < -t), \\ 0 & (\text{if } -t \leq x < t), \\ 2 & (\text{if } t \leq x). \end{cases} \tag{5.25}$$

Note that $C(\cdot)$ defined in (5.25) does not require a pre-determined constant, unlike (5.23) which uses a constant $\mathcal{C}$. Moreover, $C(\cdot)$ does not output ± 1. Hence, computing $C(\cdot)$ as shown in (5.25) is as fast as computing $C(\cdot)$ as in (5.23).

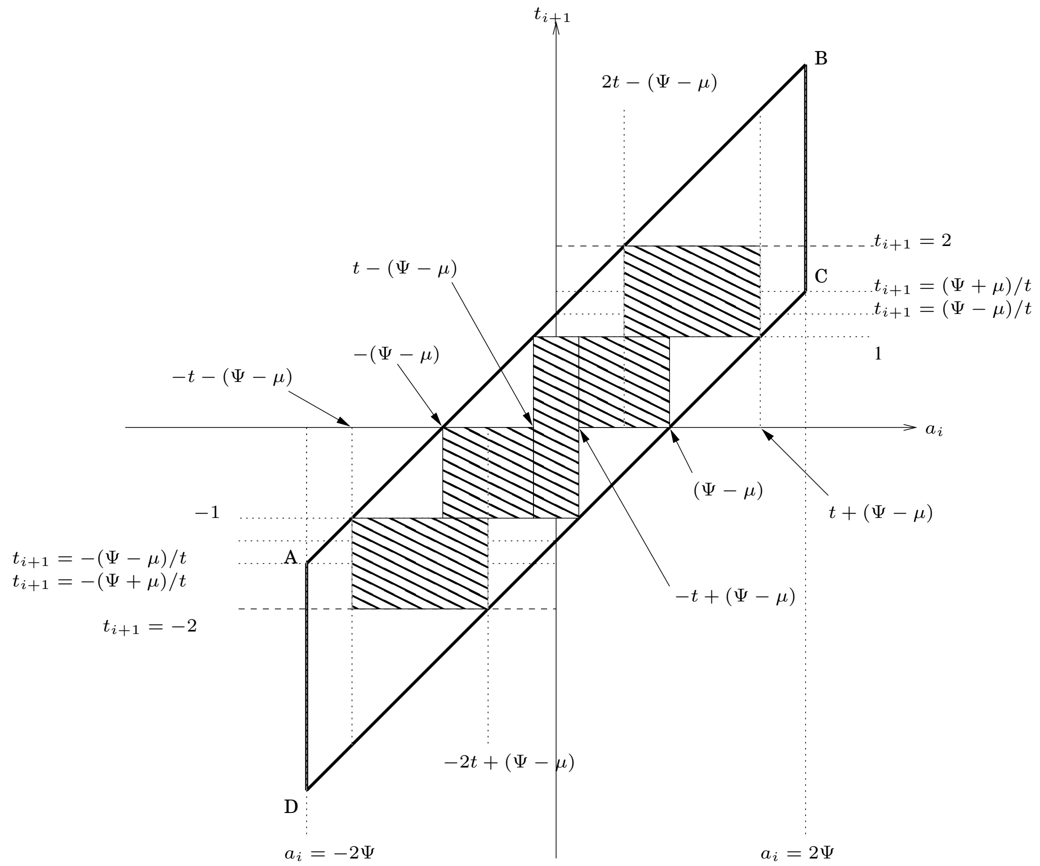

Figure 5.2: Range of Transfer Digit t_{i+1} ($\mu=2(\xi+1)$)

Table 5.1: Comparison of Algorithm 2.4 and Algorithm 5.2

Instruction	Algorithm 2.4	Algorithm 5.2
`mul`	$(nl+1)^2$	$ln^2 + (\tau k + 3l - 1)n + 2l + \tau k - 1$
`add`	$2(nl+1)$	$(n+1)(5l + 2\tau - 3)$
`adc`	$2n^2l^2 + 2nl$	$2ln^2 + (2\tau k + 3l - 1)n + 2\tau k + l - 1$

5.5 Comparisons

In this section, we compare our new coefficient reduction algorithm with Algorithm 2.4. For fairness of comparison, we let the modulus m used in Algorithm 2.4 be an nl-digit integer. Note that a degree-l polynomial $f(t)$ with n-digit t generates an nl-digit LWPFI.

We use the same technique that we use in Section 5.2.2 to analyze Algorithm 2.4. In line 3, only one `mul` is required. In line 4, the computation of $u_i \cdot m$ requires nl `mul`, 1 `add` and $(nl-1)$ `adc` instructions. Adding $u_i \cdot m$, which is at most $(nl+1)$ digits long, to T_i requires 1 `add` and $(nl+1)$ `adc` instructions. Therefore, Algorithm 2.4 requires the following number of instructions, not considering the final subtraction:

$$\#\texttt{mul} = q(nl+1),$$
$$\#\texttt{add} = 2q,$$
$$\#\texttt{adc} = 2qnl.$$

Note that q in Algorithm 2.4 is not the same as the one used in Algorithm 5.2. Final subtractions in Algorithm 2.4 can be avoided in the case $b \geq 4$ by simply letting $q = nl + 1$. In Algorithm 5.2, we suppose $q = n + 1$ eliminates the necessity of final subtractions. Note that such a value for q can be chosen if ξ is reasonably small. The details on this have been discussed at the end of Section 5.3.2.

In Table 5.1, we clearly observe that Algorithm 2.4 requires $O(n^2l^2)$ operations, whereas Algorithm 5.2 requires $O(ln^2)$ operations. Hence, Algorithm 5.2 does have better asymptotic behavior than Algorithm 2.4. However, this does not mean that Algorithm 5.2 is always faster than Algorithm 2.4. If actual values for parameters n, l, τ and k are substituted in Table 5.1, the required number of operations for Algorithm 5.2 may be larger. However, it is clear that the larger n and l, the better Algorithm 5.2 will perform.

5.6 Applications of LWPFI Modular Multiplications

Many cryptosystems rely on the ability to perform modular arithmetic modulo large integers. Among the modular arithmetic, modular multiplication is the most frequently used operation. In most cases, the modulus has to be a prime number. One can randomly try t until $f(t)$ is a prime to use it in cryptosystems requiring modular multiplications. One may find t such that $f(t)$ has a large enough prime factor suitable for certain cryptosystems. We denote such a factor p'. In this case, we can embed any modular arithmetic modulo p' in slightly larger ring $\mathbb{Z}_{f(t)}$, where we can use efficient modular multiplications using LWPFI moduli. Note, however, this method is faster only if the modular multiplication using LWPFI is faster than the modular multiplication modulo p' using usual integer arithmetic. After all computations have been performed, the result must be converted to the usual representation of integers and be taken modulo p'.

The idea of embedding arithmetic into a larger ring, where computations are easy, is not at all new. The similar technique is used for efficient multiplication in finite fields [88] [22].

5.7 Application to Modular Number Systems?

In this section, we investigate if Algorithm 5.2 can be applied also in modular number systems (MNS) proposed in [5, 6] by Bajard et. al.

Definition 6 (Modular Number System)**.** *A Modular Number System (MNS) $\mathcal{B}$, is a quadruple (p, l, γ, ρ), such that all positive integers $0 \leq x < p$ satisfy*

$$x = \sum_{i=0}^{l-1} x_i \gamma^i \bmod p, \textit{ with } \gamma > 1 \textit{ and } |x_i| < \rho \tag{5.26}$$

The MNS has some similarities with LWPFIs. For instance, numbers are represented in polynomials and the steps for modular multiplications are quite similar: multiplication is first performed in $\mathbb{Z}[t]/f(t)$ and then the coefficients are reduced by a coefficient reduction algorithm. However, the coefficient reduction method used in MNS is different from the one used in LWPFI modular multiplication. The difference is mainly due to the fact that γ can lie in much wider range than t. In particular, $0 \leq \gamma < p$, while the size of t used in LWPFI is approximately that of $p^{1/l}$.

As a special case of MNS, adapted modular number system (AMNS) is proposed in [5]. AMNS is an MNS where $\gamma^l \bmod p = c$ for some small integer c. Each iteration of the coefficient reduction algorithm (Algorithm CR in [5]) reduces $\lceil 3s/2 \rceil$ bits to $s+1$ bits. Some repetition of CR produces polynomial whose coefficients are at most $s+1$ bits long. Modular multiplication in AMNS is very efficient, however, it appears that the suitable sets of parameters that allows efficient computation in AMNS are quite difficult to find and are scarce.

In a recent paper [6], Bajard et. al. have proposed another special case of MNS called the polynomial modular number system (PMNS).

Theorem 7 (Fundamental theorem of an MNS). *Let* $p, l > 1$. *Also define* $E(X) = X^l + \alpha X + \beta$, *with* $\alpha, \beta \in \mathbb{Z}$, *such that* $E(\gamma) \equiv 0 \pmod{p}$, *and* E *irreducible in* $\mathbb{Z}[X]$. *Then, we can define a modular number system* $\mathcal{B} =$ *MNS*(p, l, γ, ρ) *provided that*

$$\rho \geq (|\alpha| + |\beta|)p^{1/l}. \tag{5.27}$$

Definition 7. *An MNS,* $\mathcal{B} =$ *MNS*(p, l, γ, ρ)*, which satisfies the conditions of Theorem 7 is called a Polynomial Modular Number System (PMNS).*

The coefficient reduction algorithms presented in [6] reduces one bit at a time in each iteration using a look-up table.

To perform the Montgomery reduction algorithm for MNS , we need an $l \times l$ matrix having similar properties as the matrix F used in Algorithm 5.2. However, we have not been able to find such a matrix for MNS. We have considered two candidate matrices, but they do not lead to satisfactory results.

Consider a matrix A as follows.

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ -\gamma & 1 & \cdots & 0 & 0 & 0 \\ 0 & -\gamma & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & 0 \\ 0 & 0 & \cdots & -\gamma & 1 & 0 \\ 0 & 0 & \cdots & 0 & -\gamma & p \end{bmatrix} \tag{5.28}$$

The matrix A is constructed similarly as (5.7) except for the last column. The column vectors

in A are all congruent to 0 modulo p and A has inverse modulo b if and only if $\gcd(p, b) = 1$ and $p \neq 0$. Hence, the matrix A is a good candidate for the matrix F in Algorithm 5.2, to compute $x(t) \cdot b^{-q} \bmod p$. Unfortunately, the entries in A are too large ($0 \leq \gamma < p$) to compute line 4 of Algorithm 5.2 efficiently. Due to the same reason, it is clear that Algorithm 5.2 will not reduce the coefficients of input polynomial if F is given as (5.28), regardless of the choice of q.

Let $\mathcal{L}(A)$ denote the lattice generated by the column vectors of A. Let v be a vector in $\mathcal{L}(A)$ such that $||\vec{v}||_\infty \leq p^{1/l}$. Note that there is always a vector $\vec{v}$ such that $||\vec{v}||_\infty \leq \sqrt[l]{\det A}$ in any lattice $\mathcal{L}(A)$. We construct another lattice $\mathcal{L}(B)$ generated by B, where

$$B = \begin{bmatrix} v_{l-1} & v_{l-2} & \cdots & (v_0 - \alpha v_{l-1}) \\ \vdots & \vdots & \vdots & \vdots \\ v_2 & v_1 & \cdots & (-\beta v_3 - \alpha v_2) \\ v_1 & v_0 - \alpha v_{l-1} & \cdots & (-\beta v_2 - \alpha v_1) \\ v_0 & -\beta v_{l-1} & \cdots & -\beta v_1 \end{bmatrix}. \tag{5.29}$$

Let $\vec{b_i}$ denote the i-th column vector of B. Then, $\vec{b_i} = \gamma^i \vec{v} \bmod (\gamma^l + \alpha\gamma + \beta)$. It is easily seen that $\max(||\vec{b_i}||_\infty) \leq (|\alpha| + |\beta|)p^{1/l}$. Clearly, $L(B) \subseteq L(F)$. Therefore, the matrix B is a good candidate for the matrix F in Algorithm 5.2. If $F = B$ in Algorithm 5.2, the algorithm will reduce the output to a certain extent. However, efficient coefficient reduction is still not achievable since computing $B \cdot \vec{u_i}$ (at line 4 of Algorithm 5.2) is not any faster than computing general matrix-vector product.

5.8 Conclusions

In this chapter, we have extended LWPFIs presented in Chapter 4, and have proposed a new coefficient reduction reduction based on Algorithm 2.4. Our new coefficient reduction algorithm have been analyzed using the extended definition of LWPFIs. Performance have been analyzed in terms of the number of digit-level multiplications, additions/subtractions and additions/subtractions with carry. Bounds on input and output of the new coefficient reduction algorithm have been carefully analyzed to eliminate the final subtractions in the new coefficient reduction algorithm. As a side result, we have presented methods for performing additions and subtractions modulo an LWPFI in a carry/borrow-free manner. We

have also considered applying our coefficient reduction algorithm to modular number systems proposed by Bajard et. al. but have not been successful in finding a good F that can lead to efficient coefficient reduction.

Chapter 6

Asymmetric Squaring Formulae

In this chapter, we present new 3, 4 and 5-way squaring formulae based on the Toom-Cook multiplication algorithm. To the best of our knowledge, previously known squaring algorithms use multiplication algorithms with two identical inputs. Such algorithms are symmetric in the sense that they use only squaring for multiplying coefficients. However, our squaring formulae are asymmetric and use at least one multiplication. Our squaring algorithms are not advantageous for large operand sizes, but have much less overhead than any other squaring algorithms. In most practical cases, the schoolbook method performs the best for small operands. Therefore, medium size operands, our squaring algorithm is likely to be faster than other algorithms. Our experimental results show that one of our 3-way squaring algorithms outperforms the squaring function in GNU multiprecision library v4.2.1 for some ranges of input sizes.

6.1 Further Details on the Toom-Cook Multiplication Algorithm

In Section 2, we have reviewed various multiplication algorithms including the Toom-Cook multiplication algorithm. In this section, we look into details on the Toom-Cook algorithm, especially on its interpolation step. The interpolation step can be easily performed by using

the Lagrange interpolation polynomial.

$$C(x) = \sum_{j=1}^{2n-1} C_j(x),$$

where

$$C_j(x) = C(x_j) \prod_{1 \le k \le 2n-1, k \ne j} \frac{x - x_k}{x_j - x_k}.$$

Alternatively, the Chinese remainder theorem (CRT) can be used. We can view the evaluation of a polynomial at point x_i as computing modulo a linear polynomial $(x - x_i)$, since computing $C(x) = A(x)B(x) \bmod (x - x_i)$ for $i = 1, \ldots, 2n-1$ is equivalent to computing $C(x_i)$'s. The CRT can combine the $(2n-1)$ distinct equivalence relations to compute the unique polynomial $C(x)$.

$$C(x) = \sum_{i=1}^{2n-1} C(x_i) M_i M_i',$$

where,

$$\begin{aligned} M &= \prod_{i=1}^{2n-1} (x - x_i), \\ M_i &= M/(x - x_i), \\ M_i' &= M_i^{-1} \bmod (x - x_i) = \frac{1}{\prod_{1 \le j \le 2n-1, i \ne j} (x_i - x_j)}. \end{aligned} \tag{6.1}$$

Even though the description of the CRT method looks quite different from the Lagrange interpolation method, the actual computation of the former is exactly the same as that of the latter, since $\prod_{1 \le k \le 2n-1, k \ne j} \frac{x - x_k}{x_j - x_k} = M_i M_i'$. Note that, in (6.1), M_i's and M_i''s can be pre-computed, for fixed x_i's.

By noticing that the interpolation step in Algorithm 2.1 solves a system of $(2n-1)$ linear equations with $(2n-1)$ unknown values (the coefficients of $C(x)$), we can construct the

following linear system:

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^{2n-2} \\ 1 & x_2 & \cdots & x_2^{2n-2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{2n-1} & \cdots & x_{2n-1}^{2n-2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-2} \end{bmatrix} = \begin{bmatrix} C(x_1) \\ C(x_2) \\ \vdots \\ C(x_{2n-1}) \end{bmatrix}. \tag{6.2}$$

The $(2n-1) \times (2n-1)$ matrix on the left hand side of (6.2) is called the Vandermonde matrix. We denote it by V. A Vandermonde matrix has a known determinant, $D = \prod_{1 \leq j < i \leq 2n-1}(x_i - x_j)$. The system (6.2) is uniquely solvable, since x_i's are all distinct in Algorithm 2.1. Computing the inverse matrix can be pre-computed for a fixed set of x_i's. Therefore, the coefficients c_i's can be easily obtained by multiplying the inverse matrix to the both sides of (6.2). Zuras uses this approach for 3, 4 and 5-way Toom-Cook multiplication algorithms [90]. However, this interpolation method is hardly useful in practice. It requires at most n^2 constant multiplications and at most n constant divisions for matrix-vector product. Among such constant divisions, at least one divisor must have an odd, nontrivial factor if $n > 2$.

Theorem 8. *There is no set of distinct integers $\{x_1, \ldots, x_s\}$'s such that $D = \det V$ is a power of 2; $D = \prod_{1 \leq j < i \leq s}(x_i - x_j) = \pm 2^k$ for some positive integer k, if $s > 3$.*

Proof. We need to show that there exists at least one pair (x_i, x_j), where $i \neq j$, such that $|x_i - x_j|$ is not a power of 2, if $s > 3$. We prove this by contradiction. We suppose we have a set of $s \geq 4$ distinct integers $\{x_1, \ldots, x_s\}$ such that D is a power of 2. Then, without loss of generality, x_2, x_3 and x_4 can be expressed as follows:

$$\begin{aligned} x_2 &= x_1 + (-1)^{u_1} 2^{v_1}, \\ x_3 &= x_1 + (-1)^{u_2} 2^{v_2}, \\ x_4 &= x_1 + (-1)^{u_3} 2^{v_3}, \end{aligned}$$

where u_i's are 0 or 1 and $v_i \in \mathbb{N} \cup \{0\}$. If $u_i = u_j$, then $v_i \neq v_j$. There are two cases:

1. All u_i's are the same: $u_1 = u_2 = u_3$.

It follows that

$$|x_2 - x_3| = |2^{v_1} - 2^{v_2}|,$$
$$|x_3 - x_4| = |2^{v_2} - 2^{v_3}|,$$
$$|x_4 - x_2| = |2^{v_3} - 2^{v_1}|.$$

Note that v_i's must be distinct, since otherwise x_i's are not distinct. The value $|2^{v_1} - 2^{v_2}|$ can be a power of 2 if and only if $|v_1 - v_2| \leq 1$. But $v_1 \neq v_2$. Therefore, $v_1 = v_2 \pm 1$. Without loss of generality, we can let $v_1 = v_2 + 1$. Then there is no such v_3 that satisfies both $|v_3 - v_2| = \pm 1$ and $|v_3 - v_2 - 1| = \pm 1$. If $v_3 - v_2 = 1$, then $|v_3 - v_1| = |v_3 - v_2 - 1| = 0$. If $v_3 - v_2 = -1$, then $|v_3 - v_1| = 2$.

2. One of u_i's is different.

   Without loss of generality, we can let $u_1 = u_2 \neq u_3$. It follows that

$$|x_2 - x_3| = |2^{v_1} - 2^{v_2}|,$$
$$|x_3 - x_4| = |2^{v_2} + 2^{v_3}|,$$
$$|x_4 - x_2| = |2^{v_3} + 2^{v_1}|.$$

Note that $v_1 \neq v_2$ and v_3 may be equal to either one of v_1 or v_2. It is easily seen that the value $|x_3 - x_4|$ is a power of 2 if and only if $v_2 = v_3$. Suppose $v_2 = v_3$. However, $|x_4 - x_2|$ can not be a power of 2, since v_3 must be different from v_1. □

Theorem 9. *In the Toom-Cook multiplication algorithm, at least one nontrivial constant division must occur for $n > 2$.*

Proof. The $(2n-1)$-th row vector of V^{-1} is $(L_1, L_1, \ldots, L_{2n-1})$, where

$$L_i = \prod_{1 \leq j \leq 2n-1, j \neq i} \frac{1}{x_i - x_j}.$$

Hence, the inverse matrix V^{-1} must have entries whose denominators are factors of D. Moreover, L_i's have all the factors of D, since $D^2 = |\prod_{i=1}^{2n-1} L_i|$. Due to Theorem 8, the odd, nontrivial factor of D must divide at least one of $1/L_i$'s. Therefore, the Toom-Cook multiplication algorithm must have at least one nontrivial constant division in the interpolation

step, for $n > 2$. □

There are heuristic approaches for small n to reduce the number of constant divisions and its sizes as much as possible. Such methods perform elementary row operations on both sides of (6.2) until the system is solved, rather than multiplying the inverse matrix of V. For instance, Paul Zimmermann's implementation in GNU multiple precision library (GMP) v4.2.1 uses only one constant division by 3 for the 3-way Toom-Cook multiplication algorithm as shown in Section 2.1.1. We believe Zimmermann's method is by far the best 3-way Toom-Cook multiplication algorithm.

Even though there exist methods for fast exact division by a constant [38], [47], divisions by constants are very time consuming operation. Figure 6.1 shows the timing ratio of GMP's exact division by 3 to the fastest available large integer squaring. In GMP, exact division by 3 is implemented in the function `mpn_divexact_by3()` and `mpz_mul()` calls a squaring subroutine when it is called with equal operands. When timing `mpz_mul()` and `mpn_divexact_by3()`, we used $3u$-bit and $(2u + 6)$-bit operands, respectively. Note that, if the input size is $3u$-bit for the 3-way Toom-Cook multiplication algorithm, Algorithm 2.2 requires one exact division by 3 of at most $(2u+6)$-bit operand. We can easily observe that the exact division by 3 is very slow compared to the entire squaring operation for small operand sizes.

The choice of evaluation points is very important for Algorithm 2.1, since it significantly affects the performance of the Toom-Cook multiplication algorithm. Toom and Cook proposed $x_i \in \{-n + 1, \ldots, -1, 0, 1, \ldots, n - 1\}$ and $x_i \in \{0, 1, 2, \ldots, 2n - 2\}$, respectively. Knuth proposed the use of powers of 2 and their negatives [43], [8]. Winograd proposed ∞ as one of the evaluation points [87]. Winograd also noted that x_i's can be fractions, e.g., $x_i = p/q$, where evaluation at a rational point p/q means computing $q^{n-1}A(p/q)$. Note that it can be proven that the inclusion of ∞ and rational numbers for evaluation points does not change the fact that the determinant of the matrix in (6.2) must have factors other than 2 for $n > 2$. In 1994, Zuras proposed the use of reciprocally symmetric set, e.g., $x_i \in \{1, \infty, 0, 2, 1/2, -2, -1/2, \ldots\}$ [90]. Harley used the same evaluation points as Zuras ($\{1, \infty, 0, 2, 1/2\}$) in implementing the Toom-Cook multiplication routine in GNU multiple precision (GMP) arithmetic library version 4.1.4 [26]. He used simple row operations to solve the system (6.2) instead of multiplying an inverse matrix. He performed interpolation by using only one exact division by 3 for $n = 3$. Paul Zimmermann recently improved

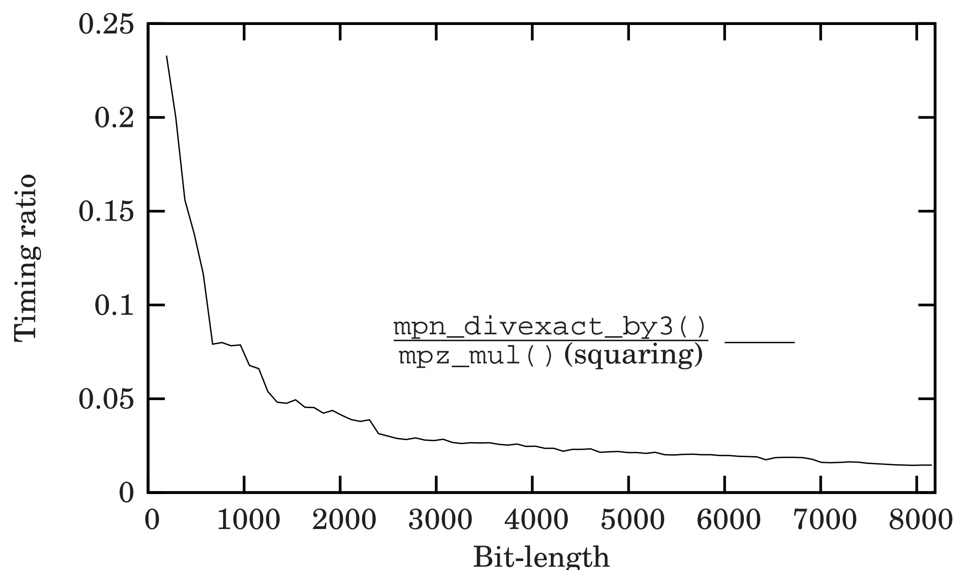

Figure 6.1: Timing Ratio of `mpn_divexact_by3()` to `mpz_mul()`

Harley's method using a simpler set: $x_i = \{0, 1, -1, 2, \infty\}$. This algorithm has been implemented in GMP 4.2.1 [26].

In [87], Winograd proves that Algorithm 2.1 uses the least possible number of coefficient multiplications. Unfortunately, the cost involved in evaluation and interpolation steps cannot be ignored even for small n. In fact, the evaluation and interpolation cost overwhelms the entire computation time for multiplying polynomials having only small coefficients. To reduce the amount of overhead, Winograd proposes the use of remainder arithmetic by modulo cyclotomic polynomials, whose zeros are on unit circle in complex domain. For example, he proposed using $x, (x-1), (x+1)$ and (x^2+1) for 3-way multiplications. The Winograd algorithm can be viewed as the Toom-Cook algorithm for $x_i = \{0, 1, -1, j, -j\}$. However, this method needs one more coefficient multiplication than the Toom-Cook algorithm, since $A(x) \cdot B(x) \bmod (x^2+1)$ requires three coefficient multiplications. However, it has an advantage that there is no constant division during the interpolation step.

6.2 New Squaring Formulae

To the best of our knowledge, no sub-quadratic multiplication algorithms reviewed in Section 2 have been considerably specialized for squaring. We attempt to fill in this gap in the literature. Of course, there is no squaring algorithm which is asymptotically faster than the fastest multiplication algorithm [90] and it is not a goal of this work to find such squaring algorithms.

In Section 6.1, we have seen that nontrivial constant divisions in the Toom-Cook algorithm are unavoidable. There are multiplication algorithms not requiring the constant division, but they use more than $(2n-1)$ coefficient multiplications. Winograd shows methods for avoiding such constant divisions and reducing overhead in interpolation, but it is always at the sacrifice of an increased number of coefficient multiplications [87]. NTT based multiplication algorithms do not require nontrivial constant divisions if N is a power of 2, but this means that N must be greater than $2n-1$.

However, squaring can be performed without the nontrivial constant division using exactly $(2n-1)$ multiplications, at least for $n = 3, 4$ and 5. In this section, we present three potentially useful explicit formulae for 3-way squaring that do not require a nontrivial constant division. Our new squaring algorithms are similar to the Toom-Cook multiplication algorithm, but we use different approach for constructing a linear system on c_i's to achieve faster evaluation and interpolation. This new approach allows us to find squaring formulae that do not require any nontrivial constant divisions. Our squaring formulae require only the theoretic minimum number of coefficient multiplications, which is five for 3-way multiplication. We present only one explicit formula for each of 4-way and 5-way squaring in Section 6.5.

All sub-quadratic multiplication algorithms we have reviewed in Section 2 are *symmetric* algorithms in the sense that all point-wise multiplications are squarings when $A(x) = B(x)$. On the other hand, our new squaring formulae are *asymmetric* algorithms, since they involve at least one point-wise multiplication of two different values.

Compared to the Toom-Cook multiplication algorithm, our algorithms are not advantageous for squaring very large size operands, since squaring operation is usually faster than multiplication operation for all ranges of operand sizes in practice. The schoolbook squaring algorithm performs better than the schoolbook multiplication algorithm. Similarly, the same holds true for symmetric sub-quadratic algorithms, since most implementations of

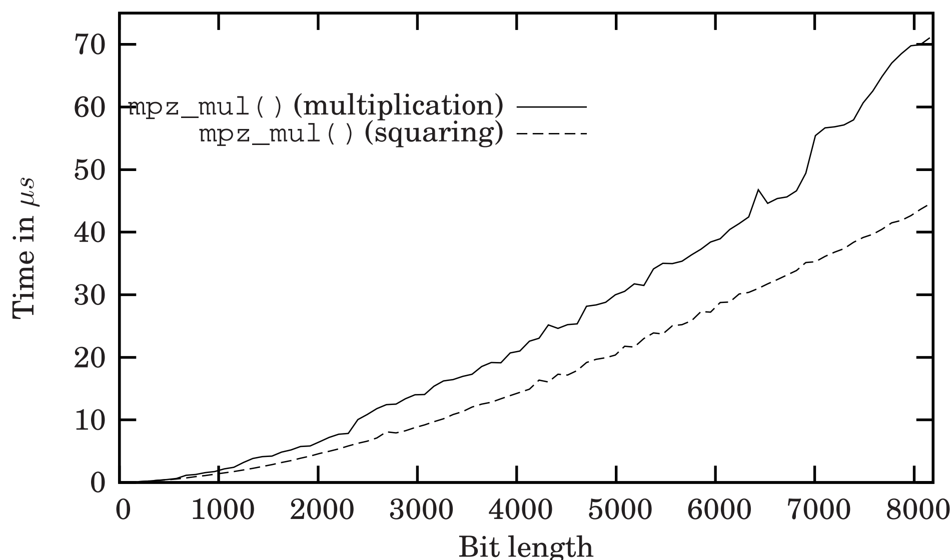

Figure 6.2: Timing Results of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring)

sub-quadratic algorithms use the schoolbook methods for the base case. Figure 6.2 shows the timing results of multiplication and squaring routines (both are called from `mpz_mul()`) in GMP.

The timing difference between multiplication and squaring is not significant for small operands, but the difference becomes larger as operand size grows. Hence, it easily follows that symmetric squaring algorithms are more advantageous for squaring very large size operands. However, there is a possibility that the proposed asymmetric algorithms are advantageous for squaring relatively small operand sizes, for which the effect of reduced overhead in evaluation and interpolation steps is greater than that of the lost efficiency in asymmetric point-wise multiplication step. In fact, our experimental results in Section 6.4, show that one of our squaring formulae performs better than the long integer squaring function in GMP for a certain range of operand sizes.

6.2.1 Our Approach

In Section 6.1, we have shown that the Toom-Cook multiplication algorithm requires $(2n-1)$ distinct evaluation points for constructing a system of $(2n-1)$ linear equations having $(2n-$

1) unknown values (the coefficients of $C(x) = A(x) \cdot B(x)$). As shown in Theorems 8 and 9, such a construction method always introduces at least one nontrivial constant division in the interpolation step. For $n = 3$, even the best known 3-way Toom-Cook multiplication method, shown in Section 2.1.1, requires one constant division by 3 during interpolation step.

To completely eliminate the constant divisions, we take a different approach for constructing a linear system. Below we give detailed methods for obtaining linear equations on c_i's that cannot be derived by directly evaluating $C(x) = A(x)^2$. Our approach allows us to find linear equations of c_i's such that the corresponding linear system does not involve a Vandermonde matrix.

1. Taking modulo $(x^2 + ux + v^2)$, where u and v are some integers: By taking modulo $(x^2 + ux + v^2)$ on both sides of $C(x) = A(x)^2$, we obtain

$$c'_1 x + c'_0 \equiv (a'_1 x + a'_0)^2 \pmod{(x^2 + ux + v^2)}, \tag{6.3}$$

where $a'_1 x + a'_0 = A(x) \bmod (x^2 + ux + v^2)$ and $c'_1 x + c'_0 = C(x) \bmod (x^2 + ux + v^2)$. Then it follows that

$$c'_1 x + c'_0 \equiv a'_1(2' a'_0 - u a'_1)x + (a'_0 - v a'_1)(a'_0 + v a'_1) \pmod{(x^2 + ux + v^2)}. \tag{6.4}$$

It is interesting to see that computing both c'_0 and c'_1 requires only two coefficient multiplications. Hence, we obtain two useful linear equations for c'_is as follows:

$$\begin{aligned} c'_1 &= a'_1(2' a'_0 - u a'_1), \\ c'_0 &= (a'_0 - v a'_1)(a'_0 + v a'_1). \end{aligned} \tag{6.5}$$

Therefore, by choosing some small integers u and v, we can obtain useful linear equations on c_i's. Such equations cannot be obtained by simply evaluating $C(x) = A(x)^2$. For example, suppose that $A(x)$ is a degree-2 polynomial, $A(x) = a_2 x^2 + a_1 x + a_0$. Let $u = 0$ and $v = 1$. We take modulo $(x^2 + 1)$ on both sides of $C(x) = A(x)^2$ and obtain

$$\begin{aligned} c_4 - c_2 + c_0 &= (a_0 - a_2)^2 - a_1^2 = (a_0 - a_2 + a_1)(a_0 - a_2 - a_1), \\ c_1 - c_3 &= 2a_1(a_0 - a_2). \end{aligned} \tag{6.6}$$

This method is different from the Winograd algorithm. In the Winograd algorithm, after taking modulo a second degree polynomial, c'_0 and c'_1 are simultaneously computed using KA which requires three coefficient multiplications [87]. However, the computation of c'_0 is independent from that of c'_1 in (6.6). Hence, we can select only one of the two linear equations in (6.6) without sacrificing the efficiency.
We remark that a special case of this idea is known for efficient implementation of finite field squaring in $\mathbb{Z}_{p^2}[x]/f(x)$ where $f(x) = x^2 + x + 1$ [77].

2. Hermite interpolation: Interpolation using the evaluations of derivatives is known as the Hermite interpolation. Interestingly, for squaring, each evaluation of the first derivative of $C(x)$ requires only one coefficient multiplication, since $C'(x) = 2A(x) \cdot A'(x)$.
Evaluating the first derivative of $C(x)$ gives linear relations, some of which may not be obtained by evaluating $C(x) = A(x)^2$. For example, when $A(x)$ is a degree-2 polynomial, the first derivative of $C(x) = A(x)^2$ results in the following:
$$4c_4x^3 + 3c_3x^2 + 2c_2x + c_1 = 2(a_2x^2 + a_1x + a_0)(2a_2x + a_1). \tag{6.7}$$
Some interesting evaluations of (6.7) are given below.
   (a) $x = 0$: $c_1 = 2a_0a_1$.
   (b) $x = \infty$: $c_4 = a_2^2$. (The same result can be obtained also by evaluating $C(x) = A(x)^2$ at $x = \infty$.)
   (c) $x = 1$: $4c_4 + 3c_3 + 2c_2 + c_1 = 2(a_2 + a_1 + a_0)(2a_2 + a_1)$.
   (d) $x = -1$: $-4c_4 + 3c_3 - 2c_2 + c_1 = 2(a_2 - a_1 + a_0)(-2a_2 + a_1)$.

   All of the above linear equations are reasonably simple and requires only one coefficient multiplication for each evaluation point.

3. $A(x_i)^2 - A(x_j)^2 = (A(x_i) + A(x_j)) \cdot (A(x_i) - A(x_j))$ for $x_i \neq x_j$.
Using this method, we can combine two distinct evaluations of $A(x)$ into one.

4. Duality: any function computing c_i can be used to compute c_{2n-1-i} with no changes. In other words, if $c_i = f(a_0, \ldots, a_{n-2}, a_{n-1})$, then $c_{2n-1-i} = f(a_{n-1}, \ldots, a_1, a_0)$ [78].
Hence, we can safely substitute c_i to c_{2n-1-i} and a_j to a_{n-1-j} for all $0 \leq i \leq 2n - 1$

and $0 \leq j \leq n-1$ in any linear equations on c_i's. For example, $c_3 = 2a_2a_1$ is a dual of $c_1 = 2a_0a_1$. This is a well-known fact and a similar argument holds for multiplications.

6.2.2 New 3-way Squaring

Let $\vec{C} = (c_4, c_3, c_2, c_1, c_0)$. To construct a 3-way squaring algorithm computing $C(x) = A(x)^2$ that requires only five coefficient multiplications, we need to find a five-tuple $(i_0, i_1, i_2, i_3, i_4)$, where i_j's are all distinct, such that

- There exists a u_{i_j}, which is a product of two elements (not necessarily distinct) from $L(A)$, for each vector $\vec{L}_{i_j}$ such that $\vec{L}_{i_j} \circ \vec{C} = u_{i_j}$, where $\circ$ is a dot product.
- The set of vectors $\{\vec{L}_{i_0}, \dots, \vec{L}_{i_3}, \vec{L}_{i_4}\}$ forms a basis in $\mathbb{Z}^5$.

Let $M = (\vec{L}_{i_4}, \dots, \vec{L}_{i_1}, \vec{L}_{i_0})^T$. If we can find a five-tuple $(i_0, \dots, i_3, i_4)$ which makes $\det M$ a power of 2, we get a squaring algorithm that require only 5 coefficient multiplications and no nontrivial constant divisions.

We have identified 20 potentially useful $\vec{L}_i$'s and u_i's by directly evaluating $C(x) = A(x)^2$ and by using our new construction methods given above, and show them in Table 6.1. Note that $\vec{L}_9$–$\vec{L}_{29}$ have been obtained using the methods given above and they cannot be obtained by simply evaluating $C(x) = A(x)^2$.

There are a total of 42504 possible combinations of $(i_0, i_1, i_2, i_3, i_4)$ and 34268 of them make $\{\vec{L}_{i_0}, \dots, \vec{L}_{i_3}, \vec{L}_{i_4}\}$ a linearly independent set. We divide these 34268 combinations into the following three sets:

Set I: there are three or more i_j's such that $i_j \geq 9$.

Set II: there are only two i_j's such that $i_j \geq 9$.

Set III: there is only one i_j such that $i_j \geq 9$.

Sets I, II and III have 13584, 5946 and 1012 combinations, respectively. In Set I, it is easily seen that combinations $(1,2,9,10,15)$, $(1,2,9,10,17)$, $(1,2,9,10,18)$, $(1,2,9,10,19)$ and $(1,2,9,10,20)$ lead to the simplest interpolation step. Note that $\vec{L}_1$, $\vec{L}_2$, $\vec{L}_9$, $\vec{L}_{10}$ immediately give the coefficients c_0, c_1, c_3 and c_4 of $C(x)$. The remaining coefficient c_2 can be obtained

Table 6.1: List of Candidate Vectors

i	$\vec{L}_i$	$u_i = \vec{L}_i \circ \vec{C}$	Comment
1	$(0,0,0,0,1)$	a_0^2	$C(0)$
2	$(1,0,0,0,0)$	a_2^2	$C(\infty)$
3	$(1,1,1,1,1)$	$(a_2+a_1+a_0)^2$	$C(1)$
4	$(1,-1,1,-1,1)$	$(a_2-a_1+a_0)^2$	$C(-1)$
5	$(16,8,4,2,1)$	$(4a_2+2a_1+a_0)^2$	$C(2)$
6	$(16,-8,4,-2,1)$	$(4a_2-2a_1+a_0)^2$	$C(-2)$
7	$(1,2,4,8,16)$	$(a_2+2a_1+4a_0)^2$	$2^4 \cdot C(1/2)$
8	$(1,-2,4,-8,16)$	$(a_2-2a_1+4a_0)^2$	$2^4 \cdot C(-1/2)$
9	$(0,0,0,1,0)$	$2a_0a_1$	$C'(0)$
10	$(0,1,0,0,0)$	$2a_1a_2$	Dual of 9
11	$(4,3,2,1,0)$	$2(a_2+a_1+a_0)(2a_2+a_1)$	$C'(1)$
12	$(-4,3,-2,1,0)$	$2(a_2-a_1+a_0)(-2a_2+a_1)$	$C'(-1)$
13	$(0,1,2,3,4)$	$2(a_2+a_1+a_0)(2a_0+a_1)$	Dual of 11
14	$(0,1,-2,3,-4)$	$2(a_2-a_1+a_0)(a_1-2a_0)$	Dual of 12
15	$(1,0,-1,0,1)$	$(a_0-a_2+a_1)(a_0-a_2-a_1)$	Constant term of $C(x) \bmod (t^2+1)$
16	$(0,-1,0,1,0)$	$2a_1(a_0-a_2)$	t's coefficient of $C(x) \bmod (t^2+1)$
17	$(-1,0,1,1,0)$	$(a_1-a_2+2a_0)(a_1+a_2)$	t's coefficient of $C(x) \bmod (t^2-t+1)$
18	$(0,-1,-1,0,1)$	$(a_0-a_1-2a_2)(a_0+a_1)$	Constant term of $C(x) \bmod (t^2-t+1)$
19	$(1,0,-1,1,0)$	$(a_2+a_1-2a_2)(a_2-a_1)$	t's coefficient of $C(x) \bmod (t^2+t+1)$
20	$(0,1,-1,0,1)$	$(a_0+a_1-2a_2)(a_0-a_1)$	Constant term of $C(x) \bmod (t^2+t+1)$
21	$(1,0,0,0,-1)$	$(a_0+a_2)(a_0-a_2)$	$A(0)^2-A(\infty)^2$
22	$(0,1,0,1,0)$	$2a_1(a_2+a_0)$	$(A(1)^2-A(-1)^2)/2$
23	$(0,4,0,1,0)$	$2a_1(4a_2+a_0)$	$(A(2)^2-A(-2)^2)/4$
24	$(0,1,0,4,0)$	$2a_1(4a_0+a_2)$	$4(A(1/2)^2-A(-1/2)^2)$

by at most two additions/subtractions. Among the five contenders, $(1,2,9,10,15)$ is the best choice, since computing u_{15} is easier than computing u_{17}, u_{18}, u_{19} and u_{20}.

In Set II, there are 124 combinations of $(i_0,\ldots,i_3,i_4)$ such that $|\det M| = 1$. To narrow down our search, we have considered only the combinations that lead to M such that the entries of M^{-1} are relatively small. Combinations $(1,2,3,9,10)$, $(1,2,4,9,10)$, $(1,2,4,9,22)$ and $(1,2,4,10,22)$ are the best, and they lead to the simplest form of M^{-1}. Combination $(1,2,4,9,10)$ is more advantageous than $(1,2,3,9,10)$, since computing u_4 is more efficient than computing u_3. Note that $(a_2+a_1+a_0)$ could be at most 1 bit longer than $(a_2-a_1+a_0)$. Moreover, $(1,2,4,9,10)$ is better than $(1,2,4,9,22)$ and $(1,2,4,10,22)$ since computing u_9 and u_{10} is faster than computing u_9 and u_{22} or computing u_{10} and u_{22}. We have also considered combinations that results in $|\det M| = 2,4,8$ and 16, but could not find a better combination than $(1,2,3,4,9)$ and $(1,2,3,4,10)$.

In Set III, there is no combination that makes $|\det M| = 1$, but there are 26 combinations that makes $|\det M| = 2$. Among these 26 combinations, $(1,2,3,4,9)$ and $(1,2,3,4,10)$ lead to the most efficient squaring algorithm. We have also considered combinations that result in $|\det M| = 4,8$ and 16, but could not find a better combination than $(1,2,3,4,9)$ and $(1,2,3,4,10)$.

We have derived three new squaring methods from Set I, II and III.

1. Squaring Method 1 (SQR$_1$)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_0-a_2+a_1)(a_0-a_2-a_1) \\ 2a_1a_0 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{6.8}$$

The computation of S_i's requires 3 coefficient multiplications and 2 coefficient squarings. The determinant of the 5×5 matrix in (6.8) is -1, meaning the interpolation can be performed without bit shift or constant division. In fact, the coefficients c_0, c_1, c_3 and c_4 are already given. The coefficients c_2 can be computed with one addition and one subtraction: $c_2 = S_0 + S_4 - S_2$.

2. Squaring Method 2 (SQR_2)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_2 - a_1 + a_0)^2 \\ 2a_1a_0 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{6.9}$$

This algorithm requires 2 coefficient multiplications and 3 coefficient squarings. The coefficients c_0, c_1, c_3 and c_4 are already given. The remaining coefficient c_2 can be obtained using only 4 additions/subtractions: $c_2 = S_2 + S_1 + S_3 - S_0 - S_4$.

3. Squaring Method 3 (SQR_3)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_2 - a_1 + a_0)^2 \\ (a_2 + a_1 + a_0)^2 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{6.10}$$

This algorithm requires 1 coefficient multiplication and 4 coefficient squarings. The coefficients c_0, c_3 and c_4 are already given. The coefficients c_1 and c_2 are computed using only 5 additions/subtractions and 1 bit shift.

$$\begin{aligned} T_1 &= (S_1 + S_2)/2, \\ c_1 &= S_1 - T_1 - S_3, \\ c_2 &= T_1 - S_4 - S_0. \end{aligned} \tag{6.11}$$

6.3 Analysis

In this section, we analyze the squaring algorithms SQR_1, SQR_2 and SQR_3 presented in Section 6.2. The analysis may differ depending on how the various squaring algorithms are used in specific applications (e.g., long integer squaring, squaring in extension field $GF(p^m)$, polynomial squaring in $\mathbb{Z}[x]$,...). In this section, we assume that our squaring formulae are

applied to the arithmetic in $\mathbb{Z}[x]$. However, the results shown in this section are relevant to other applications. For applications in $GF(p^m)$, after the polynomial squaring has been completed, one needs to perform reduction by an irreducible polynomial for $GF(p^m)$ and then reduce each coefficient modulo p. These reduction operations are not dependent on the algorithm used for the polynomial squaring. For long integer squaring, an integer is interpreted as a polynomial and a polynomial squaring is performed. Then, one needs to overlap the coefficients and perform carry propagations. The overlapping and carry propagation is also not related to the algorithm used for the polynomial squaring.

We compare our algorithms with other known 3-way squaring algorithms: schoolbook squaring algorithm, 3-way KA-like formula and Zimmermann's 3-way Toom-Cook algorithms.

We denote the digit size of the representation by b. Addition or subtraction of two u-digit integers requires $\mathcal{A}(u)$ time. Multiplication and squaring of two u-digit integers require $\mathcal{M}(u)$ and $\mathcal{S}(u)$ times, respectively. Bit shift of u digit integers require $\mathcal{B}(u)$ time. During evaluation and interpolation step, there are cases when the operands to addition/subtraction and shift are a few bits larger than u or $2u$ digits, where the coefficients of $A(x)$ are at most u digits long. For simplicity, we ignore this overhead caused by carries and borrows. However, we do not ignore the overhead involved in multiplying two integers that are slightly longer than u digits. For example, an integer s and t are only 1-bit longer than u digits. Then we can write $s = s_h b + s_l$ and $t = t_h b + t_l$, where $|s_h|$, $|t_h| \leq 1$ and $0 \leq s_l$, $t_l < b^u$. The time required to compute $s \cdot t$ is at most $\mathcal{M}(u) + 2\mathcal{A}(u)$. For simplicity, we ignore the cost for multiplying carries, i.e., s_h and t_h.[1] The time required to compute s^2 is $\mathcal{S}(u) + \mathcal{B}(u) + \mathcal{A}(u)$ in the worst case. When computing a product $2a_ia_j$, we always compute a_ia_j first and then perform the bit shift later. It is reasonable to assume that $\mathcal{A}(\cdot)$ and $\mathcal{B}(\cdot)$ are linear functions; $\mathcal{A}(fu + gv) = f\mathcal{A}(u) + g\mathcal{A}(v)$ and $\mathcal{B}(fu + gv) = f\mathcal{B}(u) + g\mathcal{B}(v)$. The exact division by 3 of an u-digit integer used in the 3-way Toom-Cook algorithm shown in Section 2.1.1 is denoted by $\mathcal{D}_3(u)$.

We assume that $A(x) = a_2x^2 + a_1x + a_0$ is the input, where a_i's are u digits long. Table 6.2 shows our analysis results. Table 6.3 shows the conditions for which our squaring

[1] Note, however, that the 3-way Toom-Cook multiplication algorithm in GMP v4.2.1 stores carries and borrows in the most significant digit place instead of handling them separately with extra variables. This method has a trade-off. Using extra digit reduces the number of additions and subtractions, but coefficient multiplications and squarings becomes slower. We have tested both methods and found that it is better to use extra variables for carries and borrows on architectures on which we have performed our experiments.

Table 6.2: Analysis Results of Various Squaring Algorithms

Algorithm	$\mathcal{S}\&\mathcal{M}$	Overhead
3-way Toom-Cook	$5\mathcal{S}(u)$	$14\mathcal{B}(u) + 25\mathcal{A}(u) + \mathcal{D}_3(2u)$
Schoolbook sqr.	$3\mathcal{S}(u) + 3\mathcal{M}(u)$	$6\mathcal{B}(u) + 2\mathcal{A}(u)$
3-way KA-like	$6\mathcal{S}(u)$	$3\mathcal{B}(u) + 20\mathcal{A}(u)$
SQR_1	$2\mathcal{S}(u) + 3\mathcal{M}(u)$	$5\mathcal{B}(u) + 9\mathcal{A}(u)$
SQR_2	$3\mathcal{S}(u) + 2\mathcal{M}(u)$	$5\mathcal{B}(u) + 11\mathcal{A}(u)$
SQR_3	$4\mathcal{S}(u) + 1\mathcal{M}(u)$	$6\mathcal{B}(u) + 15\mathcal{A}(u)$

Table 6.3: Conditions for Which SQR_i's Are Faster Than Other 3-way Algorithms

i	SQR_i vs. 3-way Toom-Cook
1	$3\mathcal{M}(u) < 3\mathcal{S}(u) + 9\mathcal{B}(u) + 16\mathcal{A}(u) + \mathcal{D}_3(2u)$
2	$2\mathcal{M}(u) < 2\mathcal{S}(u) + 9\mathcal{B}(u) + 14\mathcal{A}(u) + \mathcal{D}_3(2u)$
3	$\mathcal{M}(u) < \mathcal{S}(u) + 8\mathcal{B}(u) + 10\mathcal{A}(u) + \mathcal{D}_3(2u)$

i	SQR_i vs. Schoolbook sqr.
1	$7\mathcal{A}(u) < S(u) + \mathcal{B}(u)$
2	$9\mathcal{A}(u) < \mathcal{M}(u) + \mathcal{B}(u)$
3	$\mathcal{S}(u) + 13\mathcal{A}(u) < 2\mathcal{M}(u)$

i	SQR_i vs. 3-way KA-like
1	$3\mathcal{M}(u) + 2\mathcal{B}(u) < 4\mathcal{S}(u) + 11\mathcal{A}(u)$
2	$2\mathcal{M}(u) + 2\mathcal{B}(u) < 3\mathcal{S}(u) + 9\mathcal{A}(u)$
3	$\mathcal{M}(u) + 3\mathcal{B}(u) < 2\mathcal{S}(u) + 5\mathcal{A}(u)$

algorithms are superior to the other algorithms.

Table 6.3 shows that there is apparently no single algorithm that is absolutely superior to the others. Without considering the actual values $\mathcal{S}(u)$, $\mathcal{M}(u)$, $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$, which are very application specific, it is not easy to decide which algorithm is faster than the rest. However, one thing that is clear from Table 6.3 is that the 3-way Toom-Cook algorithm becomes the best for squaring polynomials as u increases. The timings $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$ grow linearly with u, but timings of multiplication ($\mathcal{M}(u)$) and squaring ($\mathcal{S}(u)$) grow quadratically or sub-quadratically depending on the methods used for point-wise multiplications. It is obvious that, for large u, the effect of reduced overhead in our algorithms will be offset by the timing difference in multiplication and squaring.

However, SQR_i's have very little amount of overhead compared to the 3-way Toom-Cook multiplication algorithm. Hence, it is possible that, for some small u, the timing difference

of multiplication and squaring is small enough that some of the conditions in Table 6.3 will be satisfied. In fact, our implementation results given in Section 6.4 confirm that there is a range of u where some conditions of Table 6.3 are satisfied.

6.4 Implementation Results

To verify the practical usefulness of our algorithms, we have implemented in software the functions for large integer squaring, and the functions for degree-2 polynomial squaring in $\mathbb{Z}[x]$ using SQR_1, SQR_2 and SQR_3 presented in Section 6.2. Our experiments have been performed on Linux (kernel version 2.6.15.26) running on Intel Pentium IV Prescott 3.2GHz, Pentium II MMX 300MHz, Pentium III Mobile 1.13GHz. We have used GCC 4.0.3 to compile all programs. We have compiled GMP library v4.2.1 in two passes. Between the first and the second passes of compilations, we have performed GMP's tuneup program (with an option '-p 100000000' for better precision than default) so that GMP uses the optimal threshold values between multiplication algorithms. We compiled all our source codes using the same compiler options used for compiling GMP library. We have ensured that our program does not link with the shared library of GMP, since shared libraries have a performance penalty due to the runtime address resolution[2]. The testing program has been run at the highest priority to minimize the risk of interference by other running processes.

6.4.1 Application to Large Integer Squaring

We have compared our implementation of SQR_i's with GMP's squaring function. For fair comparison, our algorithms have been written so that it can replace `mpn_toom3_sqr_n()` function in GMP. Note that `mpn_toom3_sqr_n()` is a low level implementation of the algorithm shown in Section 2.1.1 for squaring case.[3]

Our squaring algorithms have been written using the same coding style that Harley used for implementing `mpn_toom3_sqr_n()` in GMP 4.1.4. Note that Zimmermann's algorithm is theoretically better than Harley's, but the implementations of the two algorithms in GMP v4.1.4 and v4.2.1 use different coding styles. Harley stores the carries in separate variables in GMP 4.1.4, but Zimmermann stores them in the most significant digit place in GMP

[2] I thank Augusto Jun Devegili letting me know about the runtime address resolution.

[3] Even though the 3-way Toom-Cook squaring is separately implemented, it uses the same 3-way Toom-Cook multiplication algorithm given in Section 2.1.1.

4.2.1. Zimmermann's method increases the digit length of input to coefficient multiplications, additions and subtractions. However, such a method reduces the number of functions calls to additions and subtractions. Since it is not fair to compare two different algorithms written in different coding style, we have implemented Zimmermann's algorithm presented in Section 2.1.1 using Harley's coding style. Our implementation ourperformed Zimmermann's impelemtation in GMP v4.2.1 on Pentium II MMX 350MHz, III Mobile 1.13GHz, IV 3.2GHz. We have also implemented SQR_1, SQR_2 and SQR_3 in both ways and found that Harley's coding style is always better. Therefore, our implementation results and comparisons in the following are based on Zimmermann's 3-way Toom-Cook squaring and our new squaring algorithms both written in Harley's style. We provide the source code of our improved Zimmermann's 3-way Toom-Cook squaring in Appendix A.

We have not used any optimization tricks or special functions other than those used in `mpn_toom3_sqr_n()`. We have ensured that our implementations produce correct results for varying bit lengths of input. We provide the source code for SQR_3 in Appendix A.

When timing our squaring algorithms we have replaced `mpn_toom3_sqr_n()` with our functions and called from the top level function `mpz_mul()`. To prevent `mpz_mul()` from using the schoolbook squaring algorithm and KA, we have modified `mpn_sqr_n()`, which chooses the best one among various squaring algorithms depending on the input size, so that only our algorithms are called for all ranges of input sizes. For timing the 3-way Toom-Cook multiplication algorithm, we have forced `mpn_sqr_n()` to choose only the original `mpn_toom3_sqr_n()`.

Figure 6.3 shows the timing results of SQR_3, `mpz_mul()` and the 3-way Toom-Cook multiplication algorithm. Figure 6.4 shows the timing ratio of `mpz_mul()` and the 3-way Toom-Cook multiplication algorithm to SQR_3. On Pentium IV 3.2GHz, `mpz_mul()` uses the schoolbook squaring algorithm for small operands, KA for input longer than 1984 bits, the 3-way Toom-Cook algorithm for input longer than 3744 bits. In our experiments, we have found that SQR_1 and SQR_2 are slower than `mpz_mul()` for all sizes of input. Thus, we have not included their timing results. Our experiments show that SQR_3 outperforms `mul_mul()` for operands that are about 2300–6900 bits long. Relative performance improvements of SQR_3 over mpz_mul() near 10000 bit input is observed in Figure 6.4. It is due to the fact that our SQR_3 recurses into itself, which is faster than `mpz_mul()` with operands of size approximately 3300 bits.

We have also performed the same experiments on Pentium II MMX 350MHz and Pen-

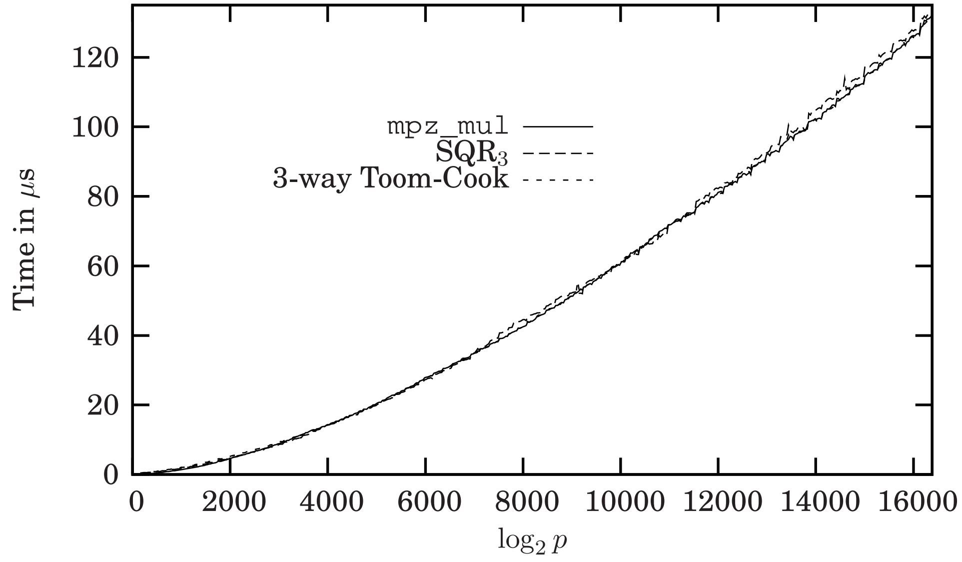

Figure 6.3: Timing Results of Squaring Algorithms (Pentium IV 3.2GHz)

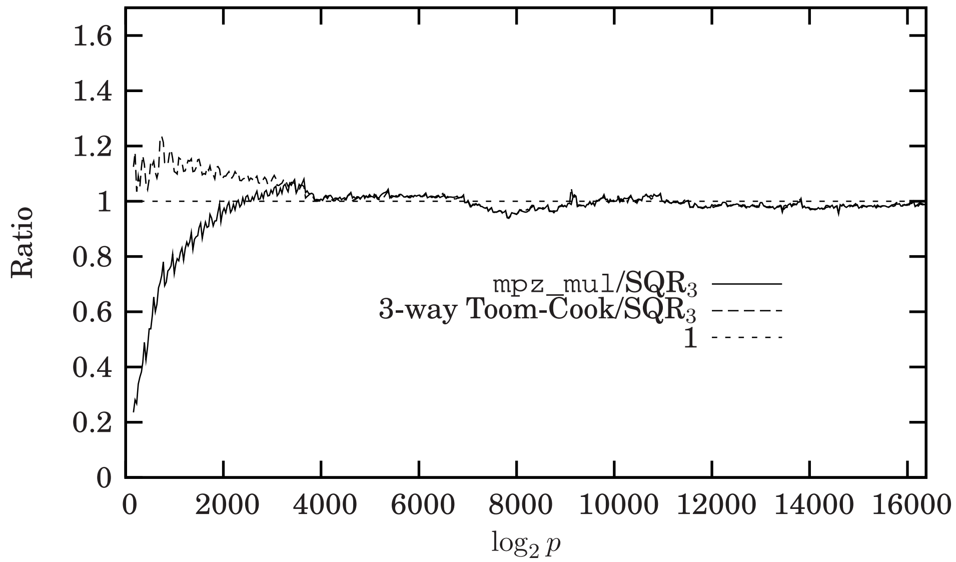

Figure 6.4: Timing Ratio of SQR_3 vs. Other Algorithms on Pentium IV 3.2GHz

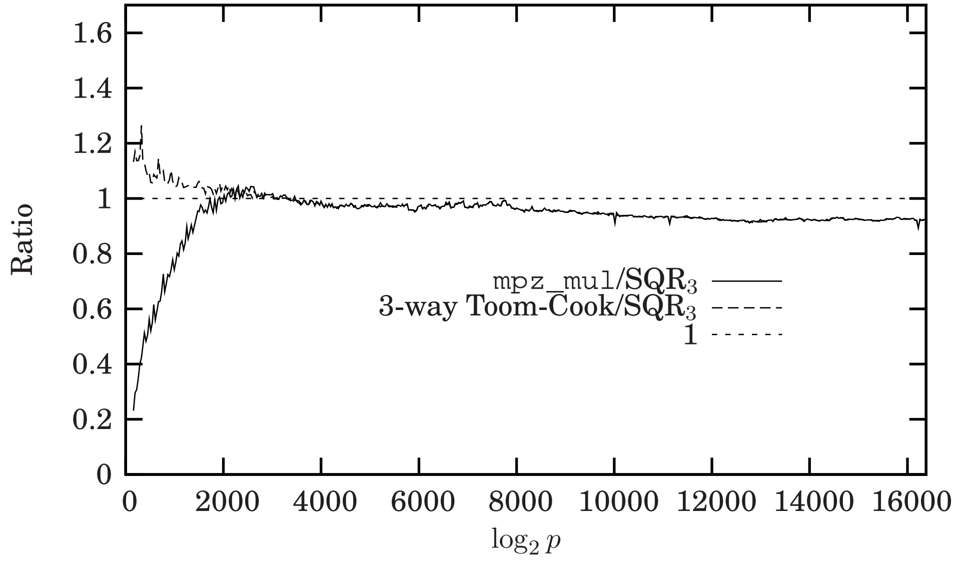

Figure 6.5: Timing Ratio of SQR$_3$ vs. `mpz_mul()` (Pentium II MMX 350MHz)

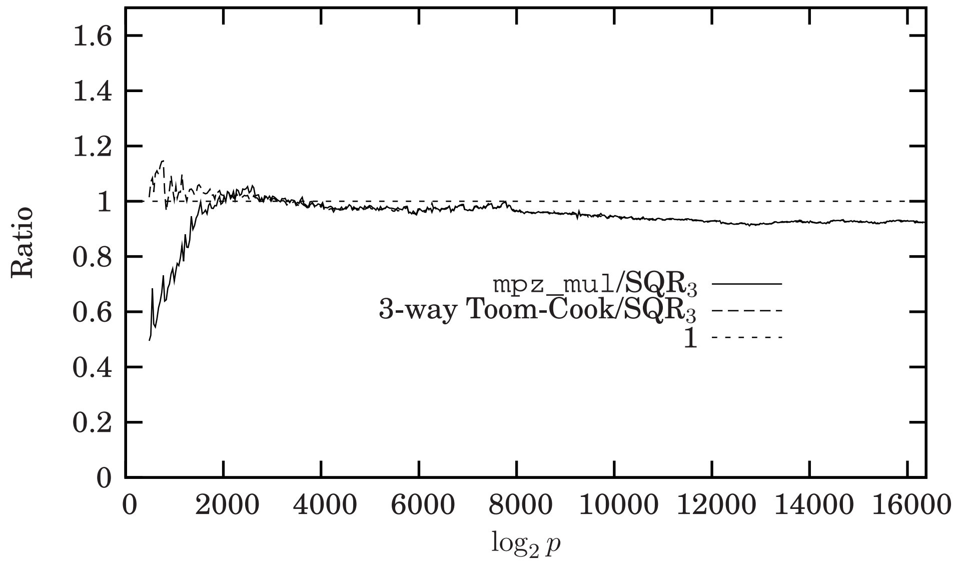

Figure 6.6: Timing Ratio of SQR$_3$ vs. `mpz_mul()` (Pentium III M 1.13GHz)

Table 6.4: Timing Results of Polynomial Squaring on Pentium IV 3.2GHz (unit= $\mu s.$)

Bit length	SQR_1	SQR_2	SQR_3	3-way Toom-Cook	3-way KA-like	Schoolbook sqr.
32	**0.23**	0.32	0.37	0.58	0.43	0.25
256	**1.12**	1.42	1.33	1.68	1.80	1.26
576	**3.43**	3.96	3.59	4.09	4.79	4.15
608	3.89	4.25	**3.85**	4.42	5.25	4.42
768	6.09	6.41	**5.44**	5.87	7.29	6.99
1024	8.37	9.65	**8.17**	8.47	11.12	10.72
1216	11.28	12.67	**10.73**	10.83	14.17	14.38
1248	12.84	14.11	11.51	**11.17**	15.21	16.29
1502	17.15	18.94	15.42	**14.90**	19.85	22.02
1536	19.37	19.52	15.55	**15.21**	20.40	22.31

tium III Mobile 1.13GHz. We have plotted the results in Figures 6.5 and 6.6. On Pentium II MMX 350MHz processors, SQR_3 performed better than `mpz_mul()` for about 2000–3300-bit operands by up to 3-4%. On Pentium III Mobile 1.13GHz processors, SQR_3 performed better than `mpz_mul()` for about 1900–3500-bit operands by up to 4-5%. The GMP tuneup program has determined that the crossover between the classical multiplication and the KA is 48 words (1536 bits) and the crossover between the KA and 3-way Toom-Cook multiplication algorithm is 83 words (2656 bits) on both Pentium II MMX 350MHz and Pentium III Mobile 1.13GHz.

6.4.2 Application to Polynomial Squaring in $\mathbb{Z}[x]$

We have applied our squaring algorithm for performing polynomial multiplication in $\mathbb{Z}[x]$. We have implemented functions for squaring degree-2 polynomials in $\mathbb{Z}[x]$. The implementation uses the functions from GMP library. The timing results on Pentium IV 3.2GHz are shown in Table 6.4. The first column in Table 6.4 shows the sizes of coefficients in bits. In the table, the best timing for each bit length is indicated in bold. SQR_1 is the most efficient squaring algorithm for squaring polynomials having small coefficients of up to 576 bits. For polynomials with coefficients up to 1216 bits, SQR_3 is the most efficient. However, the 3-way Toom-Cook algorithm becomes the fastest algorithm for squaring degree-2 polynomials whose coefficients are at least 1216 bits long.

6.5 4-way and 5-way Squaring Formulae

We have constructed 4-way and 5-way squaring formulae that do not require any nontrivial constant divisions. We have used the same technique that we applied to construct 3-way formulae. We have chosen the algorithms that have the simplest interpolation among the many candidates we have considered so far. The results shown in this section are to illustrate that the technique we have developed in Section 6.2 can also be applied to construct n-way squaring formulae for $n > 3$. Note that 5-way is not the limit where nontrivial divisions in interpolation step can be eliminated. Future research will show further results on 4, 5, 6, 7-way squaring formulae.

6.5.1 New 4-Way Squaring

Let $A(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. To compute $C(x) = \sum_{i=0}^{6} c_ix^i = A(x)^2$, we first compute S_i's as shown below:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_0^2 \\ 2a_0a_1 \\ (a_0+a_1-a_2-a_3)(a_0-a_1-a_2+a_3) \\ (a_0+a_1+a_2+a_3)^2 \\ 2(a_0-a_2)(a_1-a_3) \\ 2a_3a_2 \\ a_3^2 \end{bmatrix}. \quad (6.12)$$

The linear combinations of a_i's in (6.12) can be computed using the following:

$$\begin{aligned} T_1 &= a_0 - a_2 \\ T_2 &= a_1 - a_3 \\ T_3 &= T_1 + T_2 \\ T_4 &= T_1 - T_2 \\ T_5 &= a_0 + a_1 + a_2 + a_3. \end{aligned} \quad (6.13)$$

The determinant of the 7×7 matrix in (6.12) is 2. This method uses 3 coefficient squar-

Algorithm 6.1 New 4-Way Toom-Cook Interpolation for Squaring

Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$ as in (6.12).
Ensure: $C(x) = A(x) \cdot B(x)$.

1: $T_1 \leftarrow S_3 + S_4$. $(= c_5 + 2c_4 + c_3 + c_1 + 2c_0)$
2: $T_2 \leftarrow (T_1 + S_5)/2$. $(= c_5 + c_4 + c_1 + c_0)$
3: $T_3 \leftarrow S_2 + S_6$. $(= c_5 + c_1)$
4: $T_4 \leftarrow T_2 - T_3$. $(= c_4 + c_0)$
5: $T_5 \leftarrow T_3 - S_5$. $(= c_3)$
6: $T_6 \leftarrow T_4 - S_3$. $(= c_6 + c_2)$
7: $T_7 \leftarrow T_4 - S_1$. $(= c_4)$
8: $T_8 \leftarrow T_6 - S_7$. $(= c_2)$
9: **return** $C(x) = S_7x^6 + S_6x^5 + T_7x^4 + T_5x^3 + T_8x^2 + S_2x + S_1$.

ings and 4 coefficient multiplications. Note that KA requires 9 coefficient squarings for squaring a polynomial using 4-way split. Interpolation method is given in Algorithm 6.1. Algorithm 6.1 requires only 8 additions/subtractions and 1 bit shift.

Using the same analysis methods in 6.3, we obtain that our 4-way squaring algorithm requires $3\mathcal{S}(u) + 4\mathcal{M}(u) + 28\mathcal{A}(u) + 13\mathcal{B}(u)$.

6.5.2 New 5-term Squaring Method

Let $A(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. To compute $C(x) = \sum_{i=0}^{8} c_ix^i = A(x)^2$, we first compute S_i's as shown below:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \\ S_8 \\ S_9 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_8 \\ c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

$$= \begin{bmatrix} a_0^2 \\ a_4^2 \\ (a_0 + a_1 + a_2 + a_3 + a_4)^2 \\ (a_0 - a_1 + a_2 - a_3 + a_4)^2 \\ 2(a_0 - a_2 + a_4)(a_1 - a_3) \\ (a_0 + a_1 - a_2 - a_3 + a_4)(a_0 - a_1 - a_2 + a_3 + a_4) \\ (a_1 + a_2 - a_4)(a_1 - a_2 - a_4 + 2(a_0 - a_3)) \\ 2a_0a_1 \\ 2a_3a_4 \end{bmatrix}. \tag{6.14}$$

The above system needs 4 squarings and 5 multiplications. The linear combinations of a_i's can be computed as follows using 14 additions or subtractions and 1 shift:

$$T_1 = a_0 + a_4, \qquad T_8 = T_5 - T_2 = \mathbf{a_0 - a_1 + a_2 - a_3 + a_4},$$

$$T_2 = a_1 + a_3, \qquad T_9 = T_6 + T_4 = \mathbf{a_0 + a_1 - a_2 - a_3 + a_4},$$

$$T_3 = a_1 - a_4, \qquad T_{10} = T_6 - T_4 = \mathbf{a_0 - a_1 - a_2 + a_3 + a_4},$$

$$T_4 = \mathbf{a_1 - a_3}, \qquad T_{11} = T_3 + a_2 = \mathbf{a_1 + a_2 - a_4},$$

$$T_5 = T_1 + a_2 = a_0 + a_2 + a_4, \qquad T_{12} = T_3 - a_2 = a_1 - a_2 - a_4,$$

$$T_6 = T_1 - a_2 = \mathbf{a_0 - a_2 + a_4}, \qquad T_{13} = T_{12} - 2(a_0 - a_3) = \mathbf{a_1 - a_2 - a_4 - 2(a_0 - a_3)}$$

$$T_7 = T_5 + T_2 = \mathbf{a_0 + a_1 + a_2 + a_3 + a_4}.$$

Interpolation can be performed by Algorithm 6.2. The algorithm requires 18 additions

Algorithm 6.2 New 5-Way Toom-Cook Interpolation for Squaring

Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)$ as in (6.14).

Ensure: $C(x) = A(x) \cdot B(x)$.

1: $T_1 \leftarrow S_1 + 2 \cdot S_2 - S_7 + 2 \cdot S_8 + S_9$ $(= c_8 + c_5 + c_4 - c_2 + c_1 + c_0)$
2: $T_2 \leftarrow S_3 - S_4$ $(= 2c_7 + 2c_5 + 2c_3 + 2c_1)$
3: $T_3 \leftarrow 2 \cdot S_5$ $(= -2c_7 + 2c_5 - 2c_3 + 2c_1)$
4: $T_4 \leftarrow T_2 + T_3$ $(= 4c_5 + 4c_1)$
5: $T_5 \leftarrow T_2 - T_3$ $(= 4c_7 + 4c_3)$
6: $T_6 \leftarrow T_4/4$ $(= c_5 + c_1)$
7: $T_7 \leftarrow T_5/4 - S_9$ $(= c_3)$
8: $T_8 \leftarrow T_1 - T_6 - S_6$ $(= c_6)$
9: $T_9 \leftarrow T_6 - S_8$ $(= c_5)$
10: $T_{10} \leftarrow S_3 + S_6$ $(= 2c_8 + c_7 + c_5 + 2c_4 + c_3 + c_1 + 2c_0)$
11: $T_{11} \leftarrow (T_{10} + S_4 + S_6)/4$ $(= c_8 + c_4 + c_0)$
12: $T_{12} \leftarrow T_{11} - T_1 - T_2$ $(= c_4)$
13: $T_{13} \leftarrow (T_{10} + S_5)/2$ $(= c_8 + c_5 + c_4 + c_1 + c_0)$
14: $T_{14} \leftarrow T_{13} - T_1$ $(= c_2)$
15: **return** $C(x) = S_2 t^8 + S_9 t_7 + T_8 t^6 + T_9 t^5 + T_{11} t^4 + T_7 t_3 + T_{13} t^2 + S_8 t + S_1$.

and subtractions, 7 shifts and no divisions by constants.

Note that Montgomery's 5-way formulae requires 13 squarings. Hence, if the ratio of squaring to multiplication is greater than 5.6, there is a very good possibility that our algorithm is superior to Montgomery's 5-way formula.

Using the same analysis technique and assuming that each coefficient a_i is u-digit integer, we obtain that our 5-way squaring algorithm requires at most $4\mathcal{S}(u) + 5\mathcal{M}(u) + 60\mathcal{A}(u) + 26\mathcal{B}(u)$. Montgomery's 5-way algorithm requires at most $13\mathcal{S}(u) + 65\mathcal{A}(u) + 10\mathcal{B}(u)$ when two operands are identical. Therefore, if $5\mathcal{M}(u) + 16\mathcal{B}(u) < 9\mathcal{S}(u) + 5\mathcal{A}(u)$, then our algorithm is superior. Ignoring the overhead terms ($\mathcal{A}$ and $\mathcal{B}$), our algorithm is superior if squaring/multiplication ratio is more than $5/9 \approx 0.56$. This condition appears to be easily satisfied in practice. Figure 6.7 shows the timing ratio of squaring and multiplication routines in GMP library. In the figure, the GMP's squaring/multiplication timing ratio is between 0.6–0.8 for operand sizes larger than 500-bits on Pentium 4 Prescott 3.2GHz.

Usually, additions and subtractions are slower than bit shifts by a small factor. If the bit shift is more than 3.2 times faster than additions/subtractions, then our 5-way squaring algorithm is clearly faster than Montgomery's 5-way algorithm for all ranges of input sizes.

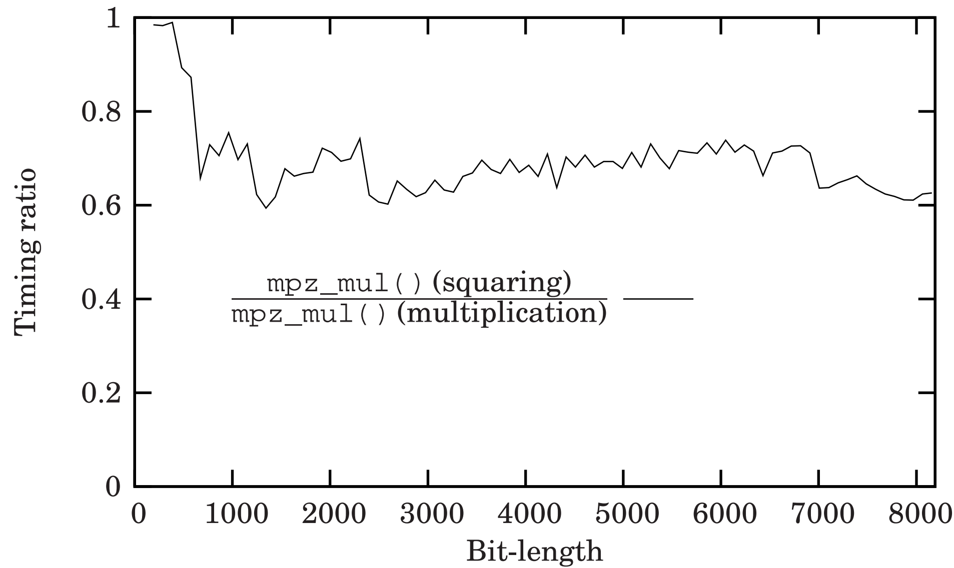

Figure 6.7: Timing Ratio of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring)

6.6 Conclusions

In this chapter, we have presented new 3, 4 and 5-way polynomial squaring formulae. Our new formulae are based on the Toom-Cook multiplication algorithm and they require the same number of coefficient multiplications used in the Toom-Cook multiplication algorithm. However, our approach eliminates the need for nontrivial constant divisions always required in the n-way Toom-Cook multiplication algorithms for $n \geq 3$. Our experimental results confirm that one of our 3-way formulae is slightly faster than GMP's squaring routine for squaring integers of size about 2300–6900 bits on Pentium IV 3.2GHz. Moreover, according to our implementation results, our squaring formulae are the best for squaring degree-2 polynomials whose coefficients are shorter than about 1200 bits on the same processor. However, symmetric squaring algorithms are advantageous for squaring very large size operands, since our asymmetric squaring algorithms use at least one point-wise multiplication that cannot be computed by squaring.

Chapter 7

Side Channel Attack on XTR Cryptosystems

In this chapter, we attempt to attack the XTR double exponentiation algorithm using a simple side channel attack. We analyze the statistical behavior of the XTR double exponentiation algorithm and use the result in our attack. Our experimental results show that, in order to determine the correct exponent pair (a, b), one would require $U^{1.25}$ tries where $U = \max(a, b)$. This result immediately shows that an adversary needs to make $U^{0.625}$ tries on average to determine the correct secret key used in the XTR single exponentiation algorithm based on the XTR double exponentiation algorithm.

7.1 Identifying Elementary XTR Operations

Our side channel attack on Algorithm 3.3 is based on the following simple assumptions.

1. It is possible to distinguish long integer multiplications and modular reductions (or Montgomery reductions) using side channel information.
2. An adversary can somehow verify whether any given integer is the correct key or not.

The first assumption is reasonable, since long integer multiplication and modular reduction are quite different operations. We believe this assumption remains reasonable, even if the Montgomery arithmetic is used. The Montgomery reduction can be distinguished from long

integer multiplication, since the former uses an additional single precision multiplication (Line 3 of Algorithm 2.4) for each loop.

In XTR exponentiation algorithms, operations such as c_{u+v}, c_{2u}, c_{3u}, c_{u+2}, c_{2u-1} and c_{2u+1} are used. However, we observe that only three operations among them are sufficient to implement XTR exponentiations. Let $w, x, y, z \in GF(p^2)$. The elementary XTR operations are:

1. $A(w, x, y, z) = w \cdot x - x^p \cdot y + z$.
2. $D(x) = x^2 - 2x^p$.
3. $T(x) = x^3 - 3x^{p+1} + 3$.

It is easy to verify that all nontrivial trace operations used in XTR cryptosystems are derived from elementary XTR operations as shown below. Computing the p-th power operation does not involve any computation, i.e., $x^p = x_2\alpha + x_1\alpha^2$ where $x = x_1\alpha + x_2\alpha^2$, $x_1, x_2 \in GF(p)$, $p \bmod 3 \equiv 2$ and $\alpha^2 + \alpha + 1 = 0$.

$$\begin{aligned} c_{u+v} &= A(c_u, c_v, c_{u-v}, c_{u-2v}) \ , & c_{2u+1} &= A(c_{u+1}, c_u, c_1, c_{u-1}^p) \ , \\ c_{u+2} &= A(c_{u+1}, c_1, c_u, c_{u-1}) \ , & c_{2u-1} &= A(c_{u-1}, c_u, c_1^p, c_{u+1}^p) \ , \\ c_{2u} &= D(c_u) \ , & c_{3u} &= T(c_u) \ . \end{aligned}$$

Let $m(\cdot)$ denote a long integer multiplication and $r(\cdot)$ a modular reduction by p (or a Montgomery reduction). According to the improved $GF(p^2)$ arithmetic shown in [77], $GF(p^2)$ operations are implemented as follows.

$$\begin{aligned} M(x, y) &= x \cdot y = r(t-u)\alpha + r(s-u)\alpha^2, \\ & s = m(x_1, y_1) \ , \quad t = m(x_2, y_2), \\ & u = m(x_1 + x_2, y_1 + y_2) - s - t, \\ S(x) &= x^2 = r(m(x_2, x_2 - 2x_1))\alpha + r(m(x_1, x_1 - 2x_2))\alpha^2, \\ X(x, y, z) &= x \cdot z - y \cdot z^p \\ &= r(m(z_1, y_1 - x_2 - y_2) + m(z_2, x_2 - x_1 + y_2))\alpha \\ &\quad + r(m(z_1, x_1 - x_2 + y_1) + m(z_2, y_2 - x_1 - y_1))\alpha^2, \end{aligned}$$

where $x = x_1\alpha + x_2\alpha^2$, $y = y_1\alpha + y_2\alpha^2$ and $z = z_1\alpha + z_2\alpha^2$ are in $GF(p^2)$, and $\alpha^2 + \alpha + 1 = 0$. We only consider long integer multiplications and modular reductions (or Montgomery reductions), since additions and subtractions are performed in very short time and much harder to be detected by side channel attacks. The integer operation sequence for each $GF(p^2)$ operation may vary depending on implementations. However, $GF(p^2)$ operations can be clearly identified if an adversary can distinguish integer multiplications and modular reductions (or Montgomery reductions). The following shows all possible integer operation sequences for M, S and X.

- M : $mmmrr$
- S : $mrmr$ or $mmrr$
- X : $mmrmmr$, $mmmmrr$ or $mmmrmr$

From above, we clearly see that $GF(p^2)$ operations can be identified by identifying integer operation sequences. We rewrite elementary XTR operations in terms of M, S and X as follows.

$$
\begin{aligned}
A(w, x, y, z) &= X(w, y, x) + z, \\
D(x) &= S(x) - 2x^p, \\
T(x) &= M(x, S(x) - 3x^p) + 3.
\end{aligned}
$$

This clearly shows that operations A, D and T are also distinguishable by identifying integer operation sequences.

Here we give a short example on how to determine the elementary XTR operation sequence from $GF(p)$ operation sequence. We assume that the operations M, S and X are implemented as $mmrr$, $mmmrr$ and $mmmmrr$, respectively.

Suppose that an adversary observes the sequence of $GF(p)$ operations as follows:

$$mmmmrrmmrrmmrrmmmmrrmmrrmmrrmmrrmmmrrmmmmrrmmmmrr.$$

Then he/she can determine the $GF(p^2)$ operation sequence from the observed $GF(p)$ operation sequence as follows.

Table 7.1: Update Rules for Algorithm 3.3

Notation	Condition	Update $(d, e, c_u, c_v, c_{u-v}, c_{u-2v})$	XTR Seq.	# Muls.
If $d > e$				
S_0	i. If $d \leq 4e$	$(e, d-e, \mathbf{c_{u+v}}, c_u, c_v, c_{v-u})$	A	3
S_1	ii. Else if d is even	$(\frac{d}{2}, e, \mathbf{c_{2u}}, c_v, \mathbf{c_{2u-v}}, \mathbf{c_{2(u-v)}})$	ADD	7
S_2	iii. Else if e is odd	$(\frac{d-e}{2}, e, \mathbf{c_{2u}}, \mathbf{c_{u+v}}, c_{u-v}, \mathbf{c_{-2v}})$	ADD	7
S_3	v. Else (e is even)	$(\frac{e}{2}, d, \mathbf{c_{2v}}, c_u, c_{2v-u}, \mathbf{c_{2(v-u)}})$	DD	4
If $d < e$				
S_4	i. If $e \leq 4d$	$(d, e-d, \mathbf{c_{u+v}}, c_v, c_u, c_{u-v})$	A	3
S_5	ii. Else if e is even	$(\frac{e}{2}, d, \mathbf{c_{2v}}, c_u, c_{2v-u}, \mathbf{c_{2(v-u)}})$	DD	4
S_6	iii. Else if d is odd	$(\frac{e-d}{2}, d, \mathbf{c_{2v}}, \mathbf{c_{u+v}}, c_{v-u}, \mathbf{c_{-2u}})$	ADD	7
S_7	vi. Else (d is even)	$(\frac{d}{2}, e, \mathbf{c_{2u}}, c_v, \mathbf{c_{2u-v}}, \mathbf{c_{2(u-v)}})$	ADD	7

$$\underbrace{mmmmr}_{X}\underbrace{mmrr}_{S}\underbrace{mmrr}_{S}\underbrace{mmmmrr}_{X}\underbrace{mmrr}_{S}\underbrace{mmrr}_{S}\underbrace{mmrr}_{S}\underbrace{mmmrr}_{M}\underbrace{mmmmrr}_{X}\underbrace{mmmmrr}_{X}$$

The adversary has the sequence of $GF(p^2)$ operations, "$XSSXSSSMXX$". Using this result, the adversary successfully determines the sequence of elementary XTR operations, "$ADDADDTAA$".

Table 7.1 shows the update rules used in lines 10 and 13 of Algorithm 3.3. The first column of Table 7.1 lists the short notation for each sub-step. The second last column of Table 7.1 lists the corresponding elementary XTR operation sequence for each sub-step. Note that the elementary XTR operation sequences for sub-steps S_1, S_2, S_6 and S_7 may vary depending on implementations. However, we will assume that they are implemented in the way specified in Table 7.1. This makes the sub-steps S_1, S_2, S_6, S_7, S_0S_3, S_0S_5, S_4S_3 and S_4S_5 indistinguishable, and it may prevent easy detection of some sub-steps.

7.2 Simple Side Channel Attack

In this section, we first state our observations on the statistical behavior of Algorithm 3.3. Then we discuss how to obtain useful information for recovering the two exponents used in Algorithm 3.3 by using the simple side channel attack.

In exponentiation algorithms used in other cryptosystems, one can determine the exponent bit one at a time by observing which part of an algorithm is executed in each iteration. In Algorithm 3.3, the knowledge of execution path within each iteration of the algorithm does not lead to a discovery of a single key bit. The whole exponent can be reconstructed only after all necessary information is assessed. Note that lines 10 and 13 have four sub-steps each. To reconstruct a secret exponent, one need to know the sequence of these sub-steps, along with f_2, f_3 and d value in line 17 of Algorithm 3.3. Then he/she can recover two input exponents a and b.

According to Table 7.1, each sub-step S_i, $(d,\, e)$-pair is linearly transformed by matrix T_i, where transformation matrices T_i's are defined as follows.

$$\begin{aligned}
T_0 &= \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}, \quad T_1 = \begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix}, \quad T_2 = \begin{bmatrix} 1/2 & -1/2 \\ 0 & 1 \end{bmatrix}, \quad T_3 = \begin{bmatrix} 0 & 1/2 \\ 1 & 0 \end{bmatrix}, \\
T_4 &= \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}, \quad T_5 = \begin{bmatrix} 0 & 1/2 \\ 1 & 0 \end{bmatrix}, \quad T_6 = \begin{bmatrix} -1/2 & 1/2 \\ 1 & 0 \end{bmatrix}, \quad T_7 = \begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix}.
\end{aligned} \tag{7.1}$$

Let us define two more matrices, F_2 and F_3,

$$F_2 = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix}, \quad F_3 = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix}. \tag{7.2}$$

Suppose that an adversary knows the sub-step sequence, $S_{l_1} S_{l_1} \ldots S_{l_n}$, where $l_i \in [0,\, 7]$. Then he/she can solve (7.3) to recover the input exponents a and b:

$$\begin{bmatrix} d \\ d \end{bmatrix} = T_{l_n} T_{l_{n-1}} \cdots T_{l_1} \cdot F_3^{f_3} \cdot F_2^{f_2} \cdot \begin{bmatrix} b \\ a \end{bmatrix}. \tag{7.3}$$

Since the matrices T_i's for $i = 0 \ldots 7$, F_2 and F_3 are all invertible, the unique solution $(b,\, a)^T$ exists.

7.2.1 Markov Chain Model

We have made some important observations on the statistical behavior of Algorithm 3.3. We use Markov Chain model to analyze the statistical behavior of Algorithm 3.3. This method

is based on the ideas proposed in [82, 67] for attacking MIST and ECC.

We have found that some sub-steps in lines 10 and 13 are used more frequently than the others. Moreover, some sub-steps cannot be even reached from other sub-steps. To prove that all sub-steps are not equally probable, we will use a Markov chain model. We first compute the conditional probabilities $Pr(S_j|S_i)$ for $i,\, j = 0, \ldots 7$, where $Pr(S_j|S_i)$ represents the probability that the next sub-step is S_j given current state S_i. Then the steady-state vector π will be determined from state transition matrix. In the following, we let $S_a = \{S_0,\, S_1,\, S_2,\, S_3\}$ and $S_b = \{S_4,\, S_5,\, S_6,\, S_7\}$.

Transition from S_0

Suppose that $(d_1,\, e_1)$ are the values of $(d,\, e)$ before entering S_0. Then $(d_1,\, e_1)$ must meet the condition, $e_1 < d_1 \leq 4e_1$. After executing sub-step S_0, $(d,\, e)$-pair is updated to $(d_2,\, e_2)$, where $d_2 = e_1$, $0 < e_2 \leq 3e_1$. Then it is easy to see that,

$$-2e_1 \leq d_2 - e_2 < e_1.$$

Hence the next sub-step is in $S_a = \{S_0,\, S_1,\, S_2,\, S_3\}$ with the probability $1/3$ and it will be in $S_b = \{S_4,\, S_5,\, S_6,\, S_7\}$ with the probability $2/3$. Suppose the algorithm enters one of S_a. Then,

$$-e_1 < 4e_2 - d_2 \leq 11e_1.$$

Hence the next step is again S_0 with the probability $11/12$.

$$Pr(S_0|S_0) = \frac{1}{3}\frac{11}{12} = \frac{11}{36},$$

Since $d_2 = e_1$ is even with the probability $1/2$, the next step is S_1 with the probability,

$$Pr(S_1|S_0) = \frac{1}{3} \cdot \frac{1}{12} \cdot \frac{1}{2} = \frac{1}{72}.$$

Similarly,

$$Pr(S_2|S_0) = \frac{1}{3} \cdot \frac{1}{12} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{144}.$$

It is clear that,

$$Pr(S_3|S_0) = Pr(S_2|S_0) = \frac{1}{144}.$$

Suppose the algorithm enters the second half, S_b. Then,

$$0 < e_1 \leq 4d_2 - e_2 < 4e_1.$$

Therefore, the next step must be S_4. Hence,

$$\begin{aligned}
Pr(S_4|S_0) &= \frac{2}{3}, \\
Pr(S_5|S_0) &= 0, \\
Pr(S_6|S_0) &= 0, \\
Pr(S_7|S_0) &= 0.
\end{aligned}$$

This shows that some sub-steps are not reachable from some sub-steps as well as some sub-steps are preferred than the others. In above example, S_0 and S_4 are executed after S_0 with dominating probability.

Transition from S_1

Suppose that (d_1, e_1) are the values of (d, e) before entering S_1. Then (d_1, e_1) must meet the condition, $4e_1 < d_1$ and d_1 is even. After executing sub-step S_1, (d, e)-pair is updated to (d_2, e_2), where $d_2 = d_1/2$ and $e_2 = e_1$. Then it must be the case that,

$$0 < e_1 < d_2 - e_2.$$

Hence the next sub-step must be in S_a. However, the upper bound of d_1 is not defined. Thus, we cannot compute $Pr(S_0|S_1)$. Say,

$$Pr(S_0|S_1) = s_1,$$

where $0 \leq s_1 \leq 1$. Suppose $4e_2 < d_2$. Then the next sub-step must be either S_1 or S_2. Note that S_3 cannot occur since d_1 and e_1 cannot have 2 as a common factor due to lines 2-4 of Algorithm 3.3. Since d_2 is even with the probability $1/2$,

$$Pr(S_1|S_1) = Pr(S_2|S_2) = \frac{1 - s_1}{2}.$$

Transition from S_2

Suppose that (d_1, e_1) are the values of (d, e) before entering S_2. Then (d_1, e_1) must meet the condition, $4e_1 < d_1$ and both d_1 and e_1 are odd. After executing sub-step S_2, (d, e)-pair is updated to (d_2, e_2), where $d_2 = (d_1 - e_1)/2$ and $e_2 = e_1$. Then it must be the case that,

$$0 < \frac{e_1}{2} < d_2 - e_2.$$

Hence the next sub-step must be in S_a. Since we do not know the upper bound of d_1, we cannot compute $Pr(S_0|S_2)$. Say,

$$Pr(S_0|S_2) = s_2,$$

where $0 \leq s_2 \leq 1$. Suppose $4e_2 < d_2$. Then the next sub-step must be either S_1 or S_2. Therefore,

$$Pr(S_1|S_2) = Pr(S_2|S_2) = \frac{1 - s_2}{2}.$$

Transition from S_3

Suppose that (d_1, e_1) are the values of (d, e) before entering S_3. Then (d_1, e_1) must meet the condition, $4e_1 < d_1$, where d_1 is odd and e_1 is even. After executing sub-step S_3, (d, e)-pair is updated to (d_2, e_2), where $d_2 = e_1/2$ and $e_2 = d_1$. Then it must be the case that,

$$d_2 - d_2 < -\frac{7}{2}e_1 < 0.$$

Hence, the next sub-step must be in S_b. Since,

$$4d_2 - e_2 < -2e_1 < 0,$$

the next sub-step cannot be S_4. S_5 cannot occur since $e_2 = d_1$ is odd. The remaining sub-steps S_6 and S_7 occur with equal probabilities, $1/2$.

$$Pr(S_6|S_3) = Pr(S_7|S_3) = 0.5.$$

Transition from S_4

Suppose that (d_1, e_1) are the values of (d, e) before entering S_4. Then (d_1, e_1) must meet the condition, $d_1 \leq e_1 \leq 4d_1$. After executing sub-step S_4, (d, e)-pair is updated to (d_2, e_2), where $d_2 = d_1$ and $e_2 = e_1 - d_1$. Then it must be the case that,

$$-2d_1 \leq d_2 - e_2 \leq d_1.$$

Hence, the next sub-step belongs to S_a with the probability $1/3$ and S_b with the probability $2/3$. Suppose that the next sub-step is in S_a. Since

$$-d_1 \leq 4e_2 - d_2 \leq 11d_1,$$

the next sub-step is S_0 with the probability,

$$Pr(S_0|S_4) = \frac{1}{3} \cdot \frac{11}{12} = \frac{11}{36}.$$

The probability that d_2 is even is $1/2$. Thus,

$$Pr(S_1|S_4) = \frac{1}{3} \cdot \frac{1}{12} \cdot \frac{1}{2} = \frac{1}{72}.$$

The probability that e_2 is odd is $1/2$. Therefore,

$$Pr(S_2|S_4) = Pr(S_3|S_4) = \frac{1}{3} \cdot \frac{1}{12} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{144}.$$

Suppose that the next sub-step is in S_b. Since

$$0 < d_1 \leq 4d_2 - e_2 \leq 4d_1,$$

the next sub-step must be S_4. Therefore,

$$Pr(S_4|S_4) = \frac{2}{3}.$$

Transition from S_5

Suppose that (d_1, e_1) are the values of (d, e) before entering S_5. Then (d_1, e_1) must meet the condition, $4d_1 < e_1$ and e_1 is even. After executing sub-step S_5, (d, e)-pair is updated to (d_2, e_2), where $d_2 = e_1/2$ and $e_2 = d_1$. Then it must be the case that,

$$0 < d_1 < d_2 - e_2.$$

Hence the next sub-step must be in S_a. However, we cannot compute $Pr(S_0|S_5)$, since we do not know the upper bound of e_1. Say,

$$Pr(S_0|S_5) = s_5,$$

where $0 \leq s_5 \leq 1$. Suppose $4e_2 < d_2$. Then the next sub-step must be either S_1 or S_2, since entering S_3 requires that e_1 and d_1 be both even. Therefore,

$$Pr(S_1|S_5) = Pr(S_2|S_5) = \frac{1 - s_5}{2}.$$

Transition from S_6

Suppose that (d_1, e_1) are the values of (d, e) before entering S_6. Then (d_1, e_1) must meet the condition, $4d_1 < e_1$ and both d_1 and e_1 are odd. After executing sub-step S_6, (d, e)-pair is updated to (d_2, e_2), where $d_2 = (e_1 - d_1)/2$ and $e_2 = d_1$. Then it must be the case that,

$$0 < \frac{d_1}{2} < d_2 - e_2.$$

Hence, the next sub-step must be in S_a. Since there is no upper limit for e_1, we cannot compute $Pr(S_0|S_6)$. Say,

$$Pr(S_0|S_6) = s_6,$$

where $0 \leq s_6 \leq 1$. Suppose $4e_2 < d_2$. Then the next sub-step must be either S_1 or S_2, since entering S_3 would mean that d_1 is even, which contradicts the assumption. Therefore,

$$Pr(S_1|S_6) = Pr(S_2|S_6) = \frac{1 - s_6}{2}.$$

Transition from S_7

Suppose that (d_1, e_1) are the values of (d, e) before entering S_7. Then (d_1, e_1) must meet the condition, $4d_1 < e_1$, e_1 is odd and d_1 is even. After executing sub-step S_7, (d, e)-pair is updated to (d_2, e_2), where $d_2 = d_1/2$ and $e_2 = e_1$. Then it must be the case that,

$$d_2 - e_2 < -\frac{7}{2}d_1 < 0.$$

Hence, the next sub-step must be in S_b. Since,

$$4d_2 - e_2 < -2d_1 < 0,$$

the next sub-step cannot be S_0. Moreover S_5 cannot occur since $e_2 = e_1$ is odd. It is straightforward that the remaining sub-steps S_6 and S_7 occur with equal probabilities, 1/2.

$$Pr(S_6|S_7) = Pr(S_7|S_7) = 0.5.$$

7.2.2 Statistical Behavior of Algorithm 3.3

By the arguments above, the transition matrix $B = [b_{(i,j)}]$, where $b_{(i,j)} = P(S_j|S_i)$ for $i, j \in [0, 7]$, is computed as follows,

$$B = \begin{bmatrix} 0.30056 & 0.01389 & 0.00694 & 0.00694 & 0.66667 & 0 & 0 & 0 \\ s_1 & \frac{1-s_1}{2} & \frac{1-s_1}{2} & 0 & 0 & 0 & 0 & 0 \\ s_2 & \frac{1-s_2}{2} & \frac{1-s_2}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0.30056 & 0.01389 & 0.00694 & 0.00694 & 0.66667 & 0 & 0 & 0 \\ s_5 & \frac{1-s_5}{2} & \frac{1-s_5}{2} & 0 & 0 & 0 & 0 & 0 \\ s_6 & \frac{1-s_6}{2} & \frac{1-s_6}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix},$$

for some s_1, s_2, s_5, s_6 where $0 \leq s_1, s_2, s_5, s_6 \leq 1$. It is clearly seen that some steps cannot be reached from other steps. For example, step S_5, S_6 and S_7 cannot follow the step S_0.

Note that the above analysis may give imprecise results, since we made some assumptions that may be invalid. For example, in our analysis, we assumed that d_1 takes on a value

Table 7.2: Sub-step Probabilities

Sub-step Probabilities			
$P(S_0) = 0.361$	$P(S_1) = 0.129$	$P(S_2) = 0.132$	$P(S_3) = 0.042$
$P(S_4) = 0.251$	$P(S_5) = 0.000$	$P(S_6) = 0.043$	$P(S_7) = 0.042$

in range $(e_1, 4e_1]$ with *uniform* probability at the beginning of step S_0. Similar assumption was applied to the step S_4. In fact, initially $d \approx e$. Therefore, our analysis is not precise for the first and the fourth row of B.

We have performed an experiment to determine the values of $P(S_i|S_j)$ for all combinations of $i,\, j \in [0,\, 7]$. The following sub-step transition matrix $B = [b_{(i,\, j)}]$ shows our experimental results,

$$B = \begin{bmatrix} 0.336 & 0.062 & 0.038 & 0.056 & 0.507 & 0.0 & 0.0 & 0.0 \\ 0.394 & 0.302 & 0.304 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.453 & 0.273 & 0.274 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.503 & 0.497 \\ 0.505 & 0.043 & 0.090 & 0.087 & 0.274 & 0.0 & 0.0 & 0.0 \\ 0.499 & 0.248 & 0.253 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.054 & 0.470 & 0.473 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.503 & 0.497 \end{bmatrix}. \tag{7.4}$$

Note that more than half of the elements in matrix B are zeros. This implies that some combinations of sub-step sequences never occur. Even though there does not appear to be an easy way to prove all the probabilities in matrix B, all of the zero entries can be easily proven. The zero occurrences in B are important. No matter how small the probability is. It will be shown in Section 7.3 that the exact values of probability are not important.

Table 7.2 shows probabilities of each sub-step. The results in Table 7.2 can be obtained by computing the steady state vector using B. Note that the probability $P(S_5)$ is shown to be zero in Table 7.2, but it is not exactly zero. This is because the sub-step S_5 can appear only once at the beginning and it does if only if the input exponents a and b satisfy the condition, $a > 4b$. We observe from the matrix B that no sub-step can be followed by S_5. Hence S_5 must appear only once at the beginning of sub-step sequence.

We have determined $P(Q_1 Q_2 \cdots Q_n)$ up to $n = 4$ to identify impossible sub-step sequences, and we list them in Table 7.3. Only the sub-step sequences that are not obvious

Table 7.3: Impossible Sub-step Sequences

Impossible sub-step sequences	
$S_0S_4S_4S_4$	$S_4S_4S_4S_4$
$S_1S_0S_0$, $S_1S_0S_1$ $S_1S_0S_2$, $S_1S_0S_3$, $S_1S_0S_4S_1$	$S_5S_0S_0$, $S_5S_0S_1$ $S_5S_0S_2$, $S_5S_0S_3$, $S_5S_0S_4S_1$
$S_2S_0S_1$, $S_2S_0S_2$, $S_2S_0S_3$ $S_2S_0S_0S_2$, $S_2S_0S_0S_4$, $S_2S_0S_4S_1$	$S_6S_0S_1$, $S_6S_0S_2$, $S_6S_0S_3$ $S_6S_0S_0S_2$, $S_6S_0S_0S_4$, $S_6S_0S_4S_1$
$S_3S_6S_0S_0$, $S_3S_7S_6S_0$	$S_7S_6S_0S_0$, $S_7S_7S_6S_0$

from the matrix B are listed in Table 7.3.

Graphical representation of our Markov chain model is given in Figure 7.1. Notice that there is no path leading to S_5. Thus, our Markov chain model is not irreducible. For our purpose, we can simply ignore S_5, since it can only occur once in the beginning of the algorithm.

7.2.3 Determining f_2, f_3 and Sub-step Sequence

Determining f_2 and f_3 is quite simple. In lines 18 and 19, the algorithm uses f_2 times of D and f_3 times of T. Hence, we only need to count these operations to determine the values of f_2 and f_3. Alternatively, we can determine them at lines 3 and 6 by counting the number of bit shifts and divisions by 3, though this seems to be harder.

Apparently, there seems to be no easy way to analyze the expected number of tries until the correct key pair is found. Moreover the fact that there are many impossible sub-step sequences listed in 7.3 makes things even more complicated. Nevertheless, if we take into account only the impossible sub-step sequences $S_1S_0S_3$, $S_2S_0S_3$, $S_5S_0S_3$, $S_6S_0S_3$, we can compute the expected number of possible exponent pairs for a randomly given elementary XTR operation sequence. For example, let S_1 be the current sub-step and the next elementary XTR sequence to be processed is ADD, then there are only 2 possible choices for determining the next sub-step: S_1 or S_2. The sub-step sequence S_0S_3 also leads to ADD but $S_1S_0S_3$ is an impossible sequence.

We give all the detailed calculations in Table 7.4. The third column of Table 7.4 lists $P(q|S_i)$'s, the probabilities of observing elementary XTR operation sequence q after sub-step S_i, for $i = 0, \dots 7$ and $q = A(A)$ (an A which is not directly followed by DD), ADD (which comes from a single sub-step or two consecutive sub-steps) and DD. The fourth column lists

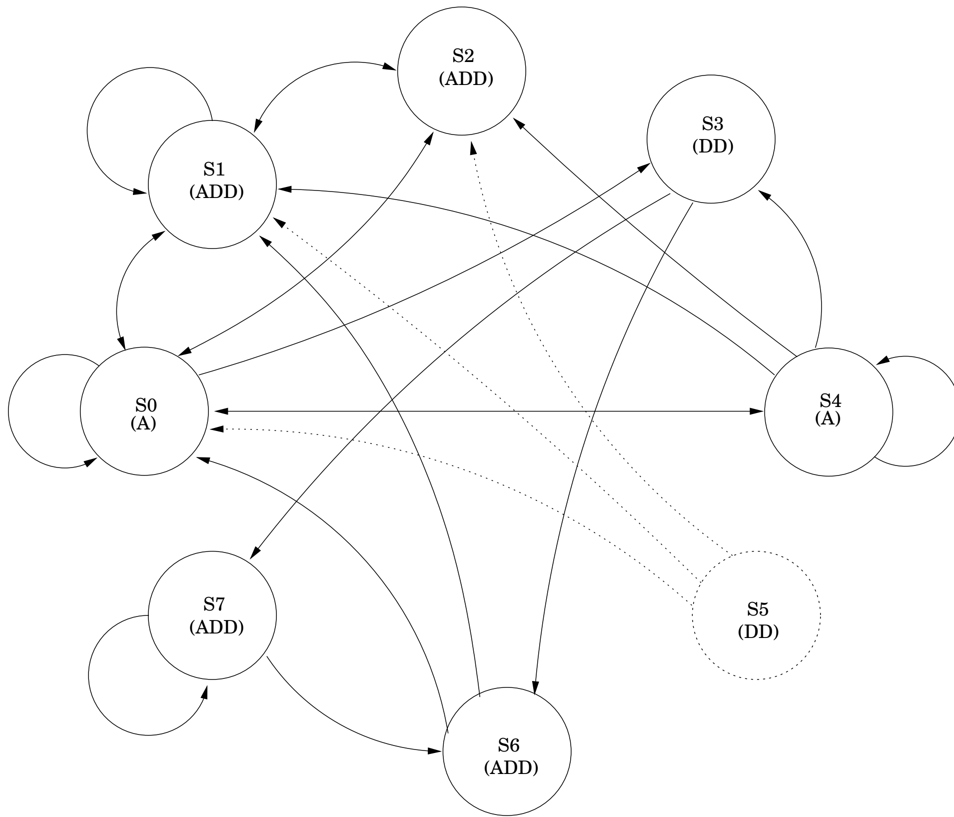

Figure 7.1: State Transition Diagram of Sub-steps

Table 7.4: Expected Number of Choices at a Sub-step

S_i	q	$P(q\|S_i)$	# Choices	E(# Choices$\|S_i$)	E(# Choices$\|S_i) \cdot P(S_i)$
S_0	$A(A)$	0.780	2		
	ADD	0.163	4	2.269	0.819
	DD	0.057	1		
S_1	$A(A)$	0.394	1	1.606	0.207
	ADD	0.606	2		
S_2	$A(A)$	0.450	1	1.550	0.205
	ADD	0.550	2		
S_3	ADD	1.000	2	2.000	0.084
S_4	$A(A)$	0.700	2		
	ADD	0.212	4	2.337	0.587
	DD	0.088	1		
S_5	$A(A)$	0.501	1	1.499	0.000
	ADD	0.499	2		
S_6	$A(A)$	0.054	1	1.946	0.084
	ADD	0.946	2		
S_7	ADD	1.000	2	2.000	0.084
Expected # of choices for each sub-step					2.069

the number of possible choices for the next sub-step given a current sub-step S_i and one of $A(A)$, ADD or DD, the next elementary XTR operation sub-sequence.

Now, using the sub-step probabilities in Table 7.2 and the number of multiplications for sub-steps in Tables 3.1 and 3.2, the average number of multiplications required for each sub-step is easily calculated to be 4.426. Since the average number of multiplication required for Algorithm 3.3 is conjectured to be $6\log_2 \max(a,\, b)$ as in Conjecture 1, the average number of sub-steps required in one double exponentiation is $6\log_2 \max(a,\, b)/4.426 = 1.356\log_2 \max(a,\, b)$. The expected size of search space is $2.069^{1.356\log_2 U} = U^{1.422}$ for a randomly given sub-step sequence. If we utilize more impossible cases from Table 7.3, the search space will be reduced significantly.

We have experimentally determined the average number of tries required for a randomly given XTR operation sequence. Figure 7.2 shows the results. The slope of the line, for which only four impossible cases are considered, is determined to be about 1.418 showing that our calculation is quite close to the correct value. The slope of the other line, for which we used all impossible cases in Table 7.3, is determined to be about 1.25. Even though the range of

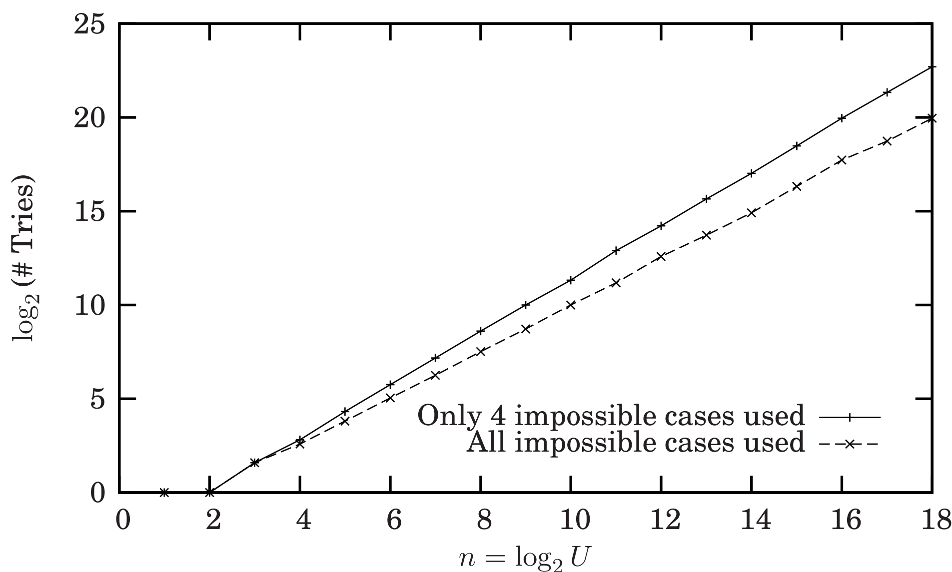

Figure 7.2: The Average Number of Tries Required and the Size of Search Space

n is limited to 18 only, the graph looks straight enough to believe that it takes about $U^{1.25}$ tries until the correct key pair is found for longer size exponents.

7.2.4 Determining d at Line 17 of Algorithm 3.3

Let us denote d at line 17 of Algorithm 3.3 by g. Suppose an adversary somehow knows that $g = 1$, then he/she will just ignore Line 17 since no computation occurs there. However if $g \neq 1$ he/she has to determine the exact value of it by attacking Algorithm 3.2, or attack Algorithm 3.3. This could be an obstacle especially if g is large. However we have found that g is very small in most cases. In fact, our rigorous calculation shows that $g = 1$ occurs about 91.2% of the time and the expected value is only 34.14 assuming 160-bit input exponents.

Let us denote d at line 17 of Algorithm 3.3 by g and (d, e) after line 8 by (d', e'). Then it is not difficult to see that $g = \gcd(d', e')$. In the following subsections, we rigorously calculate $Pr(\gcd(d', e') = 1)$ and the expected value of $\gcd(d', e')$. In our computations, we assume that $1 \leq d', e' \leq n$, unless otherwise specified.

Suppose that we pick two integers a and b independently and randomly from interval

$[1,\, n]$. Then, for large n, it is easily seen that the probability that p divides $\gcd(a,\, b)$ is,

$$P(p|\gcd(a,\, b)) = P(p|a)P(p|b) = \left(\frac{\lfloor n/p \rfloor}{n}\right)^2 \approx \frac{1}{p^2}, \tag{7.5}$$

where $p \in [1,\, n]$. We define p_i to be the i-th prime number, i.e., $p_1 = 2$, $p_2 = 3$, *etc*. Then,

$$P(g = 1) = \prod_{i=3}^{s} P(p_i \nmid \gcd(a,\, b)) \approx \prod_{i=3}^{s} \left(1 - \frac{1}{p_i^2}\right), \tag{7.6}$$

where p_s is the largest prime number in $[1,\, n]$. Note that the product in (7.6) begins from $i = 3$, since there is no common factor of 2 or 3 between d' and e'.

It is well-known that the probability of choosing two co-primes from infinite interval is $[\zeta(2)]^{-1} = 6/\pi^2$, where $\zeta(z)$ is the Riemann zeta function. For large n,

$$P(\gcd(a,\, b) = 1) \approx \prod_{i=1}^{s} \left(1 - \frac{1}{p_i^2}\right) \approx \frac{6}{\pi^2}. \tag{7.7}$$

Therefore,

$$\begin{aligned} P(g = 1) &\approx \frac{P(\gcd(a,\, b) = 1)}{(1 - 1/p_1^2)(1 - 1/p_2^2)} \\ &\approx \frac{6/\pi^2}{(1 - 1/2^2)(1 - 1/3^2)} \approx \frac{9}{\pi^2} = 0.91189. \end{aligned} \tag{7.8}$$

Now, we compute the expected value of $\gcd(a, b)$. Let $E(x)$ denote the expected value of x. Then,

$$\begin{aligned} E(g) &= \sum_{x=1}^{n} x \cdot P(g = x) \\ &= \sum_{p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s} \le n} p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s} P(g = p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s}), \end{aligned} \tag{7.9}$$

where $0 \le k_i \le \lfloor \log_{p_i} n \rfloor$ for $i = 3, \ldots, s$. Since having $p_i^{k_i}$ as a common factor and $p_j^{k_j}$ as a

common factor are independent events if $i \neq j$,

$$\begin{aligned} P(g = p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s}) &= \prod_{l=3}^{s} P(p_l^{k_l} | g \cap p_l^{k^l+1} \nmid g) \\ &\approx \prod_{l=3}^{s} \left(\frac{1}{p_l^2}\right)^{k_l} \cdot \left(1 - \frac{1}{p_l^2}\right) \ . \end{aligned} \tag{7.10}$$

Substitute (7.10) in (7.9) to get,

$$\begin{aligned} E(g) &\approx \sum_{p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s} \leq n} \prod_{l=3}^{s} \frac{1}{p_l^{k_l}} \cdot \left(1 - \frac{1}{p_l^2}\right) \\ &\approx \frac{9}{\pi^2} \cdot \sum_{p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s} \leq n} \frac{1}{p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s}} \\ &\approx \frac{9}{\pi^2} \cdot Z_n, \end{aligned} \tag{7.11}$$

where,

$$Z_n = \sum_{i=0}^{\lfloor (n-1)/6 \rfloor} \frac{1}{6i+1} + \sum_{i=0}^{\lfloor (n-5)/6 \rfloor} \frac{1}{6i+5}. \tag{7.12}$$

Let H_n be the n-th harmonic number, $H_n = \sum_{i=1}^{n} 1/i$ and let H'_n be defined as,

$$H'_n = \sum_{i=1}^{n} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots \ . \tag{7.13}$$

It is well-known that $H'_n = \ln 2 + (-1)^n [H_{(n-1)/2} - H_{n/2}] \approx \ln 2$ for large n. Then it is not too hard to prove that,

$$Z_n \approx \frac{H_n}{2} - \frac{H_{n/3}}{6} + \frac{\ln 2}{3}. \tag{7.14}$$

Since $H_n \approx \ln n + \gamma + 1/2n$, where γ is Euler-Mascheroni constant,

$$Z_n \approx \frac{\ln(12n^2)}{6} + \frac{\gamma}{3}. \tag{7.15}$$

The numerical value of γ is,

$$\gamma \approx 0.57721566490153286060651209008240243104 2\ldots$$

However, since Z_n is a partial sum of harmonic series, error may accumulate due to the approximation in (7.5). If we use exact probability $P(p|a) = \lfloor n/p \rfloor / n$, we have better approximation for Z_n, which we denote Z'_n.

$$\begin{aligned}
Z'_n &\approx \sum_{p_3^{k_3} p_4^{k_4} \cdots p_s^{k_s} \le n} \prod_{i=3}^{s} \frac{\lfloor n/p_i \rfloor^{k_i}}{n^{k_i}} \\
&= 1 + \frac{\lfloor n/5 \rfloor}{n} + \frac{\lfloor n/7 \rfloor}{n} + \frac{\lfloor n/11 \rfloor}{n} + \frac{\lfloor n/13 \rfloor}{n} + \frac{\lfloor n/17 \rfloor}{n} + \frac{\lfloor n/19 \rfloor}{n} + \frac{\lfloor n/23 \rfloor}{n} \\
&+ \frac{\lfloor n/5 \rfloor^2}{n^2} + \frac{\lfloor n/29 \rfloor}{n} + \frac{\lfloor n/31 \rfloor}{n} + \frac{\lfloor n/5 \rfloor \cdot \lfloor n/7 \rfloor}{n^2} + \cdots \\
&= 1 + \frac{\lfloor n/5 \rfloor}{n} + \frac{\lfloor n/7 \rfloor}{n} + \frac{\lfloor n/11 \rfloor}{n} + \frac{\lfloor n/13 \rfloor}{n} + \frac{\lfloor n/17 \rfloor}{n} + \frac{\lfloor n/19 \rfloor}{n} + \frac{\lfloor n/23 \rfloor}{n} \\
&+ \frac{\lfloor n/25 \rfloor}{n} + \frac{\lfloor n/29 \rfloor}{n} + \frac{\lfloor n/31 \rfloor}{n} + \frac{\lfloor n/35 \rfloor}{n} \\
&= \frac{1}{n} \left(\sum_{i=0}^{\lfloor (n-1)/6 \rfloor} \left\lfloor \frac{n}{6i+1} \right\rfloor + \sum_{i=0}^{\lfloor (n-5)/6 \rfloor} \left\lfloor \frac{n}{6i+5} \right\rfloor \right).
\end{aligned} \tag{7.16}$$

Then the difference between Z_n and Z'_n is,

$$\begin{aligned}
E_n &= Z_n - Z'_n \\
&\approx \frac{1}{n} \left(\frac{n \bmod 5}{5} + \frac{n \bmod 7}{7} + \frac{n \bmod 11}{11} + \frac{n \bmod 13}{13} + \cdots \right).
\end{aligned} \tag{7.17}$$

Due to de la Vallée Poussin [20],

$$\gamma = \lim_{n \to \infty} \frac{1}{n} \cdot \sum_{i=1}^{n} \left(\left\lfloor \frac{n}{i} \right\rfloor - \frac{n}{i} \right). \tag{7.18}$$

Using this fact, it is not too hard to prove that $E_n \approx (1 - \gamma)/3$. Therefore,

$$E(g) \approx \frac{9}{\pi^2} \cdot Z'_n = \frac{3}{\pi^2} \left(\frac{\ln(12n^2)}{2} + 2\gamma - 1 \right). \tag{7.19}$$

Hence, for $n = 2^{160}$, $E(g) \approx 34.135$.

We have tried to validate these results experimentally. We picked 10^6 set of randomly integers a and b from interval $[1,\, 2^{160}]$, where a and b do not have 2 and 3 as common factors. Then we counted the number of occurrences of $\gcd(a,\, b) = 1$. The estimated probability was consistently about 0.9119.

For the validation of the expected value $E(g)$, we tried to average the GCD's from the first experiment, but it turned out that the number of experiment, 10^6, is too small compared to $n = 2^{160}$. As a result we could only get unstable average values between 3 and 4. So, we used a small value of n. Figure 7.3 compares the actual average values of $\gcd(d',\, e')$ and expected values computed from (7.19) for $n \leq 10000$.

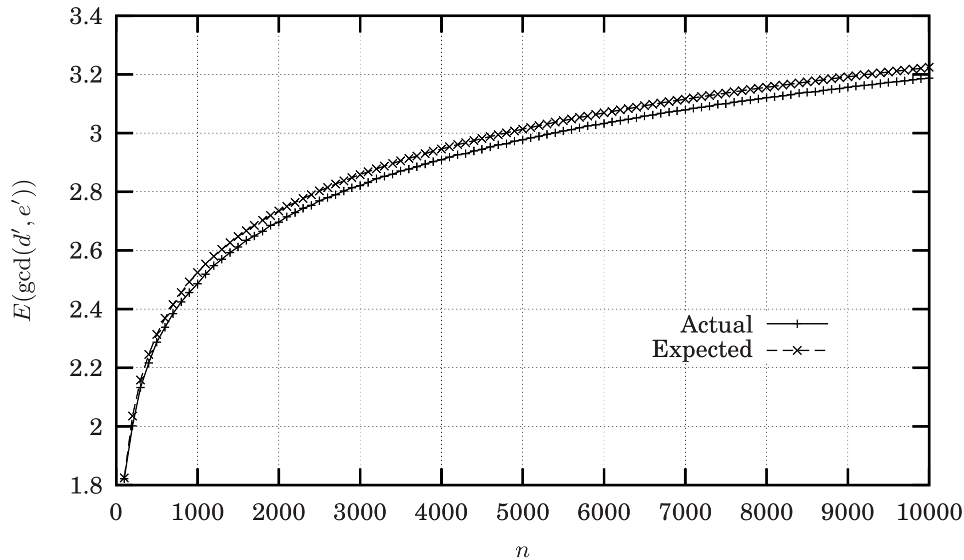

Figure 7.3: Comparison Between Actual Values and (7.19)

We observe from Figure 7.3 the small, but noticeable difference between actual values and expected values computed from (7.19). However, the error is very small. According to Figure 7.4 we can see that ϵ converges to a constant value between 0.035 and 0.040.

Suppose that Algorithm 3.2 is used at line 17. It is easy enough to see from the description of Algorithm 3.2 that an adversary can determine the bit length and the least significant bit of g. Note that to make Algorithm 3.1 resistant against side channel attack, the elementary XTR operation sequences for the two cases in lines 5 and 9 should be identi-

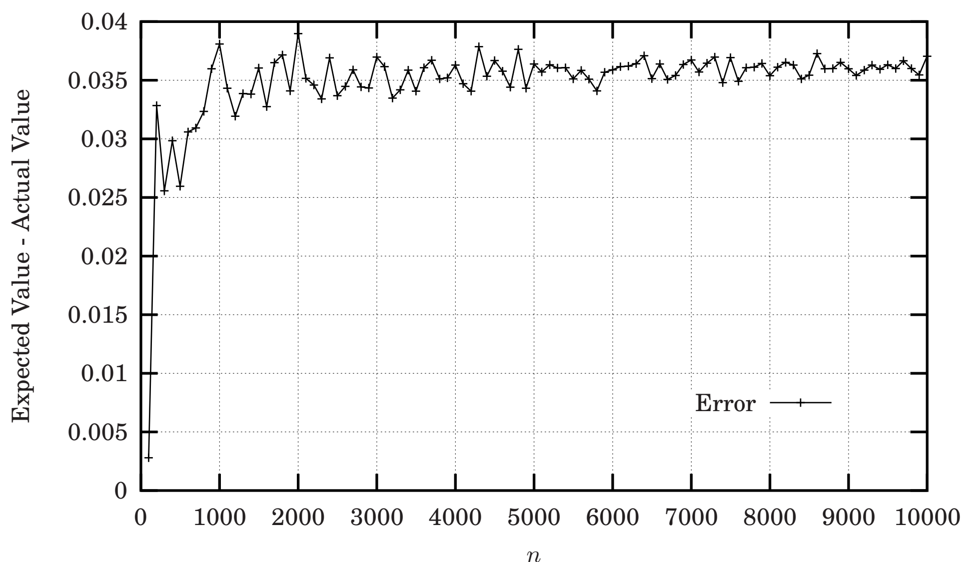

Figure 7.4: Error Between Actual Values and (7.19)

cal [53]. The bit length of g can be obtained by counting the number of repetitions of ADD, DAD or DDA during the execution of Algorithm 3.1. If Algorithm 3.2 is used at line 17, an adversary has to attack Algorithm 3.3 recursively.

7.2.5 Determining Boundaries Between Steps

We now discuss whether it is possible to determine the exact boundaries between steps in Algorithm 3.3 by observing an elementary XTR operation sequence.

- **When** d = 1 **at line** 17

  If $d = 1$, no elementary XTR operation is performed. The elementary XTR operation sequence from line 16 to the end of the algorithm will then look like:

$$A\underbrace{DD\cdots D}_{f_2}\underbrace{TT\cdots T}_{f_3}.$$

  Since operation T can only occur at line 19, it can be clearly identified. Whether $f_2 = 0$ or not, one needs to look for the last A. Then it must be from line 16. Operation D's

between A and T are from line 18. Since $d = 1$ happens about 91.2% of the time, boundaries between steps can be found easily in most cases.

- d $\neq$ 1 **and Algorithm 3.2 is used at line** 17
  Suppose lines 5 and 9 in Algorithm 3.1 leave elementary XTR operation sequence ADD (DAD and DDA are also possible but they all lead to the same result). Then the elementary XTR operation sequence from line 16 to the end of the algorithm will look like:

$$AD\underbrace{ADDADD\cdots ADD}_{\lfloor\log_2(d)\rfloor}\underbrace{DD\cdots D}_{f_2}\underbrace{TT\cdots T}_{f_3}.$$

  Note that the first D occurs when constructing $S_1(c_{u+v})$ from c_{u+v}. An adversary can clearly determine that $d \neq 1$, if he/she observes the sequence AD followed by A. Note that the operation A from line 4 of Algorithm 3.1 never occurs, since d at line 17 cannot be a multiple of 2.

- d $\neq$ 1 **and Algorithm 3.4 is used at line** 17
  In this case, there appears to be no easy way to locate the boundaries between steps and an exhaustive search is needed for the value of d by trying from the smallest possible one. Note that the adversary only has to try values that are not multiple of 2 or 3.

Therefore we conclude that boundaries between steps are exactly identified if Algorithm 3.2 is used. If Algorithm 3.2 is used for line 17, it is not clear how to determine the boundaries. In that case, exhaustive search has to be done. Note that this does not seriously harm the effectiveness of the attack. Using the fact that $d = 1$ occurs 91.2% of the time and that the probability significantly decreases as d increases, it is expected that an attacker should guess the exact d by trying only a few values.

7.3 Effectiveness of Markov Chain Method

In [67], Oswald shows how the Markov chain method can help enhancing simple power analysis attacks on elliptic curve point multiplication algorithms. However such a method is not very useful for the attack described here.

In [67], it was possible to partition an elliptic curve operation sequence into several partitions of small lengths such that only 3-bit patterns are possible for each partition. Using this weakness, Oswald could prune significant number of spurious keys. However, apparently there is no such a weakness found in Algorithm 3.3.

In fact, we have found that the Markov method itself may not be so effective in practice as it is claimed. According to [67], an elliptic curve operation sequence resulted from the point multiplication algorithm [62] are broken into length-l partitions, where for each partition only 3-bit patterns are possible. Hence, an adversary has to try all $3^{3n/2l}$ keys in the worst case, where n is the bit length of the key. According to [67], one of the 3 bit patterns occurs with probability 1/2 and the others 1/4 each. However, it has been reported in [67] that, when an adversary takes into account the probabilities for the 3-bit patterns, the expected number of tries is $2^{3n/2l}$, which appears to be incorrect.

Suppose that there are k partitions. We assume that an adversary always tries from the highest probable keys to the least probable one. When $k = 1$, the expected number of tries in this case is $1 = 1 \cdot 1/2 + 2 \cdot 1/4$ (If the first two tries fail, the third one must be the correct key). Now, suppose that there are two partitions, i.e., $k = 2$. Note carefully that an adversary can only test complete keys but not partial key bits, since it is assumed that the adversary only knows a plaintext/ciphertext pair. The expected number of tries in this case is calculated as $3.3125 = (1/2)^2 + (2 + 3 + 4 + 5) \cdot (1/2)(1/4) + (6 + 7 + 8) \cdot (1/4)^2$.

Since it is difficult to generalize this calculation for any k, we used computer program to do this calculation up to $k = 50$ partitions and Figure 7.5 shows the result.

We easily see from Figure 7.5 that the difference between the average and the worst cases is not very significant and our calculation is far from that of [67]. For $l = 16$ and $n = 163$, the number of partitions is 15.28. In such a case, the expected number of tries is about 2^{22} as shown in Figure 7.5. However in [67], it has been reported that the expected number of tries is $2^{15.28}$, which does not match with our result.

Nevertheless, Figure 7.5 shows an evidence that the Markov method does result in a significant exponential improvement for cases such as one pattern occurs with very high probability, but the others very small probability. In such a case, as seen from Figure 7.5, the Markov method significantly saves the number of tries. The result implies that the Markov method is useful only when the probabilities are significantly 'skewed'.

The Markov method may not be useful for the attack described here, as the probabilities for sub-steps (see Table 7.2 and the matrix B in section 7.2) do not appear to be skewed

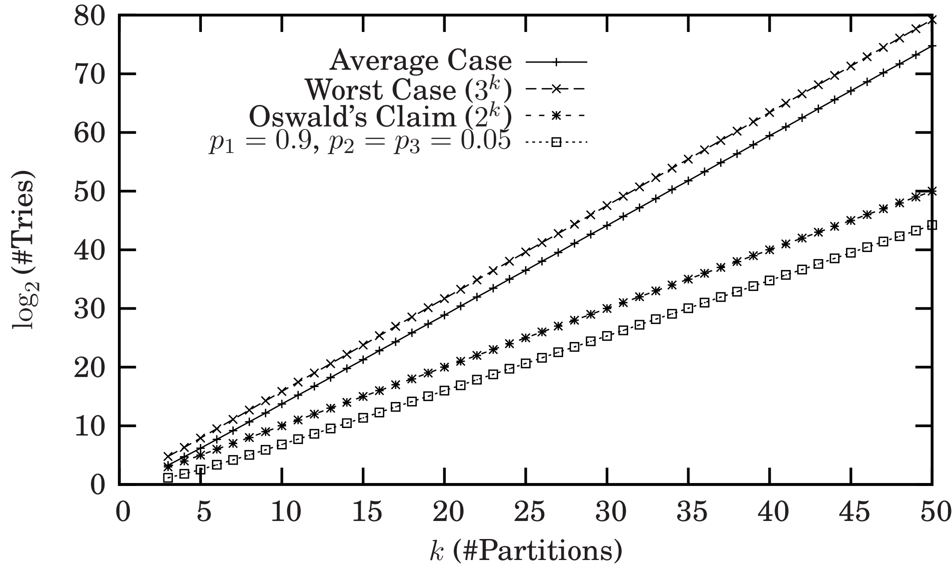

Figure 7.5: Number of Tries

enough. Moreover it would be infeasible to sort all the exponential number of spurious keys according to their probabilities.

7.4 Extension to Single Exponentiation Algorithms

Our cryptanalysis result can be directly applied to Algorithm 4.2 in [77] and Algorithm 3.4. Note that the side channel attack is meaningful only if it is applied to single exponentiation algorithms, but not the double exponentiation algorithms. Since double exponentiation algorithms are usually used in signature verification [66, 23, 65], all the input values to a double exponentiation algorithm are public information. However, Algorithm 3.3 is used for the efficient implementation of single XTR exponentiation algorithms.

- Algorithm 4.2 in [77]
  Suppose the input exponent U is n bits long. Then the exponent is split into two $n/2$-bit integers. These two integers are the input exponents to Algorithm 3.3. Therefore our cryptanalysis is expected to take $\sqrt{U}^{1.25} = U^{0.625}$ tries until the correct exponent is found.

- Algorithm 3.4
  Suppose the input exponent U is n bits long. Exponent is split into two integers a and b at step 1, but they have almost the same bit length as U. However, according to Proposition 2, for the first $\log_\Phi \sqrt{U}$ iterations, where $\Phi = (3 - \sqrt{5})/2$, only S_0 is executed and the values of d and e reduce to about $\sqrt{U}$ after first $\log_\Phi \sqrt{U}$ iterations. This means one only needs to attack the second half of the algorithm. Therefore it takes about $U^{0.625}$ tries until the correct exponent is found.

7.5 Other Researchers' Results

Since the availability of this work [14] in public domain, other researchers have also tried to attack XTR cryptosystems in various ways.

In [31], Han et. al. show that Algorithm 3.2 is vulnerable to SPA, data-bit DPA [18], address-bit DPA [36] and doubling attack [24]. They also propose countermeasures for these attacks. The field isomorphism method is proposed to thwart data-bit DPA and doubling attack, but it slows down the performance about 129 times for 170-bit p. In [30], Han et. al. apply the refined power analysis attack [27] and the zero-value attack [2] to attack Algorithm 3.2. They propose random exponent splitting method as a countermeasure, but it nearly doubles the computation time. However, in [9], Bevan shows that all of the input values used in [30] in attacking Algorithm 3.2 are not valid. Moreover, Bevan shows that Algorithm 3.2 becomes a finite state machine for the input values that are used in [30].

In [69], the authors use the same approach as ours, but have obtained a better results. According to their experiments, it takes on average $U^{1.09}$ tries to find correct exponent pairs (d, e). For single exponentiation, $U^{0.55}$ tries are required on average. Unfortunately, no details are given on how their experiments have been conducted. They propose several SPA and DPA counter measures for Algorithms 3.2 and 3.3.

In [16], fault analysis attacks [10, 11, 39, 89] on Algorithm 3.2 are discussed. They consider four plausible situations where fault may harm the security of XTR cryptosystems; 1) random bit-fault on a random $S_k(c)$, 2) random faults on a chosen c_i, 3) erasing faults on a coordinate of c_{k+1} of a random $S_k(c)$, 4) random bit-faults on the secret exponent.

In [32], the authors use simple side channel attack on Algorithm 3.3. They use additional assumption that adversaries can detect whether the operands are equal in two D operations. They conclude that about 2^{40} tries are required to break Algorithm 3.3 for 160-bit exponents.

7.6 Conclusions

In this chapter, we have analyzed the XTR double exponentiation algorithm against a side channel attack using an assumption that an adversary has the ability to distinguish between multiplication and modular reduction (or Montgomery reduction). We have applied the Markov chain model used in [67] to obtain statistical behavior of Algorithm 3.3. Our analysis shows that $U^{1.25}$ tries on average where $U = \max(a, b)$ are needed to find the correct exponent pair (a, b) for Algorithm 3.3. It immediately follows that an adversary is expected to make $U^{0.625}$ tries on average until he/she finds the correct input exponent to Algorithm 4.2 in [77] and Algorithm 3.4. We also remark that the Markov chain model presented in [67] may not be useful in the attack described here.

We remark that the side channel attack shown in this chapter is not computationally better than well-known square-root type algorithms (baby step giant step or Pollard's Rho algorithms). Such square-root algorithms require only $O(\sqrt{U})$ efforts. However, our results are obtained under a very simple assumption. More sophisticated attackers may be able to extract more information from side channel leakages, and further research has yet to be done.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, low-weight polynomial form integers (LWPFIs) have been introduced to devise an efficient modular multiplication method. In addition, new squaring formulae based on the Toom-Cook multiplication algorithm and a side channel attack on XTR cryptosystems have been presented.

In Chapter 4, LWPFI are defined as a family of integers expressed in polynomial form $p = f(t)$, where t is a positive integer and $f(t)$ is a monic polynomial whose coefficients are limited to 0 and ± 1. Our analysis shows that LWPFI modular multiplication has better asymptotic behavior than other general modular reduction methods. Our implementation results show that LWPFI modular multiplication is faster than Montgomery reduction for moduli of large sizes. We have shown techniques that can speed up LWPFI modular multiplication.

In Chapter 5, we have slightly extended the low-weight polynomial form integers. Improved coefficient reduction algorithm based on the Montgomery reduction algorithm and its analysis results have been presented. The bound on the input and output has been analyzed and the conditions for eliminating the final subtractions have been determined. We have also presented methods for performing additions and subtractions modulo an LWPFI in a carry/borrow-free manner. We have also considered applying our coefficient reduction algorithm to modular number systems proposed by Bajard et. al. but have not been able to find a good F for efficient coefficient reduction.

GMN or pseudo-Mersenne number based modular multiplication would be faster than LWPFI based one, however there are not that many GMNs and pseudo-Mersenne numbers. LWPFI has its advantage that the implementation does not have to be specific to a single modulus and that LWPFI provides a considerably larger choice of moduli than GMN. Hence, one may consider LWPFI as a trade-off between general integer and other special type of moduli such as GMNs and pseudo-Mersenne numbers.

In Chapter 6, we have presented new 3-way polynomial squaring formulae. Our squaring squaring formulae enhance modular squaring modulo an LWPFI, since the first step of modular multiplication using an LWPFI moduli is a multiplication in $\mathbb{Z}[x]/f(t)$. Our squaring formulae are based on the Toom-Cook multiplication algorithm and they require the same number of coefficient multiplications used in the Toom-Cook multiplication algorithm. However, our 3-way formulae have less amount of overhead than the best known 3-way Toom-Cook multiplication algorithm. Moreover, our formulae do not require any nontrivial constant divisions which always occur in all n-way Toom-Cook multiplication algorithms for $n \geq 3$. Our experimental results confirm that one of our 3-way formulae is superior to GMP's squaring routine for squaring integers of size 2300–6900 bits on Pentium IV 3.2GHz. Moreover, according to our implementation results, our squaring formulae are the best for squaring degree-2 polynomials in $\mathbb{Z}[x]$ whose coefficients are shorter than 1216 bits on the same processor. However, symmetric squaring algorithms are advantageous for squaring very large size operands, since our asymmetric squaring algorithms use at least one pointwise multiplication that cannot be computed by squaring.

In Chapter 7, we have attempted a side channel attack on XTR exponentiation algorithms under the assumption that an adversary has the ability to distinguish between multiplication and modular reduction (or Montgomery reduction). We have shown that an adversary is expected to make about $U^{1.25}$ tries on average where $U = \max(a, b)$ to find the correct exponent pair (a, b) for Algorithm 3.3. It immediately follows that an adversary is expected to make $U^{0.625}$ tries on average until he/she finds the correct input exponent to Algorithm 3.4 and Algorithm 4.2 in [69]. We remark that the Markov chain model presented in [67] may not be useful in the attack described here.

8.2 Future Work

- In this thesis, methods for modular addition, subtraction and multiplication (squaring) have been presented. We have not yet developed a method to perform modular inversion using LWPFI moduli. It appears that modular inverse computation can hardly take advantage of the special form moduli. However, it would be interesting to have a modular inversion algorithm which works on integers represented in polynomial form in t.

- The security aspect of using LWPFI in cryptography needs to be thoroughly investigated. For instance, it is not known whether LWPFI can be used in RSA cryptosystems without degrading its security. It is also not known whether LWPFI can be used in the cryptosystems based on the hardness of the discrete logarithm problem. In [73], it is conjectured that discrete logarithm problem based on integer moduli having low Hamming weight is significantly easier than the number field sieve on general moduli. However, this result does not apply to discrete logarithm problem based on LWPFI moduli, since LWPFIs do not have low Hamming weight.

- There are a number of tricks and enhancement methods for the Montgomery reduction algorithm. For instance, pipelining of sub-steps in a loop significantly improves the critical path delay for hardware implementation. We have not thoroughly considered all techniques with regard to their effective usability in Algorithm 5.2 presented in Chapter 5.

- Throughout this thesis, only software implementation has been considered. It would be very interesting to devise hardware architectures for modular multiplication using LWPFI moduli and integer/polynomial squaring using our asymmetric squaring formulae.

- It would be very interesting to find a formulae for cubing of a polynomial (or integer) which can perform faster than multiply-then-square approach.

Bibliography

[1] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, August 2002.

[2] T. Akishita and T. Takagi. Zero-value point attacks on elliptic curve cryptosystem. In *Information Security Conference - ISC 2003*, LNCS 2851, pages 218–233. Springer-Verlag, 2003.

[3] A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *IRE Transaction on Computers*, EC-10:389–400, 1961.

[4] Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[5] Jean-Claude Bajard, Laurent Imbert, and Thomas Plantard. Modular number systems: Beyond the Mersenne family. In *Selected Areas in Cryptography 2004*, LNCS 3357, pages 159–169. Springer-Verlag, 2004.

[6] Jean-Claude Bajard, Laurent Imbert, and Thomas Plantard. Arithmetic operations in the polynomial modular number system. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH'05, pages 206–213, 2005.

[7] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86*, LNCS 263, pages 311–323. Springer-Verlag, 1987.

[8] Daniel Bernstein. Multidigit multiplication for mathematicians, 1991. Available at `http://cr.yp.to/papers/m3.pdf`.

[9] Régis Bevan. Improved zero value attack on XTR. In *ACISP 2005*, LNCS 3574, pages 207–217. Springer-Verlag, 2005.

[10] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97*, LNCS 1294, pages 513–525. Springer-Verlag, 1997.

[11] D. Boneh, R. DeMillo, and R. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology - EUROCRYPT '97*, LNCS 1233, pages 37–51. Springer-Verlag, 1997.

[12] Wieb Bosma, James Hutton, and Eric R. Verheul. Looking beyond XTR. In *Advances in Cryptology - ASIACRYPT 2002*, LNCS 2501, pages 46–63. Springer-Verlag, 2002.

[13] Jaewook Chung and Anwar Hasan. More generalized Mersenne numbers. In *Selected Areas in Cryptography - SAC 2003*, LNCS 3006, pages 335–347. Springer-Verlag, 2003.

[14] Jaewook Chung and Anwar Hasan. Security analysis of XTR exponentiation algorithms against simple power analysis attack, 2004. Available at `http://www.cacr.math.uwaterloo.ca/techreports/2004/cacr2004-05.pdf`.

[15] Jaewook Chung and M. Anwar Hasan. Low-weight polynomial form integers for efficient modular multiplication, 2006. To appear in IEEE Transactions on Computers. Available at `http://vlsi.uwaterloo.ca/~ahasan/web_papers/technical_reports/web_lwpfi.pdf`.

[16] Mathieu Ciet and Christophe Giraud. Transient fault induction attacks on XTR. In *ICICS 2004*, LNCS 3269, pages 440–451. Springer-Verlag, 2004.

[17] S. A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, May 1966.

[18] J. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems - CHES '99*, LNCS 1717, pages 292–302. Springer-Verlag, 1999.

[19] Richard E. Crandall. Method and apparatus for public key exchange in a cryptographic system (oct. 27, 1992). U.S. Patent # 5,159,632.

[20] C. J. de la Vallée Poussin. Untitled communication. *Annales de la Soc. Sci. Bruxelles*, 22:84–90, 1898.

[21] Jean-François Dhem. Efficient modular reduction algorithm in $\mathbb{F}_q[x]$ and its application to "left to right" modular multiplication in $\mathbb{F}_2[x]$. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, LNCS 2779, pages 203–213, 2003.

[22] Germain Drolet. A new representation of elements of finite fields $GF(2^m)$ yielding small complexity arithmetic circuits. *IEEE Transactions on Computers*, 47(9), 1998.

[23] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[24] P. A. Fouque and F. Valette. The doubling attack: why upwards is better than downwards. In *Workshop on cryptographic hardware and embedded systems - CHES 2003*, LNCS 2779, pages 269–280. Springer-Verlag, 2003.

[25] Freescale Semiconductor, Inc. MCF5307 ColdFire, integrated microprocessor user's manual, 2005. Available at `http://www.freescale.com/files/soft_dev_tools/doc/ref_manual/MCF5307BUM.pdf`.

[26] GNU. GNU multiple precision arithmetic library, 2005. Available at `http://www.swox.com/gmp`.

[27] L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *Public Key Cryptography - PKC 2003*, LNCS 2567, pages 199–211. Springer-Verlag, 2003.

[28] Torbjörn Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, 2005. Available at `http://swox.com/doc/x86-timing.pdf`.

[29] Gaël Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pages 293–301. Springer-Verlag, 2000.

[30] Dong-Guk Han, Tetsuya Izu, Jongin Lim, and Kouichi Sakurai. Modified power-analysis attacks on XTR and an efficient countermeasure. In *ICICS 2004*, LNCS 3269, pages 305–317. Springer-Verlag, 2004.

[31] Dong-Guk Han, Jongin Lim, and Kouichi Sakurai. On security of XTR public key cryptosystems against side channel attacks. In *ACISP 2004*, LNCS 3108, pages 454–465. Springer-Verlag, 2004.

[32] Dong-Guk Han, Tsuyoshi Takagi, Tae Hyun Kim, Ho Won Kim, and Kyo Il Chung. Collision attack on XTR and a countermeasure with a fixed pattern. In *The First International Workshop on Security in Ubiquitous Computing Systems - SecUbiq 2005*, LNCS 3823, pages 864–873. Springer-Verlag, 2005.

[33] M. Hasan. Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pages 93–108. Springer-Verlag, 2000.

[34] Seong-Min Hong, Sang-Yeop Oh, and Hyunsoo Yoon. New modular multiplication algorithms for fast modular exponentiation. In *Lecture Notes in Computer Science*, LNCS 1070, pages 166–177. Springer-Verlag, 1996.

[35] IEEE. *P1363: Standard Specification for Public Key Cryptography*. Institute of Electrical and Electronics Engineers, 2000.

[36] K. Itoh, T. Izu, and M. Takenaka. Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA. In *Workshop on cryptographic hardware and embedded systems - CHES 2002*, LNCS 2523, pages 129–143. Springer-Verlag, 2002.

[37] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Public Key Cryptography - PKC 2002*, LNCS 2274, pages 280–296. Springer-Verlag, 2002.

[38] Tudor Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15:169–180, 1993. Research report version available at `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-35.ps.gz`.

[39] M. Joye, A. Lenstra, and J. Quisquater. Chinese remaindering based cryptosystems in the presence of faults. *Journal of Cryptology*, 12:241–245, 1999.

[40] M. Joye and C. Tymen. Protections against differential analysis for elliptic curve cryptography-an algebraic approach. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, LNCS 2162, pages 377–390. Springer-Verlag, 2001.

[41] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady (English translation)*, 7(7):595–596, 1963.

[42] S. Kawamura, K. Takabayashi, and A. Shimbo. A fast modular exponentiation algorithm. *IEICE Transactions*, E-74(8):2136–2142, August 1991.

[43] D.E. Knuth. *The Art of Comuter Programming, Vol. 2, Seminumerical Algorithms*. Addison-Wesley, 2nd edition edition, 1981.

[44] Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48:203–209, January 1987.

[45] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96*, LNCS 1109, pages 104–113. Springer-Verlag, 1996.

[46] P. Kocher, J. Jeffe, and B. Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99*, LNCS 1666, pages 388–397. Springer-Verlag, 1999.

[47] Werner Krandick and Tudor Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21:441–455, 1996. Early technical report version available at `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1994/94-50.ps.gz`.

[48] Arjen K. Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields. In *Proceedings of Australian Conference on Information Security and Privacy*, pages 127–138. Springer-Verlag, 1997.

[49] Arjen K. Lenstra and H.W. Lenstra Jr. The development of the number field sieve. In *Lecture Notes in Mathematics*, 1554, pages 11–42, 1993.

[50] Arjen K. Lenstra, H.W. Lenstra Jr, M.S. Manasse, and J.M. Pollard. The factorization of the ninth Fermat number. *Mathematics of Computation*, 61(203):319–349, 1993.

[51] Arjen K. Lenstra and Eric R. Verheul. Key improvements to XTR. In *Advances in Cryptology - ASIACRYPT 2000*, LNCS 1976, pages 220–233. Springer-Verlag, 2000.

[52] Arjen K. Lenstra and Eric R. Verheul. An overview of the XTR public key system. In *The Proceedings of the Public Key Cryptography and Computational Number Theory Conference*, 2000.

[53] Arjen K. Lenstra and Eric R. Verheul. The XTR public key system. In *Advances in Cryptology - CRYPTO 2000*, LNCS 1880, pages 1–19. Springer-Verlag, 2000.

[54] Chae Hoon Lim, Hyo Sun Hwang, and Pil Joong Lee. Fast modular reduction with precomputation. In *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC '97)*, pages 65–79, Seoul, 1997.

[55] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[56] T. Messerges. Using second-order power analysis to attack DPA resistant software. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pages 238–251. Springer-Verlag, 2000.

[57] T. Messerges, E. Dabbish, and R. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51:541–552, 2002.

[58] V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85*, LNCS 218, pages 417–426. Springer-Verlag, 1986.

[59] P. L. Montgomery. Evaluation recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, Jan. 1992. Available at ftp://ftp.cwi.nl/pub/pmontgom/Lucas.pz.gz.

[60] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[61] Peter L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transaction on Computers*, 54(3):362–369, 2005.

[62] F. Morain and J. Olivos. Speeding up the computation on an elliptic curve using addition-subtraction chains. *RAIRO: R. A. I. R. O. Informatique Theorique et Applications / Theoretical Informatics and Applications*, 24:119–129, 1990.

[63] D. Naccache and H. M'Silti. A new modulo computation algorithm. *Recherche Opérationnelle - Operations Research (RAIRO-OR)*, 24:307–313, 1990.

[64] National Institute of Standards and Technology. Recommended elliptic curves for federal government use, July, 1999.

[65] K. Nyberg and R. Rueppel. A new signature scheme based on the DSA giving message recovery. In *Proceedings of the First ACM Conference on Computer and Communications Security (ACM CCS 1993)*, pages 58–61. ACM Press, 1993.

[66] National Institute of Standards and Technology. Digital signature standard (DSS). FIPS Publication 186-2, January, 2000.

[67] Elisabeth Oswald. Enhancing simple power-analysis attacks on elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, LNCS 2523, pages 82–97. Springer-Verlag, 2002.

[68] Elisabeth Oswald and Mangred Aigner. Randomized addition subtractions chains as a countermeasure against power attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, LNCS 2162, pages 39–50. Springer-Verlag, 2001.

[69] Daniel Page and Martijn Stam. On XTR and side-channel analysis. In *Selected Areas in Cryptography - SAC 2004*, LNCS 3357, pages 54–68. Springer-Verlag, 2005.

[70] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[71] R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[72] Werner Schindler. A timing attack against RSA with the Chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pages 109–124. Springer-Verlag, 2000.

[73] Oliver Schirokauer. The number field sieve for integers of low weight. Technical report, 2006. IACR ePrint Archive 2006/107.

[74] Arnold Schönhage and Volker Strassen. Schnelle mutiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[75] Oliver Shirokauer. The special function field sieve. *SIAM Journal on Discrete Mathematics*, 16(1):81–98, 2002.

[76] Jerome A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo, 1999. Available at `http://cacr.uwaterloo.ca/techreports/1999/corr99-39.ps`.

[77] Martijn Stam and Arjen K. Lenstra. Speeding up XTR. In *Advances in Cryptology - ASIACRYPT 2001*, LNCS 2248, pages 125–143. Springer-Verlag, 2001.

[78] Berk Sunar. A generalized method for constructing subquadratic complexity $GF(2^k)$ multipliers. *IEEE Transactions on Computers*, 53:1097–1105, 2004.

[79] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Math*, 3:714–716, 1963.

[80] Eric R. Verheul. Evidence that XTR is more secure than supersingular elliptic curve cryptosystems. In *Advances in Cryptology - EUROCRYPT 2001*, LNCS 2045, pages 195–210. Springer-Verlag, 2001.

[81] C. D. Walter. Sliding windows succumbs to Big Mac attack. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, LNCS 2162, pages 286–299. Springer-Verlag, 2001.

[82] C. D. Walter. Some security aspects of the MIST randomized exponentiation algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, LNCS 2523, pages 276–290. Springer-Verlag, 2002.

[83] Colin D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.

[84] Colin D. Walter. Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli. In *Topics in Cryptology - CT-RSA 2002*, LNCS 2271, pages 30–39. Springer-Verlag, 2002.

[85] Colin D. Walter and Susan Thompson. Distinguishing exponent digits by observing modular subtractions. In *Progress in Cryptology - CT-RSA 2001*, LNCS 2020, pages 192–207. Springer-Verlag, 2001.

[86] André Weimerskirch and Christof Paar. Generalization of the Karatsuba algorithm for efficient implementations. Technical report, Ruhr-Universität Bochum, Gemany, 2003. Available at `http://www.crypto.ruhr-uni-bochum.de/en_publications.html`.

[87] Shmuel Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics 33. Society for Industrial and Applied Mathematics, 1980.

[88] Huapeng Wu, M. Anwar Hasan, Ian F. Blake, and Shuhong Gao. Finite field multiplier using redundant representation. *IEEE Transactions on Computers*, 51(11):1306– 1316, 2002.

[89] S. M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49:967–970, 2000.

[90] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.

Appendix A

Source Codes for SQR$_3$ and Improved Zimmermann's 3-way Toom-Cook Squaring Algorithm

We show the source codes that we used to obtain the experimental results presented in Section 6.4. The functions `mpn_sqr_asymmetric3_n()` and `mpn_sqr_zimmermann3_n()` have the same interface as the Toom-Cook squaring routine (`mpn_sqr_toom3_n()`) in GMP. Since we have referenced Harley's Toom-Cook squaring routine in GMP 4.1.4 to implement SQR$_3$, there are some common parts with Harley's code in the beginning (variable declaration and variable setup) and at the end (overlapping and carry adding up). We store carries in separate variables (`carryA`, `carryB`, `carryC`, `carryD`, `carryW` and `tempC`) as Harley did in the Toom-Cook multiplication and squaring routines in GMP 4.1.4.

The argument `a` is the pointer to the operand of length $n \geq 5$. The argument `p` is a pointer to memory where result is stored. There must be at least $2n$ words allocated to `p`. The argument `ws` is a pointer to a working space having at least $6 \cdot \lceil n/3 \rceil$ words.

One can easily replicate the results presented in Section 6.4 by replacing the existing `mpn_sqr_3_n()` function in GMP libary with the following codes. Since the following functions become faster than KA for shorter operands than does the 3-way Toom-Cook multiplication in GMP, one needs to change the threshold value (`SQR_TOOM3_THRESHOLD`) in the file `gmp-mparam.h` between KA and 3-way Toom-Cook squaring algorithm to obtain the optimal results. The threshold varies significantly depending on the architecture used and can

be estimated experimentally.

A.1 Source Code for SQR_3

```
#define MSB 1<<(BITS_PER_MP_LIMB-1)
void mpn_asymmetric3_n (mp_ptr p, mp_srcptr a, mp_size_t n, mp_ptr ws)
{
   mp_limb_signed_t carryB, carryC, carryD, carryW, tempC;
   mp_limb_t *A,*B,*C,*D,*E,*W;
   mp_size_t l,l2,l3,l4,l5,ls;

   ASSERT (n>=5);

   /* Break n words into chunks of size l, l and ls.
   * n = 3*k => l = k, ls = k
   * n = 3*k+1 => l = k+1, ls = k-1
   * n = 3*k+2 => l = k+1, ls = k
   */
   {
      mp_limb_t m;
      l = ls = n / 3;
      m = n - l * 3;
      if (m != 0)
      ++l;
      if (m == 1)
      --ls;
      l2 = l * 2;
      l3 = l * 3;
      l4 = l * 4;
      l5 = l * 5;
      A = p;
      B = ws;
      C = p + l2;
      D = ws + l2;
      E = p + l4;
      W = ws + l4;
```

```
}
/* ws = | W | D | B |
   p  = | E | C | A | */

/* [A] = a_0^2 = S0*/
TOOM3_SQR_REC (A, a, l, W);

/* [E] = a_2^2 = S4 */
TOOM3_SQR_REC (E, a+l2, ls, W);

/* [carryB:B] = a_0 + a_2 */
carryB = mpn_add_n(B, a, a+l2, ls);
if(ls!=l) carryB = mpn_add_1(B+ls, a+ls, l-ls, carryB);

/* [carryD:D] = a_0 + a_1 + a_2, carryD=0,1 or 2 */
carryD = carryB + mpn_add_n(D, B, a+l, l);

/* [carryB:B] = a_0 - a_1 + a_2, carryB= -1, 0 or 1 */
carryB = carryB - mpn_sub_n(B, B, a+l, l);

/* [carryW:W] = (a_0 + a_1 + a_2)^2 = S1, 0 <= carryW <= 6 */
TOOM3_SQR_REC (W, D, l, C);
carryW = carryD*carryD;
if(carryD==1)
{
   carryW +=mpn_lshift(C, D, l, 1);
   carryW +=mpn_add_n(W+l, W+l, C, l);
}
else if (carryD==2)
{
   carryW +=mpn_lshift(C, D, l, 2);
   carryW +=mpn_add_n(W+l, W+l, C, l);
}

/* [carryD:D] = (a_0 - a_1 + a_2)^2 = S2, 0 <= carryD <= 3 */
TOOM3_SQR_REC (D, B, l, C);
carryD = carryB*carryB;
```

```
if(carryB>0)
{
   carryD += mpn_lshift(C, B, l, 1);
   carryD += mpn_add_n(D+l, D+l, C, l);
}
else if (carryB<0)
{
   carryD -= mpn_lshift(C, B, l, 1);
   carryD -= mpn_sub_n(D+l, D+l, C, l);
}

/* [carryC:C] = T1 = (S1+S2)/2 */
carryC= carryW+carryD+mpn_add_n(C, W, D, l2);
mpn_rshift(C, C, l2, 1);
if(carryC&1) C[l2-1]|=MSB;
carryC>>=1;

/* [carryD:D] = S1 - T1 */
carryD=carryW-(carryC+mpn_sub_n(D, W, C, l2));

/* [carryC:C] = T1 - S4 - S0 */
carryC-= mpn_sub_n(C, C, A, l2);
if(ls!=l)
{
   tempC = mpn_sub_n(C, C, E, ls<<1);
   carryC-=mpn_sub_1(C+(ls<<1), C+(ls<<1), l2-(ls<<1), tempC);
}
else
{
   carryC-= mpn_sub_n(C, C, E, l2);
}

/* [B] = a_0 * a_1 */
TOOM3_MUL_REC (B, a, a+l, l, W);

/* [carryB:B] = S3 */
carryB = mpn_lshift(B, B, l2, 1);
```

```
   /* [carryD:D] = S1-T1-S3 */
   carryD-=(carryB+mpn_sub_n(D, D, B, l2));

   /* overlapping */
   carryB+=mpn_add_n(p+l, p+l, B, l2);
   carryD+=mpn_add_n(p+l3, p+l3, D, l2);

   /* adding up carries */
   MPN_INCR_U (p + l3, 2 * n - l3, (mp_limb_t)carryB);
   MPN_INCR_U (p + l4, 2 * n - l4, (mp_limb_t)carryC);
   MPN_INCR_U (p + l5, 2 * n - l5, (mp_limb_t)carryD);
}
```

A.2 Source Code for Improved Zimmermann's 3-way Toom-Cook Squaring Algorithm

```
void mpn_zimmermann3_sqr_n (mp_ptr p, mp_srcptr a, mp_size_t n, mp_ptr ws)
{
   mp_limb_signed_t carryA, carryB, carryC, carryD, carryW;
   mp_limb_t *A,*B,*C,*D,*E, *W;
   mp_size_t l,l2,l3,l4,l5,ls;

   ASSERT (n>=5);
   /* Break n words into chunks of size l, l and ls.
    * n = 3*k   => l = k,   ls = k
    * n = 3*k+1 => l = k+1, ls = k-1
    * n = 3*k+2 => l = k+1, ls = k
    */
   {
      mp_limb_t m;

      l = ls = n / 3;
      m = n - l * 3;
      if (m != 0)
         ++l;
```

```
      if (m == 1)
         --ls;

      l2 = l * 2;
      l3 = l * 3;
      l4 = l * 4;
      l5 = l * 5;
      A = p;
      B = ws;
      C = p + l2;
      D = ws + l2;
      E = p + l4;
      W = ws + l4;
   }

   /* ws = | W | D | B |
       p = | E | C | A | */

   /* [carryB:B] = a_0 + a_2 */
   carryB = mpn_add_n(B, a, a+l2, ls);
  if(ls!=l) carryB = mpn_add_1(B+ls, a+ls, l-ls, carryB);

   /* [carryD:D] = a_0 + a_1 + a_2, carryD = 0, 1 or 2 */
   carryD = carryB + mpn_add_n(D, B, a+l, l);

   /* [carryB:B] = a_0 - a_1 + a_2, carryB = -1, 0, or 1 */
   carryB -= mpn_sub_n(B, B, a+l, l);

   /* [carryC:C] = (a_0 - a_1 + a_2)^2 = c_4 - c_3 + c_2 + c_1 + c_0 */
   TOOM3_SQR_REC (C, B, l, W);
   carryC = carryB*carryB;
   if(carryB>0)
   {
      carryC +=mpn_lshift(W, B, l, 1);
      carryC +=mpn_add_n(C+l, C+l, W, l);
   }
   else if (carryB<0)
```

```
{
   carryC -=mpn_lshift(W, B, l, 1);
   carryC -=mpn_sub_n(C+l, C+l, W, l);
}

/* [carryB:B] = (a_0 + a_1 + a_2)^2 = c_4 + c_3 + c_2 + c_1 + c_0 */
TOOM3_SQR_REC (B, D, l, W);
carryB = carryD*carryD;
if(carryD!=0)
{
   carryB += mpn_lshift(W, D, l, carryD);
   carryB += mpn_add_n(B+l, B+l, W, l);
}

/* [carryD:D] = (4a_2 + 2a_1 + a_0), 0<=carryD<=6 */
carryD=mpn_lshift(D, a+l, l, 1);
carryD+=mpn_add_n(D, D, a, l);
carryW=mpn_lshift(W, a+l2, ls, 2);
if(l!=ls)
{
   carryW+=mpn_add_n(D, D, W, ls);
   carryD+=mpn_add_1(D+ls, D+ls, l-ls, carryW);
}
else
{
   carryD+=(carryW+mpn_add_n(D, D, W, l));
}

/* [carryA:A] = (4a_2 + 2a_1 + a_0)^2 = 16c_4 + 8c_3 + 4c_2 + 2c_1 + c_0 */
carryA=carryD*carryD;
TOOM3_SQR_REC (A, D, l, W);

if(carryD==1)
{
   carryW = mpn_lshift(W, D, l, 1);
   carryA += (carryW+mpn_add_n (A+l, A+l, W, l));
}
```

```
if (carryD==2)
{
   carryW = mpn_lshift(W, D, l, 2);
   carryA += (carryW+mpn_add_n(A+l, A+l, W, l));
}
if (carryD==4)
{
   carryW = mpn_lshift(W, D, l, 3);
   carryA += (carryW+mpn_add_n(A+l, A+l, W, l));
}
if(carryD!=0)
{
   carryA += mpn_addmul_1(A+l, D, l, carryD<<1);
}

/* [carryD:D] = 2(c_4 - c_3 + c_2 - c_1 + c_0) */
carryD=(carryC<<1)+mpn_lshift(D, C, l2, 1);

/* [carryD:D] = 18c_4 + 6c_3 + 6c_2  + 3c_0 */
carryD+=(carryA+mpn_add_n(D, D, A, l2));

/* [carryD:D] = 6c_4 + 2c_3 + 2c_2 + c_0 */
carryD = ((carryD - mpn_divexact_by3 (D, D, l2))
       * MODLIMB_INVERSE_3) & GMP_NUMB_MASK;

/* [A] = a_0^2 = c_0 */
TOOM3_SQR_REC (A, a, l, W);

/* [carryD:D] = 3c_4 + c_3 + c_2 +  c_0 */
carryD+=mpn_add_n(D, D, A, l2);
mpn_rshift(D, D, l2, 1);
if(carryD&1)
{
   D[l2-1]|=MSB;
}
carryD>>=1;
```

```
/* [E] = a_2^2 = c_4 */
TOOM3_SQR_REC (E, a+l2, ls, W);

/* [carryW:W] = 2c_4 */
carryW=mpn_lshift(W, E, ls<<1, 1);

/* [carryD:D] = c_4 + c_3 + c_2 + c_0 */
if(l!=ls)
{
   carryW+=mpn_sub_n(D, D, W, ls<<1);
   carryD-=mpn_sub_1(D+(ls<<1), D+(ls<<1), l2-(ls<<1), carryW);
}
else
{
   carryD-=(carryW+mpn_sub_n(D, D, W, l2));
}

/* [carryC:C] = c_4 + c_2 + c_0 */
carryC+=(carryB+mpn_add_n(C, C, B, l2));
mpn_rshift(C, C, l2, 1);
if(carryC&1)
{
   C[l2-1]|=MSB;
}
carryC>>=1;

/* [carryB:B] = c_1 */
carryB-=(carryD+mpn_sub_n(B, B, D, l2));

/* [carryD:D] = c_3 */
carryD-=(carryC+mpn_sub_n(D, D, C, l2));

/* [carryC:C] = c_2 */
carryC-=mpn_sub_n(C, C, A, l2);
if(l!=ls)
{
   carryW=mpn_sub_n(C, C, E, ls<<1);
```

```
        carryC-=mpn_sub_1(C+(ls<<1), C+(ls<<1), l2-(ls<<1), carryW);
    }
    else
    {
        carryC-=mpn_sub_n(C, C, E, l2);
    }

    /* overlapping */
    carryB+=mpn_add_n(p+l, p+l, B, l2);
    carryD+=mpn_add_n(p+l3, p+l3, D, l2);

    /** Final stage: add up the coefficients. **/
    MPN_INCR_U (p + l3, 2 * n - l3, (mp_limb_t)carryB);
    MPN_INCR_U (p + l4, 2 * n - l4, (mp_limb_t)carryC);
    MPN_INCR_U (p + l5, 2 * n - l5, (mp_limb_t)carryD);
}
```