

AN ABSTRACT OF THE THESIS OF

Mehmet Musa for the degree of Master of Science in Electrical & Computer Engineering presented on June 21, 2000.

Title:

Improved Montgomery Algorithms Using Special Primes and
Impact on Elliptic Curve Digital Signature

Abstract approved: _____

Çetin K. Koç

The Montgomery multiplication methods constitute the core of the modular exponentiation operation, which is the most popular method in public-key cryptography for encrypting and signing digital data. In this thesis, we concentrate on improving versions of Montgomery Algorithms using a new family of special prime numbers of the form $2^k + 2^i + 1$ ($k, i \in N, k > i$). We show that the multiplication operation ($z = x.y.r^{-1}$ in the field $GF(p)$) can be implemented significantly faster than the standard multiplication when using special primes of the form $2^k + 2^i + 1$. The optimized Montgomery algorithms have been implemented in C. The analysis and actual performance results indicate that the Coarsely Integrated Operand Scanning (CIOS) method, detailed in this thesis, can be best optimized to take advantage of these special primes. We also study how the Elliptic Curve Digital Signature (ECDS) benefits dramatically from a new implementation of Montgomery multiplication algorithms that take advantage of primes of the form $2^k + 2^i + 1$.

©Copyright by Mehmet Musa

June 21, 2000

All Rights Reserved

Improved Montgomery Algorithms Using Special Primes and
Impact on Elliptic Curve Digital Signature

by

Mehmet Musa

A THESIS submitted
to
Oregon State University

in partial fulfillment of the
requirements for the degree of
Master of Science

Completed June 21, 2000
Commencement June 2001

APPROVED:

Major Professor, representing Electrical & Computer Engineering

Head of Electrical & Computer Engineering Department

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Mehmet Musa, Author

ACKNOWLEDGMENTS

First, I'd like to thank my family for their endless support and for encouraging me along the way during my studies.

I give special thanks to Dr. Koç, my major professor, for recognizing my potential, for his sponsorship, and for his reviews and help in producing the paper we have been working on, which formed a basis for this thesis.

I also thank Dr. Minoura, Dr. Tenca and Dr. Nielsen for dedicating their time to participate in my graduate committee.

I would also like to thank the following people who have provided valuable help during my research: Murat Aydos, Francisco Rodríguez-Henríquez, Erkay Savaş, and Tuğrul Yanık.

I gratefully acknowledge review of an earlier draft provided by Tim Worley.

Finally, I acknowledge financial support for this work from Secured Information Technology, Inc..

Mehmet Musa

Corvallis, Oregon

TABLE OF CONTENTS

	<u>Page</u>
1	INTRODUCTION 1
2	CRYPTOGRAPHIC SYSTEMS 4
2.1	Information Security 4
2.2	Types of Cryptographic Systems 5
2.2.1	Symmetric Key Cryptosystems 6
2.2.2	Public Key Cryptosystems 7
2.3	Security Services using Public-Key Cryptosystems 8
2.3.1	Public-Key Encryption – Confidentiality 8
2.3.2	Digital Signatures – Data Integrity, Non-repudiation 8
2.4	Hard Mathematical Problem 9
2.4.1	Integer Factorization Problem 10
2.4.2	Discrete Logarithm Problem 11
2.4.3	Elliptic Curve Discrete Logarithm Problem 11
2.5	Comparison of Public-Key Cryptosystems 12
2.6	Elliptic Curves and Cryptography 14
2.6.1	Mathematical Background 14
2.6.2	Background in Elliptic Curve 15
2.6.3	Elliptic Curve Digital Signature Algorithm (ECDSA) 16
2.6.4	ECDSA Key Generation 16
2.6.5	ECDSA signature Generation 17
2.6.6	ECDSA signature Verification 17
3	IMPROVING MONTGOMERY ALGORITHMS 18
3.1	Introduction 18
3.2	Previous Work 18
3.3	Montgomery Multiplication 19
3.4	Optimizing Montgomery Product using Special Primes 24

TABLE OF CONTENTS (CONTINUED)

	<u>Page</u>
3.5 Separated Operand Scanning (SOS) Method	26
3.5.1 SOS Method with Arbitrary Primes	26
3.5.2 Optimized SOS Method with Special Primes	29
3.5.3 SOS Speed Improvement with Special Primes	31
3.6 Coarsely Integrated Operand Scanning (CIOS) Method	32
3.6.1 CIOS Method with Arbitrary Primes	32
3.6.2 Optimized CIOS Method with Special Primes	33
3.6.3 CIOS Speed Improvement with Special Primes	35
3.7 Finely Integrated Operand Scanning (FIOS) Method	36
3.7.1 FIOS Method with Arbitrary Primes	36
3.7.2 Optimized FIOS Method with Special Primes	36
3.7.3 FIOS Speed Improvement with Special Primes	39
3.8 Finely Integrated Product Scanning (FIPS) Method	39
3.8.1 FIPS Method with Arbitrary Primes	39
3.8.2 Optimized FIPS Method for Special Primes	41
3.8.3 FIPS Speed Improvement with Special Primes	42
3.9 Coarsely Integrated Hybrid Scanning (CIHS) Method	43
3.9.1 CIHS Method with Arbitrary Primes	43
3.9.2 Optimized CIHS Method for Special Primes	44
3.9.3 CIHS Speed Improvement with Special Primes	45
3.10 Montgomery Algorithm and ECDSA Timings	45
3.10.1 Running Times for Montgomery Algorithms	47
3.10.2 Running Times for ECDSA Using CIOS Method	49
4 CONCLUSIONS	51
4.1 Discussion of Results	51
4.2 Summary of Contributions	52
4.3 Future Work	53

TABLE OF CONTENTS (CONTINUED)

	<u>Page</u>
BIBLIOGRAPHY	54
APPENDICES	57
Appendix A	Table of Low-Weight Special Primes: $p = 2^k + 2^i + 1$ 57
Appendix B	Table of Low-Weight Special Primes: $p = 2^k + 2^i - 1$ 67

LIST OF TABLES

Table	Page
2.1. Computing power required to compute elliptic curve logarithm	12
2.2. Computing power required to factor integers	13
2.3. Size of system parameters and key pair	13
2.4. Signature size for 2000-bit messages	13
2.5. Encrypted message size for 100-bit messages	14
2.6. Correspondence between Z_p^* and $E(Z_p)$	16
3.1. SOS method operation breakdown	28
3.2. Optimized SOS method operation breakdown	31
3.3. SOS speed improvement breakdown	32
3.4. CIOS method operation breakdown	33
3.5. Optimized CIOS operation breakdown	35
3.6. CIOS speed improvement breakdown	36
3.7. FIOS method operation breakdown	37
3.8. Optimized FIOS operation breakdown	38
3.9. FIOS speed improvement	39
3.10. FIPS algorithm operation breakdown	40
3.11. Optimized FIPS algorithm operation breakdown	42
3.12. FIPS speed improvement	43
3.13. CIHS operation breakdown	44
3.14. Optimized CIHS operation breakdown	46
3.15. CIHS speed improvement	47
3.16. Special primes used for timings	47
3.17. SOS timings and improvement	48

LIST OF TABLES (CONTINUED)

	<u>Page</u>
3.18. CIOS timings and improvement	48
3.19. FIOS timings and improvement	49
3.20. FIPS timings and improvement	49
3.21. CIHS timings and improvement	50
3.22. ECDSA timings and improvement	50

To my family...

for their endless love and support

Improved Montgomery Algorithms Using Special Primes and Impact on Elliptic Curve Digital Signature

Chapter 1 INTRODUCTION

Governments and businesses in a number of commercial arenas today recognize information as their most valuable asset. They are all storing information in electronic form which gives them a number of advantages over previous physical storage. However, with the electronic revolution, information faces new and potentially more damaging security threats. Unlike information printed on paper, information in electronic form can potentially be stolen from a remote location. It is much easier to intercept and alter electronic communication than its paper-based predecessors. The fundamental goal of cryptography is to protect users from such attacks. This objective is being met by public and private-key cryptosystems which require implementations of cryptographic functions. More specifically, these include encryption, decryption, authentication, digital signature algorithms and message-digest functions. The security of such functions is based on the computational complexity of an underlying mathematical problem (believed to be hard to solve), such as factoring large numbers or computing discrete logarithms.

In 1985, Neal Koblitz and Victor Miller introduced the use of elliptic curves in cryptography with a new mathematical hard problem: the elliptic curve discrete logarithm. This problem is believed to be extremely hard, much harder than the analogous ones defined before. Due to the high difficulty of computing the discrete logarithm problem in elliptic curves over finite fields, the same security provided by other cryptosystems can be achieved with smaller fields, hence shorter key lengths. The benefit of such particularity is well appreciated when memory and processing power is limited, such as in smart cards and other small devices.

High performance implementations of elliptic curve cryptosystems depend heavily on the efficiency of the arithmetic operations needed for the elliptic curve operations. This fact leads us to focus our efforts on the field operations (multiplications and squarings, etc.) of such cryptosystems.

In this thesis, we concentrate on improving algorithms for number theoretic cryptosystems. Our work is mainly focused on implementing elliptic curve cryptosystems efficiently, which require space and time-efficient implementations of arithmetic operations over finite fields.

We introduce new prime numbers and methods for fast arithmetic operations over finite fields. We choose to work on special primes of the form $2^k + 2^i + 1$ ($k, i \in \mathbb{N}$, $k > i$), mainly because the Montgomery multiplication and exponentiation can be significantly accelerated.

Since the efficiency of elliptic curve cryptosystems inherently depends on the efficiency of multiplication, the main issue is then to manage multiplication efficiently, which is usually followed by a reduction. Therefore, in this thesis, we mainly focused on multiplication algorithms.

Chapter 2 section 2.1 through section 2.3 provides general information about security services and different types of cryptosystems including public-key and private-key cryptosystems. Common cryptographic terminology is also introduced in these sections.

Chapter 2 section 2.4 and section 2.5 explore different mathematically hard problems on which public-key cryptosystems are based. Strong and weak points of the systems are also presented in order to compare the level of security they provide. Then the elliptic curve logarithm problem is compared to the discrete logarithm problem and the integer factorization problem.

Chapter 2 section 2.6 includes the definitions related to the elliptic curves. It mainly describes how a set of points on an elliptic curve form an Abelian group, and how to manage the group operations on this set. It also explains the addition operation formula for elliptic curves and provides an introduction to

arithmetic operations in finite fields. Finally the elliptic curve digital signature algorithm scheme is discussed.

Chapter 2 section 3.1 through section 3.3 give the background necessary to understand the motivation and the contents of the contributions made, which are presented in Chapters 3 section 3.4 through 3.10. Results of some previous implementations are also summarized.

In Chapter 3 section 3.5 through section 3.9, we present a set of five optimized algorithms that are used in elliptic curve cryptosystems. New methods for performing modular reduction using special primes are described. This method can be used to obtain fast software implementations of the finite field multiplication and squaring operations.

In Chapter 3 section 3.10, we show the impact of our new optimized algorithms on the the elliptic curve digital signature.

Chapter 4 concludes the thesis with the summary of the results, contributions and discussions.

In Appendix A and B, we provide a list of low-weight special primes of the form $2^k + 2^i + 1$ and $2^k + 2^i - 1$ for $64 \leq k \leq 511$, $64 \leq i \leq k-1$.

Chapter 2

CRYPTOGRAPHIC SYSTEMS

2.1 Information Security

Information security describes all measures taken to prevent unauthorized use of electronic data – whether this unauthorized use takes the form of disclosure, alteration, substitution, or destruction of the data concerned. Information security can be classified as the provision of the following three services:

- **Confidentiality** – concealment of data from unauthorized parties.
- **Integrity** – assurance that data is genuine.
- **Availability** – the system still functions efficiently after security provisions are in place (services are not denied).

Cryptographic systems are used to offer the services listed above. Broadly speaking, a cryptographic system transforms electronic data to a modified form. Depending on the security services required, the assurance may be that the data cannot be altered without detection, or it may be that the data is unintelligible to all but authorized parties. Cryptographic systems are controlled by the use of a key to determine the transformation performed. The key itself also takes the form of an electronic string. The owner of the cryptographic key must continue to ensure the security of the information by guarding the key itself.

In order to clarify and demonstrate how cryptographic systems (cryptosystems) are employed, confidentiality and integrity are further sub-classified into five services that can be thought of as the building blocks of a secure system:

- **Confidentiality** – concealment of data from unauthorized parties.
- **User Authentication** – assurance that the parties involved in a real-time transaction are who they say they are.
- **Data Origin Authentication** – assurance of the source of a message.
- **Data Integrity** – assurance that the data has not been modified by unauthorized parties.
- **Non-repudiation** – the binding of an entity to a transaction in which it participates, so that the transaction cannot later be repudiated. That is, the receiver of a transaction is able to demonstrate to a neutral third party that the claimed sender did indeed send the transaction.

Different security services use different types of cryptographic systems. In the next section, we will describe how public-key and private-key cryptosystems are used to provide these security services.

2.2 Types of Cryptographic Systems

Preparing a message for a secure, private transfer involves the process of encryption. Encryption transforms data in user or machine readable form, called the plaintext, to an illegible version, called the ciphertext. The conversion of plaintext to ciphertext is controlled by an electronic key \mathbf{k} . The key is simply a binary string which determines the effect of the encryption function. The reverse process of transforming the ciphertext back into plaintext is called decryption, and is controlled by a related key \mathbf{i} .

There are two broad classes of cryptosystems, known as symmetric-key cryptosystems and public-key cryptosystems. The relationship between \mathbf{k} and \mathbf{i} differentiates the two.

2.2.1 Symmetric Key Cryptosystems

In a symmetric-key cryptosystem, the same key is used for both encryption and decryption. Since the keys are the same, two users wishing to communicate in confidence must agree and maintain a common secret key. They can do this by physically meeting, but this can be impractical or sometimes even impossible, or they might use the services of a trusted courier.

Let \mathcal{M} denote the set of all possible plaintext messages, \mathcal{C} the set of all possible ciphertext messages and \mathcal{K} the set of all possible keys.

A *private key cryptosystem* consists of a family of pairs of functions

$$E_k : \mathcal{M} \longrightarrow \mathcal{C} \quad \text{and} \quad D_k : \mathcal{C} \longrightarrow \mathcal{M} \quad k \in \mathcal{K} ,$$

such that

$$D_k(E_k(m)) = m \quad \text{for all } m \in \mathcal{M} \text{ and } k \in \mathcal{K} .$$

The main disadvantages of the private key cryptosystems are:

1. Key distribution problem (a secure channel may not be available)
2. Key management problem (if the number of pairs is large then the number of keys becomes unmanageable)
3. No signatures possible

Some common examples of symmetric-key systems include:

- DES [18]
- IDEA [18]
- RC5 [18]

2.2.2 Public Key Cryptosystems

In public-key cryptosystems (introduced as recently as 1976 by Whitfield Diffie and Martin Hellman, the abilities to perform encryption and decryption are separated. The encryption rule employs a *public-key* e (that is $k=e$), while the decryption rule requires a different (but mathematically related) *private-key* d (that is $i=d$). Knowledge of the public-key allows encryption of plaintext but does not allow decryption of the ciphertext. And in terms of an arbitrary group it can be described as:

1. (Setup) A and B publicly select a (multiplicatively written) finite group G and an element $\alpha \in G$.
2. A generates a random integer a , computes α^a in G , and transmits α^a to B over a public communications channel.
3. B generates a random integer b , computes α^b in G , and transmits α^b to A over a public communications channel.
4. A receives α^b and computes $(\alpha^b)^a$.
5. B receives α^a and computes $(\alpha^a)^b$.

A and B now share the common group element α^{ab} . Note that an eavesdropper knows G, α, α^a and α^b , and his task is to use this information to reconstruct α^{ab} . This problem is commonly referred to as *Diffie-Hellman problem*. The problem of computing a , given G, α and α^a is called the *discrete logarithm problem*. Some common examples of public-key systems include:

- ECC [6, 12]
- ElGamal [3]
- RSA [16]

2.3 Security Services using Public-Key Cryptosystems

Public-key cryptosystems are capable of fulfilling all of the main objectives of information security. This section outlines how each of these services can be provided. For illustrative purposes, each service is discussed in the context of a hypothetical communication between two users, Alice and Bob. Bob's private key will be denoted by e_{bob} and his public key by d_{bob} .

2.3.1 Public-Key Encryption – Confidentiality

Suppose Alice wishes to send a secret message to Bob. During system set-up, Bob makes e_{bob} , his public key, available to all users by publishing it in a public directory, the electronic equivalent of a phone book. To communicate message M to Bob, Alice first looks up e_{bob} in the public directory. Alice then encrypts M by performing the public-key transformation using $E_{e_{bob}}$, to transform M into ciphertext C . This process is denoted by:

$$C = E_{e_{bob}}(M)$$

Finally Alice sends C to Bob. Bob retrieves M by transforming C using $D_{d_{bob}}$ by computing:

$$M = D_{d_{bob}}(C)$$

2.3.2 Digital Signatures – Data Integrity, Non-repudiation

Digital signatures are the equivalent of traditional handwritten signatures. Electronic signatures cannot be formed by simply appending a fixed string to a message since this would make it easily forgeable. To avoid compromise in this way, digital signatures are performed in a more complex manner using a public-key cryptosystem. The essential difference between the use of a public-key cryptosystem for signing and its use for encrypting is that the order in which the keys are used is reversed. In data encryption first Alice applied $E_{e_{bob}}$ to M ,

then Bob decrypted using $D_{d_{bob}}$. In digital signatures, first Bob applies $E_{d_{bob}}$ to compute his signature, then Alice checks, or *verifies*, the signature using $D_{e_{bob}}$.

Suppose now that Bob wishes to sign a message M . Bob first transforms M using a *hash function*. The output of the hash function is a value which is specific to the content of the message itself. This output, denoted $h(M)$, is called a *message digest* and can be thought of as a "fingerprint" of the message. Bob signs M by transforming $h(M)$ using $E_{d_{bob}}$ to obtain:

$$S = E_{d_{bob}}(h(M))$$

Bob now sends M and S to Alice as his signature on M . If Alice wants to *verify* Bob's signature on M , she first retrieves e_{bob} . Then she recomputes the message digest, $h(M)$, from M using the publicly available hash function. Finally, Alice transforms S using $D_{e_{bob}}$ and compares the result with $h(M)$. If Alice finds that:

$$D_{e_{bob}}(S) = h(M)$$

then she accepts Bob's signature as valid. Otherwise Alice concludes that S is not Bob's signature for that message M which has been modified.

This signature process provides the services of data origin authentication, data integrity, and non-repudiation since changing M would change S , and Bob is the only one who can transform the message to obtain S .

2.4 Hard Mathematical Problem

All cryptographic systems rely on the difficulty of a mathematical problem for their security. A mathematical problem is said to be difficult if the fastest algorithm to solve the problem takes a long time relative to the input size. An algorithm runs quickly relative to the size of its input if it is a polynomial time algorithm, and slowly if it is an exponential time algorithm.

Thus, when looking for a mathematical problem on which to base a public-key cryptographic system, cryptographers are searching for a problem for which

the fastest algorithm takes exponential time. Today, only three types of systems should be considered both secure and efficient. The systems, classified according to the mathematical problem on which they are based, are:

- Integer Factorization System (RSA)
- Discrete Logarithm Systems (DSA)
- Elliptic Curve Discrete Logarithm Systems (ECC)

In the following sections, we will describe the underlying problems these systems are based on.

2.4.1 Integer Factorization Problem

The first cryptographic system, called RSA, relies on the difficulty of the Integer Factorization Problem. The problem is defined as follow:

Given an integer n , which is the product of two large primes, determine these factors, i.e., find primes p and q such that $p \cdot q = n$ (e.g., $3 \cdot 5 = 15$).

An RSA public-key consists of a pair (n, e) , where e is a number between 1 and $n - 1$, and n is the product of two large primes. To provide short term security n should be at least 500 bits. RSA can be used for both encryption and digital signatures by performing modular arithmetic. *Modular addition* and *modular multiplication* modulo n works just like ordinary addition and multiplication, except that the answer is reduced to its remainder on division by n (e.g., $3 \cdot 5 = 1 \pmod{7}$).

Modular arithmetic plays a central role in the implementation of all three types of public-key cryptosystems. When RSA is used as an encryption scheme or as a digital signature scheme, exponentiation modulo n must be performed. Suppose m , a number between 0 and $n - 1$, represents a message. Then the modular exponentiation, $m^e \pmod{n}$, must be calculated for some number e (private-key) when m is transformed. This modular exponentiation dominates the time required to perform RSA.

2.4.2 Discrete Logarithm Problem

Another mathematical problem defined in terms of modular arithmetic is the *discrete logarithm problem* modulo a prime p . Fix a number p , then given an integer g between 0 and $p - 1$ we have the following relation between g and y :

$$y = g^x \pmod{p}$$

for some x . The discrete logarithm problem modulo p is to determine the integer x for a given pair g and y .

Like the integer factorization problem, no efficient algorithm is known to solve the discrete logarithm problem modulo p .

2.4.3 Elliptic Curve Discrete Logarithm Problem

In 1985, Neil Koblitz [6] and Victor Miller [12] independently proposed the Elliptic Curve Cryptosystem (ECC), whose security rests on the discrete logarithm problem over the points on an elliptic curve. ECC can be used to provide both a digital signature scheme and encryption scheme.

An elliptic curve, defined modulo a prime p , is the set of solutions (x, y) to an equation of the form:

$$y^2 = x^3 + ax + b \pmod{p}$$

for two numbers a and b . If (x, y) satisfies the equation then $P = (x, y)$ is a *point* on the elliptic curve. Addition of two points can be defined on the curve. Suppose P and Q are both points on the curve, then

$$P + Q$$

will always be another point on the curve. And the elliptic curve discrete logarithm problem can be defined as follows:

Given two points P and Q on the curve such that Q equals $x.P$ ($Q = x.P$), for some x , determine x (where the value $x.P$ represents the point P added to itself x times).

As for the integer factorization problem and the discrete logarithm problem modulo p , no fast (polynomial time) algorithm is known to solve the elliptic curve discrete logarithm problem. The existing algorithms require fully exponential time.

2.5 Comparison of Public-Key Cryptosystems

When examining the theoretical security of a public-key cryptosystem, breaking the system requires solving the underlying mathematical problem. The question we are trying to answer in this section is the following: Which is, of the three, the hardest problem? The integer factorization problem, the discrete logarithm problem modulo p , or the elliptic curve discrete logarithm problem.

As a concrete example, Table 2.1 and 2.2 [1] compares the time required to break ECC with the time required to break RSA or DSA for various modulus size. The values are computed in MIPS years (a MIPS year represents a computing time of one year on a machine capable of performing one million instructions per second).

Table 2.1. Computing power required to compute elliptic curve logarithm

Field size (bits)	MIPS years
163	$9.6 * 10^{11}$
191	$7.9 * 10^{15}$
239	$1.6 * 10^{23}$
359	$1.5 * 10^{41}$
431	$1.0 * 10^{52}$

To achieve reasonable security, RSA and DSA should employ a 1024-bit modulus, while a 160-bit modulus should be sufficient for ECC. The security gap between the systems grows as the key size increases (300-bit ECC is a great deal more secure than 2000-bit RSA).

Table 2.2. Computing power required to factor integers

Size of n (bits)	MIPS years
512	3×10^4
768	2×10^8
1024	3×10^{11}
1536	3×10^{16}
2048	3×10^{20}

Table 2.3, 2.4, and 2.5 show the efficiency of the different systems in terms of key size (bits required to store key pairs and system parameters) and bandwidth (bits that must be communicated to transfer encrypted message) in bits.

Table 2.3. Size of system parameters and key pair

	System parameters	Public-Key	Private Key
RSA	n/a	1088	2048
DSA	2208	1024	160
ECC	481	161	160

It is clear from Table 2.3 that the system parameters and key pairs are shorter for ECC than for either RSA or DSA

Table 2.4. Signature size for 2000-bit messages

	Signature size
RSA	1024
DSA	320
ECC	320

From Table 2.4 and 2.5, we can conclude that ECC offers considerable bandwidth savings over the other types of public-key cryptographic systems when being used to transform short messages.

Table 2.5. Encrypted message size for 100-bit messages

	Encrypted message size
RSA	1024
DSA	2048
ECC	321

2.6 Elliptic Curves and Cryptography

2.6.1 Mathematical Background

We begin by introducing some basic mathematical terminology. A *group* is an abstract mathematical object consisting of a set G together with an operation $*$ defined on pairs of elements of G . The operation has the following properties:

- *closure* – $a * b \in G$ for all $a, b \in G$
- *associativity* – $a * (b * c) = (a * b) * c$ for all $a, b, c \in G$
- *existence of identity* – $\exists e \in G$ such that $e * a = a * e = a$ for all $a \in G$
- *existence of inverse* – $\forall a \in G \exists b \in G$ such that $a * b = b * a = e, b = a^{-1}$

A group G is said to be *abelian* if $\forall a, b \in G, a * b = b * a$. The order of a group is the number of elements in G . For example, the integers modulo n , namely $Z_n = 0, 1, 2, \dots, n - 1$, forms a group of order n under the operation of addition modulo n . The additive identity of this group is 0. If p is a prime number, then the non-zero elements of Z_p , namely $Z_p^* = 1, 2, \dots, p - 1$, forms a group of order $p - 1$ under the operation of multiplication modulo p . The multiplicative identity of this group is 1. The order of a group element $g \in G$ is the least positive integer n such that $g^n \pmod{p} = 1$. For example, in the group Z_{11}^* , the element $g = 3$ has order 5.

2.6.2 Background in Elliptic Curve

We now give a brief introduction to the theory of elliptic curves. For more details, consult [6, 12]. In this thesis, we restrict our discussion to the elliptic curves over Z_p . An elliptic curve E over Z_p is defined by an equation of the form

$$y^2 = x^3 + ax + b, \quad (2.1)$$

where $a, b \in Z_p$, and $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$, together with a special point \mathcal{O} , called the *point at infinity*. The set $E(Z_p)$ consists of all points (x, y) , $x \in Z_p, y \in Z_p$, which satisfies the defining equation 2.1, together with \mathcal{O} .

An addition operation of two points on an elliptic curve $E(Z_p)$ can be defined to give a third elliptic curve point. Together with this addition operation, the sets of points $E(Z_p)$ forms a group with \mathcal{O} serving as identity. It is this group that is used in elliptic curve cryptosystems. The identity \mathcal{O} is such that

$$P + \mathcal{O} = \mathcal{O} + P = P,$$

for all $P \in E(Z_p)$. If $P = (x, y) \in E(Z_p)$, then $(x, y) + (x, -y) = \mathcal{O}$. The point $(x, -y)$ is denoted $-P$, and is called the *negative* of P ; observe that $-P$ is indeed a point on the curve. Let $P = (x_1, y_1) \in E(Z_p)$ and $Q = (x_2, y_2) \in E(Z_p)$, where $P \neq -Q$. Then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2$$

and

$$y_3 = \lambda(x_1 - x_3) - y_1,$$

where

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q, \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P = Q. \end{cases}$$

Observe that the addition of two elliptic curve points $E(Z_p)$ requires a few arithmetic operations (addition, subtraction, multiplication, and inversion) in the underlying finite field Z_p .

For historical reasons, the group operation for an elliptic curve $E(Z_p)$ has been called *addition*. By contrast, the group operation in Z_p^* is *multiplication*. Table 2.6 shows the correspondence between notation used for the two groups Z_p^* and $E(Z_p)$.

Table 2.6. Correspondence between Z_p^* and $E(Z_p)$

Group	Z_p^*	$E(Z_p)$
Group elements	integers $1, 2, \dots, p - 1$	Points (x, y) on E plus \mathcal{O}
Group operation	multiplication modulo p	addition of points
Notation	Elements: g, h Multiplication: $g * h$ Inverse: g^{-1} Division: g/h Exponentiation: g^a	Elements: P, Q Addition: $P + Q$ Negative: $-P$ Subtraction: $P - Q$ Multiple: aP
DLP	Given $g \in Z_p^*$ and $h = g^a \text{ mod } p$, find a	Given $P \in E(Z_p)$ and $Q = aP$, find a

2.6.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is composed of three steps:

- key generation
- signature generation
- signature verification

These three procedures are described in the following sections.

2.6.4 ECDSA Key Generation

Each entity does the following:

- Select an elliptic curve $E(Z_p)$ with $\#E(Z_p)$ divisible by a large prime n .

- Select a point $P \in E(\mathbb{Z}_p)$ of order n
- Select an unpredictable integer d in $[1, n - 1]$.
- Compute $Q = d.P$
- Public key is (E, P, n, Q) ; private key is d .

2.6.5 ECDSA signature Generation

To sign a message m , Bob does the following:

- Select an unpredictable k in $[1, n - 1]$.
- Compute $k.P = (x_1, y_1)$ and $r = x_1 \bmod n$ (if $r = 0$ go to step 1).
- Compute $k^{-1} \bmod n$.
- Compute $s = k^{-1}(h(m) + d.r) \bmod n$, where h is the Secure Hash Algorithm (SHA-1)
- If $s = 0$ go to step 1.
- The signature for the message m is the pair of integers (r, s) .

2.6.6 ECDSA signature Verification

To verify a signature (r, s) , an entity does the following:

- Obtain an authentic copy of Bob's public key (E, P, n, Q) .
- Verify that r and s are in the interval $[1, n - 1]$.
- Compute $w = s^{-1} \pmod n$ and $h(m)$
- Compute $u_1 = h(m).w \pmod n$ and $u_2 = r.w \pmod n$
- Compute $u_1P + u_2Q = (x_0, y_0)$ and $v = x_0 \pmod n$
- Accept the signature if and only if $v = r$.

Chapter 3

IMPROVING MONTGOMERY ALGORITHMS

3.1 Introduction

The arithmetic operations in Galois field $GF(p)$ have several applications in cryptography. Examples of cryptographic applications are elliptic curve cryptosystems (Koblitz 1994), over the field $GF(p)$, and the Diffie-Hellman key exchange algorithm based on the discrete exponentiation (Diffie and Hellman 1976). Both applications require the implementation of modular multiplication which can be written efficiently using the Montgomery method [13].

In order to achieve improved performance, several different methods have been proposed [9]. However the proposed algorithms are not taking advantage of the binary structure of the field modulus.

In this thesis, we optimize the five different Montgomery multiplication algorithms (SOS, CIOS, FIOS, FIPS, and CIHS) studied in [9] using special primes of the form $2^k + 2^i + 1$ ($k, i \in \mathbb{N}$, $k > i$). We have performed a thorough analysis of the optimized versions of different Montgomery methods and compared them to the original methods. We show that the new algorithms are significantly faster. Some timing results of the algorithms together with digital signature (ECDSA) timings are given in Section 3.10.

3.2 Previous Work

The introduction of the Montgomery multiplication [13] has been one of the most interesting and useful advances in modular arithmetic. This method is used to compute fast multiplication by replacing division by a modulus n with division by a power of 2.

Several versions of the Montgomery multiplication algorithm were proposed [9] in an attempt to modify the original method in order to obtain more efficient software implementations. These algorithms differ on the way multiplication and reduction are combined.

Some special classes of elliptic curves defined over $GF(p)$, where p is a special prime, also allow efficient implementations. A reduction algorithm, for modulus of the special form $m = b^t \pm c$ (where m is a t -digit base b positive integer and c an l -digit base b positive integer), is given in [11].

In [4], a technique for accelerating arithmetic, based on a modulus of the form $e2^a \pm 1$ is presented ($e2^a$ within a single word). This method is suitable for both hardware and software implementations.

Generalized Mersenne primes are another type of special modulus ($2^{k_n \cdot W} \pm 2^{k_{n-1} \cdot W} \pm \dots \pm 1$, where $k_n > k_{n-1} > \dots$) which can be exploited for fast reduction. In this case, the residue can be expressed as a sum or difference (mod m) of small number of terms [17].

From the software implementation point of view of elliptic curves over $GF(p)$, no papers have discussed methods for improving the Montgomery multiplication using special fields. Montgomery multiplication is used with random curves over arbitrary prime fields. However, the formula given for implementing the field multiplication is not taking advantage of the modulus pattern.

The remainder of the thesis is organized as follows. In Section 3, we present a short introduction to Montgomery multiplication in $GF(p)$. The proposed algorithms are described and analyzed in sections 4 thru 9. Some running times of the improved algorithms are presented in Section 10. The overall impact on the Elliptic Curve Digital Signature is also given.

3.3 Montgomery Multiplication

Here, we present a brief introduction to Montgomery multiplication; more information on Montgomery multiplication can be found in [13, 8, 9].

The Montgomery multiplication is an efficient way for computing the modular product of two integers x and y as follows:

$$z = x.y \pmod{n}, \quad (3.2)$$

where x , y and n are k -bit binary numbers, i.e., $2^{k-1} \leq x, y, n < 2^k$. The algorithm is used to speed up modular multiplications and squarings required during the modular exponentiation process. It is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery algorithm produces the resulting k -bit number z in (3.2) without performing a division by the modulus n . The Montgomery algorithm computes

$$\text{MontMult}(x, y) = x.y.r^{-1} \pmod{n} \quad (3.3)$$

given $x, y < n$ and r such that $\text{gcd}(r, n) = 1$. The algorithm works for any r which is relatively prime to n . However, some additional efficiency is obtained when r is taken to be a power of 2 (r is chosen to be 2^k). If n is odd, the requirement $\text{gcd}(n, r) = 1$ is satisfied.

To summarize the basic idea behind the Montgomery multiplication algorithm, we first define the n -residue of an integer $x < n$ as

$$\bar{x} = x.r \pmod{n}. \quad (3.4)$$

It is straightforward to show that the set

$$\{ x.r \pmod{n} \mid 0 \leq x \leq n-1 \} \quad (3.5)$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus there is one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set.

The Montgomery multiplication algorithm exploits this property by introducing a much faster multiplication routine which computes the n -residue of the

product of two integers whose n -residue is given. Given two n -residue \bar{x} and \bar{y} , the Montgomery product is defined as the n -residue

$$\bar{z} = \bar{x}.\bar{y}.r^{-1} \pmod{n}, \quad (3.6)$$

where r^{-1} is the inverse of r modulo n , i.e., it is the number with the property

$$r^{-1}.r = 1 \pmod{n}.$$

The resulting number \bar{z} in (3.6) is indeed the n -residue of the product

$$z = x.y \pmod{n},$$

since

$$\begin{aligned} \bar{z} &= \bar{x}.\bar{y}.r^{-1} \pmod{n} \\ &= x.r.y.r^{-1} \pmod{n} \\ &= x.y.r \pmod{n}. \end{aligned}$$

An additional quantity n' is needed to describe the Montgomery multiplication algorithm. It is the integer with the property

$$r.r^{-1} - n.n' = 1 .$$

The integers n' and r can both be computed by the Extended Euclidean algorithm [5]. The computation of the Montgomery product is given as follows:

function *MontMult*(\bar{x}, \bar{y})

Step 1. $t := \bar{x}.\bar{y}$

Step 2. $u := (t + (t.n' \pmod{r}).n)/r$

Step 3. **if** $u \geq n$ **then return** $u - n$ **else return** u

An important feature of the Montgomery multiplication algorithm is that it does involve multiplication modulo r and division by r rather than division by n .

Since preprocessing and postprocessing operations (converting from residue to n -residue, computing n' , and converting the result back to ordinary residue) are rather time-consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus n are needed. Such is the case for modular exponentiation, i.e., computation of $x^e \pmod n$. Using the binary method for computing the powers [5], we replace the exponentiation operation by a series of squaring and multiplication operations modulo n . Let $(e_{k-1}, e_{k-2}, \dots, e_0)$ be the binary expansion of the exponent e . The exponentiation algorithm computes $z = x^e \pmod n$ with $O(k)$ calls to the Montgomery multiplication algorithm. Step 4 of the modular exponentiation algorithm computes y using \bar{y} via the property of the Montgomery algorithm: $\text{MontMult}(\bar{y}, 1) = \bar{y} \cdot 1 \cdot r^{-1} = y \cdot r \cdot r^{-1} = y \pmod n$.

function $\text{ModExp}(x, e, n)$

Step 1. $\bar{x} = x \cdot r \pmod n$

Step 2. $\bar{y} = 1 \cdot r \pmod n$

Step 3. for $i = k - 1$ downto 0

Step 4. $\bar{y} = \mathbf{MontMult}(\bar{y}, \bar{y})$

Step 5. **if** $e_i = 1$ **then** $\bar{y} = \mathbf{MontMult}(\bar{y}, \bar{x})$

Step 6. **return** $y := \mathbf{MontMult}(\bar{y}, 1)$

Thus, we are converting the ordinary residue x into its n -residue \bar{x} using a division-like operation. This latter operation can be achieved by a series of shift and subtract operations. Additionally, Steps 2 and 3 require divisions. However, once the preprocessing is complete, the inner loop of the binary exponentiation method uses the Montgomery product operation which performs only

multiplications modulo 2^k and division by 2^k . The binary method produces the n -residue \bar{z} of the quantity $z = x^e \pmod{n}$. Then, the ordinary residue is obtained from the n -residue by executing the `MontMult` function with arguments \bar{y} and 1. This is easily shown to be correct, since

$$\bar{y} = y.r \pmod{n}$$

immediately implies that

$$\begin{aligned} y &= \bar{y}.r^{-1} \pmod{n} \\ &= \bar{y}.1.r^{-1} \pmod{n} \\ &= \text{MontMult}(\bar{y}, 1). \end{aligned}$$

The resulting algorithm is quite fast and efficient, as has been demonstrated by many researchers and engineers. An implementation example can be found in [2].

In typical implementations, operations on large numbers are performed by breaking the numbers into words. If w is the word size of the computer, then a number can be thought of as a sequence of integers each represented in radix $W = 2^w$. If these "multi-precision" numbers require s words in the radix W representation, then we take r as $r = 2^{sw}$. However, the above algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. The paper by Dussé and Kaliski in [2] describes improved algorithms, including a simple and efficient way for computing n' . The modular exponentiation algorithm is used in cryptography. Examples are the ElGamal signature scheme [3], and the Digital Signature Standard (DSS) of the National Institute for Standards and Technology [15].

In the following sections, we will present several improved algorithms for performing Montgomery multiplication `MontMult`(x, y) [9] and analyze the impact of our special primes with respect to their time requirements. The time

analysis is performed by counting the total number of multiplications, additions (subtractions and shifts: \ll), and memory read and write operations in terms of the input size parameter s . For example the following operation

$$(C, S) := t[i + j] + a[j] * b[i] + C$$

is assumed to require three memory reads, two additions, and one multiplication since most microprocessors multiply two one-word numbers, leaving the two-word result in one or two registers. Similarly the following operation

$$(C, S) := t[i] + m \ll \alpha$$

is assumed to require two memory reads and two additions.

Multi-precision integers are assumed to reside in memory throughout the computations. Therefore, the assignment operation performed within a routine corresponds to the read or write operation between a register and memory . They are counted to calculate the proportion of the memory access time in the total running time of the Montgomery multiplication algorithm. In our analysis, loop or conditional construct establishment and index computations are not taken into account. The only registers we assume are available are those to hold the carry C and the sum S as above (or equivalently borrow and difference for subtraction). Obviously, in many microprocessors there will be more registers, but this gives a first-order approximation of the running time, sufficient for a general comparison of the algorithms.

3.4 Optimizing Montgomery Product using Special Primes

Several families of special primes have already been exploited in order to speed up modular arithmetic. In [4], the authors take advantage of the elliptic curve prime modulus $p = e2^a \pm 1$, where the leading coefficients are within a word – to implement fast arithmetic. In [11], a fast reduction method is described using prime modulus of the form $b^t \pm c$, where c is an l -digit base b positive integer

(for some $l < t$). Another family special prime modulus, called *generalized Mersenne numbers*, provides more efficient modular multiplication. *Mersenne primes* are of the form $2^{6^4 k_{t-1}} \pm 2^{6^4 k_{t-2}} \pm \dots \pm 2^{6^4 k_0} \pm 1$, where k_t 's are usually very short integers (less than ten).

In this thesis, we are introducing a new family of special primes which are primes of the form $2^k + 2^i \pm 1$ (for some integers k and i where $k > i$). In the sequel, we will analyze and optimize Montgomery algorithms based on a subfamily of primes p ,

$$2^k + 2^i + 1,$$

$k, i \in N, k > i$. The advantage of such primes is that they have only 3 bits set to 1: the most significant bit, the least significant bit and a middle bit. Hence, instead of processing all words in the modulus as in standard modular multiplication, we process only the words containing the bits set to 1. In any case, we will be processing only three words (actually only two words because the least significant word will be processed in the initial settings of the algorithm).

The organization of the algorithms we will analyze is based on two factors. The first factor is whether multiplication and reduction are *separated* or *integrated*. In the separated approach, we first multiply x and y , then perform a Montgomery reduction. This integration can be either *coarse-grained* or *fine-grained*, depending on how often we switch between multiplication and reduction (specifically, after processing an array of words, or just one word).

The second factor is the general form of the multiplication and reduction steps. One form is the *operand scanning*, where an outer loop moves through words of one of the operands; another form is the *product scanning*, where the loop moves through words of the product itself. It is possible for multiplication to have one form and reduction to have the other form, even in the integrated approach.

In this thesis, we will focus on the optimization of five main implementations of the Montgomery multiplication (implemented in C). These methods have

been described and analyzed previously in [9]. In the sequel, we will further analyze and enhance them. These algorithms are the following:

- Separated Operand Scanning (SOS) (Section 5)
- Coarsely Integrated Operand Scanning (CIOS) (Section 6)
- Finely Integrated Operand Scanning (FIOS) (Section 7)
- Finely Integrated Product Scanning (FIPS) (Section 8)
- Coarsely Integrated Hybrid Scanning (CIHS) (Section 9)

In the next sections, we will optimize these methods using special primes of the form $2^k + 2^i + 1$.

3.5 Separated Operand Scanning (SOS) Method

In this section, we first briefly analyze the SOS algorithm. Following this, we study the different optimization points of the algorithm using prime modulus of the form $2^k + 2^i + 1$. In the last subsection, we give a theoretical speed up of the new optimized algorithm for different field sizes.

3.5.1 SOS Method with Arbitrary Primes

The first method to be optimized in this thesis, for computing $\text{MontMult}(x, y)$, is what is called the Separated Operand Scanning method (see [9]). In this method, we first compute the product and then we reduce it. The complete separation of the product and the reduction computation makes this method suitable for special prime optimization. The product is performed with two embedded loops. Each loop goes through the index of one of the operands and the inner j -loop updates the final product array t by adding the appropriate word product and carry. The result $x.y$ is computed as follows:

```

for i=0 to s-1
  C := 0
  for j=0 to s-1
    (C,S) := t[i+j] + x[j]*y[i] + C
    t[i+j] := S
  t[i+s] := C

```

The final value is the $2s$ -word integer t which is initially assumed to be zero. The next step is to nullify the lower half of the product residing in t . To do so, we add the product $m.n$ to t , where $m = t.n' \bmod r$. The code below updates t in order to compute $t + m.n$.

```

for i=0 to s-1
  C := 0
  m := t[i]*n'[0] mod W
  for j=0 to s-1
    (C,S) := t[i+j] + m*n[j] + C
    t[i+j] := S
  ADD(t[i+s],C)

```

Since $m = t.n' \bmod r$ and the reduction process proceeds word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n' . This observation was first made in [2], and applies to all five methods optimized in this thesis. The `ADD` function shown above performs a carry propagation, adding C to the input array given by the first argument. Starting from the first element ($t[i+s]$), it then propagates the carry until no further carry is generated. The computed value of t is then divided by r which is realized by simply ignoring the lower s words of t . The following code describes this division.

```

for j=0 to s
  u[j] := t[j+s]

```

Finally, the multi-precision subtraction in Step 3 of MontMult is computed (for details see [9]). Step 3 is performed the same way for all algorithms analyzed in this thesis, and requires $2s + 2$ additions, memory reads and memory writes

The value n'_0 , which is defined as the inverse of the least significant word of n modulo 2^w , i.e., $n'_0 = -n_0^{-1} \pmod{2^w}$, can be computed using a simple algorithm described in [2].

The total number of operations is calculated by counting each operation within a loop, and multiplying this number by the iteration count. Table 3.1 below gives a breakdown of the operations performed in the SOS algorithm. It requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 3$ reads, and $2s^2 + 7s + 3$ writes. The ADD function is counted as two memory reads, two additions, and two memory writes in this analysis [9].

Table 3.1. SOS method operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[j] + x[i]*y[j] + C	1	2	3	0	s ²
t[i+j] := S	0	0	0	1	s ²
t[i+s] := C	0	0	0	0	s
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
m := t[i]*n'[0] mod W	1	0	2	1	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[i+j] + m*n[j] + C	1	2	3	0	s ²
t[i+j] := S	0	0	0	1	s ²
ADD(t[i+s],C)	0	2	2	2	s
for i=0 to s	-	-	-	-	-
t[j] := t[j+s]	0	0	1	1	s + 1
Final Subtraction	0	2(s + 1)	2(s + 1)	2(s + 1)	1
	2s² + s	4s² + 4s + 2	6s² + 7s + 3	2s² + 7s + 3	

3.5.2 Optimized SOS Method with Special Primes

Let the modulus p be $2^{k_1*w+a} + 2^{k_0*w+b} + 1$, where w is the word size of the machine architecture and k_1, k_0, a and b integers ($k_1 = s - 1$, where s is the size in words of the modulus p , $0 \leq a, b \leq w - 1$). The binary expansion of p is $(1_{k_1}, 0, \dots, 0, 1_{k_0}, 0, \dots, 0, 1)$. This decomposition of the field prime p will be used thru the remaining of this thesis.

Since the multiplication part of the Montgomery product does not involve the modulus p , the optimization will focus on the reduction part. The original algorithm computes

$$m = t[i] * n'[0] \text{ mod } W. \quad (3.7)$$

Since $n'[0] = -n[0]^{-1} \text{ (mod } 2^w)$, we can save this multiplication by judiciously choosing $n'[0]$. By choosing $n[0] = 1$, we obtain $n'[0] = -1$. This slightly restricts the range of possible modulus p ($k_0 > 0$). Hence (3.7) becomes

$$\begin{aligned} m &= -t[i] \text{ mod } W \\ m &= \sim t[i] + 1, \end{aligned}$$

where $\sim t[i]$ is the bit complement of $t[i]$.

The next optimization has to do with loop unrolling. By unrolling the inner j -loop, we save the processing time required to compute the reduction steps for the words of the modulus that are equal to zero. Indeed, if a given word of the modulus p is set to zero, then the final product t remains unchanged when that particular word is involved. In addition, the carry of each word level product is not propagated anymore. Therefore the propagation of the carry must be performed after each reduction step. This latter step is carried out by the **ADD** function described in this previous section. Since the modulus has only three non-zero words to process, the inner j -loop of the reduction part becomes:

```

(C,S) := t[i] + m*n[0] + C           j = 0
t[i] := S
ADD(t[i+1],C)
(C,S) := t[i + k0] + m*n[k0]       j = k0
t[i + k0] := S
ADD(t[i + k0 + 1],C)
(C,S) := t[i + k1] + m*n[k1]       j = k1
t[i + k1] := S
ADD(t[i + k1 + 1],C)

```

And since $n[0] = 1$, $n[k_0] = 2^b$ and $n[k_1] = 2^a$, the unrolled code above becomes

```

(C,S) := t[i] + m                       j = 0
t[i] := S
ADD(t[i+1],C)
(C,S) := t[i + k0] + m<<b           j = k0
t[i + k0] := S
ADD(t[i + k0 + 1],C)
(C,S) := t[i + k1] + m<<a           j = k1
t[i + k1] := S
ADD(t[i + k1 + 1],C)

```

Table 3.2 gives a breakdown of the operations performed in the optimized SOS algorithm. It requires s^2 multiplications, $2s^2+12s+2$ additions, $3s^2+13s+3$ reads, and $s^2 + 12s + 3$ writes.

In the table above and in the following sections, we consider the substraction operation as an addition. The shift operation (\ll) is also counted as an addition.

Table 3.2. Optimized SOS method operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[j] + x[i]*y[j] + C	1	2	3	0	s ²
t[i+j] := S	0	0	0	1	s ²
t[i+s] := C	0	0	0	1	s
for i=0 to s-1	-	-	-	-	-
m := t̃[i] + 1	0	1	1	0	s
ADD(t[i+1],1)	0	1	1	1	s
(C, S) := t[i + k ₀] + m ≪ b	0	2	2	0	s
t[i + k ₀] := S	0	0	0	1	s
ADD(t[i + k ₀ + 1],C)	0	2	2	2	s
(C, S) := t[i + k ₁] + m ≪ a	0	2	2	0	s
t[i + k ₁] := S	0	0	0	1	s
ADD(t[i + k ₁ + 1],C)	0	2	2	2	s
for i=0 to s	-	-	-	-	-
t[j] := t[j+s]	0	0	1	1	s + 1
Final Subtraction	0	2(s + 1)	2(s + 1)	2(s + 1)	1
	s ²	2s ² + 12s + 2	3s ² + 13s + 3	s ² + 11s + 3	

The ADD function, when the carry C is equal to one, is counted as one memory read, one addition, and one memory write.

3.5.3 SOS Speed Improvement with Special Primes

In this subsection, we give the theoretical speed improvement for each operation (multiplication, addition, memory read and write). Since we are only processing three words of the modulus p , the larger the modulus, the better the improvement. We are obtaining over 50% of speed improvement for the multiplication operation whatever the field size is. The next significant saving is in the memory read operation (starting from 24% with 160 bits). The other operations start contributing in higher field size (192 bits and up). Table 3.3 details theoretical gain.

Table 3.3. SOS speed improvement breakdown

Field Size	Operations			
	Multiplications	Additions	Memory Reads	Memory Writes
$p=160$ bits ($s=5$)	54.5%	8.2%	24.0%	5.7%
$p=192$ bits ($s=6$)	53.8%	14.1%	27.6%	10.3%
$p=256$ bits ($s=8$)	53.0%	22.1%	32.5%	17.1%
$p=512$ bits ($s=16$)	51.5%	35.2%	40.7%	30.6%
$p=1024$ bits ($s=32$)	50.8%	42.4%	45.2%	39.4%
...
$p \rightarrow \infty$ ($s \rightarrow \infty$)	50%	50%	50%	50%

3.6 Coarsely Integrated Operand Scanning (CIOS) Method

The next method, the Coarsely Integrated Operand Scanning method, integrates the multiplication and reduction steps.

3.6.1 CIOS Method with Arbitrary Primes

Instead of computing the entire product $x.y$, then reducing, we alternate multiplication and reduction between the outer loops. We can do this since the value of m in the i th iteration of the outer loop for reduction depends only on the value of $\mathfrak{t}[i]$, which is completely computed by the i th iteration of the outer loop of the multiplication. Even though the multiplication and reduction steps are not completely separate, this algorithm can still be optimized efficiently. This is due to the fact that the value of m depends on the value of $\mathfrak{t}[i]$ only. The algorithm is detailed in the table 3.4.

The last j -loop, the reduction loop, integrates the shifting of the result one word to the right (i.e., division by 2^w), hence the reference to $\mathfrak{t}[j]$ and $\mathfrak{t}[0]$ instead of $\mathfrak{t}[i+j]$ and $\mathfrak{t}[i]$.

Table 3.4. CIOS method operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[j] + y[j]*x[i] + C	1	2	3	0	s ²
t[j] := S	0	0	0	1	s ²
(C,S) := t[s] + C	0	1	1	0	s
t[s] := S	0	0	0	1	s
t[s+1] := C	0	0	0	1	s
m := t[0]*n'[0] mod W	1	0	2	1	s
(C,S) := t[0] + m*n[0]	1	1	3	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + m*n[j] + C	1	2	3	0	s(s-1)
t[j-1] := S	0	0	0	1	s(s-1)
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	2s ² + s	4s ² + 4s + 2	6s ² + 7s + 2	2s ² + 5s + 2	

3.6.2 Optimized CIOS Method with Special Primes

In the CIOS algorithm, m is computed as $m = t[0] * n'[0] \bmod W$, and after improvement, becomes $m = \tilde{t}[0] + 1$. Since the shifting to the right is integrated in the reduction step, unrolling the loop would not be very advantageous. By moving the shifting to the right into the multiplication loop, we can unroll the reduction j -loop. And the new multiplication loop becomes

```

for j=0 to s-1
    (C,S) := t[j] + x[i]*y[j] + C
    [j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1]+C

```

The reduction step depends on the value of m . Since m is a function of $t[0]$, we need to unroll the multiplication loop for $j = 0$. This gives us the following algorithm.

```

for i=0 to s-1
  C := 0
  (C,S) := t[0] + y[0]*x[i]
  m := ~t[0] + 1
  for j=0 to s-1
    (C,S) := t[j] + b[j]*a[i] + C
    [j-1] := S
  (C,S) := t[s] + C
  t[s-1] := S
  t[s] := t[s+1] + C

```

The reduction step is similar to the SOS reduction.

```

S := 0; C := 1                                j = 0
ADD(t[0],C)
(C,S) := t[k0 - 1] + m<<b                    j = k0 - 1
t[k0 - 1] := S
ADD(t[k0],C)
(C,S) := t[k1 - 1] + m<<a                    j = k1 - 1
t[k1 - 1] := S
ADD(t[k1],C)

```

By moving the processing of the first reduction step ($\text{ADD}(t[0], 1)$) into the multiplication loop ($C := C+1$), we can further optimize this method. This can be done since the carry C is propagated starting from the same index of the product array t . The final optimized algorithm with its operation breakdown is given in table 3.5.

Table 3.5. Optimized CIOS operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
(C,S) := t[0] + y[0]*x[i]	1	1	3	0	s
m := t[0] + 1	0	1	1	1	s
C := C + 1	0	1	0	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + y[j]*x[i] + C	1	2	3	0	s(s-1)
t[j-1] := S	0	0	0	1	s(s-1)
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
(C,S) := t[k ₀ - 1] + m \ll b	0	2	2	0	s
t[k ₀ - 1] := S	0	0	0	1	s
ADD(t[k ₀],C)	0	2	2	2	s
(C,S) := t[k ₁ - 1] + m \ll a	0	2	2	0	s
t[k ₁ - 1] := S	0	0	0	1	s
ADD(t[k ₁],C)	0	2	2	2	s
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	s^2	$2s^2 + 13s + 2$	$3s^2 + 13s + 2$	$s^2 + 10s + 2$	

3.6.3 CIOS Speed Improvement with Special Primes

The multiplication operation shows again the best performance – over 50 percent of reduction. This is actually the case for all five algorithms optimized in this thesis. The number of memory reads are also significantly reduced – starting from 24 percent of improvement at 160 bits and up to 45 percent at 1024 bits. The other operation reductions can be observed at higher field size values (192 bits and up). The maximum theoretical improvement is 50 percent for all operations. Results are detailed in Table 3.6

Table 3.6. CIOS speed improvement breakdown

Field Size	Operations			
	Multiplications	Additions	Memory Reads	Memory Writes
$p=160$ bits ($s=5$)	54.5%	4.1%	24.1%	0%
$p=192$ bits ($s=6$)	53.8%	10.6%	27.7%	5.8%
$p=256$ bits ($s=8$)	53.0%	19.3%	32.6%	14.1%
$p=512$ bits ($s=16$)	51.5%	33.8%	40.7%	29.6%
$p=1024$ bits ($s=32$)	50.8%	41.6%	45.2%	39.1%
...
$p \rightarrow \infty$ ($s \rightarrow \infty$)	50%	50%	50%	50%

3.7 Finely Integrated Operand Scanning (FIOS) Method

This method integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop.

3.7.1 FIOS Method with Arbitrary Primes

In this case $\mathfrak{t}[0]$ must be computed before entering the loop since m depends on this value which corresponds to unrolling the first iteration of the loop for $j = 0$.

The difference between the CIOS method and this method is that the FIOS method has only one inner loop. The use of the `ADD` function is required in the inner j -loop since there are two distinct carries, one arising from the multiplication and the other from the reduction. This method requires about s^2 more additions, writes, and reads than for the CIOS method. Table 3.7 describes the algorithm and its corresponding operation breakdown.

3.7.2 Optimized FIOS Method with Special Primes

The preliminary code (computing m) processed before entering the inner j -loop, is written as follows:

Table 3.7. FIOS method operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
(C,S) := t[0] + x[0]*y[i]	1	1	3	0	s
ADD(t[1],C)	0	2	2	2	s
m := S*n'[0] mod W	1	0	1	1	s
(C,S) := S + m*n[0]	1	1	2	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + b[j]*a[i] + C	1	2	3	0	s(s-1)
ADD(t[j+1],C)	0	2	2	2	s(s-1)
(C,S) := S + m*n[j]	1	1	2	0	s(s-1)
t[j-1] := S	0	0	0	1	s(s-1)
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	$2s^2 + s$	$5s^2 + 3s + 2$	$7s^2 + 5s + 2$	$3s^2 + 4s + 2$	

```

for i=0 to s-1
    (C,S) := t[0] + y[0]*x[i]
    ADD(t[1],C)
    m := S*n'[0] mod W
    (C,S) := S + m*n[0]

```

We can optimize this code by taking advantage of the pattern of the modulus array n . After optimization, the above code becomes:

```

for i=0 to s-1
    (C,S) := t[0] + y[0]*x[i]
    ADD(t[1],C)
    m := ~S + 1
    C := C + 1

```

Since the multiplication and reduction steps are finely integrated, the optimization required some conditional constructs to be added. The reduction line of the original code

$$(C, S) := S + m * n[j]$$

is transformed into

```

if (j == k0) then
    (C, S) := S + m ≪ b
if (j == k1) then
    (C, S) := S + m ≪ a

```

Table 3.8 details the optimized FIOS algorithm.

Table 3.8. Optimized FIOS operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
(C,S) := t[0] + x[0]*y[i]	1	1	3	0	s
m := $\tilde{S} + 1$	0	1	0	1	s
C := C + 1	0	1	0	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + y[j]*x[i] + C	1	2	3	0	s(s-1)
ADD(t[j+1],C)	0	2	2	2	s(s-1)
if (j == k ₀) then	-	-	-	-	s
(C, S) := S + m ≪ b	0	2	1	0	s
if (j == k ₁) then	-	-	-	-	s
(C, S) := S + m ≪ a	0	2	1	0	s
t[j-1] := S	0	0	0	1	s(s-1)
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	s^2	$4s^2 + 6s + 2$	$5s^2 + 4s + 2$	$3s^2 + 2s + 2$	

3.7.3 FIOS Speed Improvement with Special Primes

In the optimized FIOS method, the number of writes remains almost identical. From the three algorithms optimized so far, the FIOS method is the least improved one. As for the SOS and CIOS methods, multiplications are best reduced, followed by memory reads. The number of additions and memory writes are not significantly changed. Table 3.9 shows the improvement.

Table 3.9. FIOS speed improvement

Field Size	Operations			
	Multiplications	Additions	Memory Reads	Memory Writes
p=160 bits (s=5)	54.5%	3.5%	27.3%	10.3%
p=192 bits (s=6)	53.8%	6.0%	27.5%	9.0%
p=256 bits (s=8)	53.0%	9.2%	27.8%	7.1%
p=512 bits (s=16)	51.5%	14.4%	28.2%	3.8%
p=1024 bits (s=32)	50.8%	17.2%	28.4%	2.0%
...
p → ∞ (s → ∞)	50%	20%	28.6%	0%

3.8 Finely Integrated Product Scanning (FIPS) Method

Like the previous method, the FIPS method interleaves the computations of the product and the reduction. However, in this case, both computations are in the product-scanning form.

3.8.1 FIPS Method with Arbitrary Primes

The first outer loop computes the product and reduces it. The three-word array acc , i.e., $acc[0]$, $acc[1]$, $acc[2]$, is used as the partial product accumulator for the product $x.y$ and $m.n$. The use of a three-word array assumes that $s < 2^w$ (see [9]). The algorithm can be easily modified to handle a larger accumulator. In this loop, the i th word of m is computed using n'_0 and stored in $t[i]$, and

then the least significant word of $m.n$ is added to t . Since the least significant word of t always becomes zero, the shifting can be carried out one word at a time in each iteration. The second i -loop completes the computation by forming the final result in the memory space of t . The most significant bit of the result remains in $\text{acc}[0]$ (the values $\text{acc}[1]$ and $\text{acc}[2]$ are zero at the end). The complete algorithm is described in Table 3.10.

Table 3.10. FIPS algorithm operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for $i=0$ to $s-1$	-	-	-	-	-
for $j=0$ to $i-1$	-	-	-	-	-
$(C,S) := \text{acc}[0] + x[j]*y[i-j]$	1	1	3	0	$s(s-1)/2$
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	$s(s-1)/2$
$(C,S) := S + t[j]*n[i-j]$	1	1	2	0	$s(s-1)/2$
$\text{acc}[0] := S$	0	0	0	1	$s(s-1)/2$
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	$s(s-1)/2$
$(C,S) := \text{acc}[0] + x[i]*y[0]$	1	1	3	0	s
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	s
$t[i] := S*n'[0] \bmod W$	1	0	1	1	s
$(C,S) := S + t[i]*n[0]$	1	1	2	0	s
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	s
$\text{acc}[0] := \text{acc}[1]$	0	0	1	1	s
$\text{acc}[1] := \text{acc}[2]$	0	0	1	1	s
$\text{acc}[2] := 0$	0	0	0	1	s
for $i=s$ to $2s-1$	-	-	-	-	-
for $j=i-s+1$ to $s-1$	-	-	-	-	-
$(C,S) := \text{acc}[0] + x[j]*y[i-j]$	1	1	3	0	$s(s-1)/2$
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	$s(s-1)/2$
$(C,S) := S + t[j]*n[i-j]$	1	1	2	0	$s(s-1)/2$
$\text{acc}[0] := S$	0	0	0	1	$s(s-1)/2$
$\text{ADD}(\text{acc}[1],C)$	0	2	2	2	$s(s-1)/2$
$t[i-s] := \text{acc}[0]$	0	0	1	1	s
$\text{acc}[0] := \text{acc}[1]$	0	0	1	1	s
$\text{acc}[1] := \text{acc}[2]$	0	0	1	1	s
$\text{acc}[2] := 0$	0	0	0	1	s
Final Subtraction	0	$2(s+1)$	$2(s+1)$	$2(s+1)$	1
	$2s^2 + s$	$6s^2 + 2s + 2$	$9s^2 + 8s + 2$	$5s^2 + 9s + 2$	

3.8.2 Optimized FIPS Method for Special Primes

Since both i -loops compute partly the reduction, some conditional constructs need to be added in both loops for optimization. Before optimization the multiplication and reduction code is as follows:

```

for j=0 to i-1
    (C,S) := acc[0] + x[j]*y[i-j]
    ADD(acc[1],C)
    (C,S) := S + t[j]*n[i-j]
    ADD(acc[1],C)
    acc[0] := S

```

After optimization, the multiplication and reduction code becomes:

```

for j=0 to i-1
    (C,S) := acc[0] + x[j]*y[i-j]
    ADD(acc[1],C)
    if ((i - j) == k0) then
        (C,S) := S + t[j] ≪ b
        ADD(acc[1],C)
    if ((i - j) == k1) then
        (C,S) := S + t[j] ≪ a
        ADD(acc[1],C)
    acc[0] := S

```

A similar optimization can be carried out in the second i -loop. In addition, the line of code $t[i] := S * n'[0] \bmod W$, is optimized as

$$t[i] = \tilde{S} + 1.$$

Table 3.11 gives the complete optimized algorithm.

Table 3.11. Optimized FIPS algorithm operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
for j=1 to i-1	-	-	-	-	-
(C,S) := acc[0] + x[j]*y[i-j]	1	1	3	0	$s(s-1)/2$
ADD(acc[1],C)	0	2	2	2	$s(s-1)/2$
if ((i - j) == k ₀) then	-	-	-	-	-
(C,S) := S + t[j] << b	0	2	1	0	s
ADD(acc[1],C)	0	2	2	2	s
if ((i - j) == k ₁) then	-	-	-	-	-
(C,S) := S + t[j] << a	0	2	1	0	s
ADD(acc[1],C)	0	2	2	2	s
acc[0] := S	0	0	0	1	$s(s-1)/2$
(C,S) := acc[0] + x[i]*y[0]	1	1	3	0	s
ADD(acc[1],C)	0	2	2	2	s
t[i] := S + 1	0	1	0	1	s
ADD(acc[1],1)	0	1	1	1	s
acc[0] := acc[1]	0	0	1	1	s
acc[1] := acc[2]	0	0	1	1	s
acc[2] := 0	0	0	0	1	s
for i=s to 2s-1	-	-	-	-	-
for j=i-s+1 to s-1	-	-	-	-	-
(C,S) := acc[0] + x[j]*y[i-j]	1	1	3	0	$s(s-1)/2$
ADD(acc[1],C)	0	2	2	2	$s(s-1)/2$
if ((i - j) == k ₀) then	-	-	-	-	-
(C,S) := S + t[j] << b	0	2	1	0	s
ADD(acc[1],C)	0	2	2	2	s
if ((i - j) == k ₁) then	-	-	-	-	-
(C,S) := S + t[j] << a	0	2	1	0	s
ADD(acc[1],C)	0	2	2	2	s
acc[0] := S	0	0	0	1	$s(s-1)/2$
t[i-s] := acc[0]	0	0	1	1	s
acc[0] := acc[1]	0	0	1	1	s
acc[1] := acc[2]	0	0	1	1	s
acc[2] := 0	0	0	0	1	s
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	s^2	$3s^2 + 20s + 2$	$5s^2 + 20s + 2$	$3s^2 + 18s + 2$	

3.8.3 FIPS Speed Improvement with Special Primes

In the FIPS algorithm, the speed improvement is quite considerable too. However, we must not forget that some conditional constructs have been added. Table 3.12 details theoretical results.

Table 3.12. FIPS speed improvement

Field Size	Operations			
	Multiplications	Additions	Memory Reads	Memory Writes
p=160 bits (s=5)	54.5%	-9.3%	15.0%	2.9%
p=192 bits (s=6)	53.8%	0%	19.2%	7.6%
p=256 bits (s=8)	53.0%	11.9%	24.9%	14.2%
p=512 bits (s=16)	51.5%	30.6%	34.2%	25.8%
p=1024 bits (s=32)	50.8%	40.2%	39.2%	32.5%
...
p → ∞ (s → ∞)	50%	50%	44%	40%

3.9 Coarsely Integrated Hybrid Scanning (CIHS) Method

This method is a modified version of the SOS algorithm, illustrating yet another approach to the Montgomery multiplication. This method uses less temporary space than the SOS method without changing the general flow of the algorithm. It is called "hybrid scanning" because it mixes the product-scanning and operand-scanning forms of multiplication while the reduction remains in the operand-scanning form.

3.9.1 CIHS Method with Arbitrary Primes

The computation of $x.y$ is split into two loops. The computation of the first half of the product is sufficient to process the reduction. The second i -loop alternates operand-scanning reduction and multiplication. The splitting is possible because the computation of m requires only the i th word of the product. Thus the multiplication of $x.y$ can be simplified by postponing the computation of the most significant half to the second i -loop. Table 3.13 below gives the detailed algorithm.

Table 3.13. CIHS operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-i-1	-	-	-	-	-
(C,S) := t[i+j] + x[j]*y[i] + C	1	2	3	0	s(s+1)/2
t[i+j] := S	0	0	0	1	s(s+1)/2
(C,S) := t[s] + C	0	1	1	0	s
t[s] := S	0	0	0	1	s
t[s+1] := C	0	0	0	1	s
for i=0 to s-1	-	-	-	-	-
m := t[0]*n'[0] mod W	1	0	2	1	s
(C,S) := t[0] + m*n[0]	1	1	3	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + m*n[j] + C	1	2	3	0	s(s-1)
t[j-1] := S	0	0	0	1	s(s-1)
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
t[s+1] := 0	0	0	0	1	s
for j=i+1 to s-1	-	-	-	-	-
(C,S) := t[s-1] + y[j]*x[s-j+i]	1	1	3	0	s(s-1)/2
t[s-1] := S	0	0	0	1	s(s-1)/2
(C,S) := t[s] + C	0	1	1	0	s(s-1)/2
t[s] := S	0	0	0	1	s(s-1)/2
t[s+1] := C	0	0	0	1	s(s-1)/2
Final Subtraction	0	2(s+1)	2(s+1)	2(s+1)	1
	$2s^2 + s$	$4s^2 + 4s + 2$	$6.5s^2 + 6.5s + 2$	$3s^2 + 6s + 2$	

3.9.2 Optimized CIHS Method for Special Primes

In this algorithm, the optimization takes place in the second j -loop. The computation of m within the second i -loop is performed as follows:

$$m := t[0]*n'[0] \bmod W$$

$$(C,S) := t[0] + m*n[0]$$

Here, we also take advantage of the modulus pattern to save two multiplication operations. The optimized code becomes $m := \tilde{t}[0] + 1; C := 1$.

The reduction j -loop

```

for j=1 to s-1
    (C,S) := t[j] + m*n[j] + C
    t[j-1] := S

```

is optimized as

```

for j=1 to s-1
    ADD(t[j],C)
    if (j == k0) then
        (C,S) := t[k0] + m ≪ b
    if (j == k1) then
        (C,S) := t[k1] + m ≪ a
    t[j-1] := S

```

Table 3.14 outlines the optimized breakdown of the CIHS method.

3.9.3 CIHS Speed Improvement with Special Primes

With this method, we end up having more memory writes in the optimized version than the original. However, the number of multiplications, additions and memory reads go down significantly, which gives us an interesting overall optimization. Theoretical results are given in Table 3.15

3.10 Montgomery Algorithm and ECDSA Timings

In this section, we show timings for the five Montgomery multiplication method (SOS, CIOS, FIOS, FIPS and CIHS) and compare them to their optimized versions. We also show the impact of the CIOS method on the Elliptic Curve Digital Signature (ECDSA). The computation is based on C source codes com-

Table 3.14. Optimized CIHS operation breakdown

STATEMENT	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-i-1	-	-	-	-	-
(C,S) := t[i+j] + x[j]*y[i] + C	1	2	3	0	$s(s+1)/2$
t[i+j] := S	0	0	0	1	$s(s+1)/2$
(C,S) := t[s] + C	0	1	1	0	s
t[s] := S	0	0	0	1	s
t[s+1] := C	0	0	0	1	s
for i=0 to s-1	-	-	-	-	-
m := t[0] + 1	0	1	1	1	s
C := 1	0	0	0	0	s
for j=1 to s-1	-	-	-	-	-
ADD(t[j],C)	0	1	1	1	$s(s-1)$
if (j == k ₀) then	-	-	-	-	s-1
(C,S) := t[k ₀] + m ≪ b	0	2	2	0	s-1
if (j == k ₁) then	-	-	-	-	s-1
(C,S) := t[k ₁] + m ≪ a	0	2	2	0	s-1
t[j-1] := S	0	0	0	1	$s(s-1)$
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
t[s+1] := 0	0	0	0	1	s
for j=i+1 to s-1	-	-	-	-	-
(C,S) := t[s-1] + y[j]*x[s-j+i]	1	1	3	0	$s(s-1)/2$
t[s-1] := S	0	0	0	1	$s(s-1)/2$
(C,S) := t[s] + C	0	1	1	0	$s(s-1)/2$
t[s] := S	0	0	0	1	$s(s-1)/2$
t[s+1] := C	0	0	0	1	$s(s-1)/2$
Final Subtraction	0	$2(s+1)$	$2(s+1)$	$2(s+1)$	1
	s^2	$3s^2 + 9s - 2$	$4.5s^2 + 8.5s - 2$	$4s^2 + 5s + 2$	

piled with Microsoft Visual C++ 5.0. An Intel Pentium II 32-bit microprocessor running at 450 Mhz was used to conduct the benchmarking.

To compute the timings, we choose curves E defined over the finite field $GF(p) : E : y^2 = x^3 + Ax + B$, where $A, B \in GF(p)$ with $4A^3 + 27B^2 \neq 0 \pmod{p}$. For the mathematical background, see [6]. While A and B are randomly chosen, the values of p are judiciously selected as described in table 3.16 (for a listing of low-weight special primes see Appendix A and B).

Table 3.15. CIHS speed improvement

Field Size	Operations			
	Multiplications	Additions	Memory Reads	Memory Writes
$p=160$ bits ($s=5$)	54.5%	3.3%	22.3%	-18.7%
$p=192$ bits ($s=6$)	53.8%	5.9%	23.3%	-20.5%
$p=256$ bits ($s=8$)	53.0%	9.7%	24.7%	-23.1%
$p=512$ bits ($s=16$)	51.5%	16.5%	27.3%	-27.7%
$p=1024$ bits ($s=32$)	50.8%	20.5%	29.0%	-30.4%
...
$p \rightarrow \infty$ ($s \rightarrow \infty$)	50%	25%	31%	-33%

Table 3.16. Special primes used for timings

Field Size	p
160 bits	$2^{159} + 2^{107} + 1$
192 bits	$2^{191} + 2^{111} + 1$
256 bits	$2^{255} + 2^{166} + 1$
512 bits	$2^{511} + 2^{322} + 1$
1024 bits	$2^{1023} + 2^{249} + 1$

3.10.1 Running Times for Montgomery Algorithms

The tables in the following subsections show the actual performance of our optimized Montgomery algorithms. We are comparing our new approaches of field multiplication and modular reduction using special primes to the classical methods.

3.10.1.1 *SOS Method*

For the SOS method, the timing results show about 28 percent of improvement with 160-bit fields and close to 50 percent with 1024-bit fields (see Table 3.17). This method shows the best improvement among all five algorithms.

Table 3.17. SOS timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	6.9 μ s	5 μ s	28%
192 bits	9.6 μ s	6.1 μ s	37%
256 bits	15.8 μ s	9.8 μ s	38%
512 bits	57.2 μ s	30.5 μ s	47%
1024 bits	216.5 μ s	106.5 μ s	51%

3.10.1.2 *CIOS Method*

For the CIOS method, the timing results show from 30 percent (160-bit field) to close to 50 percent (1024-bit field) improvement (see Table 3.18). Even though this method doesn't give the best improvement, it is still the fastest of all five methods.

Table 3.18. CIOS timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	5.6 μ s	3.9 μ s	30%
192 bits	7.9 μ s	5.4 μ s	32%
256 bits	13.4 μ s	8.7 μ s	35%
512 bits	47.3 μ s	26.9 μ s	43.1%
1024 bits	190 μ s	99 μ s	48%

3.10.1.3 *FIOS Method*

With the FIOS method, the performance deteriorates. The results (Table 3.19) show very little improvement for 160-bit (4 percent) field size and is even slower than the original algorithm for larger field size (-15 percent for 1024 bit). This is due to conditional constructs added in the optimized version.

Table 3.19. FIOS timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	5.2 μ s	5 μ s	4%
192 bits	7.3 μ s	6.9 μ s	5.5%
256 bits	12.3 μ s	13.2 μ s	-7.5%
512 bits	43.6 μ s	49 μ s	-12%
1024 bits	173 μ s	199 μ s	-15%

3.10.1.4 *FIPS Method*

The FIPS method shows from 20 percent to 38 percent improvement in the 160-1024 bit range (see Table 3.20).

Table 3.20. FIPS timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	6.9 μ s	5.5 μ s	20%
192 bits	9.5 μ s	7.6 μ s	20%
256 bits	16.5 μ s	12.2 μ s	26%
512 bits	62.4 μ s	40 μ s	36%
1024 bits	248 μ s	153 μ s	38%

3.10.1.5 *CIHS Method*

The CIHS method shows the least performance improvement ranging from 4 to 7.5 percent (see Table 3.21).

3.10.2 Running Times for ECDSA Using CIOS Method

We replaced the standard Montgomery algorithm by our optimized CIOS method, and observed the impact of our work on the Elliptic Curve Digital Signature.

Table 3.21. CIHS timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	$6.5\mu s$	$6.2\mu s$	4%
192 bits	$9.4\mu s$	$9.0\mu s$	4%
256 bits	$15.7\mu s$	$15.0\mu s$	4.5%
512 bits	$56.3\mu s$	$52.3\mu s$	7%
1024 bits	$213\mu s$	$197\mu s$	7.5%

Table 3.22 shows a comparison of the running time for the Elliptic Curve Digital Signature. The fastest multiplication scheme, the CIOS method was used for the signature timing. The impact on the signature ranges from 15 percent for 160-bit prime field to nearly 40 percent for 1024-bit prime field. The improvement then stagnates around 42-43 percent for higher field sizes.

Table 3.22. ECDSA timings and improvement

Field Size	Classical Method	Improved Method	Percentage Improvement
160 bits	$16.65ms$	$14.15ms$	15%
192 bits	$24.87ms$	$20.65ms$	17%
256 bits	$52.35ms$	$40.82ms$	22%
512 bits	$321ms$	$219ms$	32%
1024 bits	$2289ms$	$1401ms$	39%

Chapter 4

CONCLUSIONS

This chapter summarizes the results of this thesis, lists the most significant contributions, and finally discusses future research directions in this area.

4.1 Discussion of Results

In this thesis we studied the optimization of Montgomery algorithms using special primes of the form $2^k + 2^i + 1$. The optimized algorithms presented in this thesis reduce by half the required number of multiplications in the Montgomery product. The fastest method (CIOS) requires s^2 multiplications, $2s^2 + 13s + 2$ additions, $3s^2 + 13s + 2$ memory reads and $s^2 + 10s + 2$ writes.

Chapter 3 is based on the paper [9]. In this chapter, we proposed a new reduction method for performing the standard and Montgomery multiplication operations in $GF(k)$. The proposed method yields word-level algorithms, enabling software implementations for finite field arithmetic operations which find applications most notably in elliptic curve cryptography. We analyzed the algorithms in details and their complexity in terms of the number of basic arithmetic operations. The proposed algorithms (optimized SOS, CIOS, FIPS, and CIHS) are more efficient than the previously published results in terms of the number of operations (multiplications, additions, memory reads, and writes) used. In the FIOS case, we obtain better performance for low field sizes only.

In order to measure the actual performance of these algorithms, we have implemented them in C on an Intel Pentium II 450 Mhz system. Tables in Section 3.10 summarize the timings of these methods for different field sizes (160, 192, 256, 512 and 1024 bits). The timings given are the average values

over several thousand executions. Table 3.22 summarizes the average signature (ECDSA) times over a thousand executions. By judiciously choosing the Montgomery multiplication algorithm, we can achieve from over 30 percent to over 50 percent of improvement in the modular multiplication and from over 15 percent to close to 40 percent of speed up in the Elliptic Curve Digital Signature.

As analyzed in [9], CIOS was operating faster on the selected Intel Pentium-60 Linux system compared to the other Montgomery multiplication algorithms. However, on our Intel Pentium II platform, FIOS happened to be preferable in the non-optimized cases. This is mainly due to the hardware architecture (caching system, memory accesses) which makes the looping constructs more efficient in the FIOS method. In the optimized cases, the CIOS performs faster since the conditional constructs added in the FIOS method slow it down significantly and make it the slowest of all five optimized methods.

On a general-purpose processor, the optimized CIOS is probably best, as it is the simplest of all five methods, requires fewer operations than the other four methods, and can be used to obtain fast software implementation of the exponentiation over the field $GF(p)$.

4.2 Summary of Contributions

Below is the summary of the contributions made:

- A new set of special primes suitable for fast field arithmetic.
- A new word-level optimized reduction scheme for integer arithmetic.
- Five optimized versions of Montgomery multiplication algorithms.
- Detailed analyses of all methods proposed
- Fast elliptic curve digital signature implementation

4.3 Future Work

This work can be continued with a focus on primes of the form $2^k + 2^i - 1$. This would give us an alternative to the algorithms we studied in the event that we could not find a prime of the form $2^k + 2^i + 1$ for a given field size. From our experimental results (see Appendices A and B), we can notice that for any field size (verified for primes up to 1024 bits) there exists at least one prime of the form $2^k + 2^i \pm 1$. Even though the optimized algorithms for the set $2^k + 2^i - 1$ versions can yield more complex and less efficient algorithms than the set studied in this thesis, their frequency might make them more interesting. Future research with this alternative set may bring up totally different optimization points. Another subfamily of special primes than could be of interest is the $2^k + 2^j + 2^i + 1$ subfamily ($k, j, i \in N, k > j > i$). To extend this research on special primes, we could also focus on the bit pattern of primes of the form $e2^a + 1$, where e is within a machine word (one multiplication operation for $e2^a + 1$ primes instead of two shift operations for $2^k + 2^i + 1$ primes). Interesting comparisons between the $2^k + 2^i + 1$ and $e2^a + 1$ sets could be carried out.

BIBLIOGRAPHY

- [1] Certicom. White papers on elliptic curve cryptography. <http://www.certicom.com>, 2000.

- [2] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In H. Imai and Y. Zheng, editors, *Advances in Cryptology – EUROCRYPT 90, Lecture Notes in Computer Science, No. 473*, pages 230–244, New York, NY: Springer-Verlag, 1990.

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

- [4] T. Hasegawa, J. Nakajima, and M. Matsui. A practical implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer. *First International Workshop on Practice and Theory in Public Key Cryptography*, Lecture Notes in Computer Science, No. 1431, pages 182–194. New York, NY: Springer-Verlag, 1998.

- [5] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, Volume 2. Reading, MA: Addison-Wesley, 1998.

- [6] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

- [7] Ç. K. Koç. Montgomery reduction with even modulus. *IEE Proceedings: Computers and Digital Techniques*, 141(5):314–316, September 1994.

- [8] Ç. K. Koç and T. Acar. Montgomery multiplication in $GF(2^k)$. *Design, Codes and Cryptography*, 14(1):57–69, April 1998.

- [9] Ç. K. Koç, T. Acar and B. S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
- [10] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. *Cryptographic Hardware and Embedded Systems – CHES’99, Lecture Notes in Computer Science, No 1717*, pages 316–327, Worcester, MA: Springer-Verlag, 1999.
- [11] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.
- [12] V. Miller. Uses of elliptic curves in cryptography. *Advances in Cryptology – CRYPTO 85, Lecture Notes in Computer Science, No. 218*, pages 417–426, New York, NY: Springer-Verlag, 1986.
- [13] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
- [14] M. Musa and Ç. K. Koç. Improved Montgomery Algorithms using Special Primes and Impact on Elliptic Curve Digital Signature. *To be submitted*, June 2000.
- [15] National Institute for Standards and Technology. Digital Signature Standard (DSS). *Federal Register*, 56–169, August 1991.
- [16] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [17] J. A. Solinas. Efficient Implementation of Koblitz Curves and Generalized Mersenne Arithmetic. *ECC ’99*, November 1999.
- [18] W. Stallings. *Cryptography and Network Security*. New Jersey, NJ: Prentice-Hall, 1998.

APPENDICES

Appendix A

Table of Low-Weight Special Primes: $p = 2^k + 2^i + 1$

Field Size	(k,i) tuples
64	-
65	-
66	(65,47)
67	(66,59),(66,63)
68	(67,37),(67,54),(67,66)
69	(68,41),(68,65)
70	(69,34),(69,36),(69,43),(69,44),(69,55),(69,58)
71	(70,67)
72	(71,66)
73	-
74	(73,36),(73,39),(73,52),(73,55)
75	(74,57)
76	(75,55),(75,61),(75,66)
77	(76,33)
78	(77,51),(77,75)
79	(78,51)
80	(79,34),(79,57)
81	(80,33)
82	(81,39),(81,44),(81,52)
83	(82,75)
84	(83,38),(83,53)
85	(84,81)
86	(85,39),(85,42),(85,52)
87	(86,35),(86,71)
88	(87,59),(87,67),(87,73),(87,85)
89	-
90	-
91	(90,41),(90,57),(90,65),(90,75)
92	(91,34),(91,57),(91,61),(91,66),(91,70),(91,85)
93	(92,81)
94	(93,55),(93,67),(93,79)
95	(94,81)
96	(95,39),(95,63)
97	-
98	(97,55)
99	(98,33),(98,41),(98,65),(98,93)
100	(99,66),(99,93)
101	(100,49)
102	(101,44),(101,62)

Field Size	(k,i) tuples (cont'd)
103	-
104	(103,90)
105	-
106	(105,34),(105,36),(105,52),(105,54),(105,62)
107	(106,69),(106,79),(106,91)
108	(107,42),(107,62),(107,74),(107,77),(107,89),(107,90)
109	(108,33)
110	-
111	(110,83),(110,89)
112	(111,90)
113	-
114	(113,42),(113,60),(113,63),(113,66),(113,80),(113,84),(113,90)
115	(114,55),(114,69),(114,77),(114,79),(114,95)
116	(115,79)
117	-
118	(117,43),(117,51)
119	(118,91),(118,111)
120	(119,50),(119,66),(119,107),(119,110)
121	-
122	(121,79),(121,100)
123	(122,53)
124	(123,89),(123,118)
125	(124,61)
126	(125,66),(125,102),(125,116)
127	(126,39),(126,67),(126,85),(126,109)
128	(127,93),(127,102),(127,105),(127,106)
129	-
130	(129,86),(129,127)
131	(130,51),(130,99)
132	(131,42),(131,47),(131,87),(131,89)
133	(132,45),(132,81)
134	(133,60)
135	-
136	(135,54),(135,75),(135,102),(135,126),(135,129)
137	(136,121)
138	(137,36),(137,50),(137,54),(137,92),(137,114),(137,134)
139	(138,85),(138,109),
140	(139,66)
141	-
142	(141,36),(141,44),(141,51),(141,62)
143	(142,45)
144	(143,38),(143,87),(143,95),(143,114),(143,126)
145	(144,65)
146	-
147	(146,59),(146,65),(146,129)
148	(147,41),(147,61),(147,77),(147,90),(147,93),(147,95),(147,101),(147,129)
149	(148,33)
150	(149,47),(149,78),(149,119),(149,138)
151	(150,67),(150,79),(150,99),(150,135)
152	(151,34),(151,81),(151,90),(151,109),(151,127),(151,147)
153	-

Field Size	(k,i) tuples (cont'd)
154	(153,34),(153,40),(153,44),(153,62),(153,70),(153,75),(153,82),(153,139)
155	(154,45),(154,69),(154,87),(154,99),(154,117)
156	(155,89),(155,122)
157	(156,133)
158	(157,60),(157,79),(157,151)
159	(158,45),(158,143)
160	(159,59),(159,63),(159,107),(159,135)
161	-
162	(161,36),(161,72),(161,92),(161,99),(161,122)
163	(162,93),(162,145),(162,147),(162,153)
164	(163,82),(163,106),(163,150)
165	(164,81)
166	(165,111),(165,151),(165,162)
167	(166,63),(166,69),(166,75),(166,117),(166,141)
168	(167,86),(167,90),(167,111)
169	(168,105)
170	(169,90)
171	(170,83),(170,89)
172	(171,141),(171,154)
173	(172,81),(172,117)
174	(173,102),(173,155)
175	(174,43),(174,57),(174,115),(174,151),(174,165)
176	(175,34),(175,51),(175,57),(175,114),(175,121),(175,133),(175,142),(175,147)
177	-
178	(177,34),(177,130)
179	(178,63)
180	-
181	(180,85),(180,113),(180,133)
182	(181,138)
183	(182,45),(182,71)
184	(183,37),(183,54),(183,130),(183,138),(183,155),(183,166),(183,174)
185	(184,129)
186	-
187	(186,45),(186,53),(186,67),(186,137),(186,145),(186,153),(186,155)
188	(187,175)
189	(188,53),(188,101),(188,173)
190	(189,75),(189,108),(189,170)
191	(190,189)
192	(191,111),(191,150),(191,158)
193	-
194	(193,36),(193,76),(193,78),(193,115),(193,132)
195	(194,65),(194,167)
196	(195,34),(195,65),(195,135),(195,162),(195,177)
197	(196,45)
198	(197,63),(197,140),(197,171)
199	(198,41),(198,145)
200	(199,43),(199,45),(199,69),(199,115)
201	(200,57)
202	(201,38),(201,39),(201,52),(201,116),(201,124),(201,146),(201,166)
203	(202,63),(202,87),(202,201)
204	(203,38),(203,126),(203,135)

Field Size	(k,i) tuples (cont'd)
205	(204,185)
206	(205,75),(205,138)
207	(206,89),(206,161),(206,189)
208	-
209	-
210	(209,59),(209,108),(209,123),(209,186),(209,194),(209,206)
211	(210,35),(210,45),(210,97),(210,99),(210,145),(210,155),(210,165),(210,197),(210,209)
212	(211,78),(211,121),(211,126),(211,201),(211,205)
213	-
214	(213,43),(213,70),(213,127),(213,190)
215	(214,153),(214,211)
216	(215,51),(215,122)
217	(216,81)
218	(217,124),(217,136)
219	(218,173),(218,179)
220	(219,41),(219,126),(219,181)
221	(220,117)
222	(221,35),(221,71)
223	(222,65),(222,175),(222,211)
224	(223,55),(223,139),(223,153)
225	-
226	(225,127),(225,168),(225,182)
227	(226,33),(226,117),(226,135)
228	(227,174),(227,185),(227,197),(227,210)
229	(228,209)
230	(229,55),(229,60),(229,111),(229,195),(229,220)
231	(230,93)
232	(231,82),(231,127)
233	(232,57)
234	(233,74),(233,143),(233,198)
235	(234,41),(234,51),(234,83),(234,85),(234,131),(234,195),(234,223)
236	-
237	(236,41),(236,165)
238	(237,187)
239	(238,123),(238,211),(238,219)
240	(239,74),(239,147),(239,161),(239,230)
241	-
242	(241,90),(241,174)
243	(242,105),(242,131),(242,141)
244	(243,82),(243,90),(243,127),(243,209),(243,234)
245	-
246	(245,86),(245,131),(245,206),(245,239)
247	(246,41),(246,193),(246,209)
248	(247,129),(247,189),(247,226),(247,243)
249	-
250	(249,44)
251	(250,93),(250,129),(250,159),(250,199)
252	(251,65),(251,78)
253	-
254	(253,67),(253,99),(253,174),(253,199)
255	(254,101)

Field Size	(k,i) tuples (cont'd)
256	(255,41),(255,166),(255,227),(255,243)
257	-
258	(257,132)
259	(258,179)
260	(259,190)
261	(260,69),(260,105)
262	(261,34),(261,92),(261,126),(261,212)
263	(262,33),(262,91),(262,151),(262,201)
264	(263,158),(263,257)
265	-
266	(265,126),(265,156),(265,246)
267	(266,65),(266,191),(266,221)
268	(267,49),(267,87),(267,207),(267,239)
269	(268,181),(268,225)
270	(269,110),(269,246)
271	(270,229),(270,253)
272	(271,151),(271,177),(271,201),(271,267)
273	-
274	(273,55),(273,99),(273,130),(273,260)
275	(274,103),(274,105),(274,123),(274,211)
276	(275,134)
277	(276,161),(276,177)
278	(277,60),(277,67),(277,271),(277,276)
279	-
280	(279,34),(279,77),(279,79),(279,101),(279,187),(279,195)
281	-
282	(281,188),(281,207)
283	(282,91),(282,121)
284	(283,55),(283,142)
285	(284,221)
286	(285,78),(285,110)
287	-
288	(287,209),(287,279)
289	-
290	(289,198),(289,211)
291	(290,189),(290,275)
292	(291,54),(291,94),(291,121),(291,225),(291,229),(291,246)
293	-
294	(293,62),(293,87),(293,179),(293,267)
295	(294,65),(294,81),(294,159),(294,173),(294,287),(294,289)
296	(295,126),(295,223),(295,241)
297	(296,113)
298	(297,103),(297,120),(297,252),(297,255)
299	(298,165)
300	(299,63),(299,167)
301	(300,121),(300,149),(300,209),(300,261)
302	(301,199),(301,268)
303	-
304	(303,97),(303,153),(303,163),(303,275),(303,287),(303,294)
305	-
306	(305,104),(305,138)

Field Size	(k,i) tuples (cont'd)
307	(306,33),(306,51),(306,111),(306,119),(306,141),(306,221),(306,269),(306,285),(306,291)
308	(307,181),(307,241)
309	-
310	(309,70),(309,76),(309,210)
311	(310,99),(310,147),(310,273)
312	(311,86),(311,122),(311,131),(311,207)
313	-
314	(313,114),(313,136),(313,180),(313,207),(313,232),(313,267),(313,307)
315	(314,251),(315,54)
316	(315,114),(315,166),(315,173),(315,245),(315,262),(315,265),(315,282)
317	(316,49)
318	(317,116),(317,134)
319	(318,71),(318,161)
320	(319,103),(319,139),(319,255),(319,274),(319,283)
321	-
322	(321,46),(321,208),(321,290)
323	(322,103),(322,163)
324	(323,170),(323,270),(323,302)
325	(324,153)
326	(325,36),(325,51),(325,135),(325,306)
327	-
328	(327,65),(327,75),(327,93)
329	(328,273)
330	(329,78),(329,188),(329,215),(329,255)
331	(330,85),(330,187),(330,217),(330,263)
332	(331,222),(331,225),(331,229)
333	-
334	(333,86),(333,139),(333,163),(333,274),(333,278),(333,286)
335	(334,55),(334,81),(334,231),(334,249),(334,259)
336	(335,41),(335,114)
337	(336,65),(336,81),(336,161)
338	(337,40),(337,186),(337,216)
339	(338,165)
340	(339,95),(339,121)
341	(340,153),(340,201)
342	(341,236),(342,91)
343	-
344	(343,114),(343,115),(343,178),(343,322)
345	(344,209)
346	(345,40),(345,70),(345,102)
347	(346,123),(346,141),(346,273)
348	(347,182),(347,186),(347,230),(347,314)
349	-
350	(349,258)
351	(350,45),(350,47)
352	(351,66),(351,89),(351,179),(351,249),(351,325),(351,334),(351,347)
353	(352,97)
354	(353,158),(353,176),(353,326)
355	(354,43),(354,53),(354,79),(354,165),(354,235),(354,239),(354,275),(354,353)
356	(355,262)
357	-

Field Size	(k,i) tuples (cont'd)
358	(357,124),(357,132),(357,134),(357,220),(357,223),(357,351)
359	(358,69),(358,93)
360	(359,317)
361	-
362	(361,120),(361,291)
363	(362,305)
364	(363,175),(363,247),(363,255),(363,343)
365	(364,141),(364,213),(364,237)
366	(365,60),(365,87),(365,222)
367	(366,69),(366,331)
368	(367,237),(367,253),(367,265),(367,322)
369	(368,209)
370	(369,51),(369,86),(369,326),(369,366)
371	(370,343)
372	(371,206),(371,210),(371,290),(371,362)
373	-
374	(373,148),(373,276)
375	(374,153),(374,161)
376	(375,34),(375,65),(375,229)
377	(376,337),(376,361)
378	(377,90),(377,339)
379	(378,203),(378,301),(378,353),(378,359)
380	(379,190)
381	(380,45)
382	(381,36),(381,86),(381,99)
383	(382,43),(382,75),(382,171),(382,273)
384	(383,155),(383,233),(383,270)
385	-
386	(385,247),(385,271)
387	(386,101),(386,251),(386,311)
388	(387,47),(387,93),(387,197),(387,233),(387,322)
389	(388,133),(388,289)
390	(389,38),(389,47),(389,158)
391	(390,121),(390,135),(390,253),(390,327),(390,353)
392	(391,169),(391,309)
393	-
394	(393,268),(393,370)
395	-
396	(395,110),(395,159),(395,170),(395,290),(395,294)
397	(396,101),(396,133)
398	(397,195),(397,244)
399	(398,125),(398,203),(398,213)
400	(399,54),(399,55),(399,193),(399,329),(399,381)
401	-
402	(401,36),(401,71),(401,276),(401,306),(401,324)
403	(402,287)
404	(403,58),(403,237),(403,274),(403,397)
405	(404,233)
406	(405,36),(405,204)
407	-
408	(407,149),(407,323),(407,342)

Field Size	(k,i) tuples (cont'd)
409	(408,65)
410	(409,408)
411	(410,129),(410,149),(410,269),(410,297)
412	(411,181),(411,349)
413	(412,277)
414	(413,404)
415	(414,89),(414,105),(414,189),(414,197),(414,253),(414,327)
416	(415,186),(415,249),(415,285),(415,355)
417	-
418	(417,136),(417,334),(417,390),(417,412)
419	-
420	(419,66),(419,306)
421	(420,49),(420,77),(420,141),(420,297)
422	(421,222),(421,307),(421,318),(421,414)
423	(422,33),(422,101),(422,371)
424	(423,114),(423,234),(423,354),(423,415)
425	(424,225)
426	(425,278),(425,318),(425,338),(425,404),(425,410)
427	(426,247)
428	(427,82),(427,151),(427,309),(427,394)
429	(428,113)
430	(429,60),(429,207),(429,308),(429,350),(429,386)
431	(430,69),(430,79),(430,117),(430,141),(430,307)
432	(431,257),(431,342)
433	-
434	(433,280),(433,387),(433,414)
435	(434,383),(434,413)
436	(435,199),(435,201),(435,221),(435,382),(435,399)
437	(436,265)
438	(437,284)
439	(438,71),(438,79)
440	(439,189),(439,363),(439,385),(439,403),(439,438)
441	(440,89),(440,393)
442	(441,38),(441,255),(441,264),(441,358),(441,388)
443	(442,163)
444	(443,77),(443,402)
445	(444,137)
446	(445,55),(445,102),(445,180)
447	-
448	(447,51),(447,205),(447,213),(447,330),(447,349),(447,359),(447,367)
449	-
450	(449,60),(449,204)
451	(450,137),(450,221),(450,259),(450,321),(450,329),(450,375)
452	(451,205),(451,286),(451,369)
453	-
454	(453,166),(453,370)
455	(454,295)
456	(455,107),(455,147),(455,222)
457	(456,353)
458	(457,175),(457,204)
459	(458,89),(458,453)

Field Size	(k,i) tuples (cont'd)
460	(459,103),(459,153),(459,346)
461	(460,109)
462	(461,219),(461,338),(461,372)
463	(462,61),(462,187),(462,295),(462,413)
464	(463,75),(463,127),(463,142),(463,147),(463,235)
465	-
466	(465,144),(465,259),(465,386),(465,436)
467	(466,243),(466,351),(466,393),(466,411)
468	(467,63),(467,74),(467,290),(467,422)
469	(468,433)
470	(469,139),(469,318)
471	(470,305)
472	(471,61),(471,130),(471,165),(471,227),(471,317),(471,421)
473	-
474	(473,98),(473,335)
475	(474,157),(474,265),(474,369)
476	(475,63),(475,129),(475,151),(475,471)
477	(476,185)
478	(477,330)
479	(478,105),(478,163),(478,339),(478,441),(478,465)
480	(479,183),(479,315)
481	(480,385)
482	(481,210),(481,252),(481,264),(481,312),(481,388)
483	(482,221),(482,293),(482,365)
484	(483,37),(483,78),(483,149)
485	(484,237),(484,421)
486	(485,419),(485,447),(485,470),(485,479)
487	(486,163),(486,419),(486,443)
488	(487,177),(487,205),(487,229),(487,249),(487,411)
489	-
490	(489,180),(489,316),(489,454)
491	(490,303),(490,345),(490,375)
492	(491,327),(491,377),(491,474)
493	(492,133),(492,361)
494	(493,132),(493,228),(493,295)
495	(494,281)
496	(495,133),(495,323),(495,419),(495,471)
497	(496,129),(496,289)
498	(497,35),(497,110),(497,366),(497,392)
499	-
500	(499,94),(499,346),(499,390)
501	-
502	(501,254),(501,418),(501,430)
503	-
505	(504,97)
506	(505,108),(505,216),(505,244),(505,396)
507	(506,47),(506,101),(506,203),(506,237),(506,263),(506,345),(506,413)
508	(507,81),(507,90)
509	-
510	(509,179),(509,203),(509,227),(509,468)

Field Size	(k,i) tuples (cont'd)
511	(510,287),(510,305),(510,367),(510,369),(510,433)
512	(511,34),(511,87),(511,322),(511,334)

Appendix B

Table of Low-Weight Special Primes: $p = 2^k + 2^i - 1$

Field Size	(k,i) tuples
64	(63,34),(63,46),(63,50),(63,56)
65	(64,33),(64,60)
66	(65,64)
67	(66,36),(66,43),(66,46),(66,54),(66,59),(66,63)
68	(67,48)
69	(68,58),(68,60)
71	(70,35),(70,42),(70,43),(70,48),(70,56),(70,59)
72	(71,46),(71,48),(71,52),(71,54)
73	(72,57)
74	-
75	(74,55),(74,66),(74,72)
76	(75,36),(75,54),(75,62),(75,64)
77	(76,47)
78	(77,76)
79	(78,38),(78,48),(78,62)
80	(79,46),(79,72)
81	(80,52),(80,63),(80,67),(80,73)
82	-
83	(82,34),(82,39),(82,54),(82,58)
84	(83,58)
85	(84,72),(84,78)
86	(85,40),(85,52),(85,72)
87	(86,42),(86,52),(86,54),(86,60)
88	(87,56),(87,60)
89	(88,37),(88,51),(88,53)
90	-
91	(90,43),(90,72),(90,82)
92	(91,32),(91,38),(91,42),(91,58),(91,80)
93	(92,54),(92,64),(92,66),(92,72)
94	(93,40)
95	(94,72)
96	(95,82),(95,88),(95,94)
97	(96,33),(96,53),(96,63),(96,71),(96,91)
98	-
99	(98,64),(98,88)
100	(99,32),(99,40),(99,66),(99,76),(99,86)
101	(100,62),(100,76),(100,87),(100,93),(100,94),(100,96)
102	(101,36)

Field Size	(k,i) tuples (cont'd)
103	(102,34),(102,52),(102,64),(102,76),(102,82),(102,87),(102,99)
104	-
105	(104,52),(104,75),(104,91),(104,103)
106	(105,96)
107	(106,63)
108	(107,60),(107,90),(107,96),(107,102)
109	(108,60),(108,98)
110	(109,72)
111	(110,48),(110,52),(110,55),(110,67),(110,84),(110,99)
112	(111,60),(111,74),(111,106)
113	(112,34),(112,60),(112,71),(112,102),(112,105),(112,109)
114	-
115	(114,32)
116	(115,56),(115,66),(115,74),(115,84),(115,96)
117	(116,36),(116,70),(116,85)
118	(117,88)
119	(118,51),(118,63),(118,99)
120	(119,36),(119,40)
121	(120,35),(120,53),(120,68),(120,69),(120,89),(120,92),(120,108)
122	(121,96)
123	(122,64),(122,84)
124	(123,36),(123,40),(123,66),(123,76)
125	(124,40),(124,96),(124,98)
126	-
127	(126,40),(126,50),(126,51),(126,66),(126,84),(126,116)
128	(127,36),(127,78),(127,106),(127,110)
129	(128,115),(128,124)
130	-
131	(130,58),(130,66),(130,68),(130,90),(130,106),(130,119),(130,124)
132	(131,106)
133	(132,92)
134	(133,64)
135	-
136	(135,34),(135,42),(135,46),(135,54),(135,76)
137	(136,64),(136,122)
138	(137,48)
139	(138,55),(138,60),(138,104),(138,112)
140	(139,86),(139,106)
141	(140,55),(140,67)
142	(141,52)
143	(142,59),(142,67),(142,80),(142,95),(142,115)
144	-
145	(144,32),(144,45),(144,48),(144,50),(144,71),(144,76),(144,88),(144,98),(144,103),(144,123),(144,135),(144,143)
146	(145,80)
147	(146,115),(146,124)
148	(147,48),(147,68),(147,82),(147,126),(147,136),(147,142)
149	(148,59),(148,69),(148,70),(148,109),(148,132)
150	-
151	(150,36),(150,38),(150,128),(150,132),(150,148)
152	(151,46),(151,62),(151,82),(151,96)
153	(152,46),(152,106),(152,130)

Field Size	(k,i) tuples (cont'd)
154	-
155	(154,35),(154,100),(154,106),(154,143),(154,150)
156	(155,36),(155,130),(155,138),(155,148)
157	(156,39),(156,66),(156,67),(156,102),(156,103),(156,112),(156,115),(156,143),(156,144)
158	(157,32),(157,96),(157,112)
159	(158,108)
160	(159,88),(159,110),(159,116),(159,138)
161	(160,110),(160,117)
162	-
163	(162,50),(162,76),(162,108),(162,115),(162,135),(162,159)
164	(163,48),(163,120)
165	(164,138),(164,151)
166	-
167	(166,42),(166,66),(166,67),(166,152)
168	(167,108)
169	(168,55),(168,91),(168,162)
170	(169,120)
171	(170,60),(170,90),(170,160)
172	(171,72),(171,76),(171,98)
173	(172,40),(172,42),(172,66),(172,95),(172,117),(172,124),(172,126),(172,165)
174	-
175	(174,32),(174,88),(174,102),(174,130),(174,135)
176	(175,90),(175,126),(175,130)
177	(176,70),(176,120),(176,147)
178	(177,96)
179	(178,71),(178,72),(178,87),(178,152),(178,163)
180	(179,72),(179,100),(179,106)
181	(180,46),(180,82),(180,108),(180,142),(180,167),(180,174)
182	(181,44),(181,128),(181,164)
183	(182,39),(182,127),(182,160)
184	(183,166)
185	(184,130),(184,160)
186	(185,40),(185,136)
187	(186,34),(186,60),(186,90),(186,115),(186,116),(186,143),(186,150),(186,159),(186,163),(186,184)
188	(187,36),(187,108),(187,110),(187,138)
189	(188,82),(188,103),(188,109),(188,168)
190	(189,168)
191	(190,36),(190,63),(190,75),(190,95),(190,118),(190,140),(190,150),(190,159)
192	(191,84),(191,120)
193	(192,41),(192,65),(192,86),(192,120),(192,121),(192,132),(192,162),(192,179),(192,184)
194	-
195	(194,156)
196	(195,32),(195,50),(195,56),(195,122),(195,190)
197	(196,52),(196,60),(196,82),(196,110),(196,125),(196,141),(196,143),(196,186)
198	(197,60),(197,172)
199	(198,38),(198,124),(198,172),(198,192)
200	(199,58),(199,120),(199,168),(199,188),(199,190)
201	(200,121)
202	-
203	(202,116),(202,130),(202,192)
204	-

Field Size	(k,i) tuples (cont'd)
205	(204,72)
206	-
207	-
208	(207,86),(207,100),(207,150),(207,206)
209	(208,78),(208,107)
210	-
211	(210,64),(210,139),(210,151),(210,204)
212	(211,34),(211,36),(211,48),(211,70),(211,128),(211,150),(211,198)
213	(212,99),(212,178)
214	(213,48),(213,120),(213,208)
215	(214,102),(214,103),(214,135),(214,142),(214,160),(214,196),(214,211)
216	(215,184)
217	(216,140),(216,156)
218	(217,176),(217,216)
219	(218,43),(218,78),(218,84),(218,162)
220	(219,96),(219,156),(219,196)
221	(220,39),(220,69),(220,80),(220,133),(220,173)
222	(221,40),(221,208)
223	(222,74),(222,108),(222,155),(222,195),(222,212)
224	(223,80)
225	(224,42),(224,73)
226	-
227	(226,66),(226,107),(226,111),(226,140),(226,192)
228	-
229	(228,70),(228,99),(228,108),(228,195),(228,211)
230	-
231	(230,199),(230,211)
232	(231,40),(231,130),(231,180)
233	(232,75),(232,99),(232,169),(232,170),(232,179)
234	-
235	(234,62)
236	(235,74),(235,90),(235,120),(235,192)
237	(236,51),(236,60),(236,120),(236,171),(236,199)
238	-
239	(238,78),(238,118),(238,179),(238,188),(238,192)
240	(239,36),(239,192),(239,216)
241	(240,58),(240,98),(240,101),(240,108),(240,111),(240,155),(240,157),(240,177),(240,201)
242	-
243	(242,180),(242,196),(242,219)
244	(243,40),(243,100)
245	(244,50),(244,171),(244,172),(244,186)
246	(245,96)
247	(246,34),(246,39),(246,79),(246,192),(246,200),(246,215),(246,231)
248	(247,110),(247,206),(247,222)
249	(248,132),(248,190),(248,228)
250	(249,88)
251	(250,35),(250,52),(250,63),(250,84),(250,91),(250,111),(250,152),(250,248)
252	(251,70)
253	(252,51),(252,61),(252,78),(252,168),(252,174),(252,210),(252,214)
254	(253,56),(253,136),(253,168),(253,188)
255	(254,66),(254,88),(254,232)

Field Size	(k,i) tuples (cont'd)
256	(255,96),(255,176),(255,232)
257	(256,96),(256,106),(256,130),(256,166),(256,196),(256,203),(256,206),(256,225),(256,231),(256,252)
258	-
259	(258,32),(258,40),(258,58),(258,107),(258,128),(258,162),(258,192),(258,231),(258,242)
260	(260,37),(260,112),(260,127),(260,139),(260,163),(260,190)
261	-
262	(261,84)
263	(262,40),(262,99),(262,107),(262,162),(262,186),(262,190),(262,208)
264	(263,156),(263,240)
265	(264,88),(264,106),(264,140),(264,185),(264,203),(264,251),(264,255),(264,256)
266	(265,232)
267	(266,55),(266,100),(266,162),(266,172)
268	(267,216)
269	(268,32),(268,37),(268,64),(268,180)
270	-
271	(270,76),(270,138),(270,154),(270,206)
272	(271,42),(271,48),(271,54),(271,140),(271,156),(271,174),(271,222),(271,230),(271,266)
273	(272,52),(272,66),(272,88),(272,115),(272,126),(272,232),(272,270)
274	-
275	(274,36),(274,71),(274,95),(274,140),(274,176)
276	(275,90),(275,234)
277	(276,93),(276,117),(276,144),(276,202),(276,267),(276,274)
278	(277,268)
279	-
280	(279,98),(279,126),(279,228)
281	(280,62),(280,107),(280,108),(280,132),(280,255)
282	-
283	(282,39),(282,100),(282,104),(282,122),(282,182),(282,196),(282,198),(282,219),(282,235),(282,264),(282,270)
284	(283,78),(283,104)
285	(284,102),(284,136),(284,186),(284,211),(284,238)
286	(285,192)
287	(286,82),(286,274)
288	(287,106),(287,136),(287,142),(287,208)
289	(288,61),(288,119),(288,140),(288,142),(288,144),(288,208),(288,278)
291	(290,252),(290,258)
292	(291,144),(291,240),(291,262),(291,270)
293	(292,37),(292,58),(292,220),(292,264)
294	-
295	(294,36),(294,91),(294,166),(294,231),(294,276)
296	(295,96),(295,146),(295,180)
297	(296,51),(296,126),(296,127),(296,130)
298	-
299	(298,55),(298,231),(298,232),(298,250)
300	(299,172),(299,292)
301	(300,93),(300,142),(300,148),(300,154),(300,221),(300,259)
302	(301,48),(301,204)
303	(302,160),(302,244)
304	-
305	(304,138)
306	-

Field Size	(k,i) tuples (cont'd)
307	(306,100),(306,139),(306,194),(306,246),(306,275),(306,294)
308	(307,86),(307,182),(307,262),(307,306)
309	(308,139),(308,192),(308,240),(308,253),(308,294)
310	(309,40),(309,76)
311	(310,59),(310,90),(310,108),(310,135),(310,136),(310,162),(310,186),(310,255),(310,294),(310,304)
312	(311,36),(311,130),(311,262)
313	(312,51),(312,62),(312,64),(312,86),(312,108),(312,121),(312,165),(312,175),(312,188),(312,230),(312,290), (312,309)
314	(313,48),(313,208)
315	(314,208)
316	(315,176),(315,306)
317	(316,61),(316,74),(316,114),(316,132),(316,135),(316,176),(316,285)
318	(317,100),(317,136),(317,276)
319	(318,90),(318,95),(318,103),(318,167),(318,190),(318,290),(318,295),(318,302)
320	(319,56),(319,178)
321	(320,36),(320,159),(320,190),(320,196),(320,232),(320,286),(320,306)
323	(322,36),(322,195)
324	(323,60),(323,184),(323,258)
325	(324,46),(324,70),(324,151),(324,198),(324,202)
326	(325,104),(325,144),(325,184),(325,224),(325,324)
327	(326,67),(326,90),(326,91),(326,99),(326,216),(326,270),(326,312)
328	(327,42),(327,98),(327,106),(327,140),(327,168),(327,246)
329	(328,109),(328,121),(328,133),(328,159),(328,232),(328,238),(328,317)
330	(329,160)
331	(330,48),(330,78),(330,123),(330,130),(330,160),(330,255),(330,278)
332	(331,116)
333	(332,70),(332,91),(332,324)
334	(333,60)
335	(334,51),(334,60),(334,75),(334,156),(334,238),(334,250),(334,315)
336	(335,330)
337	(336,81),(336,283),(336,290)
338	(337,48)
339	(338,150),(338,294)
340	(339,100),(339,120),(339,128),(339,302)
341	(340,45),(340,50),(340,71),(340,72),(340,238),(340,243),(340,300)
342	(341,48),(341,64),(341,100),(341,196),(341,244),(341,280),(341,312)
343	(342,72),(342,116),(342,256),(342,287)
344	(343,58),(343,78),(343,114),(343,308)
345	(344,120),(344,195),(344,265),(344,283)
346	-
347	(346,91),(346,102),(346,104),(346,142),(346,147),(346,203),(346,327),(346,343)
348	(347,160)
349	(348,40),(348,80),(348,134),(348,141),(348,168),(348,190),(348,207),(348,229)
350	-
351	(350,60),(350,124)
352	(351,144),(351,172)
353	(352,44),(352,89),(352,165),(352,305),(352,328)
354	-
355	(354,36),(354,156),(354,162),(354,312)
356	(355,32),(355,38),(355,60),(355,134),(355,178),(355,260)

Field Size	(k,i) tuples (cont'd)
357	(356,114),(356,126),(356,163),(356,184)
358	(357,132)
359	(358,68),(358,172),(358,202),(358,320),(358,323)
360	(359,78),(359,202)
361	(360,71),(360,179),(360,181),(360,206),(360,248),(360,267),(360,280),(360,291),(360,315),(360,356)
362	(361,64),(361,224),(361,312)
363	(362,100),(362,151),(362,207),(362,294)
364	(363,36),(363,46),(363,58),(363,74),(363,94),(363,248),(363,250),(363,274)
365	(364,85),(364,142)
366	(365,64),(365,220),(365,264),(365,316),(365,336),(365,360)
367	(366,170),(366,207),(366,232),(366,252)
368	(367,128)
369	(368,78)
370	-
371	(370,224)
372	(371,64),(371,84)
373	(372,58),(372,84),(372,111),(372,192),(372,231),(372,334),(372,351),(372,359)
374	(373,160),(373,344)
375	(374,36),(374,72),(374,102),(374,132),(374,195),(374,228)
376	(375,100),(375,204),(375,212),(375,252),(375,264),(375,368)
377	(376,174),(376,232),(376,240),(376,313)
378	-
379	(378,140),(378,258),(378,354)
380	(379,48),(379,302),(379,366)
381	(380,234)
382	-
383	(382,64),(382,180),(382,300),(382,348),(382,376)
384	-
385	(384,33),(384,301),(384,341),(384,343)
386	-
387	(386,124),(386,150)
388	(387,XXX),(387,262),(387,382)
389	(388,63),(388,109),(388,123),(388,150),(388,267)
390	(389,60)
391	(390,83),(390,128),(390,230),(390,306),(390,331),(390,334),(390,338)
392	(391,122),(391,128),(391,326)
393	(392,58),(392,75),(392,135),(392,256),(392,306),(392,348),(392,391)
394	-
395	(394,343)
396	(395,36),(395,96),(395,262),(395,336),(395,376)
397	(396,44),(396,75),(396,107),(396,167),(396,202),(396,205),(396,243),(396,370),(396,386)
398	(397,272)
399	(398,54),(398,147),(398,180),(398,187),(398,219)
400	(399,130),(399,292),(399,362)
401	(400,123),(400,136),(400,312),(400,330),(400,389)
402	(402,98),(402,252),(402,299),(402,312),(402,366)
403	-
404	(403,108),(403,114),(403,216),(403,240)
405	(404,115),(404,171),(404,351),(404,363)
406	-
407	(406,102),(406,104),(406,116),(406,210),(406,232),(406,240),(406,262),(406,312),(406,316),(406,344),(406,370)

Field Size	(k,i) tuples (cont'd)
408	(407,238)
409	(408,94),(408,99),(408,103),(408,121),(408,131),(408,140),(408,175),(408,207),(408,214),(408,233),(408,255), (408,307),(408,313),(408,357)
410	(409,56),(409,240)
411	-
412	(411,140),(411,162),(411,222),(411,232),(411,292)
413	-
414	(413,220)
415	(414,90),(414,278),(414,298),(414,371),(414,402)
416	(415,84),(415,118),(415,224),(415,346)
417	(416,66),(416,73),(416,279),(416,295),(416,330),(416,396)
418	-
419	(418,102),(418,130),(418,138),(418,142),(418,211),(418,238),(418,291),(418,330),(418,415)
420	-
421	(420,50),(420,98),(420,114),(420,115),(420,126),(420,173),(420,215),(420,248),(420,256),(420,326),(420,327), (420,379),(420,388),(420,389)
422	(421,116),(421,184),(421,260),(421,288)
423	(422,162),(422,240),(422,258),(422,420)
424	(423,46),(423,240),(423,394)
425	(424,63),(424,112),(424,221),(424,363),(424,375)
426	-
427	(426,162),(426,187),(426,252),(426,415)
428	(427,222),(427,302),(427,390)
429	(428,150),(428,222),(428,234)
430	(429,396)
431	(430,78),(430,79),(430,207),(430,214),(430,286),(430,295)
432	(431,370)
433	(432,82),(432,111),(432,226),(432,232),(432,249),(432,260),(432,275),(432,344),(432,420),(432,424)
434	-
435	(434,355)
436	(435,206),(435,226),(435,288),(435,294),(435,312)
437	(436,35),(436,40),(436,147),(436,299),(436,349),(436,354),(436,384)
438	(437,172),(437,376)
439	(438,40),(438,47),(438,107),(438,322),(438,350),(438,362),(438,388)
440	(439,62),(439,162),(439,216),(439,258)
441	(440,37),(440,217),(440,240)
442	(441,336)
443	(442,95),(442,104),(442,150),(442,154),(442,166),(442,367),(442,388)
444	(443,40),(443,94),(443,330),(443,348)
445	(444,75),(444,398)
446	(445,392),(445,396)
447	(446,55),(446,280)
448	-
449	(448,289),(448,298),(448,323),(448,351),(448,382),(448,410),(448,433)
450	-
451	(450,218),(450,244),(450,326),(450,330),(450,431)
452	(451,34),(451,168),(451,230)
453	(452,114),(452,192),(452,199),(452,391),(452,445)
454	(453,208),(453,324)
455	(454,382),(454,388)
456	(455,84),(455,126),(455,292),(455,342),(455,418),(455,444)

Field Size	(k,i) tuples (cont'd)
457	(456,42),(456,83),(456,87),(456,240),(456,252),(456,327),(456,341),(456,376)
458	(457,128)
459	(458,75),(458,151)
460	(459,66),(459,138),(459,222),(459,260),(459,458)
461	(460,95),(460,108),(460,256),(460,306),(460,352),(460,432),(460,436)
462	(461,280)
463	(462,140),(462,320),(462,327),(462,355),(462,415)
464	(463,116),(463,150),(463,164),(463,366)
465	(464,343),(464,460)
466	-
467	(466,34),(466,51),(466,246),(466,259),(466,271),(466,440),(466,444)
468	(467,202),(467,462)
469	(468,122),(468,318)
470	-
471	(470,84),(470,103),(470,108)
472	(471,84),(471,242),(471,250),(471,438),(471,470)
473	(472,34),(472,144),(472,212),(472,317),(472,381),(472,431),(472,469)
474	-
475	(474,38),(474,186),(474,258),(474,266),(474,291)
476	(475,80),(475,82),(475,102),(475,192)
477	(476,139),(476,183),(476,222),(476,250),(476,270),(476,373),(476,454)
478	-
479	(478,58),(478,68),(478,120),(478,163),(478,210),(478,387)
480	-
481	(480,92),(480,122),(480,211),(480,274),(480,318),(480,371),(480,407),(480,463)
482	-
483	(482,99),(482,172),(482,387)
484	(483,128),(483,156),(483,158)
485	(484,35),(484,156),(484,243),(484,276),(484,315),(484,352),(484,406),(484,431)
486	(485,180),(485,204)
487	(486,54),(486,103),(486,132),(486,135),(486,139),(486,163),(486,204),(486,311),(486,371),(486,384),(486,403), (486,462),(486,467)
488	(487,108),(487,146),(487,468),(487,476)
489	(488,52),(488,61),(488,120),(488,133),(488,163),(488,172),(488,207),(488,277),(488,358),(488,367)
490	-
491	(490,112),(490,131),(490,143),(490,239),(490,255),(490,311),(490,323),(490,332),(490,412)
492	(491,36),(491,52),(491,274),(491,376)
493	(492,108),(492,211),(492,261)
494	-
495	(494,420)
496	(495,42),(495,78),(495,132),(495,222),(495,308),(495,408),(495,428)
497	(496,95),(496,131),(496,147),(496,170),(496,172),(496,206),(496,220),(496,360),(496,457)
498	-
499	(498,87),(498,159),(498,238),(498,319),(498,462)
500	(499,70),(499,266)
501	(500,36),(500,42),(500,60),(500,63),(500,157),(500,267),(500,342),(500,375),(500,378),(500,411),(500,447)
502	(501,172),(501,268)
503	(502,44),(502,100),(502,139),(502,164),(502,394),(502,426),(502,431),(502,475)
504	(503,448)
505	(504,88),(504,90),(504,112),(504,213),(504,375),(504,382),(504,396),(504,410),(504,495)

Field Size	(k,i) tuples (cont'd)
506	(505,40),(505,312),(505,424)
507	-
508	(507,168),(507,208),(507,226),(507,258)
509	(508,39),(508,64),(508,350),(508,364),(508,503)
510	(509,280)
511	(510,67),(510,79),(510,115),(510,183),(510,259),(510,323),(510,408),(510,499)
512	(511,102),(511,156),(511,344),(511,462),(511,466)