PARAMETER SELECTION REFINEMENT AND SOFTWARE IMPLEMENTATIONS OF SPECTRAL MODULAR EXPONENTIATION

by

Matthew Allen Estes
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

KGaj ____________ Dr. Krzysztof Gaj, Dissertation Director

____________ Dr. Jens-Peter Kaps, Committee Member

____________ Dr. Jill Nelson, Committee Member

____________ Dr. Andre Manitius, Department Chair

____________ Dr. Daniel Menascé, Senior Associate Dean

____________ Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: 7/23/2010

Summer Semester 2010
George Mason University
Fairfax, VA

Parameter Selection Refinement and Software Implementations of Spectral Modular Exponentiation

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Matthew Allen Estes
MS Computer Engineering
George Mason University, 2010

Director: Dr. Kris Gaj, Associate Professor
The Volgenau School of Information Technology and Engineering

Summer Semester 2010
George Mason University
Fairfax, VA

DEDICATION

This is dedicated to my wife and constant supporter, Kathleen. Her patience and support during these years in making up all the hours lost to my studies was critical to my success. To my parents Donald and Joan, who gave me the character and education that has enabled me to get this far. To my children Dalton, Ryan, and Kaitlyn and their frequent diversions from my work that always kept things in perspective.

ACKNOWLEDGEMENTS

I would like to thank Dr. Gaj for his consistent willingness to direct and advise me during my research. I would also like to thank Dr. Kaps, Dr. Nelson, and Dr. Fernandez for their valuable suggestions and support. Special thanks to Marcin Rogawski and all of the other GMU CERG team members that supported, advised, and listened to my talks and presentations over the years.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

PARAMETER SELECTION REFINEMENTAND SOFTWARE IMPLEMENTATIONS OF SPECTRAL MODULAR EXPONENTIATION

Matthew Allen Estes, MS Computer Engineering

George Mason University, 2010

Thesis/Dissertation Director: Dr. Kris Gaj

A consistent challenge to the widespread use public key cryptosystems, such as RSA, is the computational difficulty of performing arithmetic operations with large operands. There are many branches of mathematics and algorithms devoted to the exploration of different aspects of computer arithmetic on large integers. In this thesis, we outline several parameter selection techniques and software implementations that apply to a new technique of exponentiation, referred to as spectral modular exponentiation, which attempts to address computational efficiency of public key cryptosystems, such as RSA and Elliptic Curve Cryptosystems.

Spectral modular exponentiation (SME) is a method by which numbers are converted into spectral representations through a process known as Discrete Fourier Transform (DFT), at some initial cost in doing the transformations. The spectral domain

has the advantage of greatly reduced multiplication cost during the most costly portions of exponentiation. This thesis will describe the different algorithms that have been proposed independently by two different research groups, compare and contrast these algorithms, and describe various parameter selection techniques that apply to them. It will also cover lessons learned and some difficulties encountered in the development of a working implementation of spectral modular exponentiation. This thesis will also addresses some of the discovered concerns regarding particular approaches to spectral modular exponentiation in software implementations.

These difficulties involve the inherent limitations of the algorithm in software and the theoretical potential of performance in hardware. Variations on implementations were attempted to test different environments for the algorithm, but software implementations of spectral modular exponentiation were still characterized by performance less than that of existing algorithms, even at larger operand sizes. Included in this thesis are the actual calculated and verified results for several of these variations. These results include the initial generated parameters, internal interim values, and final results that would be necessary to verify the correctness of future algorithms and implementations.

These interim values serve as parameters and interim value references to future attempts for working implementations in both hardware and software. The hardware implementations of spectral modular exponentiation still show possibility for better comparable performance than traditional algorithms.

Also in this thesis are two proofs that demonstrate how to reliably generate parameters for a valid DFT and inverse DFT transformation. These are based on multiple previous works on characteristics of Mersenne and Fermat numbers and connecting those characteristics to the requirements for a valid DFT.

CHAPTER 1: Introduction

1.1. Cryptography

Cryptography involves the application of algorithms to transform a message into a representation of the message that is then referred to as the ciphertext. This algorithm must be able to then take that ciphertext and reverse the transformation to obtain the original message.

In the field of cryptography, symmetric and asymmetric cryptography constitute two of the major categories of algorithms. Symmetric cryptography is defined by encryption and decryption with a single identical key and is often much more efficient than the alternative method of asymmetric cryptography. Asymmetric cryptography has the characteristic of using two different keys in which one key is used for encryption and one key is used for decryption. This allows one or more parties to encrypt messages with a public key, and only the party that possesses the private key to decrypt the messages. Asymmetric cryptography enables digital signatures and public-key infrastructures, but is generally accepted to be much more computationally difficult. Although there are methods to greatly improve the efficiency of certain types of asymmetric algorithms, there is still a large focus to increase the computational efficiency of asymmetric cryptography.

One very commonly used asymmetric algorithm is called RSA. RSA involves the choice of two large prime numbers whose product forms the modulus for modular operations. Then two values are derived termed *e* and *d* which are multiplicative inverses of each other. These terms *e* and *d* are used to compute the encrypted and the decrypted message respectively. The primary operation to achieve these computations is modular exponentiation.

1.2. Exponentiation

Modular exponentiation, as stated, is a primary operation in RSA public-key cryptography. There are many different algorithms that are known to improve the efficiency of the modular exponentiation with varying degrees of complexity and each addressing different areas of modular exponentiation, but the basic mathematical operation is:

$$c = m^e \ mod \ n$$

To properly compare algorithms, modular exponentiation must be broken down into sub-components. This thesis will evaluate exponentiation by dividing exponentiation into three sub-component operations.

The first component is the algorithm of the exponentiation itself. This includes how multiplications, squarings, table look-ups and possibly other operations will be combined to properly achieve exponentiation. The most basic method of exponentiation is to multiply *m* by itself *e* times. For large values of *m* and *e* this method is much too slow to be used in practical applications.

Some popular algorithms that improve upon the efficiency of the naïve method are Left-to-Right binary exponentiation, Right-to-Left binary exponentiation, K-nary Exponentiation, and Sliding-Window Exponentiation. All of these algorithms base their improvements on the binary representations of values and the manipulation of bits or groups of bits in order to improve efficiency. Multiplication and squaring are major operations in all of these algorithms.

The second component is multiplication. The multiplication of two numbers, including the squaring of a single number, is typically an expensive operation. Thus, the type of multiplication algorithm used is highly influential on the overall efficiency of exponentiation. Some multiplication algorithms used in exponentiation are Karatsuba, Toom-3, and FFT. Montgomery and Spectral are not traditional multiplication algorithms in that they are multiplication algorithms that include reduction and operate on terms in a different domain. [8].

For multiplications not including reduction, there is the required additional component of reduction. The reduction of varying measures, such as bit sizes of intermediate values or the degree of certain polynomial representations, is necessary to maintain all interim values at a size that can be efficiently operated on within a fixed architecture. The most basic method is simple modular reduction through division.

Sometimes, the use of specific parameters can also allow efficient short-cuts to calculating modular reductions.

1.3. Existing Exponentiation Algorithms

Left-to-Right Exponentiation Algorithm

This algorithm is also called the "square and multiply" algorithm and was originally conceived in 200BC [1]. This "Left-to-Right" algorithm initializes the output value *c* to 1. It then scans the bits of *e* from highest to lowest or left to right. If the bit is one, then the algorithm calculates:

$$c = c * m \bmod n$$

Then, as it increments to the next bit it calculates the effect of shifting the exponent by one bit position, which has the effect of squaring the temporary result:

$$c = c * c \bmod n$$

The entire algorithm is:

INPUT: m, e (where e_i is the i^{th} bit of e, and t is the size of e in bits)
OUTPUT: $c = m^e$

1. $c \leftarrow 1$
2. For i from t-1 down to 0, do the following:
   2.1 $c \leftarrow c^2$
   2.2 if e_i=1 then $c \leftarrow c \cdot m$
3. Return c

Figure 1: Left-to-Right Exponentiation Algorithm

The “Right-to-Left” algorithm uses the same principal as the Left-to-Right algorithm, but runs in reverse.

The entire algorithm is:

```
INPUT:  m, e
OUTPUT: c = m^e
    4.  c ← 1, S ← m
    5.  While e ≠ 0, do the following:
        5.1 if e is odd then  c ← c·S
        5.2 e ← e/2
        5.3 S ← S · S
    6.  Return c
```

Figure 2: Right-to-Left Exponentiation Algorithm

K-ary Window Algorithm

The K-ary Window algorithm is an adaptation of the “Right to Left” algorithm except that it improves upon this algorithm by evaluating bits of the exponent in k-bit “windows” instead of in single bits [1]. The algorithm is defined as follows:

INPUT: m, e where e_i is the i^{th} digit of k bits, and t is the size of e in digits
OUTPUT: $c = m^e$

1. Precomputation
   1.1 $g_0 \leftarrow 1$
   1.2 For i from 0 to (2^k-1), do:
       $g_i \leftarrow g_{i-1} \cdot g$ (thus $g_i = g^i$)
2. $c \leftarrow 1$
3. For i from t-1 down to 0, do the following:
   $c \leftarrow (c^2)^k$
   $c \leftarrow c \cdot g_{ei}$
4. Return (c)

Figure 3: K-ary Window Algorithm

Sliding Window Algorithm

This algorithm is an adaptation of the K-ary algorithm except that it improves upon the algorithm by evaluating bits of the exponent *e* using dynamic optimized "windows" instead of k-bit static windows [1]. The algorithm has several derivations, but the sliding window algorithm is as follows:

INPUT: m, e where e_i is the i^{th} bit, and t is the size of e in bits
OUTPUT: $c = m^e$

1. Precomputation (odd g's only)
   $g_1 \leftarrow g$, $g_2 \leftarrow g^2$
   For i from 1 to $(2^{k-1}-1)$, do:
   $g_{2i+1} \leftarrow g_{2i-1} * g^2$
2. $c \leftarrow 1$, $i \leftarrow t-1$
3. while $i \geq 0$, do the following:
3.1 If $e_i = 0$ then do $c \leftarrow c^2$, $i \leftarrow i - 1$
   Otherwise, find longest bit string
   $e_i e_{i-1} \ldots e_{s+1} e_s$, such that:
   $i-s+1 \leq k$ and $e_s=1$ and do the following:
   $c \leftarrow (c^2)^{i-s+1} \cdot g_{(ei\ ei-1 \ldots es)2 \ldots,}$ $i \leftarrow s - 1$
4. Return c

Figure 4: Sliding Window Algorithm

1.4. Existing Modular Multiplication Algorithms

Multiplication and reduction are sometimes independent steps, such as when reduction is used with Karatsuba multiplication. However, multiplication and reduction are combined in Spectral Modular Multiplication as well as Montgomery Multiplication. This section will cover some combinations of reduction and multiplication, including those that are tested in the thesis.

All exponentiation algorithms discussed so far in this thesis can be used for both infinite and finite field operations, and thus have not yet included the modular reduction steps specific to finite fields. Reduction can be added to the implementation of the multiplication and squaring operations to modify exponentiation operations for use within finite fields. Because some modular multiplication operations embed reduction and cannot be discussed apart from each other, the following sections will discuss the various combinations of multiplication and reduction algorithms used during testing.

There are several alternative algorithms to reduction that can be used with classical multiplication. Within the scope of this thesis, only one type of reduction algorithm was tested.

Karatsuba Multiplication and Reduction by Division

Karatsuba is a popular algorithm for efficient multiplication [18]. It involves the recursive splitting of input numbers into smaller numbers. This split allows one large multiplication to be accomplished by 3 smaller multiplications and a few additions as shown below in Figure 5.

The split is based on the boundary B^m where B is the base for a single digit and m is the number of digits in lower half of the split. In binary 32-bit computing environments, B is sometimes chosen to be 2^{31} so as to allow additions of two 2^{31} sized numbers to take place without requiring a carry bit. A third value n represents the number of digits in the input values such that each digit is less than B.

Because Karatsuba is an algorithm that splits values into smaller values, it can be run recursively until n is small enough that the multiplications can be computed directly. The most efficient m is usually $n/2$ so that each iteration splits the values in half. A recursive version of the Karatsuba algorithm is shown below [18].

INPUT: x, y
OUTPUT: x·y

KARATSUBA[x,y] is:

1. if $n < 2$ then Return x·y
2. Split x and y into x_1, x_0 and y_1, y_0 using B^m such that:
   $x = x_1 B^m + x_0$
   $y = y_1 B^m + y_0$
   (where x_0 and y_0 are less than B^m)
3. z_2 = KARATSUBA[x_1, y_1]
   z_0 = KARATSUBA[x_0, y_0]
   $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$
   $xy = z_2 B^{2m} + z_1 B^m + z_0$
4. Return xy

Figure 5: Karatsuba Multiplication Step

Since Karatsuba does not include reduction, the reduction algorithm that will be combined with Karatsuba Multiplication is that of simple arithmetic division to determine the modular reduction.

Montgomery Multiplication

Montgomery multiplication is a technique that combines multiplication and reduction into a single operation. It achieves this by converting values into images within the Montgomery domain, computing the product of those images, and then converting back from the Montgomery domain. An image in the Montgomery domain is defined as x' = x·R mod m. The basic required operation in Montgomery Multiplication is that of Montgomery Product.

Montgomery Product, which, given n, x, y in number system b each with k digits and $R = b^k$ where $0 \le x; y \le n$ and gcd(n, b) = 1 returns xyR^{-1} mod n.

$$\text{MontgomeryProduct}(x,y) = xyR^{-1} \bmod n$$

Conversion of *x* to the Montgomery image x':

$$x' = \text{MontgomeryProduct}(x, R^2 \bmod n)$$

$$\text{because } x \cdot R^2 \cdot R^{-1} \bmod n = x \cdot R \bmod n = x'$$

Conversion of the Montgomery image x' to *x*:

$$x = \text{MontgomeryProduct}(x', 1)$$

$$\text{because } x' \cdot 1 \cdot R^{-1} \bmod n = x \cdot R \cdot 1 \cdot R^{-1} \bmod n = x$$

INPUT: m, e, n (where e_i is the ith bit of e), and t is the size of e in bits
OUTPUT: $c = m^e \bmod n$

1. $c' \leftarrow$ MontgomeryProduct(1, R^2 mod n)
2. $m' \leftarrow$ MontgomeryProduct(m, R^2 mod n)
3. While i from t-1 down to 0, do the following:
   3.1 $c' \leftarrow$ MontgomeryProduct(c',c')
   3.2 if e_i=1 then $c' \leftarrow$ MontgomeryProduct(c', m')
4. $c \leftarrow$ MontgomeryProduct(c', 1)
5. Return c

Figure 6: Left-to-right exponentiation algorithm with multiplications replaced by Montgomery Product

Montgomery multiplication gains efficiency from the ability to choose an integer ring in which the residual math will take place. If chosen properly, reductions can be achieved with shifts and additions, greatly improving the overall efficiency of exponentiation especially for larger operands.

Using Montgomery Multiplication for smaller operands of *m* and *e* generally suffer as compared to other algorithms. This is because the time necessary to convert between numbers and residual representations at the beginning and end of exponentiation outweighs the efficiencies gained during the square-and-multiply operations.

1.5. Spectral Math Overview

Spectral math offers a different way of representing values, much like Montgomery Multiplication. Spectral techniques involve the conversion of time-domain representations of numbers into the spectral domain. This is frequently used in the field of signal processing where time-domain sequences are samples from a sensor and are transformed into spectral representations that represent the spectral, or frequency, content of the time domain sequence. Sequences of time values have different mathematical properties once transformed to the spectral domain. These properties allow certain operations to be performed differently than they would have been accomplished in the time domain representation.

Because spectral techniques are often used in signal processing for very different applications, the algorithms and terminology will vary widely than those in this thesis [5]. For example, in some signal processing there is an allowance for small deviations in values and the final values are only approximations, whereas in most implementations of finite field encryption techniques, there is no allowance for any such variation in results.

While in signal processing a spectral transform is often applied to a “sequence” or array of “sample” values, in the SME it is often referred to as an “evaluation polynomial” and the different terms are treated as the coefficients of a polynomial representation of the time or spectral values. This different representation also serves to visibly differentiate signal processing techniques from techniques used in discrete math. In signal processing, the transform to the spectral domain is called the Discrete Fourier Transform (DFT), while when a DFT is applied within a finite field for an evaluation polynomial, it is called a Number Theoretical Transform (NTT), although the term DFT is still used. [1]

1.6. Spectral Modular Multiplication Overview

In the spectral domain, complex multiplication operations become d-element component-wise multiplications. The parameters used for NTTs can be chosen in such a way as to choose a spectral domain that allows for efficient modular reductions and efficient conversion to and from the spectral domain. The proposals made by Baktir, Saldamli, and Koç also show how multiplications and reductions can be made in the spectral domain with selection of specific parameters in order to ensure the spectral multiplications can take place successively without the need to convert intermediate results back into the time-domain representation for reductions, all the while avoiding potential overflows [3], [5].

This allows the application of spectral math to achieve faster spectral modular multiplications while not suffering the penalty of DFT/IDFT conversions between multiplication operations. This has direct application to modular exponentiation and the works by Baktir, Saldamli, and Koç emphasize this benefit [3], [5].

Spectral Modular Multiplication does not necessarily address the actual method of exponentiation, such as Sliding Window or Left to Right. It only addresses the initialization, the conversion to and from the spectral domain, multiplication, and reduction operations. In many ways, it operates similar to Montgomery Multiplication in that each multiplication, when given n, x, y in number system b each with k digits and $R = b^k$ where $0 \le x; y \le n$ and gcd(n, b) = 1, returns xyR^{-1} mod n.

$$\text{SpectralModularProduct}(x,y) = xyR^{-1} \bmod n$$

Evaluation Polynomials

The first step in using spectral arithmetic is to evaluate a single large value provided as input, which will be referred to as *m,* into a series of values suitable for use in a NTT, which will be referred to as *m(t)*. One method to divide the number is to split it on fixed bit boundaries such that a certain number of bits per word, *u*, and a certain number of terms *s,* will together form a series of value representing *u*s* total bits. This method is referred to as an evaluation polynomial and takes the form of:

Theorem 1: Evaluation Polynomial

$$m(t) = m_0 + m_1 b + m_2 b^2 + m_3 b^3 + m_{(s-1)} b^{(s-1)}$$

$$\text{where } b = 2^u$$

m = 645484

1	0	0	1	1	1	0	1	1	0	0	1	0	1	1	0	1	1	0	0

Evaluation with b=16 (u=4)

⇩

1 0 0 1	1 1 0 1	1 0 0 1	0 1 1 0	1 1 0 0

$$m(t) = 9b^4 + 13b^3 + 9b^2 + 6b + 12$$

$$m = 9(16^4) + 13(16^3) + 9(16^2) + 6(16) + 12 = 645484$$

Figure 7: Evaluations Polynomial Example

Be aware that the evaluation polynomial shown is known as the **base evaluation polynomial** since each value m_i is bounded by:

$$0 \leq m_i < b$$

It is possible to generate an evaluation polynomial with different terms that represent the same value by not using the base evaluation polynomial, but simply an evaluation polynomial. The following evaluation polynomial represents the same value as the previous example in Figure 7.

$$m(t) = 9b^4 + 13b^3 + 9b^2 + 5b + 28$$

Notice that a value was "borrowed" from one term to add to another. This concept of borrowing and likewise carrying is important to the reduction of terms in the Spectral Modular reduction technique.

Number Theoretical Transform

The conversion from time domain values to frequency domain values is accomplished by the Number Theoretical Transform, which is a special implementation of the Discrete Fourier Transform:

Definition 1: Number Theoretical Transform

$$A_j = \sum_{i=0}^{d-1} a_i w^{ij} \; mod \; q \;\;, 0 \le j \le d-1$$

Definition 2: Inverse Number Theoretical Transform

$$a_j = d^{-1} \sum_{i=0}^{d-1} A_i w^{-ij} \; mod \; q \;\;, 0 \le j \le d-1$$

Definition 1 defines a matrix that achieves the DFT by matrix multiplication with the time-series polynomial. An example matrix is shown for a DFT of size 5 in Table 1: DFT Transform Matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 \\ 1 & w^2 & w^4 & w^6 & w^8 \\ 1 & w^3 & w^6 & w^9 & w^{12} \\ 1 & w^4 & w^8 & w^{12} & w^{16} \end{bmatrix}$$

Table 1: DFT Transform Matrix

The value w is called the **generator** or **principal d^{th} root of unity** of the DFT while d is the size or length of the DFT. Thus, w and d are related in that $w^d = 1 \mod q$. For a certain integer ring of Z_q, w, d, and q are constrained when attempting to create a valid and invertible NTT that allows Definition 2 to exist. The requirements for existence of the invertible matrix are discussed in Chapter 3.

Spectral Modular Multiplication

The last major operation within the application of Spectral Modular Exponentiation is the Spectral Modular Multiplication. In the spectral domain, the product of two spectral numbers is accomplished by a component-wise multiplication of each term in the spectral evaluation polynomial, sometimes called the Spectral Modular Product. Once the product is calculated, however, the algorithm must ensure that the time series representation is reduced to remain properly bounded by a well constructed reduction algorithm. This algorithm must be computationally efficient and avoid overflows in terms of the time series representation while doing calculations in the spectral domain.

To perform this algorithm, several parameters are required and can be precomputed. These values include N(t), Γ(t), d^{-1}, and λ(t). The calculation of these parameters is explained during the iterim value calculations later in this thesis.

CHAPTER 2: ANALYSIS OF EXISTING WORKS

2.1. SME Algorithm Descriptions

Spectral math as a performance enhancement to cryptography is covered in several works. The application of NTT to multiplication dates back to 1971 with Schönhage and Strassen [17]. Kalach further discusses multiplication efficiency in hardware, but again does not cover exponentiation [2]. Baktir [3] discusses the application of spectral math to modular multiplication as well as Elliptic Curve Cryptography. However, Baktir covers the exploration of modular multiplication as a subset of the larger effort towards spectral applications to ECC operations, and not exponentiation. Koç and Saldamli [5] explore specifically SME as it benefits exponentiation and offer several resources in the understanding of this algorithm.

Schönhage-Strassen and Kalach

The Schönhage-Strassen multiplication algorithm is an early example of a spectral multiplication algorithm. It is asymptotic in complexity and has been shown to outperform the traditional multiplication algorithm of Karatsuba for numbers approximately larger than 2^{15} bits [17]. This algorithm is based on spectral techniques.

Kalach [2], in his exploration of spectral math, addresses improvements to the efficiency of the DFT operations. This includes the application of Fast Fourier Transforms (FFT). Both recursive and iterative algorithms for FFT are outlined to apply to spectral modular multiplications.

Baktir Spectral Multiplications

Baktir was the first Spectral Modular Exponentiation algorithm to be evaluated in the preparation for this thesis. Baktir discusses the use of the properties of Mersenne Number Theoretical transforms (MNT), Fermat Number Theoretical Transforms (FNT), pseudo-Mersenne Transforms (PMT), and pseudo-Fermat Transforms (PFT).

Baktir describes the use of the Pseudo Fermat Transform with $q= 2^{2\char`^n+1}/p$ (where *p* is a prime factor) to enable efficient Fast Fourier Transform methods as opposed to the general Discrete Fourier Transform.

In a PMT, arithmetic is achieved modulo $q=(2^n -1)/t$, an integer sub-multiple of a Mersenne Number. However, the intermediate reductions are computed modulo the original Mersenne number, $2^n -1$, and only the final result needs to be reduced modulo M_n/t. The use of PMT increases the number of available transform lengths since each integer sub-multiple has a different length. But, the downside is increased word size for intermediary transform operations (n vs. $n-\log_2 t$)

Baktir also outlines efficient parameter selection for ECC. Most of this constitutes the selection of operand sizes relevant to ECC parameters. Baktir describes that a fully recursive FFT can only be used for highly composite numbers (2^n or other powers of small primes). The allowable sequence length is either a prime number d for w=2 or 2 times a prime number 2d for w=-2.

Overall, this work focused mainly on field operations used in ECC, such as multiplications, additions, and inversions. Spectral Modular Exponentiation was not discussed by Baktir.

Koç and Saldamli SME

In much the same way as Baktir, Koç and Saldamli also cover DFT Improvements through the use of Mersenne and Fermat Theoretical Transforms. Although some papers were only written by either Koç or Saldamli, their names will be used interchangeably since they were both involved in the development of the algorithm. Parameter calculations were described using MNT with positive (w=2,4) and negative (w=-2) principal roots of unity. These papers review MNT, FNT, PMT, and PFT variations on parameters. In these works, much more time was spent addressing the parameters to achieve operations safely in the spectral domain without creating overflows of values in the time domain that would alter the represented value.

Koç also covered SME improvements through the application of a prior work in evaluating multiplication/reduction algorithms in which the CIOS Algorithm – Coarsely Integrated Operand Sum – was selected.

Chinese Remainder Theorem was outlined as a method for achieving efficient operations in larger modulus size by using CRT to remain in a smaller ring modulus. This algorithm from the very beginning was specified as one that would be efficient in hardware. It was not specifically limited to hardware, but significantly parallel operations were the core mechanism for efficiency. Parallel hardware architectures were outlined.

2.2. Building Blocks of Existing SME Algorithms

While there are potentially numerous methods by which spectral modular operations could be adopted to achieve modular exponentiation, Baktir and Koç both suggest very similar algorithms that closely resemble Montgomery multiplication. These methods calculate multiplications of residual values with an embedded reduction following the form $xyR^{-1} \bmod n$, where R is a power of two.

The algorithms require the following operations:

1. Addition and subtraction - in one example, addition or subtraction of multiples of the modulus in order to zero out the least significant term
2. Right-shift of terms - for reductions
3. Left-shifts of terms - to calculate residuals
4. Product of two numbers
5. Obtaining the first time-series value – used by reduction operations

Also necessary for the success of these operations are several constants:

1. $\Gamma(t)$ – a number that, when multiplied by another number, causes that number to shift *u* bits to the left.
2. One – a number that is the multiplicative identity.
3. <u>N</u>(t) - a number that is the spectral representation of $k \cdot n$ (i.e. a multiple of the modulus) and also has the feature of having it's least significant term set to 1. This facilitates an algorithm to manipulate the least significant term through addition without affecting the overall value, such as just before a right-shift operation during reductions.
4. $\lambda(t)$ – a pre-computed value that is used by multiplication to compute the residual of a value in time series representation.

CHAPTER 3: Parameter Selection

3.1. SME Parameters

All spectral exponentiation techniques have a unique set of parameters that must be established prior to beginning operations.

d	"size" of the DFT, i.e. number of terms of the evaluation polynomial
s	maximum number of input words (approx. d/2, see below)
q	Number Theoretic Transforms such as MNT will take place within the integer ring Z_q
u	number of bits in each DFT term
b	2^u
w	principal d^{th} root of unity, on which DFT transform is based
p,n	In MNT, q is of the form 2^p-1 and in FNT q is of the form 2^{2^n}+1. These exponents are parameters.
bits	supported bit size for operands, u·s

Table 2: List of Spectral Parameters

The parameters are interdependent, but bit size is one of the final parameters to be determined and depends on multiple other parameters. Therefore, an efficient approach to parameter selection involves pre-calculating a table of parameters and selecting the most efficient set of parameters that meet certain criteria.

3.2. Parameter Variations

Mersenne Number Transform (MNT) Parameters

One very simple manner of calculations of parameters involves using the characteristics of Mersenne Number Transform (MNT) [15]. Mersenne numbers are defined as numbers of the form $q = 2^p - 1$, where p is oftentimes required by a definition to be a prime. MNT's support DFT's that use values for w of both 2 and -2 and the corresponding d parameters are trivially determined as p and $2p$ respectively.

Theorem 2: The length of the DFT matrix d is p for $w=2$, and also d is $2p$ for $w=-2$ for DFTs over the field Z_q where q is a Mersenne Number of the form 2^p-1.

PROOF:

For w = 2:

1. Assume d=p.
2. $1 = w^d \bmod q$ by definition of principal roots of unity (Chapter 1)
3. $1 = 2^p \bmod q$ by substitution
4. $1 = 2^p \bmod 2^p - 1$ which is true

For w = -2:

1. Assume d=2p.
2. $1 = w^d \bmod q$ by definition of principal roots of unity (Chapter 1)

3. $1 = (-2)^{2p} \bmod q$ by substitution

4. $1 = 2^{2p} \bmod 2^p - 1$

5. $1 = 2^{2p} - 1 + 1 \bmod 2^p - 1$

6. $1 = (2^p - 1)(2^p + 1) + 1 \bmod 2^p - 1$

7. $1 = (0)\cdot(2^p + 1) + 1 \bmod 2^p - 1$ which is true, thus the assumption is true

Fermat Number Transform (FNT) Parameters

The Fermat Number Transform (FNT) uses Fermat numbers of the form: $q = 2^{2^n} + 1$ [14]. It is also possible to use Fermat numbers of the form $2^n + 1$, but this could potentially end up with complex roots and additionally would lose the performance benefits from having word aligned calculations. Additionally, this would necessarily need to perform additional checks on the validity of the DFT transform. Math operations are carried out with optimized Fermat arithmetic operations.

Pseudo Number Transform Parameters

In the ring Z_q, for some prime factors p that divide q, the field $Z_{q/p}$ can be useful or necessary. q itself does not have to be a prime for a valid transform and thus may have multiple small factors. However, for certain q, these small factors cause the resulting transform size for the DFT to be too short for the necessary length required by MNTs or

FNTs. Refer to Theorem 7.7 in [5] for further explanation of how small factors cause small transform sizes.

This length can be increased by dividing out certain small prime factors *p*. These resulting fields are called Pseudo Mersenne Transforms (PMT) or Pseudo Fermat Transforms (PFT). Another reason for pseudo transforms is to create new combinations of parameters that meet certain bit lengths or DFT lengths *d*.

3.3. Parameters for a Valid DFT

For a single generator in Z_q, the generator must have a multiplicative inverse in *q*. However, in NTTs every element generated in the DFT matrix generated by the Theorem 1 must itself have a multiplicative inverse in Z_q by Blahut [10]. This is because concerning the matix of the DFT transform and inverse-DFT transform, there only exists a valid inverse if and only if the determinant is non-zero [12]. Additionally, by Massey [10], the determinant must also be a unit in the field Z_q. Massey defines "unit" as an element in Z_q having a multiplicative inverse.

One example of an invalid DFT matrix is the matrix defined for a DFT of size d=4, in the ring Z_q where $q=2^4-1$. This matrix is shown in Figure 8.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 4 & 1 & 4 \\ 1 & 8 & 4 & 2 \end{bmatrix}$$

Figure 8: Invalid DFT Matrix for d=4, $q=2^4-1$

The determinate of this matrix is 3. Because 3^{-1} mod 15 does not exist, the determinant has no multiplicative inverse in Z_{15}. Therefore, this matrix is not suitable for NTTs.

Another requirement for a valid matrix that is derived from the above determinant requirement is that the size of the matrix d must evenly divide p_i-1 for any prime factor of the Mersenne Number. This will be referred to as the division test. The tests required for a valid and invertible transform are outlined in the table below.

1. Basic Invertibility – The determinant of the DFT matrix must be non-zero.
2. Invertible in Ring – The determinant of the DFT matrix must be a unit in the ring, i.e. an element having a multiplicative inverse.
3. All Elements Invertible - Every element of the DFT matrix must be a unit. This results from the definition of the IDFT matrix that contains elements of the form w^{-ij}, which are the multiplicative inverses of the DFT elements.
4. Divisibility Test - The length of the DFT matrix d must divide p_i-1 for each prime factor p_i of q, given the field Z_q used for spectal domain operations.

Table 3: Tests for Invertible DFT Matrix (NTT)

The final test is an additional test derived from the tests 1-3 for NTTs that are attempted where the field characteristic q is composite. If test 4 passes, then it can be assumed that tests 1-3 also pass.

3.4. Reliable Mersenne Parameter Production

For Mersenne Numbers of the form $q=2^p - 1$ in which p is prime, for which $w=2$, and for which the size of the matrix d is either p or $2p$, it can be shown that all invertibility tests for a valid matrix pass, including the division test.

For the first three tests, it can be shown that all tests pass. This is due to the requirement that elements of the DFT are produced by powers of 2, thus 2 is the only prime factor. Since q is always odd, q and 2 are always relatively prime, thus 2^{-ij} mod q = $((2^{-1}) \bmod q)^{ij}$ mod q is always well defined. For the division test, the proof is slightly more involved:

Theorem 3: The length of the DFT matrix d divides $r-1$ for each prime factor r of q for DFTs over the field Z_q where q is a Mersenne Number of the form 2^p-1 where p is prime and where $w=2$ or $w=-2$.

PROOF: If p is an odd prime, then any prime r that divides $q = 2^p-1$ (a Mersenne Number), must be of the form: $r = k\,2p + 1$. Or, otherwise stated, both p and $2p$ divides $r-1$ for any factor r. This holds even when $q = 2^{p-1}$ is prime. [15]

From **Theorem 2**, if $w = 2$, then d = p or if w = -2 then d = 2p.

If any prime factor r divides $2^p - 1$ then $2^p \equiv 1 \pmod{r}$. By Fermat's Little Theorem, $2^{(r-1)} \equiv 1 \pmod{r}$.

1. It is easier to attack the contra-positive, so assume p and $r - 1$ are relatively prime and once again apply Fermat's Little Theorem to derive $(r - 1)^{(p-1)} \equiv 1 \pmod{p}$.

2. If we factor out one term of (r-1), we can show that there is a number $x \equiv (r-1)^{(p-2)}$ for which $(r-1)\cdot x \equiv 1 \pmod{p}$
3. Removing the modulus, there is a number k for which $(r-1)\cdot x - 1 = k\,p$. And rearranging terms: $(r-1)x - kp = 1$
4. From step 1, $2^{(r-1)} \equiv 1 \pmod{r}$, and raising both sides of the congruence to the power x gives: $2^{(r-1)x} \equiv 1$ mod r, and since $2^p \equiv 1 \pmod{r}$, raising both sides of the congruence to the power k gives $2^{kp} \equiv 1$.
5. Since both congruencies equal 1, dividing one by the other will also be congruent to 1, thus $2^{(r-1)x} / 2^{kp} = 2^{(r-1)x-kp} \equiv 1 \pmod{r}$. Substituting the earlier equality: $(r-1)x - kp = 1$, obtains that $2^1 \equiv 1 \pmod{r}$; which is false. If this false statement is pursued further, $2\text{-}1 \equiv 1 \equiv k\,r$, thus that r divides 1, which is also false.
6. From this, it is apparent that the initial assumption that p and $r-1$ are relatively prime is untenable. Therefore, p and $r-1$ share a common factor, but since p is prime $r-1$ must be a multiple of p.
7. Therefore, since p and $2p$ divide *r-1* for any factor r of the Mersenne Number, if the length of the Mersenne Transform is either p or *2p*, then the length of the transform divides *r-1*. This length is used when MNT are used when either w=2 or w=-2, respectively. Additionally, because w=2 or w=-2, then w always has an inverse in modulo q since q is always odd.

3.5. Reliable Fermat Parameter Production

For Fermat Numbers only of the form: $q = 2^{2^n} + 1$ for which w is 2, and for which the size of the matrix is $d=2^{n+1}$, it can be shown that all invertibility tests for a valid matrix pass, including the division test. Thus, the DFT always has a valid inverse DFT matrix.

It can be shown that the first three tests pass, like in case of MNT. Since q is always odd, q and 2 are always relatively prime, thus $2^{-ij} \bmod q = ((2^{-1}) \bmod q)^{ij} \bmod q$ is always well defined. However, the division test requires more examination.

Theorem 4: The length of the DFT matrix *d* divides *r-1* for each prime factor *r* of *q* for DFTs over the field Z_q where q is a Fermat Number of the form $q = 2^{2^n} + 1$.

PROOF: If a DFT is constructed over the Fermat Number $q = 2^{2^n} + 1$, then the size of the DFT matrix, which is *d*, must divide 1 less than each prime factor *r* that divides the Fermat Number $q = 2^{2^n} + 1$. This holds even when *q* is prime.

1. If any factor *r* divides $q = 2^{2^n} + 1$ then $2^{2^n} = -1 \ (mod\ r)$ and thus:

   $$2^{2^{n+1}} = 1 \ (mod\ r)$$

   It can be seen that the order or DFT length *d* of this with a w = 2 is 2^{n+1}.

2. From Édouard Lucas improving upon Euler, any prime divisor *r* of $F_n = q = 2^{2^n} + 1$ is of the form $k2^{n+2} + 1$ whenever n is greater than one.

3. Since *d* must divide r-1 for each prime factor r, and substituting:

   $d = 2^{n+1}$ and

$r = k2^{n+2} + 1$

It is shown that:

$d \mid r-1$

$2^{n+1} \mid k2^{n+2}$

Which is true.

4. Therefore, since *d* divides *r-1* for any factor *r* of the Fermat Number, then the division test passes. This length is used when FNT is used with w=2.

3.6. Parameters for Overflow Boundaries

An overflow is when any element of the sequence exceeds the boundaries of the field. This applies to both elements in the spectral domain and to elements in the time domain. When overflows occur during calculations in the spectral domain, it alters the time domain representation of the value. To prevent overflows from occurring during spectral modular exponentiation, the numbers of terms in the DFT (or degree) must remain bounded within the size of the DFT *d*. The number of terms in both of the input values is defined as *s*. Since a sequence of size s has terms of degree 0..s-1, the maximum degree of the resulting sequence from multiplication is (s-1)+(s-1). Therefore, the degree (*d-1*) of the maximum supported size of the DFT is must be large enough to support to resulting sequence as follows:

$$d - 1 \geq s-1 + s-1 = 2s - 2$$

$d \geq 2s - 1$

$\frac{d+1}{2} \geq s$

Given an odd d , such as 7, it is seen that s is 4. With d=8, s is also 4. For integers, s can be evaluated as:

$$s = \left\lceil \frac{d}{2} \right\rceil$$

Also, not only must the number of terms be within bounds, but each coefficient in the polynomial representation must be within the field as well, unless the algorithm includes a check to try and detect overflows. In the Saldamli algorithm [6], the largest possible coefficient value is $2b^2s$. This is based on the multiplication of 2 coefficients of max b size and s number of coefficients added together. The constant 2 term comes from the possible large remainder value of alpha that could potentially double the final value.

This limit is highly dependent on the spectral modular exponentiation algorithm used and additional parameters. Each algorithm implements different numbers and types of operations and the evaluation of these operations determines the boundaries. It is not a theoretical restriction on NTT or DFT, but of the particular implementation used by Saldamli that does not attempt to detect overflows during exponentiation for performance reasons.

The Koç algorithm includes a formula for boundary testing. It was based on the boundaries necessary for multiplications to follow multiplications indefinitely. This algorithm was defined by Theorem 7.6 in [5]. However, independent calculations have shown the actual value for B(s) to be:

$$B = -\frac{2s^3}{3} + \frac{2rs^2}{3} + \frac{s^2}{3} + \frac{2rs}{3} + s + \frac{r}{9} + \frac{2}{9}$$

Further, the entire inequality can be shown below after substituting r and B into the final equation specified in Theorem 7.6 in [5].

$(b^2 + b)^2 B(s) + b^2 s < q$

It is possible to compute the resulting equation from these substitutions, but it is much simpler to calculate the value for r first after substitution, then substitute into B, and finally substitute B into the final inequality. Solving for any particular value in this inequality is computationally infeasible. The simplest method to solve this inequality is to iteratively test values for feasibility and determine the *b* that satisfies the inequality given *s* and *q*.

3.7. Mersenne Parameters

Mersenne Number Theoretical (MNT) transforms are transforms into domains of Z_q where q is a Mersenne Number and has the form:

$$q = 2^p - 1$$

Mersenne Numbers are not necessarily prime and their definition does not necessarily assume that p is prime. However, in this thesis we will assume that p is always prime. Mersenne numbers have several properties that make them suitable for modular operations. First, observe that for any Mersenne number $q=2^{p-1}$:

$$2^p mod\ q = 1$$

This provides a principal root of unity of *w=2* such that this root produces a sequence of degree *p*. Since Koç tells us that spectral transforms require a "primitive root of unity", we can meet this requirement by using an MNT with a primitive root of unity, *w=2*, and a size *d=p*.

From this first parameter of spectral math, d, the degree of the transform with a base *w* of 2, we can also derive *s* from the earlier discussion about the relationship of *s* to *d*.

$$s = \lceil d/2 \rceil$$

Remember the bits is simply *u*s* and the value "nttwords" is the number of words required to store a single term of the DFT in a 32-bit architecture. The space required to store a single term is dependent on the size of *q*, since each term undergoes spectral operations modulo *q*. In MNT, *q* is 2^p-1, and therefore can be stored in *p* bits resulting in *p/32* words.

p	d	w	u	w	bits	nttwords
17	17	2	2	9	18	1
19	19	2	2	10	20	1
23	23	2	3	12	36	1
29	29	2	4	15	60	1
31	31	2	5	16	80	1
37	37	2	6	19	114	2
41	41	2	7	21	147	2

43	43	2	7	22	154	2
47	47	2	8	24	192	2
53	53	2	10	27	270	2
59	59	2	11	30	330	2
61	61	2	11	31	341	2
67	67	2	13	34	442	3
71	71	2	14	36	504	3
73	73	2	14	37	518	3
79	79	2	16	40	640	3
83	83	2	17	42	714	3
89	89	2	18	45	810	3
97	97	2	20	49	980	4
101	101	2	21	51	1071	4
103	103	2	21	52	1092	4
107	107	2	22	54	1188	4
109	109	2	23	55	1265	4
113	113	2	24	57	1368	4
127	127	2	27	64	1728	4
131	131	2	28	66	1848	5
137	137	2	30	69	2070	5
139	139	2	30	70	2100	5
149	149	2	33	75	2475	5
151	151	2	33	76	2508	5
157	157	2	34	79	2686	5
163	163	2	36	82	2952	6
167	167	2	37	84	3108	6
173	173	2	38	87	3306	6
179	179	2	40	90	3600	6
181	181	2	40	91	3640	6
191	191	2	43	96	4128	6
193	193	2	43	97	4171	7
197	197	2	44	99	4356	7
199	199	2	45	100	4500	7
211	211	2	48	106	5088	7
223	223	2	51	112	5712	7
227	227	2	52	114	5928	8
229	229	2	52	115	5980	8

233	233	2	53	117	6201	8
239	239	2	55	120	6600	8
241	241	2	55	121	6655	8
251	251	2	57	126	7182	8
257	257	2	59	129	7611	9
263	263	2	60	132	7920	9
269	269	2	62	135	8370	9
271	271	2	62	136	8432	9
277	277	2	64	139	8896	9
281	281	2	65	141	9165	9
283	283	2	65	142	9230	9
293	293	2	68	147	9996	10
307	307	2	71	154	10934	10
311	311	2	72	156	11232	10
313	313	2	73	157	11461	10
317	317	2	74	159	11766	10
331	331	2	77	166	12782	11
337	337	2	79	169	13351	11
347	347	2	81	174	14094	11
349	349	2	82	175	14350	11
353	353	2	83	177	14691	12
359	359	2	84	180	15120	12
367	367	2	86	184	15824	12
373	373	2	88	187	16456	12
379	379	2	89	190	16910	12
383	383	2	90	192	17280	12
389	389	2	92	195	17940	13
397	397	2	93	199	18507	13
401	401	2	94	201	18894	13
409	409	2	96	205	19680	13
419	419	2	99	210	20790	14

Table 4: Mersenne NTT Parameters up to 20000 bits

3.8. Mersenne Negative W Parameters

With a negative w:

p	d	w	u	w	bits	nttwords
17	34	-2	1	17	17	1
19	38	-2	1	19	19	1
23	46	-2	2	23	46	1
29	58	-2	4	29	116	1
31	62	-2	4	31	124	1
37	74	-2	5	37	185	2
41	82	-2	6	41	246	2
43	86	-2	7	43	301	2
47	94	-2	8	47	376	2
53	106	-2	9	53	477	2
59	118	-2	10	59	590	2
61	122	-2	11	61	671	2
67	134	-2	12	67	804	3
71	142	-2	13	71	923	3
73	146	-2	14	73	1022	3
79	158	-2	15	79	1185	3
83	166	-2	16	83	1328	3
89	178	-2	17	89	1513	3
97	194	-2	19	97	1843	4
101	202	-2	20	101	2020	4
103	206	-2	21	103	2163	4
107	214	-2	22	107	2354	4
109	218	-2	22	109	2398	4
113	226	-2	23	113	2599	4
127	254	-2	26	127	3302	4
131	262	-2	27	131	3537	5
137	274	-2	29	137	3973	5
139	278	-2	29	139	4031	5
149	298	-2	32	149	4768	5
151	302	-2	32	151	4832	5
157	314	-2	34	157	5338	5
163	326	-2	35	163	5705	6
167	334	-2	36	167	6012	6
173	346	-2	38	173	6574	6
179	358	-2	39	179	6981	6
181	362	-2	40	181	7240	6

191	382	-2	42	191	8022	6
193	386	-2	43	193	8299	7
197	394	-2	43	197	8471	7
199	398	-2	44	199	8756	7
211	422	-2	47	211	9917	7
223	446	-2	50	223	11150	7
227	454	-2	51	227	11577	8
229	458	-2	51	229	11679	8
233	466	-2	52	233	12116	8
239	478	-2	54	239	12906	8
241	482	-2	54	241	13014	8
251	502	-2	57	251	14307	8
257	514	-2	58	257	14906	9
263	526	-2	60	263	15780	9
269	538	-2	61	269	16409	9
271	542	-2	62	271	16802	9
277	554	-2	63	277	17451	9
281	562	-2	64	281	17984	9
283	566	-2	65	283	18395	9
293	586	-2	67	293	19631	10
307	614	-2	71	307	21797	10

Table 5: Mersenne NTT Parameters with a negative w up to 20000 bits

3.9. Fermat Parameters

Fermat Number Theoretical (FNT) transforms are transforms into domains of Z_q where q is a Fermat Number and has the form:

$$q = 2^{2^n} + 1$$

Fermat Numbers are not necessarily prime. Fermat numbers have several properties that make them suitable for modular operations. First, observe that for any Fermat number:

$$2^{2^n} mod\ q = -1$$

thus:

$$\left(2^{2^n}\right)^2 mod\ q = 1$$

Said another way, with an FNT we know that 2 raised to a power will eventually result in unity. So, FNT also produces a primitive root of unity with a base of 2.

2^n	d	w	u	s	bits	nttwords
16	32	2	1	16	16	1
32	64	2	4	32	128	1
64	128	2	11	64	704	2
128	256	2	27	128	3456	4
256	512	2	58	256	14848	8
512	1024	2	121	512	61952	16

Table 6: Fermat NTT Parameters with a negative w up to 60000 bits

CHAPTER 4: Algorithm Compare and Critique

In comparing algorithms, the term “spectral” is used in one particular series of papers and books about the subject. But, other authors have used the term “Fast-Fourier-Transform Modular Multiplication” and “Discrete-Fourier-Transform Improvements to Montgomery Multiplication”. Likewise, some authors have used the term “spectral”, but not leveraged it for anything close to exponentiation.

4.1. Comparison between Koç and Baktir SMM

Koç and Baktir use very similar algorithms for Spectral Modular Multiplication. Baktir does not address exponentiation and calculates arithmetic in $GF(p^m)$. The primary difference between the SMM algorithms is that Koç uses addition to achieve a modular zero result before shifting, then sets the first term to 0, and lastly carries the term to the next term in order to achieve the result of setting the least significant term to zero. Baktir subtracts to set the term to zero without having to consider the carry.

$\underline{N}(t)$	spectral equivalent of a multiple of the modulus, *n*, used during modular exponentiation such that the first term is 1.
z_0 or z0	the first term of the time polynomial z(t)
β	beta = –z0 mod b

Γ(t)	A special polynomial consisting of the negative powers of w such that: $\Gamma(t) = 1 + \omega^{-1}t + \omega^{-2}t^2 + \ldots + \omega^{-(d-1)}t^{(d-1)}$ When Γ(t) is component-wise multiplied against a polynomial in the spectral domain, it computes the one term left circular shift of the polynomial equivalent in the time domain.
m	(Baktir only) the equivalent of s, number of terms in input values
A(t)	The DFT(α(t)) where α(t) is the evaluation polynomial of the integer α α is the integer time domain value of the carry value used internal by the Koç algorithm.
X	(Baktir only) x is the value of a single word that right shifts terms when multiplied, also called *b* by Koç. X^{-1} is the spectral equivalent of a left shift of a single term… same as Γ(t)
F'	(Baktir only) same as N(t) described above

Table 7: Spectral Values used in Algorithms

Table 7 is a reference for the various values used during the exponentiation algorithms discussed in this thesis.

1: $Z(t) := X(t) * Y(t)$
2: $\alpha := 0$
3: **for** $i = 0$ **to** $d - 1$
4: $z0 := d^{-1} \cdot (Z_0 + Z_1 + \ldots + Z_{d-1}) \bmod q$
5: $\beta := -(z0 + \alpha) \; mod \; b$
6: $\alpha := (z0 + \alpha + \beta)/b$
7: $Z(t) := Z(t) + \beta \cdot \underline{N}(t) \bmod q$
8: $Z(t) := Z(t) - (z0 + \beta)(t) \bmod q$
9: $Z(t) := Z(t) * \Gamma(t) \bmod q$
10: **end for**
11: $Z(t) := Z(t) + A(t)$
12: **return** $Z(t)$

Figure 9: Koç Spectral Modular Multiplication

Please refer to Table 7 for an explanation of the terms in these algorithms

```
for i = 0 to d - 1
        C_i = A_i * B_i
end for
for j = 0 to m - 2
        S = 0
        for i = 0 to d - 1
                S = S + C_i
        end for
        S = -S/d
        for i = 0 to d - 1
                C_i = (C_i + F'_i * S) * X_i^-1
        end for
end for
return (C)
```

Figure 10: Baktir Spectral Modular Multiplication

Koc

```
for i = 0 to d - 1
        Z_i = X_i * Y_i
end for
α := 0
for i = 0 to d − 1
z_0 = d^-1 (Z_0 + Z_1 + . . . + Z_{d−1})
        mod q
β = −(z_0 + α) mod b
α = (z_0 + α + β)/b
Z(t) = Z(t) + β · N(t) mod q
Z(t) = Z(t) − (z_0 + β)(t) mod q
Z(t) = Z(t) * Γ(t) mod q
end for
Z(t) := Z(t) + A(t)
return Z(t)
```

Baktir

```
for i = 0 to d - 1
        Z_i = X_i * Y_i
end for

for j = 0 to m - 2
        z_0 := d^-1 (Z_0 + Z_1 + . . . + Z_{d−1})
        mod q
        β = - z_0

        Z(t) = Z(t) + β · N(t) mod q

        Z(t) = Z(t) * Γ(t) mod q
end for

return Z(t)
```

Figure 11: Koç vs. Baktir Rewritten SMM

In this comparison Figure, the expressions in Baktir that have almost identical meaning as the SMM algorithm by Koç were converted to use the same expressions. This includes the Baktir expressions of A, B, C, S, F, and X that were changed to their counterparts of X, Y, Z, β, $\underline{N}(t)$, and $\Gamma(t)$ respectively. This allows the much easier side-by-side contrast and comparison of the two algorithms. Major differences are underlined in the figure.

4.2. Clarifications of Saldamli and Koç Works

The primary issue in the Illustrative Example in [5] is that there is an error early on in the sample calculations in step 5 where b is set to 16 and not 8. Unfortunately, due to this early error, all remaining calculations are not suitable as reference.

Another area that caused difficulty is in the addition of α(t) carry during the final step of Spectral Modular Multiplication. It is absolutely necessary to calculate the DFT of the base evaluation polynomial of α(t). The base evaluation polynomial is defined as one in which each term x_i is $0 \leq x_i < b$ [5]. If $\alpha \geq b$, then it must be broken into the base evaluation polynomial by breaking α into multiple terms each term less than b.

There are several references to more efficient methods for handling this carry, such as pp. 144 in [5]. Combined with Notation 2 on pp. 132 of [5], this appears to be a component-wise addition, which is functionally correct and very efficient. But, it was determined through testing that by not calculating the DFT of the base evaluation polynomial, the algorithm results in overflows because the carry value will cause iterative increases in the size of the first time series term, and eventual overflow. However, these overflows occur much less frequently with smaller field sizes and thus will only manifest regularly during tests of larger field sizes. Because it manifests as an overflow of time series values while in the frequency domain, it can be difficult to detect unless the testing includes careful monitoring of time series values.

This concern with $\alpha(t)$ was noted in pp. 142 of [5], but in other sections it is not described, including in the Illustrative Example. Unfortunately, this note was during the early description of just the reduction step, and later sections describing the larger algorithm used different terminology.

CHAPTER 5: Testing and Results

5.1. Sample Result and Interim Values

The following section outlines the output values produced by a functional implementation of spectral modular exponentiation as outlined by Saldamli and Koç [5]. Assume that the following input values are provided:

RSA Values: m=48644, e=5581, n=136163

The sample results start with determining the following parameters. Parameter generation is covered in Chapter 3 on parameters. The input values require 18 bits of storage. An appropriate field to support the NTT must be calculated based on the 18 bit requirement. A Mersenne Number Theoretical Transform will be selected with a positive base for this example. By parameter generation derived from the calculation of the inequality described by Koç, the following parameters as suitable for this NTT.

$$q=2^{17}-1,\ d=17,\ w=2,\ u=2,\ s=9,\ bits=18$$

The first step in initialization is to calculate the inverse of d mod q in order to compute the inverse DFT matrix.

$$d^{-1} = 17^{-1} \bmod 131071 = 123361$$

Then is the initialization of two values that will be used later in computations. The value of the Γ sequence is computed by:

$\Gamma_i = w^{-i} \bmod q \qquad$ where $0 \leq i \leq d-1$

$$\Gamma = (1, 65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2)$$

$$ONE = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$$

Then it is necessary to evaluate derived parameters from the RSA input values provided:

$$m=48644 \; e=5581 \; n=136163$$

First, compute the evaluation polynomials for the input values:

$$n(t)= (3, 0, 2, 3, 3, 0, 1, 0, 2)$$

$$m(t) = (0, 1, 0, 0, 2, 3, 3, 2, 0)$$

Then, derive an appropriate multiple of *n* such that the first term for this multiple is 1 when expressed as an evaluation polynomial. This is accomplished by examining the evaluation polynomial for *n(t)* and extracting the first term n_0. The multiple required can be calculated by determining the modular inverse of n_0 with respect to *b*.

$$\delta = n_0^{-1} \bmod b$$

$$\delta = 3$$

This multiple is used to create a new value, $\underline{n}(t)$ by the following calculation.

$$\underline{n} = 136163*3 = 408489$$

$$\underline{n}(t) = (1, 2, 2, 2, 3, 2, 3, 0, 2, 1, 0, 0, 0, 0, 0, 0)$$

As verification, the first value of $\underline{n}(t)$ is indeed 1 and thus will be effective in later reduction steps.

The next initialization is not necessarily required, it depends on the implementation of DFT that is chosen. Since DFT in MNT or FNT can take advantage of characteristics of those transforms such that the transform can be accomplished by only shifts and additions, it is possible to accomplish the transform more simply than a matrix multiplication. However, for possible reference value the DFT matrix with these parameters is computed as the following:

1	1	1	1	1	1	1	1	1
1	2	4	8	16	32	64	128	256
1	4	16	64	256	1024	4096	16384	65536
1	8	64	512	4096	32768	2	16	128
1	16	256	4096	65536	8	128	2048	32768
1	32	1024	32768	8	256	8192	2	64
1	64	4096	2	128	8192	4	256	16384
1	128	16384	16	2048	2	256	32768	32
1	256	65536	128	32768	64	16384	32	8192
1	512	2	1024	4	2048	8	4096	16
1	1024	8	8192	64	65536	512	4	4096
1	2048	32	65536	1024	16	32768	512	8
1	4096	128	4	16384	512	16	65536	2048
1	8192	512	32	2	16384	1024	64	4
1	16384	2048	256	32	4	65536	8192	1024
1	32768	8192	2048	512	128	32	8	2
1	65536	32768	16384	8192	4096	2048	1024	512

Table 8: Sample DFT Matrix (first 9 columns)

1	1	1	1	1	1	1	1
512	1024	2048	4096	8192	16384	32768	65536
2	8	32	128	512	2048	8192	32768
1024	8192	65536	4	32	256	2048	16384
4	64	1024	16384	2	32	512	8192

2048	65536	16	512	16384	4	128	4096
8	512	32768	16	1024	65536	32	2048
4096	4	512	65536	64	8192	8	1024
16	4096	8	2048	4	1024	2	512
8192	32	16384	64	32768	128	65536	256
32	32768	256	2	2048	16	16384	128
16384	256	4	8192	128	2	4096	64
64	2	8192	256	8	32768	1024	32
32768	2048	128	8	65536	4096	256	16
128	16	2	32768	4096	512	64	8
65536	16384	4096	1024	256	64	16	4
256	128	64	32	16	8	4	2

Table 9: Sample DFT Matrix (last 8 columns)

And, then compute the inverse DFT matrix, which includes the scalar multiple of the pre-computed value for the inverse of *d*, which was 123361.

123361	123361	123361	123361	123361	123361	123361	123361
123361	127216	63608	31804	15902	7951	69511	100291
123361	63608	15902	69511	115681	61688	15422	69391
123361	31804	69511	123376	15422	100231	127216	15902
123361	15902	115681	15422	115651	31804	100291	30844
123361	7951	61688	100231	31804	115681	7711	127216
123361	69511	15422	127216	100291	7711	63608	115681
123361	100291	69391	15902	30844	127216	115681	100231
123361	115681	115651	100291	100231	69511	69391	7951
123361	123376	127216	61688	63608	30844	31804	15422
123361	61688	31804	7711	69511	115651	123376	63608
123361	30844	7951	115651	61688	15902	100231	123376
123361	15422	100291	63608	69391	123376	15902	115651
123361	7711	123376	7951	127216	69391	61688	69511
123361	69391	30844	115681	7951	63608	115651	7711
123361	100231	7711	30844	123376	100291	7951	31804
123361	115651	100231	69391	7711	15422	30844	61688

Table 10: Sample Inverse DFT Matrix (first 8 columns)

123361	123361	123361	123361	123361	123361	123361	123361	123361

115681	123376	61688	30844	15422	7711	69391	100231	115651
115651	127216	31804	7951	100291	123376	30844	7711	100231
100291	61688	7711	115651	63608	7951	115681	30844	69391
100231	63608	69511	61688	69391	127216	7951	123376	7711
69511	30844	115651	15902	123376	69391	63608	100291	15422
69391	31804	123376	100231	15902	61688	115651	7951	30844
7951	15422	63608	123376	115651	69511	7711	31804	61688
7711	15902	15422	31804	30844	63608	61688	127216	123376
15902	7711	7951	69391	69511	100231	100291	115651	115681
15422	7951	100231	115681	127216	30844	15902	69391	100291
31804	69391	115681	63608	7711	100291	127216	15422	69511
30844	69511	127216	7711	115681	31804	100231	61688	7951
63608	100231	30844	100291	31804	115651	15422	115681	15902
61688	100291	15902	127216	100231	15422	123376	69511	31804
127216	115651	69391	15422	61688	115681	69511	15902	63608
123376	115681	100291	69511	7951	15902	31804	63608	127216

Table 11: Sample Inverse DFT Matrix (last 9 columns)

Now with the DFT pre-computations and the value for $\underline{n}(t)$, calculate the spectral equivalent for this value, since it will be used in the spectral domain. Note that capital names will generally be used for spectral equivalents of values.

$$\underline{N}(t) = DFT(\underline{n}(t)) = (18, 1357, 15276, 80279, 9143, 94937, 57881, 44133, 33683, 15433, 28402, 121970, 62841, 86095, 105194, 22374, 7427)$$

Now, a special spectral polynomial is required to convert each input value to the residual. This is effectively multiplying a number by a special power of 2 that can be later reduced out of the number by using right shifts. In this case, the shifts will be whole terms of the evaluation polynomial which are *u* bits in length. The Koç algorithm requires shifts of *d* terms, i.e. the size of the polynomial, thus the initial residual must calculate the residual of $2^{d*u} \bmod n$. However, to reuse the code for spectral modular multiplication, and yet make this slightly more difficult to understand, the value used to calculate the residual will be twice the shift as necessary… 2^{2d*u} *mod n*. This is because the code for spectral modular multiplication includes a reduction and thus we must compensate for the reduction in the value we use for generating residuals… if we want to reuse this code:

$$SMM(a,a) = a*b*2^{-d*u}$$

The final value to calculate residuals is:

$$\lambda = 2^{2d*u} \text{ mod n}$$

$$\lambda = 2^{2*17*2} \text{ mod } 408489$$

$$\lambda = 83327$$

$$\lambda(t) = (3, 3, 3, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$$

DFT(λ(t)) = Λ(t) = (14, 461, 83327, 37739, 105275, 36269, 37445, 84405, 107492, 8733, 80991, 73339, 97159, 42601, 64807, 125581, 62981)

With the values in place to complete the exponentiation, the one remaining value that can be pre-computed is the residual of one in the spectral domain, as this value is a starting value for *c* when using the Left-to-Right Exponentiation algorithm. This is computed by calculating the Spectral Modular Multiplication of ONE(t) and Λ(t):

Residual of ONE(t) = (14, 461, 83327, 37739, 105275, 36269, 37445, 84405, 107492, 8733, 80991, 73339, 97159, 42601, 64807, 125581, 62981)

The following values are calculated during the exponentiation itself.

$$m(t) = (0, 1, 0, 0, 2, 3, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

M(t) = (11, 578, 48644, 106542, 4521, 25396, 25420, 70534, 115200,

14880, 68233, 103472, 38449, 60548, 98381, 34288, 102400)

Now calculate the residual of m(t) in the spectral domain,

$$\underline{M}\,(t) = SMM\,(\,M(t),\,\Lambda(t)\,):$$

$$\underline{M}(t) = (49, 869, 105421, 60270, 55992, 11906, 100117, 111736, 76691, 26697, 111079, 23975, 5919, 21790, 39138, 93857, 72212)$$

With M(t) calculated, the loop of the Left-to-Right exponentiation algorithm is run. Each loop involves the calculation of SMM($\underline{C}$(t),$\underline{C}$(t)) and if the ith bit of e is set, then it also calculates SMM($\underline{C}$(t),$\underline{M}$(t)).

	0	1	2	3	4	5	6	7	8
0	106	1065	47311	5951	11623	93817	129809	68283	123781
1	49	869	105421	60270	55992	11906	100117	111736	76691
2	82	1095	64381	36438	127588	97968	92803	51307	72410
3	175	3772	12499	130566	100915	41625	43898	129863	119443
4	115	1386	82144	40385	59276	45637	8331	53590	170957
5	97	1230	79156	92940	122006	33897	77702	129612	7492
6	130	2064	55581	102683	4748	114011	17251	18074	2187
7	112	2094	60855	26552	118228	117444	12633	62807	17154
8	118	1621	94183	36565	128202	91772	24724	117868	40779
9	142	2269	111006	40078	91301	119659	99158	84383	108181
10	103	2193	126354	28627	70264	83536	24900	79418	56100
11	178	3534	385	2712	27450	91878	19572	33318	119450
12	115	1690	26673	103021	38620	120063	8713	84393	115177
13	136	2769	75845	84958	61209	106657	104076	73787	1550

Table 12: Sample Interim C Values (first 9 terms)

	9	10	11	12	13	14	15	16
0	48769	102507	26805	26417	71819	129564	211529	80942
1	26697	111079	23975	5919	21790	39138	93857	72212
2	39545	129611	47641	65447	104000	97798	112870	38944
3	114974	67622	124599	18592	40920	73965	156183	572
4	40642	50434	61875	25644	110685	34094	154234	109615
5	37532	52197	62511	113319	88659	44226	127862	109609
6	66758	12892	107808	23909	68126	90871	223149	138802
7	64694	107438	130200	106206	66452	79545	173254	34345
8	50885	61603	64883	19023	61563	58910	189113	7215
9	74955	119495	39439	37666	16846	66614	185400	114747
10	61611	70637	119424	34322	143708	118489	91730	68648
11	105756	52346	64188	41342	121759	110338	102878	152130
12	53934	82055	69340	93208	39168	93205	94533	156207
13	84687	56674	8577	84621	21586	123027	113316	45620

Table 13: Sample Interim C Values (last 8 terms)

The final value after all iterations of the loop is:

Final: $\underline{C}(t)$= (136, 2769, 75845, 84958, 61209, 106657, 104076, 73787, 1550, 84687, 56674, 8577, 84621, 21586, 123027, 113316, 45620)

Now, the value of $\Lambda(t)$ must be divided out of the final value of $\underline{C}(t)$ to get C(t), which is the similar to the step in Montgomery Multiplication where the final result is converted from the Montgomery domain to the integer domain. This step is not done for every iteration of the exponentiation loop, only the final value. Because the algorithm for Spectral Modular Multiplication includes the reduction, the operation SMM($\underline{C}(t)$, ONE(t)) will accomplish the reduction:

Final: C(t)= (148, 2212, 58671, 53306, 1748, 85985, 50727, 92180, 2964, 71396, 8067, 52678, 103271, 127500, 148101, 92341, 97851)

The spectral domain evaluation polynomial must now be converted back to the time-series representation by using the inverse DFT operation.

IDFT(C(t))= c(t) = (34, 31, 25, 20, 18, 7, 7, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0)

The time series polynomial can't simply be pasted back together as consecutive bits, since some of the terms are now larger than the original 2 bits. Therefore, the evaluation is accomplished by the following algorithm:

```
INPUT:  u, n, a(t) where a_i is the ith word of a(t)
OUTPUT: a = a(b) mod n , where b=2^u
     1. a = 0
     2. For i from d-1 down to 0, do:
             a = a * 2^u mod n
             a = a + a_i mod n
     3. Return(a)
```

Figure 12: Paste Words Algorithm

The final result from the evaluation of c(t):

c = 53579

5.2. Functional Tests

Functional testing was added during the algorithm development to detect potential problems in the calculations as the various changes and variations on implementations were tested. Some changes in code, such as restructuring loops or the reorganization of calculations, had unintended consequences. Also, the lack of prior work having a valid example with correct interim values made initial implementation and debugging very difficult. Therefore, a battery of sanity checks was developed to verify the correctness of different aspects of the spectral math required in both Koç and Baktir versions of algorithms. These tests were executed prior to performance testing during each test to verify the correctness of the algorithm.

These tests shown following were using the following parameters, should they need to be duplicated: (MNT) q=2^19-1, d=19, w=2, u=2, s=10, bits=20, $b=2^u=4$. These tests were also done using sample values of an exponentiation in the time domain of:

$c = m^e \bmod n$ where e=53 n=3141

Testing Evaluation Polynomials

Sanity Test #1 covers the conversion of an integer to an evaluation polynomial and back. The value used is 2922 and it is using u=2, which means 2-bit words and each term will be in the range from 0-3. "*a_t*" is the debugging terminology for *a(t)*, which is the evaluation polynomial of *a*. The correct output will be the original value submitted.

```
a=2922
a_t=   [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0
       [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0
 a=2922
```

Figure 13: Sanity Check #1: break/paste evaluations polynomials

Testing DFT and IDFT Conversions

The next test implemented verifies the ability of performing the spectral transformation to frequency domain and back. It shows the interim DFT results of the transformation of a_t as A_t and the inverse DFT transform, along with the evaluation of the polynomial to the original value.

```
a=2922
a_t=    [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0
        [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0
A_t=    [0]:12 [1]:134 [2]:2922 [3]:78482 [4]:70195 [5]:35418 [6]:25092  [8]:99075
        [9]:6162 [10]:10451 [11]:72802 [12]:58630 [13]:50216 [14]:37226 [7]:39190
        [15]:85762 [16]:114691
a_t=    [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0
        [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0
a=2922
```

Figure 14: Sanity Check #2: DFT/IDFT

Testing Addition of Evaluation Zeros

Test 3 verifies the ability to add multiples of the spectral evaluation of the value *n*, which is the modulus of the modular exponentiation in the time domain, to another evaluation polynomial without altering the value of the number. In this case, the value is even more specific. It is a multiple of *n* such that the first term is 1 in the time domain. This property of having the first term equal 1 in the time domain is used in the spectral domain during reductions for Koç and this polynomial is referred to as N(t) [5]. In this example, the value is denoted as n_base_t and the changing values of a(t) are shown as a_t.

```
n_base_t= [0]:1 [1]:1 [2]:0 [3]:1 [4]:0 [5]:3 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0
[12]:0 [13]:0 [14]:0 [15]:0 [16]:0
        a=2922
            Now compute a(t) = IDFT( DFT(a_t) + N(t) )
        a_t= [0]:3 [1]:3 [2]:2 [3]:2 [4]:3 [5]:5 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0
[12]:0 [13]:0 [14]:0 [15]:0 [16]:0
            Now evaluate a = a(t) and test if a is unchanged
        a=2922
DFT/IDFT+N_base_t  test: PASS
            Now again compute a(t) = IDFT( DFT(a_t) + N(t) )
        a_t= [0]:4 [1]:4 [2]:2 [3]:3 [4]:3 [5]:8 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0
[12]:0 [13]:0 [14]:0 [15]:0 [16]:0
            Now evaluate a = a(t) and test if a is unchanged
        a=2922
DFT/IDFT+N_base_t  test: PASS
            Now again compute a(t) = IDFT( DFT(a_t) + N(t) )
        a_t= [0]:5 [1]:5 [2]:2 [3]:4 [4]:3 [5]:11 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0
[12]:0 [13]:0 [14]:0 [15]:0 [16]:0
            Now evaluate a = a(t) and test if a is unchanged
        a=2922
 DFT/IDFT+N_base_t  test: PASS
```

Figure 15: Sanity Check #3: Adding Evaluations of Zero

Testing Addition of Spectral Zeros

Test 4 verifies the ability to add multiples of the spectral evaluation polynomial of the modulus *n*, which is the modulus of the modular exponentiation in the time domain, to another spectral polynomial without altering the time domain value of the number. In this case, the value is even more specific. It is a multiple of n such that the first term is 1. In this output, the value is denoted n_base_t. This property is used during reductions for Koç [5]. Also, Test 4 verifies the ability to add the correct multiples of n_base_t to set the first term to 0: (modulo b, which is 4 in this case).

$a_0 = 0 \bmod b = 0 \bmod 4$

The correct number of multiples is called "beta" in the output. [5] This value is calculated by evaluation of:

beta = $-a_0 \bmod b$ where a_0 is the first term in a_t

```
a=2922
a_t= [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0
[16]:0
a0=2
beta=2
n_base_t= [0]:1 [1]:1 [2]:0 [3]:1 [4]:0 [5]:3 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0
[15]:0 [16]:0
N_base_t= [0]:6 [1]:107 [2]:3141 [3]:98825 [4]:4137 [5]:33569 [6]:24643 [7]:151 [8]:577 [9]:7681
[10]:74754 [11]:67633 [12]:5637 [13]:57377 [14]:16653 [15]:35201 [16]:94209
A_t= [0]:12 [1]:134 [2]:2922 [3]:78482 [4]:70195 [5]:35418 [6]:25092 [7]:39190 [8]:99075 [9]:6162
[10]:10451 [11]:72802 [12]:58630 [13]:50216 [14]:37226 [15]:85762 [16]:114691
A_t+beta*N_base_t= [0]:24 [1]:348 [2]:9204 [3]:13990 [4]:78469 [5]:102556 [6]:74378 [7]:39492
[8]:100229 [9]:21524 [10]:28888 [11]:76997 [12]:69904 [13]:33899 [14]:70532 [15]:25093
[16]:40967
a_t (1st is 0)= [0]:4 [1]:4 [2]:2 [3]:3 [4]:3 [5]:8 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0
[15]:0 [16]:0   a0 is 0 mod 4
a=2922
DFT/IDFT+a0*N_base_t test: PASS
```

Figure 16: Sanity Check #4: Zeroing out 0th term of time domain polynomial by addition in spectral domain

Testing Left Shift in Spectral Domain

Test 5 verifies the ability to shift left of the terms of the evaluation polynomial in the time domain by using operations in the frequency domain. This is done by the multiplication of Γ(t) as specified by [5]. Γ(t) is a special polynomial consisting of the negative powers of w such that:

$$\Gamma(t) = 1 + \omega^{-1}t + \omega^{-2}t^2 + \ldots + \omega^{-(d-1)}t^{(d-1)}$$

When Γ(t) is component-wise multiplied against a polynomial in the spectral domain, it computes the one term left circular shift of the polynomial equivalent in the time domain.

a_t START= [0]:4 [1]:4 **[2]:2** [3]:3 [4]:3 [5]:8 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

Multiply Γ(t) against A_t in the spectral domain to left-shift a_t, then show a_t = IDFT(Γ(t) * A_t)

a_t FINAL1= [0]:4 **[1]:2** [2]:3 [3]:3 [4]:8 [5]:0 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

Multiply Γ(t) against A_t in the spectral domain to left-shift a_t, then show a_t = IDFT(Γ(t) * A_t)

a_t FINAL2= **[0]:2** [1]:3 [2]:3 [3]:8 [4]:0 [5]:0 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

Figure 17: Sanity Check #5: Left shift of one term of time domain polynomial by operations in spectral domain

Testing Multiplications in Spectral Domain

Test 6 verifies that we can actually do multiplications in the frequency domain with smaller values. This is modulo arithmetic, so actual answers will be computed modulo n.

a=2922

a_t START= [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

A_t DFT= [0]:12 [1]:134 [2]:2922 [3]:78482 [4]:70195 [5]:35418 [6]:25092 [7]:39190 [8]:99075 [9]:6162 [10]:10451 [11]:72802 [12]:58630 [13]:50216 [14]:37226 [15]:85762 [16]:114691

A_t*A_t = [0]:144 [1]:17956 [2]:18469 [3]:4821 [4]:116993 [5]:85254 [6]:74451 [7]:97193 [8]:79506 [9]:90725 [10]:41258 [11]:13177 [12]:8854 [13]:102758 [14]:92464 [15]:71479 [16]:2063

a_t FINAL= [0]:4 [1]:8 [2]:12 [3]:12 [4]:20 [5]:24 [6]:21 [7]:14 [8]:13 [9]:12 [10]:4 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

a=846 ($= 2922^2 \bmod 3141$)

Figure 18: Sanity Check #6: Linearity of multiplication in DFT domain

Testing Additions in Spectral Domain

Test 7 verifies that we can actually do additions in the frequency domain with smaller values. This is modulo arithmetic, so actual answers will be computed modulo n.

```
a=2922

a_t START= [0]:2 [1]:2 [2]:2 [3]:1 [4]:3 [5]:2 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

A_t DFT= [0]:12 [1]:134 [2]:2922 [3]:78482 [4]:70195 [5]:35418 [6]:25092 [7]:39190 [8]:99075 [9]:6162 [10]:10451 [11]:72802 [12]:58630 [13]:50216 [14]:37226 [15]:85762 [16]:114691

A_t+A_t = [0]:24 [1]:268 [2]:5844 [3]:25893 [4]:9319 [5]:70836 [6]:50184 [7]:78380 [8]:67079 [9]:12324 [10]:20902 [11]:14533 [12]:117260 [13]:100432 [14]:74452 [15]:40453 [16]:98311

a_t FINAL= [0]:4 [1]:4 [2]:4 [3]:2 [4]:6 [5]:4 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

a=2703          ( = 2922+2922 mod 3141 )
```

Figure 19: Sanity Check #7: Linearity of addition in DFT domain

Testing Spectral Modular Product

Lastly, we check SMP. If SMP works, then SME is the only piece left and it is tested by the actual SME itself and comparisons against the reference algorithms. Because SMP computes, in this case, $SMP(a)=a\cdot a\cdot R^{-1}$, we first multiply *a* by R to create a value called a_p.

$$a\_p = a\cdot R$$

Given this value, we can compute:

$$SMP(a,a\_p) = a\cdot a\cdot R^{-1}R = a^2 \bmod n$$

This value is much easier to verify by computing 2922^2 mod 3141.

a=2922

a_p=375

A_t= [0]:12 [1]:134 [2]:2922 [3]:78482 [4]:70195 [5]:35418 [6]:25092 [7]:39190 [8]:99075 [9]:6162 [10]:10451 [11]:72802 [12]:58630 [13]:50216 [14]:37226 [15]:85762 [16]:114691

A_p_t= [0]:9 [1]:41 [2]:375 [3]:4811 [4]:70419 [5]:35883 [6]:12485 [7]:51347 [8]:98692 [9]:1549 [10]:9307 [11]:68707 [12]:20871 [13]:9765 [14]:22819 [15]:59907 [16]:57348

SMP_test Return A_t^2= [0]:21 [1]:78 [2]:846 [3]:12558 [4]:66575 [5]:4134 [6]:16782 [7]:71694 [8]:98320 [9]:34 [10]:238 [11]:3214 [12]:49678 [13]:2068 [14]:8302 [15]:34318 [16]:24591

SMP_test IDFT(a_t^2)== [0]:14 [1]:0 [2]:4 [3]:0 [4]:3 [5]:0 [6]:0 [7]:0 [8]:0 [9]:0 [10]:0 [11]:0 [12]:0 [13]:0 [14]:0 [15]:0 [16]:0

SMP(a,a_p)=846 (2922*2922 mod 3141 = 846)

Figure 20: Sanity Check #8: Spectral Modular Product

5.3. Functional Results

Functional tests were run for 3 sample numbers across MNT, MNT negative w, and FNT parameter selection. Starting at 20 bits and ending at 4000 bits, all functional tests passed. Since the original implementation and testing, these tests have been tested against numerous other random input numbers during the course of performance tuning.

Functional Issues

There were many barriers along the way to resolve before the functional tests worked, especially for larger parameters. At first, for performance reasons, the multi-precision arithmetic was implemented inline by custom functions and structures. However, since the first algorithm chosen was the Baktir algorithm and this algorithm was not suitable for RSA exponentiation, it became impossible to test and verify the multi-precision operations because the algorithm itself would not produce the expected output. In the course of understand the specifics of Baktir, a well-tested and accepted multi-precision library (GMP) was adopted to resolve any possible issues that might be resulting from the custom multi-precision arithmetic.

When this did not resolve the issues, the implementation of Sanity Tests ensued to diagnose the algorithm issues. What resulted was the full battery of tests described in the Chapter 5.1 on functional testing. This battery of tests revealed that during subtraction of the spectral equivalent of the modulus *n*, the value of the time-domain equivalent changed. This was accomplished by computing the inverse DFT on the interim value and displaying the result. The Koç method was tested, which relied on addition instead of subtraction. This method worked correctly and thus the Koç method become the primary candidate.

During the ongoing functional testing, the implementation of multi-precision logic for larger and larger portions of code was necessary as bit sizes grew. Not only did each and every time series and spectral term require multi-precision operations, several unexpected values required multi-precision as well:

1. *b* – a value required for the computation of left and right shift operations of whole terms. *b* exceeds 32-bits at a Mersenne NTT that supports 2475-bits.
2. *bit masks* – most bit masks require multi-precision values, such as during the evaluation of values into time-series polynomials and also during custom modular operations designed to take advantage of MNT or FNT reduction techniques.

3. *α (carry value)* - the carry value in Koç is almost always a smaller number as it is not the size of the modulus q but rather the same size as b. But, b exceeds 32-bits at a Mersenne NTT that supports 2475-bits.

5.4. Performance Testing

Overview

One of the difficulties in performance testing was to develop a fair system to measure performance of algorithms that originated in different libraries and that had different implementations. Some of the questions encountered and resolved during the testing were:

1. How do we measure performance in a way that scores different algorithms accurately? Application of the high-resolution timer that measures CPU time only
2. Do we measure initialization code? No, initialization is not relevant to these performance measurements.
3. How much initialization do we measure? Only that which must be calculated as a result of the input value to be operated on (in our case, m). We also calculate initialization time that is included in operations that are tightly integrated into exponentiation operations and cannot be measured separately.
4. Do we use specifically chosen input parameters or randomly selected ones? Both, but only those specific or random parameters that would be likely found in RSA operations.

a. Randomized m will be used.

b. Random e or fixed e=17,65537 (typical parameters in RSA, but it is not necessary to test the fixed values of *e* used during encryption because these small values are almost always too small to benefit from the overhead of converting to the spectral domain)

5. What spectral parameters are chosen? Optimally chosen spectral parameters are chosen to match the input value bit sizes and provide the greatest performance for those given bit sizes.

Measurement Procedures

Measurements are accomplished by the application of the high-resolution timer that measures CPU time only. Floating point measurements where eventually implemented to record potentially large values in CPU time for inefficient configurations. The following code sample shows the types of functions used to measure time:

```
timespec diff(timespec start, timespec end)
….
clock_getres(CLOCK_PROCESS_CPUTIME_ID, &time1);
….
 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);
SME(&FD);
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
```

```
time_diff=diff(time1,time2);

USeconds = time_diff.tv_sec*1000000000+time_diff.tv_nsec;

printf("Time_SME,%d,%lu ns\n",test_bits, USeconds);
```

6.1. Algorithm Profiling Analysis

There were several iterations of algorithm analysis as the algorithm evolved over time. As seen in this first profile analysis, the DFT calculation time accounts for a very small percentage of total execution time (2.69%) in this release of the code. This code includes the shift improvements, but not the planned improvements to the Koç SMP algorithm. It was fairly consistent for all implementations in software that SMP was the most costly operation.

% time	self seconds		self calls	ms/call short name
83.92	1.25	448	2.79	SMP_koc
12.08	0.18	28	6.43	init_DFT
2.69	0.04	168	0.24	DFT
1.34	0.02	28	0.71	IDFT
0	0	546	0	find_max_u
0	0	140	0	break_words
0	0	56	0	init_SME
0	0	28	0	init_GAMMA
0	0	28	0	init_sme_math
0	0	28	0	init_d_inverse
0	0	28	0	SME
0	0	28	0	init_ONE
0	0	28	0	init_RSA

6.2. Performance Modifications #1

1. Removing targeted modulus operations

2. Convert divisions during SMP by numbers in the form 2^k to shifts by k positions.

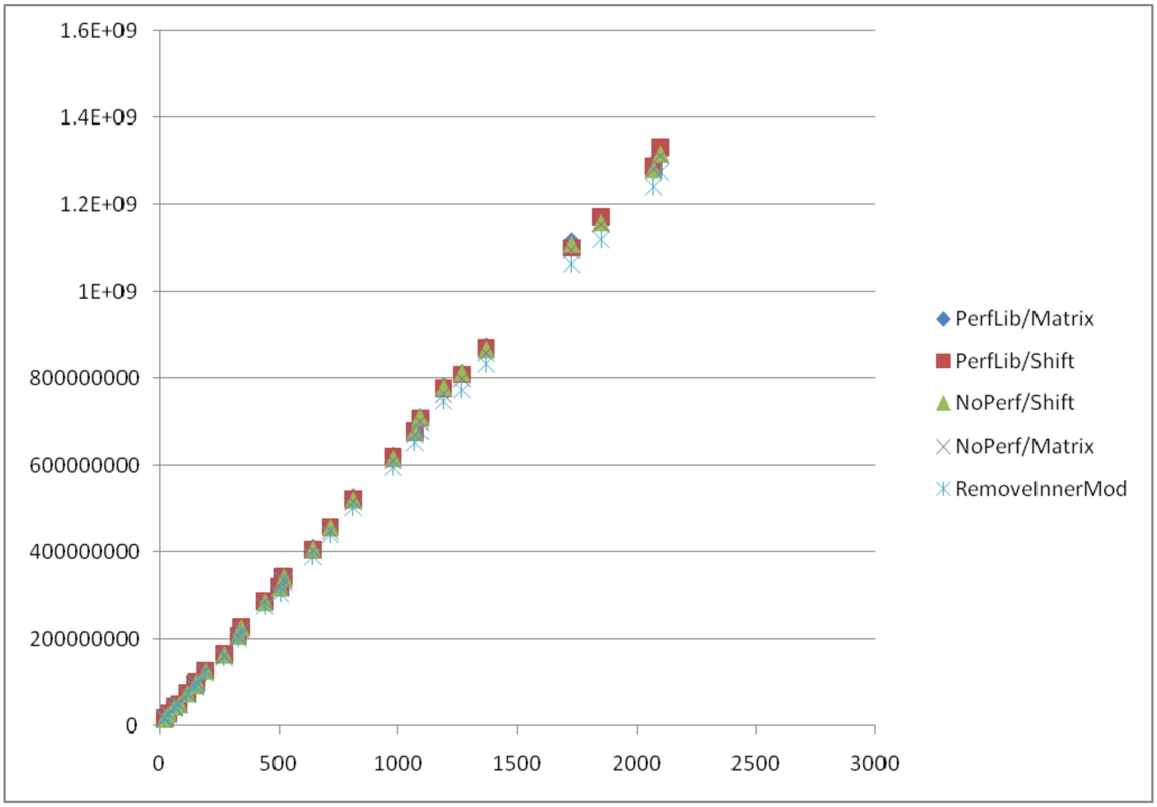

Figure 21: Timing Comparison Performance Modifications #1

This chart shows the execution time (y-axis) of these various modifications at various bit sizes of operands (x-axis). There was not a noticeable change in performance in any of these performance enhancements. These modifications are listed below:

- Perflib/Matrix – Performance Monitoring and Matrix DFT Multiplications
- Perflib/Shift – Performance Monitoring and DFT Multiplications computed by only shifts and additions.

- NoPerf/Matrix – No Performance Monitoring and Matrix DFT Multiplications
- NoPerf/Shift – No Performance Monitoring and DFT Multiplications computed by only shifts and additions.
- RemoveInnerMod – Removing certain modulus operations to measure performance improvements.

6.3. Performance Modifications #2

1. Moving multiple term calculations to the same major loop
2. Moving summation for z_0 calculation to final loop for future calculations.
3. Convert Gamma multiplication to modular shift.
4. Take advantage of the properties of MNT and FNT arithmetic.

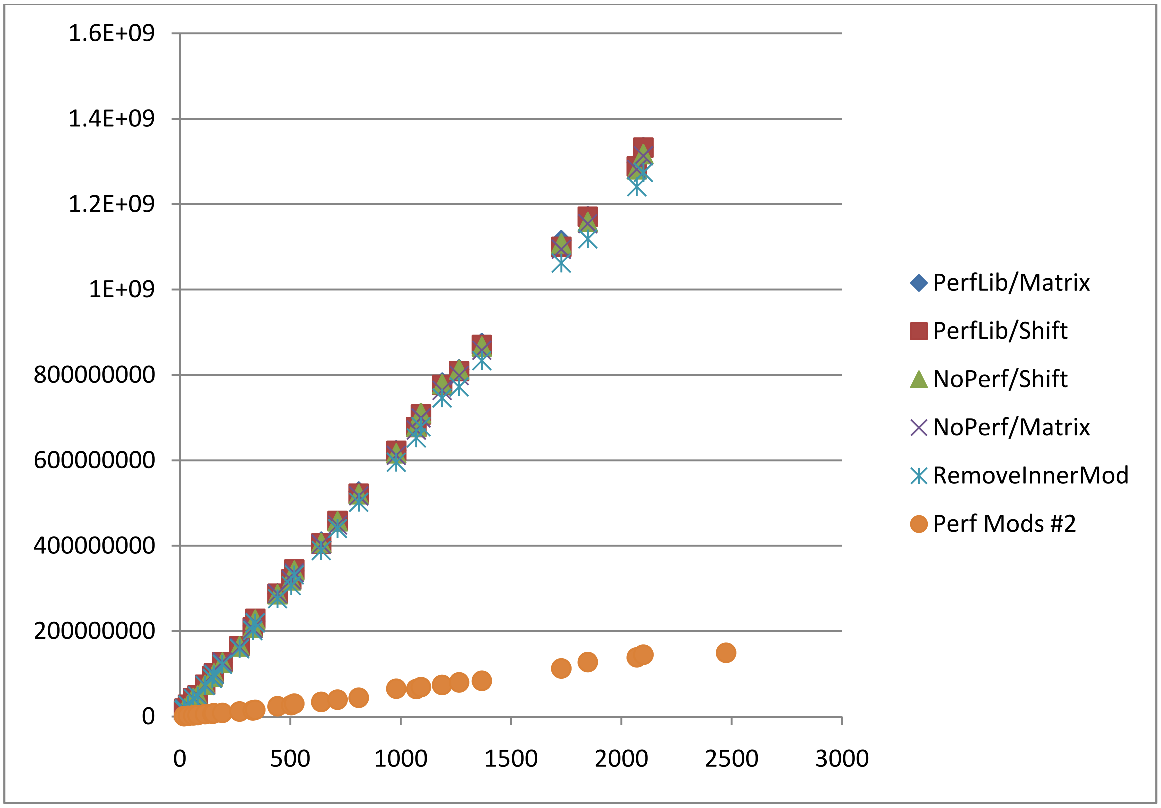

Figure 22: Timing Comparison Performance Modifications #2

It can be seen that this modification “Perf Mods #2” had a significant improvement in the overall performance in software by the SME algorithm. This chart shows the execution time (y-axis) of this modification at various bit sizes of operands (x-axis). Because the reference algorithm was implemented in a different library, a series of alternatives to the reference algorithms were implemented. These enhancements are discussed further in the next section.

- Perf Mods #2 – These modifications were extensive rewrites of the ordering of operations to improve efficiency. These changes are described in the following section.

6.4. Performance Enhancement #2 Details

Minor Algorithm Adjustments

One potential performance improvement was the attempt to add in α directly into the spectral representation. This turned out to violate the overflow controls because α is a carry value that can sometimes grow very large. Thus, the only safe way to incorporate it is to evaluate the single term across the entire time-domain representation and then take the DFT to add it into the spectral domain. This adds an additional DFT computation that is required for every Spectral Modular Product operation as seen by the Koç text "A(t) is the DFT pair of the base polynomial of α" [5].

for $i = 0$ **to** $e - 1$

  $y_0 := e^{-1} \cdot (Y_0 + Y_1 + \ldots + Y_e)$

  $\beta := (y_0 + \alpha)$ **rem** b

  $\alpha := (y_0 + \alpha)$ **div** b

  $Y(t) := Y(t) - \beta \cdot \underline{N}(t)$

  $Y(t) := Y(t) - (y_0 - \beta)(t)$

  $Y(t) := Y(t) \odot \Gamma(t)$

end for

$Y(t) := Y(t) + A(t)$, *where* $A(t)$ *is the DFT pair of the base polynomial of* α.

return $Y(t)$

> b is defined as 2^u, so it is always a power of 2. Division by b is a right shift by u.

Koç SMM Algorithm Component [5], Division Improvement

```
for i = 0 to e − 1
    y0 := e^{-1} · (Y0 + Y1 + ... + Ye)
    β := (y0 + α) rem b
    α := (y0 + α) div b
    Y(t) := Y(t) − β · N(t)
    Y(t) := Y(t) − (y0 − β)(t)
    Y(t) := Y(t) ⊙ Γ(t)
end for
Y(t) := Y(t) + A(t), where A(t) is the DFT pair of the base polynomial of α.
return Y(t)
```

> This matrix is built from w. For MNT parameters, w=2, and therefore every element is a shift… much faster than a multiply.

Koç SMM Algorithm Component [5], Left-Shift Improvement

Loop Operation Adjustments

The Koç pseudo-code is not suitable for fixed architectures when an "a+b" operation is actually adding large multi-word numbers with many reads and writes. [5]

Instead of doing a single operation across all terms, and having to fetch/save every word,

- Complete all operations on a single word before moving on.
- Cumulative operations (such as summation of all terms) can also be interwoven into loops.

Some advantages in using multi-precision libraries:

- Do not need to worry about allocation/reallocation of memory during operations.
- Do not need to worry about special values in operations (like divide by 0 or multiply by 2).

- Do not need to worry about choosing between algorithms based on parameters (like Karatsuba multiplication).

Some notable disadvantages:

- Some operations have significant overhead
- Cannot take advantage of holistic algorithm knowledge , such as efficient modulus operation with Mersenne or Fermat numbers.

Take advantage of Fermat and Mersenne Arithmetic

Mersenne and Fermat rings have certain characteristics that make modular reductions much more efficient. In MNT, $q = 2^p - 1$. The number being reduced, *a,* can be broken into higher and lower portions along the 2^p boundary such that:

$a = a_h \cdot 2^p + a_l$ but, because $2^p = 1 \bmod 2^p - 1$,

$a = a_h + a_l$ so reduction can be accomplished with shifts and addition

Similarly for FNT:

$a = a_h \cdot 2^p + a_l$ but, because $2^p = -1 \bmod 2^p + 1$,

$a = -a_h + a_l$ so reduction can be accomplished with shifts and subtraction

These are very good opportunities for performance increases, as the division to accomplish modular reduction is very expensive. However, *p* is prime in our testing, and therefore will never be word-aligned. In our software testing, this results in several additional checks and shifts in order to accomplish this reduction.

These characteristics were also combined into efficient additions and subtractions.

6.5. Performance of MNT with Negative *w*

It is possible to create appropriate parameters for an MNT with *w=-2*. This immediately increases the number of terms or “size” of the DFT by a factor of two. However, when comparing like-sized DFT in regards to overall bit size, the results are slightly unexpected.

Take the following two example sets of parameters, the first is positive *w*, the second is negative w.

q=2^101-1: w=2 u=21 s=51 d=101 bits=1071

q=2^73-1: w=-2 u=14 s=73 d=146 bits=1022

The negative *w* results in more bits with a smaller field element bit length (u) and a smaller field modulus. These are generally positive benefits. But, in software, this is at the cost of more DFT elements, *d*. Since software cannot parallelize *d*- way operations, this results in more calculations overall as seen in the following chart of comparison timings. Be aware that negative and positive *w* do not produce the same field sizes in the charted measurements.

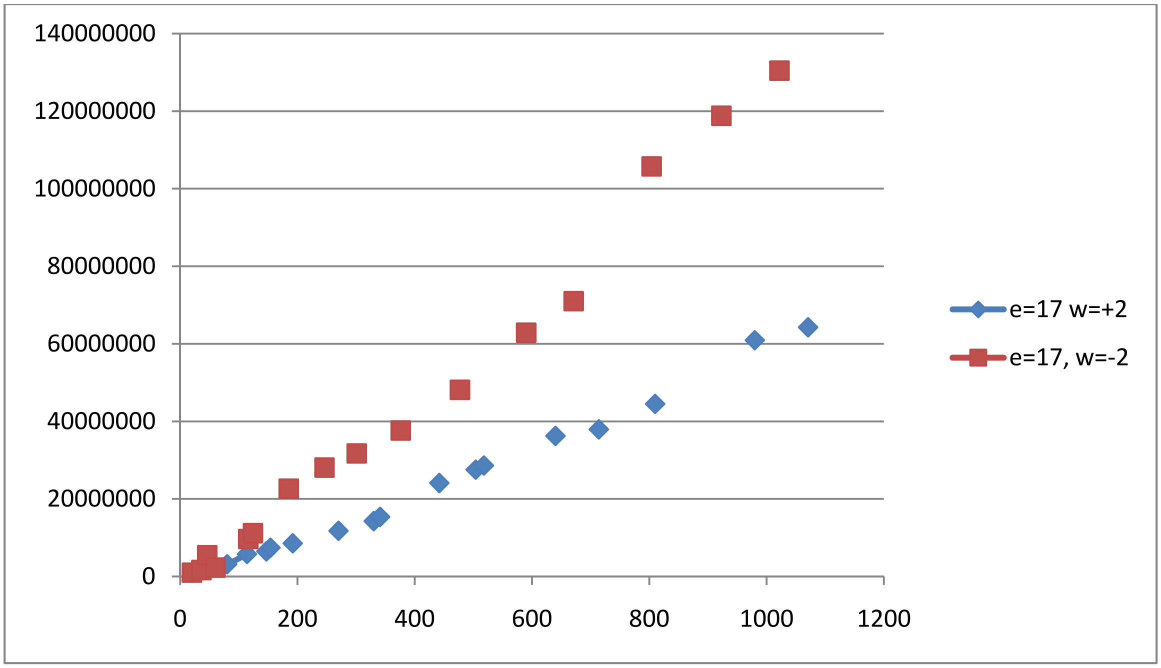

Figure 23: Timing Comparison of Positive vs. Negative W (usec)

This chart shows the execution time of the exponentiation on the y-axis vs. the bit size of the operands that is supported by the parameters on the x-axis. It shows that for w=-2 (negative), the execution time is generally higher than for positive w=2.

The one situation where this set of parameters will be beneficial is when making calculations in a fixed architecture where if the bit size exceeds the architecture capabilities it will cause a significant increase in computation time. For example, in a 32-bit architecture, if using only positive w=2 with MNT and limiting the word size to 32 bits, SME can achieve a maximum bit length of 2100 bits (u=30). With w=-2, SME can support 4832 bits (u=32).

6.6. Final Reference Timing Comparisons

In software, many iterations of field sizes and implementations were tested to find potential areas where SME performance in software would exceed that of more traditional algorithms.

Reference Algorithms

The following chart shows Spectral Modular Exponentiation timings versus the following reference implementations, all of which were tested under identical conditions. All implementations used random values for *e*, *m*, and *n* during a single exponentiation and measurements are in seconds.

1. Sliding Window – This uses the Sliding Window exponentiation algorithm along with multiplication by Karatsuba/Toom-3 and reduction using arithmetic division [8].
2. Left-to-Right - This uses the Left-to-Right or "square-and-multiply" exponentiation algorithm along with multiplication by Karatsuba/Toom-3 and reduction using arithmetic division [8].
3. Montgomery - This uses the Left-to-Right or "square-and-multiply" exponentiation algorithm along with multiplication and reduction achieved by Montgomery Multiplication.

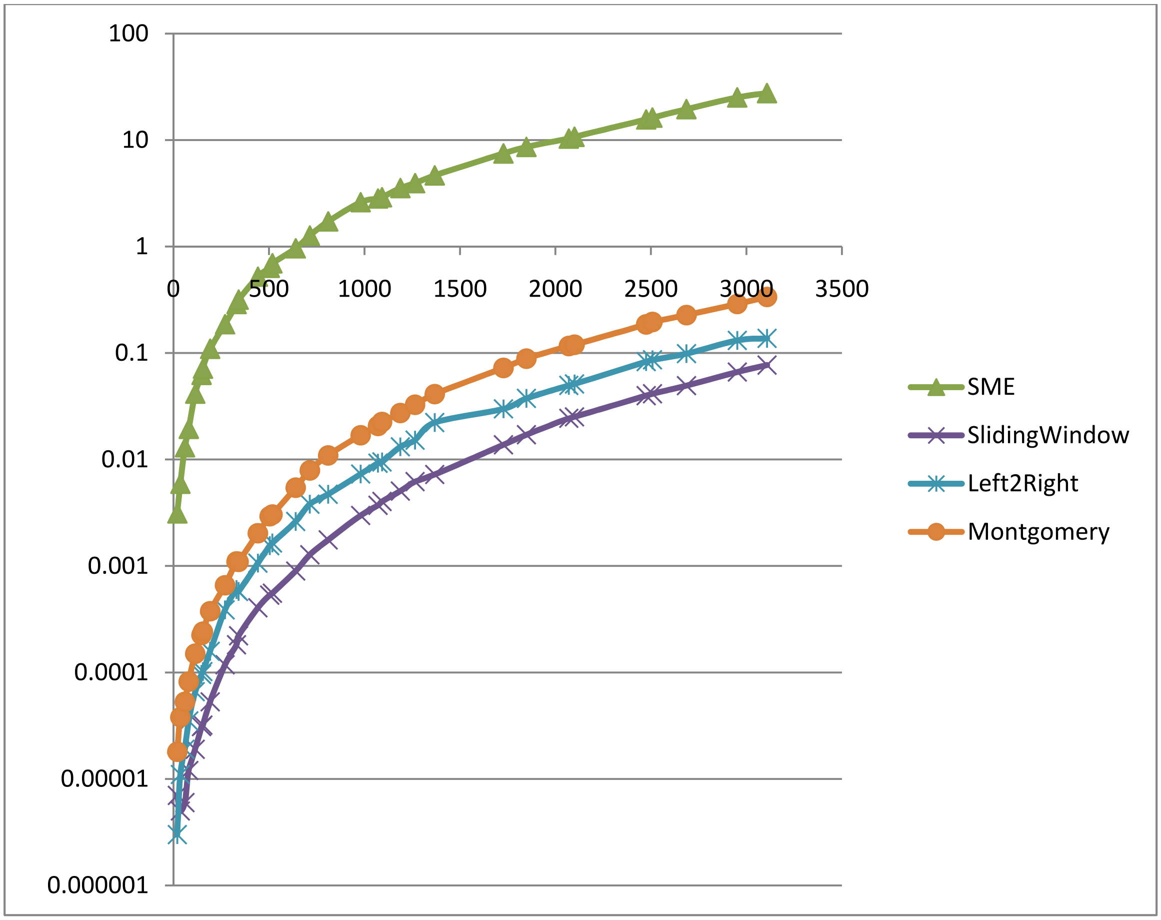

Figure 24: Spectral Timings vs. Reference Implementations (sec)

These timings in logarithmic scale demonstrate the superiority of the reference algorithms over the basic implementation of Spectral Modular Exponentiation at all bit sizes tested.

Left-to-Right Montgomery vs. Parallel Simulation Reference Timings

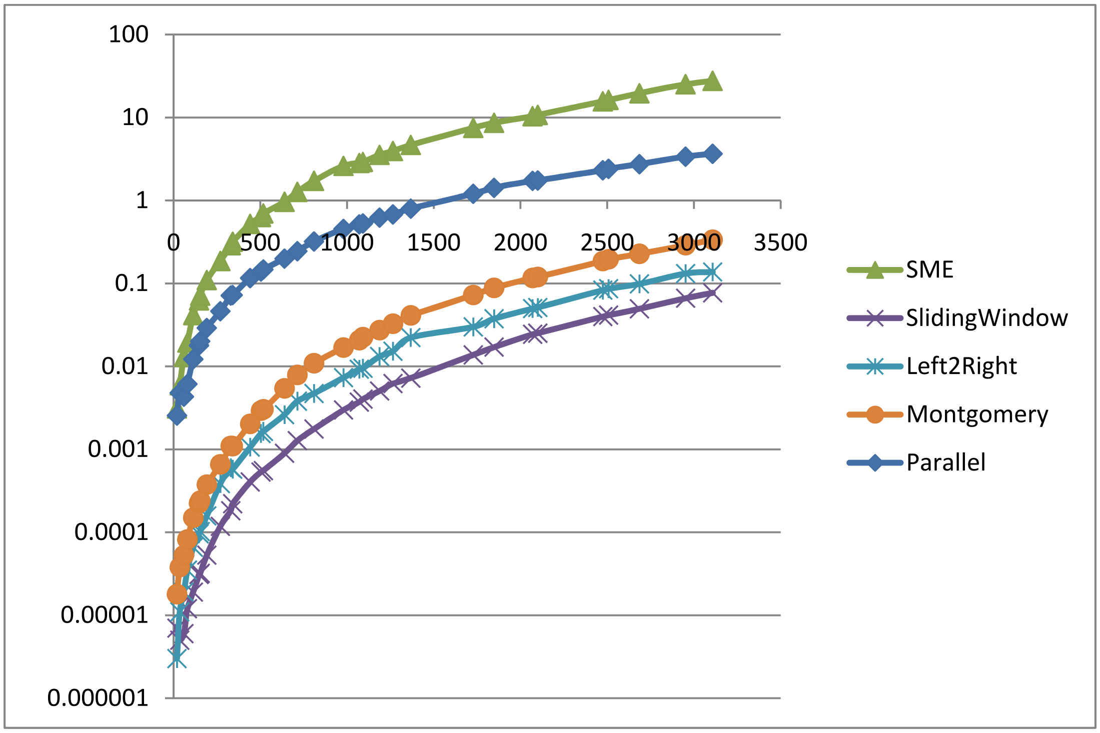

Figure 25: Theoretical Spectral Parallelized Timings vs. Reference Implementations (sec)

These timings in logarithmic scale demonstrate the superiority of the reference algorithms over the basic implementation of a theoretical timing of Spectral Modular Exponentiation with parallel operations at all bit sizes tested. The parallel operations were applied to the following segments of the SME algorithm:

7: $Z(t) := Z(t) + \beta \cdot N(t) \bmod q$
8: $Z(t) := Z(t) - (z0 + \beta)(t) \bmod q$
9: $Z(t) := Z(t) * \Gamma(t) \bmod q$

Steps 7-9 are all operations that occur independently across *d* terms, and thus are easy candidates for parallel testing. Because *d* is sometimes very large, it was not possible to provide actual values for all sizes of d, but the values provided are approximations of parallel calculation performance for steps 7-9.

CHAPTER 7: Performance Limitations

7.1. Complexity of SME in Software

There are performance limitations with the software implementation of the Spectral Modular Exponentiation algorithm that was proposed by Koç [5]. These limitations were discovered during the performance testing described in Chapter 6. In profiling the code during testing, it was discovered that some operations in SMM were executed much more often than expected.

To explore these concerns, the following complexity calculations were derived from the parameters generated of d, s, q, u, and bit size of a single spectral term and how these parameters affect the quantity of operations in SME. The complex relationship between u, d, and q as determined by the overflow inequality make it difficult in solving for efficiency values directly, so efficiency was modeled based on the tabulated parameters that resulted from the iterative solving of the inequality.

With the output of parameter generation combined with the analysis of the loops and multi-word operations in the implementations of SME, the following estimates were determined for the number of operations required to compute critical steps of spectral modular exponentiation at various operand sizes. These critical steps were called InnerLoop operations on the charts.

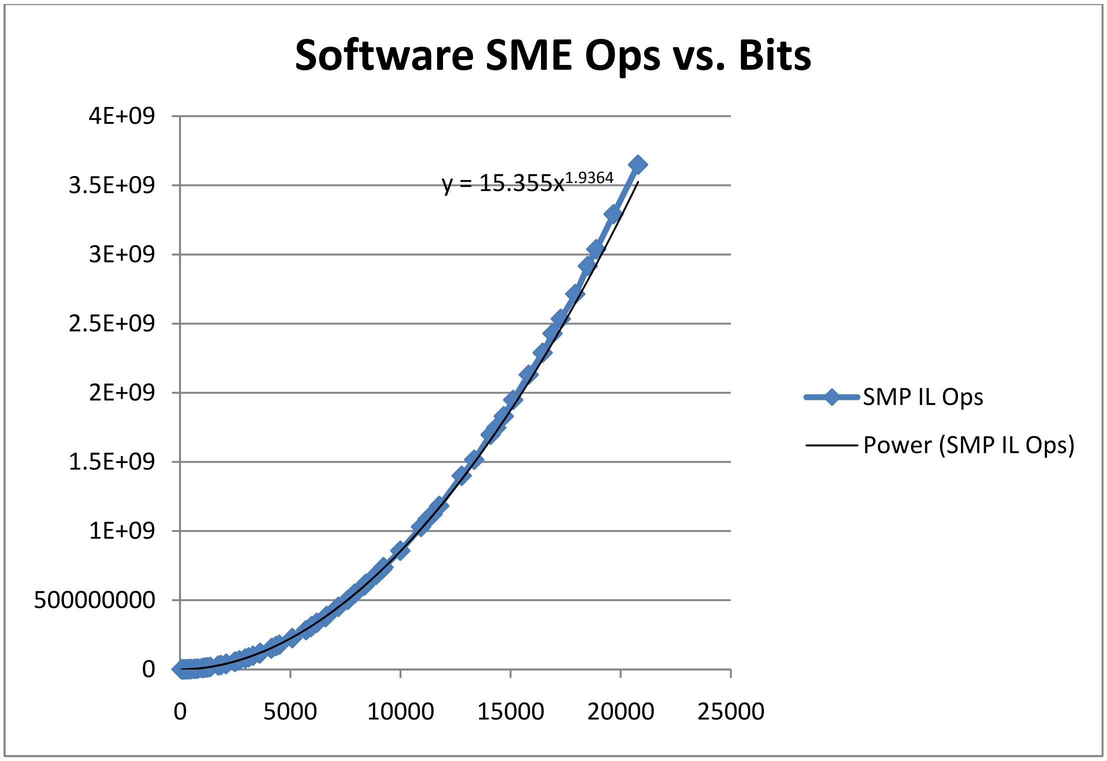

Figure 26: Calculated SME Inner-Loop Operations

This chart shows the calculated count for InnerLoop operations for SME over a variety of operand bit sizes (x-axis). It appears from the chart that the number of operations has a quadratic relationship for this model based on software calculations.

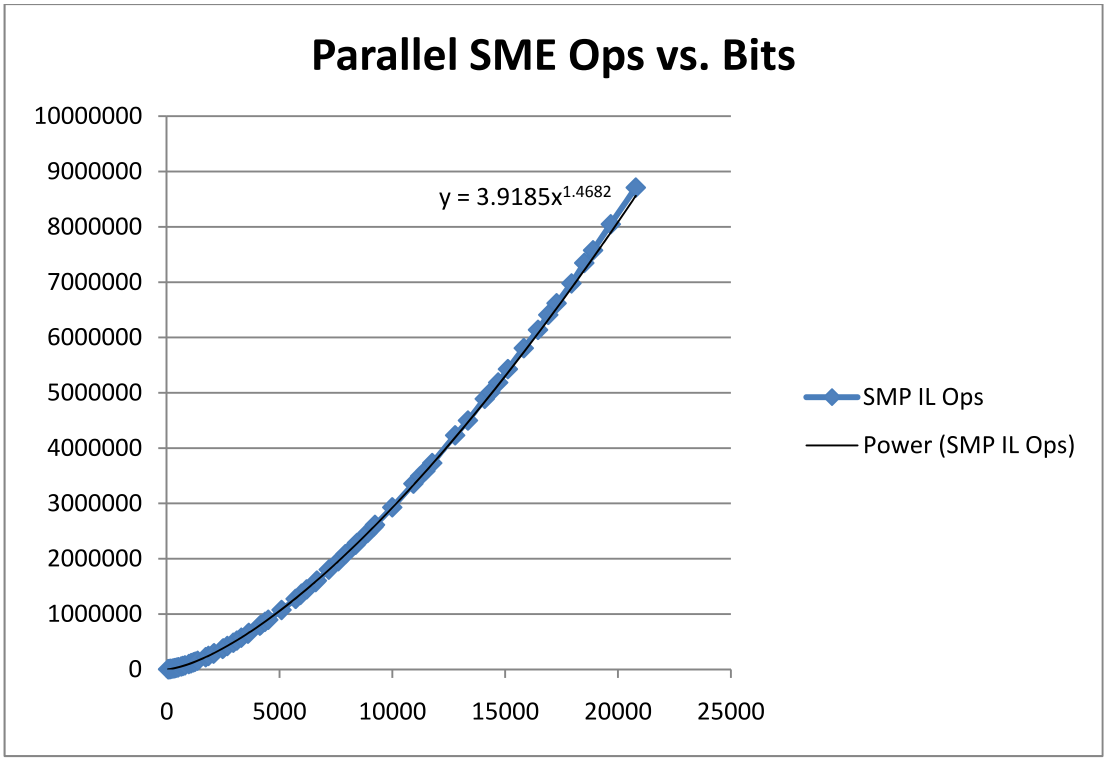

Figure 27: Calculated Parallelized SME Inner-Loop Operations

By examining these charts, it is seen that the modeled equation for the number of Inner Loop (IL) operations shows that in software the number of operations in the SMP increase as $O(n^2)$. This efficiency takes it characteristics from the fact that as bit sizes increase, so does the bit size of the terms, the number of terms, and the number of reductions steps that must be computed. Parallelization would only resolve this efficiency issue if it were possible to calculate *d* parallel operations simultaneously (where *d* varies between 29 and 419 in our test parameters).

With sufficient resources, the computations can be done in parallel for several of the internal loop operations of SMP and the computational efficiency is modeled at approximately $O(n^{1.5})$.

Chapter 8: Conclusions and Future Work

This thesis provides significant additional information for the implementation of algorithms that apply spectral modular exponentiation. Specifically, this covers some beneficial characteristics determined concerning parameter generation for both Mersenne and Fermat primes that allow the reliable generation of parameters.

The performance of software implementations was not an improvement over the reference algorithms. The lack of efficient operations to support spectral math in software and the absolute necessity for hundreds of simultaneous parallel operations made competitive performance difficult in software. In hardware, these issues might be resolved.

The production of verified intermediate values and the corresponding parameters for these successful operations will be beneficial for future implementations and assist in future attempts at hardware implementations.

REFERENCES

REFERENCES

[1] M. B. Tandrup, M. H. Jensen, R. N. Andersen, T. F. Hansen, “Fast Exponentiation In practice,” Dec 2004.

[2] K. Kalach, J. P. David, “Hardware implementation of large number multiplication by FFT with modular arithmetic,” IEEE-NEWCAS Conference, 2005. The 3rd International Volume, June 2005, pp. 267-270.

[3] S. Baktır., B. Sunar, “Achieving Efficient Polynomial Multiplication in Fermat Fields Using the Fast Fourier Transform,” *ACM SE’06 March 1012, 2006*, Melbourne, Florida, March 2006

[4] C. D. Walter, “Logarithmic Speed Modular Multiplication,” Department of Computation, UMIST.

[5] G. Saldamlı and C. K. Koç, Chapter 7, *Spectral Modular Arithmetic for Cryptography*, pp. 125-169.

[6] G. Saldamlı, Spectral Modular Arithmetic, Ph.D. thesis, Department of Electrical and Computer Engineering, Oregon State University, May 2005.

[7] G. Saldamlı, Spectral Modular Arithmetic, M.S. thesis, Department of Electrical and Computer Engineering, Oregon State University, May 2003.

[8] “Algorithms,” GMP Documentation, http://www.cims.nyu.edu/cgi-systems/info2html?%28gmp%29Algorithms

[9] S. Baktir, et al. “A State-of-the-art Elliptic Curve Cryptographic Processor Operating in the Frequency Domain”, *Mobile Networks and Applications*, vol. 12, August 2007, pp. 259-270.

[10] J. L. Massey, “The Discrete Fourier Transform in Coding and Cryptography,” San Diego , ITW, 1998.

[11] J. M. Pollard, “The Fast Fourier Transform in a Finite Field,” *Mathematics of Computation*, pp. 365-374, 1971.

[12] “Determinant,” http://en.wikipedia.org/wiki/Determinant, June 2010.

[13] H. J. Nussbaumer, “Fast Fourier Transform and Convolution Algorithms,” Berlin: Springer, 1982.

[14] Fermat Number. http://en.wikipedia.org/wiki/Fermat_number

[15] Mersenne Prime. http://en.wikipedia.org/wiki/Mersenne_Prime

[16] Fermat Number . http://mathworld.wolfram.com/FermatNumber.html

[17] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", *Computing* 7 (1971), pp. 281–292.
http://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm

[18] Karatsuba. http://en.wikipedia.org/wiki/Karatsuba

[19] A. J. Menezes, P. C. van Oorschot, Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996, Chapter 14 - Efficient Implementation.

CURRICULUM VITAE

Matthew Estes received his Bachelor of Science in Computer Engineering in 1998 from Rose-Hulman Institute of Technology. He currently works as a Lead Information Systems Engineer for the MITRE Corporation. He began his graduate degree at George Mason for Computer Engineering in 2005 and is currently a member of the Cryptographic Engineering Research Group (CERG) and the Spectral Modular Arithmetic Group (SMAG) at George Mason University.