

ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

COMPUTATIONAL METHODS FOR INTEGER FACTORIZATION



M.Sc. THESIS

Deniz KIRLIDOĞ

Department of Computational Science and Engineering

Computational Science and Engineering Programme

JUNE 2019

ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

COMPUTATIONAL METHODS FOR INTEGER FACTORIZATION



M.Sc. THESIS

**Deniz KIRLIDOĞ
(702151002)**

Department of Computational Science and Engineering

Computational Science and Engineering Programme

Thesis Advisor: Assoc. Prof. Enver ÖZDEMİR

JUNE 2019

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

ÇARPANLARA AYIRMA İÇİN HESAPLAMALI YÖNTEMLER

YÜKSEK LİSANS TEZİ

**Deniz KIRLIDOĞ
(702151002)**

Hesaplama Bilim ve Mühendislik Anabilim Dalı

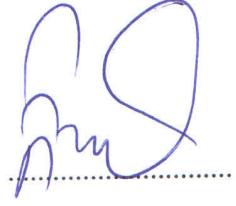
Hesaplama Bilim ve Mühendislik Programı

Tez Danışmanı: Doç. Dr. Enver ÖZDEMİR

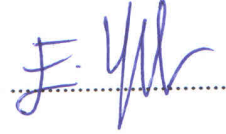
HAZİRAN 2019

Deniz KIRLIDOĞ, a M.Sc. student of ITU Informatics Institute student ID 702151002, successfully defended the thesis entitled "COMPUTATIONAL METHODS FOR INTEGER FACTORIZATION", which she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

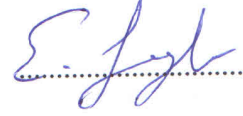
Thesis Advisor : **Assoc. Prof. Enver ÖZDEMİR**
İstanbul Technical University



Jury Members : **Assoc. Prof. Ergün YARANERİ**
İstanbul Technical University



Asst. Prof. Elif SEGAH ÖZTAŞ
Karamanoğlu Mehmetbey University



Date of Submission : 3 May 2019
Date of Defense : 14 June 2019



To my mother İkbal and father Melih,



FOREWORD

I would like to thank my advisor Enver Özdemir, my family and friends for supporting me throughout this process. I am grateful to my grandfather, aunts and cousin for encouraging me. Special thanks to Cem Kocagil for proofreading.

May 2019

Deniz KIRLIDOĞ



TABLE OF CONTENTS

	<u>Page</u>
FOREWORD	ix
TABLE OF CONTENTS	xi
ABBREVIATIONS	xiii
LIST OF TABLES	xv
LIST OF FIGURES	xvii
SUMMARY	xix
ÖZET	xxi
1. INTRODUCTION	1
1.1 Purpose of Thesis	1
2. DIFFIE-HELLMAN KEY EXCHANGE AND RSA	3
2.1 Public Key Cryptography and Diffie-Hellman Key Exchange.....	3
2.2 RSA Cryptosystem.....	8
3. INTEGER FACTORIZATION	17
3.1 Binary Quadratic Forms and Groups	17
3.2 Shanks' SQUFOF Algorithm and Improving SQUFOF.....	18
3.3 Taking Shanks' Algorithm a Step Further	26
3.4 Factoring a Semiprime	30
4. BENCHMARKS	33
4.1 Serial Code Comparison	33
4.2 Parallel Code Comparison	33
5. CONCLUSION AND FUTURE WORK	37
REFERENCES	39
CURRICULUM VITAE	41



ABBREVIATIONS

CPU	: Central Processing Unit
CRT	: Chinese Remainder Theorem
CUDA	: Compute Unified Device Architecture
GMP	: GNU Multiple Precision Arithmetic Library
GPU	: Graphics Processing Unit
HPC	: High Performance Computing
JIT	: Just-in-Time
LLVM	: Low Level Virtual Machine
MPI	: Message Passing Interface
PKCS	: Public Key Cryptography Standard
SQUFOF	: Square Forms Factorization
UHEM	: Ulusal Yüksek Başarımlı Hesaplama Merkezi



LIST OF TABLES

	<u>Page</u>
Table 3.1: GMP cuGMP Comparison.....	23
Table 3.2: Calculated Interval Beginning and Ends.....	27
Table 3.3: Factorization Results Using Intervals.....	28
Table 4.1: Julia vs C++ Serial Code Results.....	33





LIST OF FIGURES

	<u>Page</u>
Figure 2.1: Conventional Cryptosystem with Single Key Source.....	3
Figure 2.2: Public Key Cryptosystem.....	4
Figure 2.3: Diffie-Hellman Key Exchange Man-In-The-Middle Attack Scenario.....	5
Figure 2.4: Communication Interception After Man-In-The-Middle Attack.....	6
Figure 2.5: Diffie-Hellman Key Exchange Colour Analogy.....	7
Figure 2.6: Diffie-Hellman Key Exchange Flow.....	8
Figure 2.7: RSA Flow.....	9
Figure 2.8: RSA Flow When an Eavesdropper Factors n	13
Figure 3.1: Julia Performance Comparison.....	24
Figure 3.2: $\log(\langle \text{RSA_NUM}, \text{INTERVAL_END} \rangle)$	29
Figure 3.3: $\log(\langle \text{RSA_NUMBER_DIGITS} \rangle, \langle \text{INTERVAL_END} \rangle)$	30
Figure 4.1: Julia vs C++ Parallel Code Results.....	34
Figure 4.2: Julia Efficiency.....	34
Figure 4.3: Julia Speedup.....	35
Figure 4.4: C++/MPI Efficiency.....	35
Figure 4.5: C++/MPI Speedup.....	35



COMPUTATIONAL METHODS FOR INTEGER FACTORIZATION

SUMMARY

Integer factorization is the task of finding the prime factors of a composite number. Since integer factorization is a computationally expensive task, today many digital security systems rely on its difficulty. Many systems use the difficulty of integer factorization for assuring security. Today, RSA cryptosystem, which takes its name from its inventors, Rivest, Shamir and Adleman, dating back to 1977, is widely used for secure communication. RSA is a public key cryptosystem, which uses two asymmetric keys. The concept of asymmetric keys was firstly presented by Whitfield Diffie and Martin E. Hellman in 1976.

Integer factorization has always been an attractive field of research for mathematicians and computer scientists. Many scientists developed algorithms for factoring large integers, however none of them are useful with regard to the limitations of the computational power we have as of today.

Daniel Shanks' SQUFOF Algorithm, which was developed using Binary Quadratic Forms is one of the most popular algorithms for integer factorization, however it is not efficient due to the computational power it requires. Nari, Ozdemir and Yaraneri took this algorithm another step further and developed a new algorithm. By using a multiplier which lies within an interval calculated using the factors of the integer, the new algorithm can easily factor large semiprimes. The multiplier selected from the interval accelerates integer factorization. However, finding the interval without knowing the factors is difficult. Some properties of the intervals are studied in this thesis.

In this work, new SQUFOF with a multiplier algorithm's parallel and serial versions were implemented on multiple platforms. The platforms include C++ and Julia. C++ is one of the most recognised programming languages in the world due to its widespread usage for decades. In contrast, Julia is a very young programming language with rising popularity. For achieving parallelism on multiple processors, MPI is used on C++, Julia has its own libraries for supporting parallelism. The Distributed.jl library is an open source library, which is suggested for writing parallel code on Julia. Not only CPUs, but also GPUs can be used for parallelism, therefore some experiments with CUDA were conducted.

The benchmarks show that the cuGMP library written in C++ for representing Big Integers (integers larger than 64 bits) on GPUs is not successful. Julia is slower than C++ for parallel computations, however, considering its high level features which makes programming easier, it proves itself to be a fast and efficient programming language. C++ with MPI is nearly two times faster than Julia, however writing code in C++ using MPI is a more difficult task than achieving parallelism in Julia. The benchmarks are shown in this thesis.



ÇARPANLARA AYIRMA İÇİN HESAPLAMALI YÖNTEMLER

ÖZET

Whitfield Diffie ve Martin E. Hellman 1976 yılında geliştirdikleri anahtar değiştirme yöntemiyle kriptografi alanında büyük bir değişikliğe imza attılar. Bu buluşla bilgisayar dünyasının en önemli ödülü olan Turing ödülünün de sahibi olan bilim insanları, geleneksel şifreleme yöntemlerinin aksine biri kapalı (diğer insanların ulaşımına kapalı), diğeri açık (diğer insanların ulaşımına açık) olmak üzere çift anahtarla iletişim halinde olanların yeni bir anahtar oluşturup şifrelemeleri fikrini gündeme getirdiler. Açık anahtar değiştirme yöntemi, üzerinde iletişim kurulan kanal güvenilir olmasa da gizli ortak anahtar oluşmasını sağlar.

Diffie ve Hellman'ın geliştirdikleri anahtar değiştirme methodu, 1977 yılında adını onu bulanlardan alan RSA kriptosisteminin de gelişimine sebep oldu. RSA kriptosistemi, adını onu geliştiren Rivest, Shamir ve Adleman'dan almaktadır. Ortaya çıktığı 1977 yılından bu yana RSA kriptosistemi dijital güvenlik dünyasının belkemiği haline gelmiştir.

RSA kriptosisteminin güvenilirliğinin altında iki asal sayının çarpımından oluşan bir sayının çarpanlara ayrılmasının zorluğu yatar. Rivest, Shamir ve Adleman'ın sahibi olduğu RSA dijital güvenlik firması, 2007 yılına kadar "RSA Challenge" ismi altında 100 basamaklı ve daha büyük yarı asal sayıların çarpanlarına ayrılmasını teşvik eden bir yarışma düzenliyordu.

RSA kriptosistemi ile iletişim kurmak isteyen herkesin iki adet anahtar sahibi olması gerekmektedir. Bu anahtarların bir tanesi açık, bir tanesi kapalı olmalıdır. Anahtarların her biri aslında bir sayıdır, bu sayılardan kapalı olan anahtarın başkalarının eline geçmesi büyük güvenlik sorununa sebep olur. Örneğin kapalı anahtarı ele geçiren kişi, anahtarın asıl sahibiymişçesine başkalarıyla iletişim kurabilir, anahtarın asıl sahibine gelen mesajları okuyabilir. Açık anahtar ise sistemin mantığı itibariyle herkese görünmektedir.

RSA kullanarak iletişim kurmak isteyen kişi ilk olarak n yarı asal sayısını seçer, n sayısının çarpanları p ve q asal sayılardır. p ve q sayılarının asallığından emin olmak için asallık testleri yapılmalıdır, Solovay-Strassen Asallık Testi, Fermat Asallık Testi, Miller-Rabin Asallık Testi asallık durumunun kontrolü için kullanılabilir. Fermat Asallık Testi yüksek doğruluk oranına sahiptir, fakat bulmayı garanti ettiği şey asallıktan ziyade asal olmama durumudur. n sayısının faktörlerine ayrılmasının zorluğunun sebebi sayının çok büyük bir sayı olmasıdır. Örneğin Bitcoin için gizli (kapalı) anahtarın uzunluğu 256 bittir. $(p-1)*(q-1)$ sayısından Uzatılmış Öklit Algoritması ile açık anahtar e sayısının çarpmaya göre tersi bulunur. e sayısının çarpmaya göre tersi olan d sayısı kapalı (gizli) anahtardır. Gizli anahtarı bulmak isteyen bir kişinin n yarı asal sayısını çarpanlarına ayırmadan gizli anahtara ulaşması çok zordur.

Büyük tam sayıların çarpanlarına ayrılması muazzam bir hesaplama gücü gerektirdiğinden geleneksel bilgisayarlardan çok daha hızlı olan kuantum bilgisayarlarla bu işlemin yapılması çok daha hızlı olacaktır, fakat günümüzde kuantum bilgisayarların yaygınlaşmaması, kuantum bilgisayarlara ulaşımın neredeyse imkansız olması, kuantum bilgisayarlar için tasarlanan algoritmaların geleneksel algoritmalarından farklı olması gerekliliği sebebiyle çalışmalar daha çok geleneksel bilgisayarlar üzerinden yürütülmektedir. Peter Shor'un algoritması $O((\log n)^2(\log \log n)(\log \log \log n))$ adımda n sayısını çarpanlarına ayırabilmektedir.

Bu çalışmada üzerinde durulan Daniel Shanks'in yaratıcısı olduğu Shanks Algoritması ise geleneksel Von Neumann Mimarili bilgisayarlarda çalışmaktadır. Shanks Algoritması yaygın olarak kullanılmamaktadır, zira hız ve hesaplama gücünün kritik olduğu bu işlem için çok yavaş kalmaktadır. Shanks Algoritması sayılar teorisinin birtakım kavramları üzerine kurulmuştur. Bu kavramlar arasında grup teorisi, ikili kuadratik formlar gösterilebilir. Nari, Özdemir ve Yaraneri Shanks Algoritmasını geliştirerek ikili kuadratik formların kullanıldığı yeni bir algoritma geliştirmişlerdir. Bu tezde, geliştirilmiş yeni algoritma üzerinde performans karşılaştırılması yapılmıştır.

Shanks Algoritması'nın geliştirilmiş hali Özdemir ve Yaraneri'nin ortaya attığı üzere belli bir interval (sayı aralığı) arasından seçilen bir r sayısı ile hızlandırılabilir. Bu interval çarpanlarına ayrılması istenen n sayısının çarpanları p ve q kullanılarak hesaplanmaktadır. Yeterli hesaplama gücüne sahip olduğunda çarpımlara ayrılması istenen sayıya yakın daha önce çarpanlarına ayrılmış sayılardan yararlanmak mümkün olabilir.

İkili kuadratik formlar $f(x, y) = ax^2 + bxy + cy^2$ şeklinde ifade edilir. Bu formun ikili kuadratik form olabilmesi için diskriminant Δ 'nın aşağıdaki koşulları sağlaması gerekmektedir ($\Delta = b^2 - 4ac$). İlk koşul $\Delta \pmod{4}$ 'ün bir veya sıfıra eşit olmasıdır, ikinci koşul ise b sayısının $\pmod{2}$ 'de diskriminant Δ 'ya eşit olması zorunluluğudur. a, b, c sayılarının en büyük ortak bölenlerinin 1 olduğu durumda, ikili kuadratik formun ilkel olduğu söylenir. Bu çalışmada da ilkel ikili kuadratik formlar kullanılmıştır.

Bu çalışmada, aralarında denklik ilişkisi olan ikili kuadratik formlar kullanılarak yaratılan bir döngü ile yarı asal sayıları çarpanlara ayırma işleminin nasıl yapılabileceği anlatılmaktadır.

Bu kadar yüksek hesaplama gücü ve hız gerektiren bir işlemi paralelleştirmek gerekmektedir. Ayrıca, 64 bitten çok daha büyük sayılarla uğraşıldığından klasik double değişkenler bu konuda işe yaramamaktadır. Büyük tam sayıları ifade etmek, onlarla yüksek hızlarda işlemler yapabilmek için GMP kütüphanesi kullanılmıştır. GMP kütüphanesi açık kaynak kodlu ifade edebileceği sayı büyüklüğü teoride sonsuza eşit olan bir kütüphanedir. Tam sayılarla yapılabilecek neredeyse tüm işlemler bu kütüphanede bulunmaktadır. 1991 yılında ilk defa yayınlanmış olan bu kütüphane işlemlerin hızlı olmasına odaklanmıştır ve gönüllülerin katkılarıyla neredeyse her yıl yeni özellikler, hata düzeltmeler ile gelişmeye devam etmektedir, üzerine eklentiler yapılmaktadır.

GMP kütüphanesi (özellikle tamsayı fonksiyonları) genellikle kriptografi alanında çalışmalar yapan bilim insanları tarafından aktif olarak güvenle kullanılmaktadır. Bu çalışmada da kullanılan her platformda GMP kütüphanesinin tamsayı fonksiyonlarından faydalanılmıştır.

Shanks Algoritması'nın hızlandırıcı çarpanlı yeni versiyonu çeşitli platformlarda denendi. Bu platformlar arasında dünyada en çok bilinen ve on yıllardır aktif olarak kullanılan C++, yeni fakat giderek daha da yaygın olarak kullanılan Julia programlama dili de var. Ayrıca sadece Julia ve C++'ın üzerinde çalıştığı ile CPU değil, GPU üzerinde de CUDA ile çeşitli denemeler yapıldı, fakat sonuçlar yeterince başarılı bulunmadı.

GPU üzerindeki denemelerden iyi sonuçlar alınmadı, büyük sayılar için kullanılan GMP kütüphanesinin GPU mimarisi için yazılmış resmi bir kütüphanesi olmadığından Github'daki açık kaynak kodlu projeler üzerinden gidildi. Bulunan GPU üzerinde çalışan cuGMP kütüphanesi ile CPU üzerinde çalışan GMP kütüphanesinin performansları karşılaştırıldığında, CUDA için yazılan cuGMP kütüphanesinin oldukça düşük performanslı olduğu görüldü. Yapılan testlerin sonuçları raporda da yer aldı.

C++ ve Julia karşılaştırılması için hem seri, hem paralel kodlar yazıldı. Julia'nın C ve C++ kütüphanelerini rahatlıkla çağırabilmesi sebebiyle Julia kodlarında da büyük tam sayıların temsili, büyük tam sayılarla yapılan işlemler için doğrudan GMP kütüphanesi kullanıldı.

C++ ile paralelleştirme için üzerinde 1991 yılından itibaren çalışılan MPI kullanıldı. Artık standartlaşmış bir protokol haline gelmiş olan MPI için yazılmış farklı işletim sistemlerinde çalışan birçok farklı derleyici bulunmaktadır. Paralel işlemcilerin birbirleriyle haberleşmelerini sağlayan MPI için oldukça fazla programlama dilinden çağrılabilen farklı işlevde birçok fonksiyona sahiptir.

MPI proses seviyesinde paralellliği sağlamaktadır. Aynı zamanda iplik seviyesinde paralelleştirme imkanı da sağlar. Bu çalışmada ise MPI kütüphanesinin proses seviyesinde paralelleştirme olanaklarından yararlanılmıştır.

Julia, okunabilirliği oldukça yüksek olan, dolayısıyla öğrenmesi de kolay olan, popüleritesi giderek artan yeni nesil bir programlama dilidir. An itibariyle internette en çok aranan 21. programlama dili olarak listelerde yer almaktadır. Özellikle yüksek başarılı hesaplamalar için dizayn edilmiş bu dil, paralel platformlarda C++/MPI kadar yüksek performanslı olmasa da ilerleyen zamanlarda prosesler arası iletişim daha da hızlanırsa birçok alanda C++ popüleritesine ulaşma imkanına sahip olabilir. Seri kodlar karşılaştırıldığında, Julia'nın gücü görülebilmektedir, hesaplama süresi arttıkça Julia performans açısından C++'ı geride bırakmaktadır. Bu tezin de son bölümünde karşılaştırmalar yer almaktadır.

Günümüzde giderek artan hesaplama gücü ihtiyacını giderebilmek için artık elimizde birçok farklı programlama dili, standart ve kütüphane bulunmaktadır. Bu çalışmada İTÜ UHEM'in Sarıyer makinasındaki CPU ve GPUlar kullanılarak birtakım karşılaştırmalar yapıldı. Amaç, yarı asal bir sayıyı çarpanlarına ayırmakla beraber bilgisayar dünyasındaki standartları ve yenilikleri karşılaştırarak, hesaplama imkanlarını gözden geçirmektir.



1. INTRODUCTION

Integer factorization is the task of finding the factors of a large integer. Integer factorization is an active field of research since security of many digital systems depend on the difficulty of factoring a large integer into its prime factors. Today, many systems from various areas use RSA cryptosystem for confidentiality. Confidentiality is an important concept in our modern world where illegal collection of data is very common and dangerous. Cryptosystems enable us to encrypt our data for preventing people (other than we allow) to read, change or use our data. Not only today, but also in history hiding private content and messages has always been an important issue. First known cryptosystem dates back to 1900 BC and today RSA cryptosystem is widely accepted [1].

The difficulty of integer factorization lies within the insufficient computational power of today's computers. Until quite recently, it was assumed that quantum computers were going to be much more efficient than today's computers. However, this theory is now suspicious [2]. Therefore, it can be said that integer factorization will most probably be an important problem even if we start using quantum computers. Until quantum computers become feasible and cryptographers prove that RSA cryptosystem is trustworthy in these systems, research in this area will continue. As of today, we are bound to use RSA on traditional Von Neumann Architecture computers.

With technological advances, we have more opportunities and platforms for computational studies. In this thesis, new and traditional ways of computation are discussed and compared.

1.1 Purpose of Thesis

This study focuses on more than a single aim, first one is to prove the Shanks' algorithm with a multiplier method computationally and explain how to factor a semiprime using the new algorithm. The multiplier selected from an interval whose formulations are given in following chapters accelerates Daniel Shanks' SQUFOF algorithm. The computations show that, thanks to the multiplication, even 768 bit semiprime numbers

can be factored in seconds. Another purpose is to compare young and traditional platforms of computation and their parallelization potentials.

Julia is a very young programming language focusing mostly on performance and ease of use. This new programming language attracts not only software engineers and computer scientists, but also programmers from different backgrounds with different fields of expertise. C++, the backbone of many systems shows its speed, but it is difficult for people without software engineering background to use it. In today's world, in which we have limitless data, computation lies in the heart on many scientific researches. Julia may be an alternative due its readability for people who are not comfortable with low level C++ code. In this study, a brief history of RSA, public key cryptography and integer factorization is found. Methods for breaking RSA are discussed and the history of attacks to some cryptosystems are elaborated.

2. DIFFIE-HELLMAN KEY EXCHANGE AND RSA

2.1 Public Key Cryptography and Diffie-Hellman Key Exchange

RSA Cryptosystem is a public key cryptosystem. Its name comes from its inventors: Rivest, Shamir, Adleman. The concept of public key cryptosystem was firstly discussed by Diffie and Hellman's 1976 paper named "New Directions in Cryptography" [3]. In contrast to conventional old-fashioned encryption and decryption methods, Diffie and Hellman propose a new system in which communicating parties have two keys: one public and one private. Figures below taken from "New Directions in Cryptography" show the difference of conventional cryptosystems and public key cryptosystems.

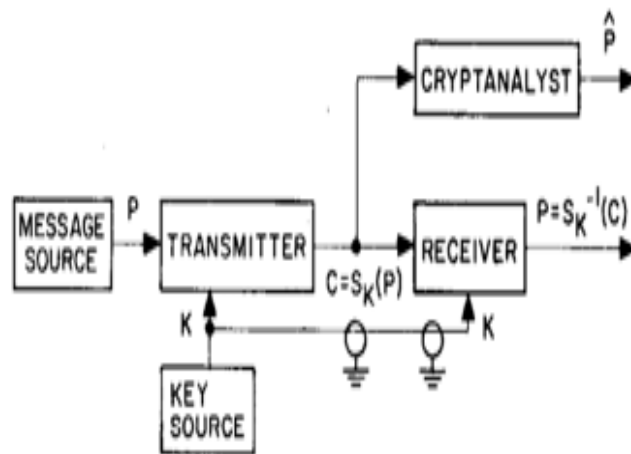


Figure 2.1: Conventional Cryptosystem with Single Key Source.

The figure above describes the conventional system: communicating parties (Alice and Bob) share one secret key: K . The message P is encrypted by K by the sender Alice using the encryption function S_K . The receiver Bob decrypts the message P using the same key K with the decryption function S_K^{-1} .

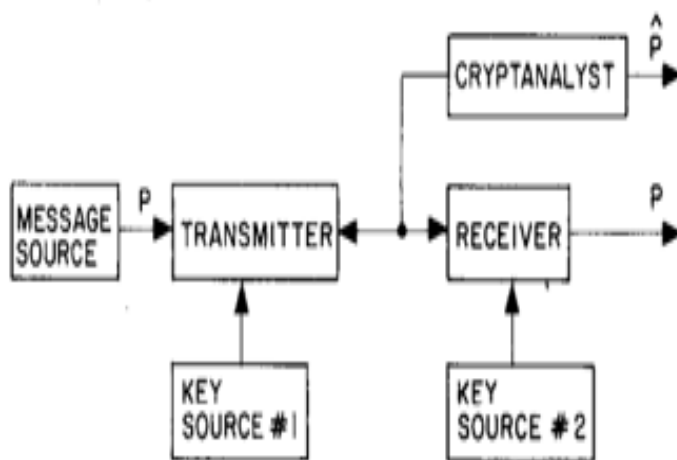


Figure 2.2: Public Key Cryptosystem.

The figure above shows the public key cryptosystem proposed by Diffie and Hellman. In this system, there are two keys: D_K is the deciphering key (private key) and E_K is the enciphering key (public key). The sender and the receiver do not have the same key source in contrast to the old-fashioned cryptosystem in Figure 2.1. The communicating parties agree on two numbers and these two numbers become their public keys. Also, both of them select an integer privately and these integers become their private keys. There are two different key sources for each communicating party. Once Discrete Logarithm Problem has a solution, Diffie-Hellman Key Exchange is no longer secure. Discrete Logarithm Problem is solved when c can be extracted from $a^c \bmod n$ (a and n are known).

Communicating parties create a key for their communication and transmit data to each other using networks. However, this kind of communication is open to Man-In-The-Middle Attacks. The scenario below shows an example for Man-In-The-Middle Attack for Diffie-Hellman Key Exchange.

Here we have Alice and Bob, who wish to exchange keys and there is Darth, a person who wants to imitate Alice or Bob (q is a global large prime) [4]. The flow of data, which is visualized in the figure below shows the Man-In-The-Middle Attack Scenario for Diffie-Hellman Key Exchange.

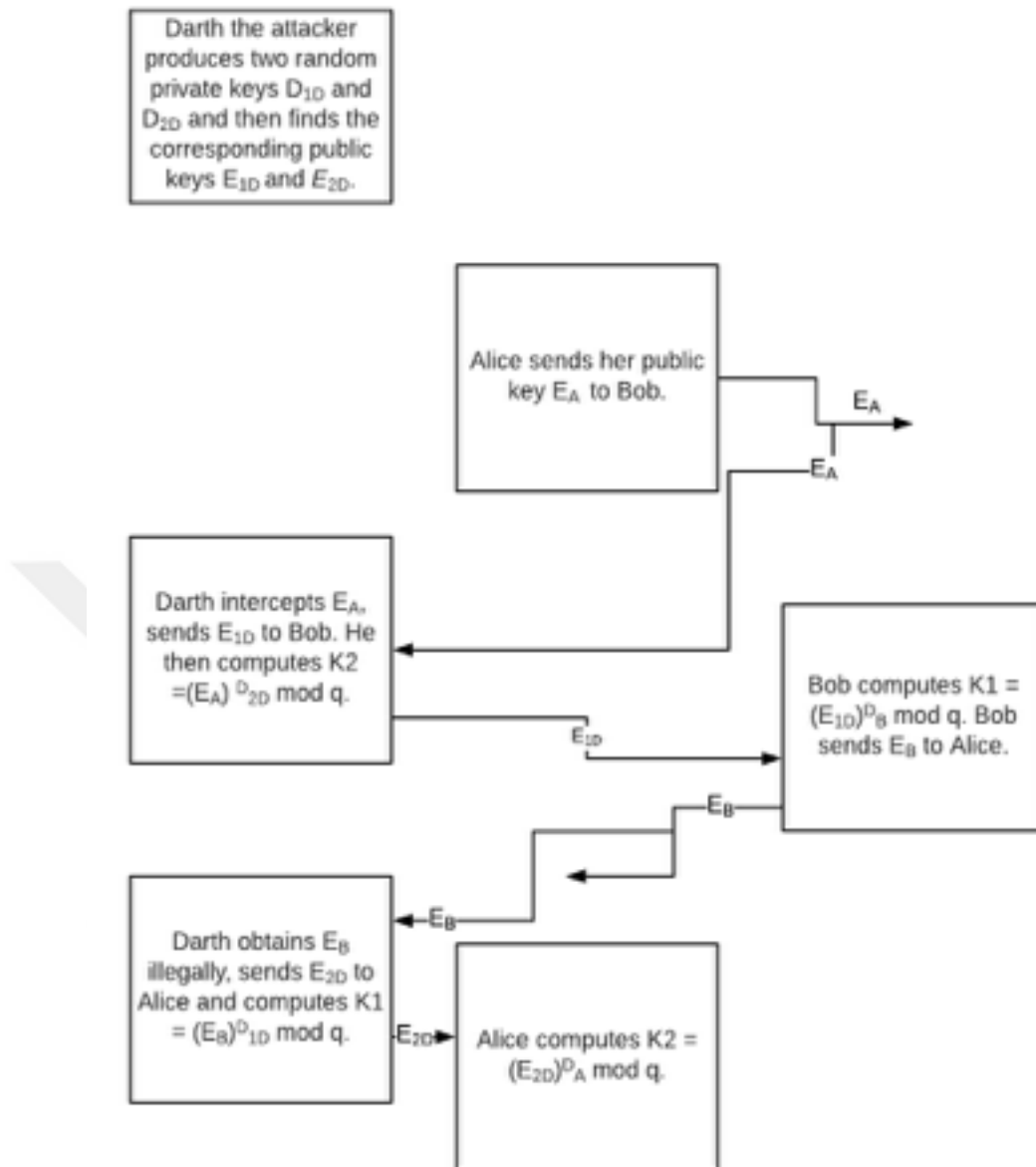


Figure 2.3: Diffie-Hellman Key Exchange Man-In-The-Middle Attack Scenario.

Now, even though Alice and Bob think they are communicating with each other, they are actually communicating with Darth. The eavesdropper Darth can pretend to be Alice or Bob. He can easily read private messages written by Alice and Bob. Darth now shares the key $K1$ with Bob and $K2$ with Alice. Alice and Bob are unaware of the situation. Below, you can see the flow of communication.

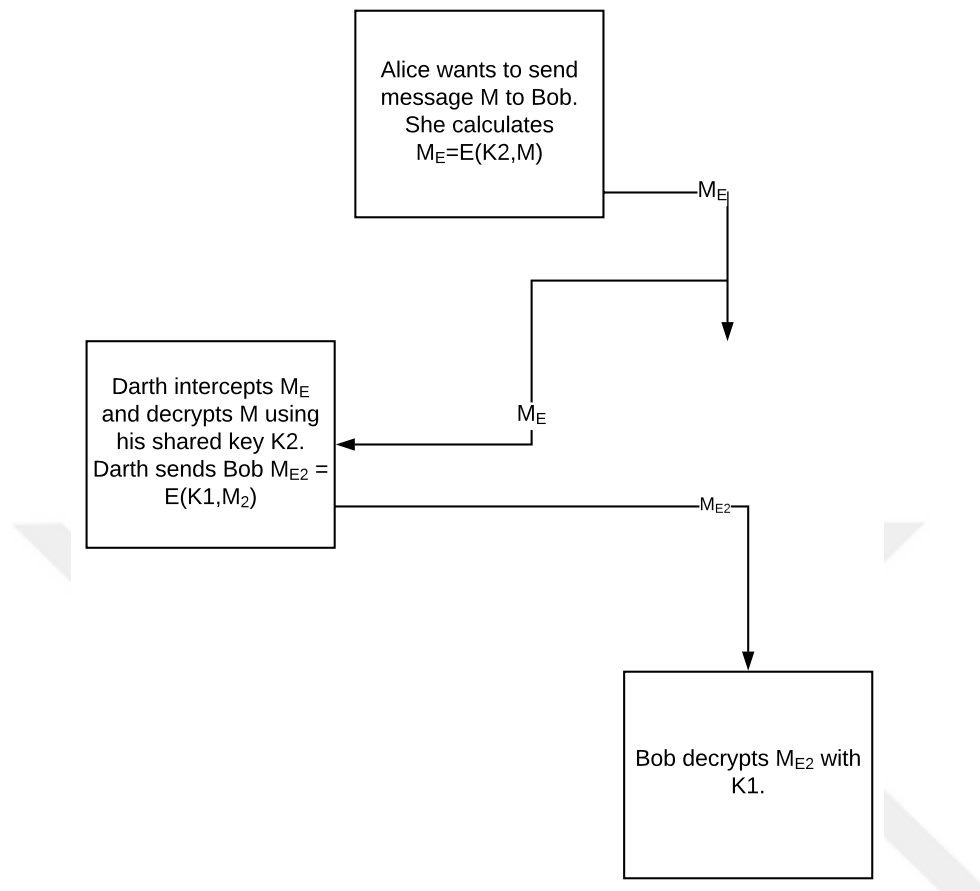


Figure 2.4: Communication Interception After Man-In-The-Middle Attack.

The flow shows that Darth can read any message sent to Bob by using his key with Alice. Also, he can imitate Alice and send messages to Bob as if he is Alice. He can alter the original message from Alice or he can send a completely different message. This security flaw in key exchange can be fixed using authentication. If communicating parties authenticate each other, Man-in-the-Middle Attack is no longer a problem. There are different ways of authenticating each other. Digital signatures and public-key certificates can be used for solving this issue [4].

Let's take a look at digital signatures and public-key certificates which are widespread solutions for communicating parties to acknowledge each other. Digital signatures are a good way of preventing imitators like Darth in the example above. Their benefits are not only preventing imitators but also preventing dispute among communicating parties. One of the popular digital signatures, El Gamal Digital Signature keeps date/time, content of the message at the time of signature within. In case of a dispute, a neutral third party can expose the truth.

The National Institute of Standards and Technology proposed Digital Signature Standard based on El Gamal and Schnorr and today it is widely used as a standard in order to prevent claims of forgery. Public-key certificates consist of public key, owner identifier and a block signed by a trusted third party. Third party should be a trustworthy organisation like governments or international firms [4]. The certificate makes sure that the owner has the corresponding private key of the public key. Diffie-Hellman Key Exchange is often explained visually with the colour analogy, which is shown below [5]:

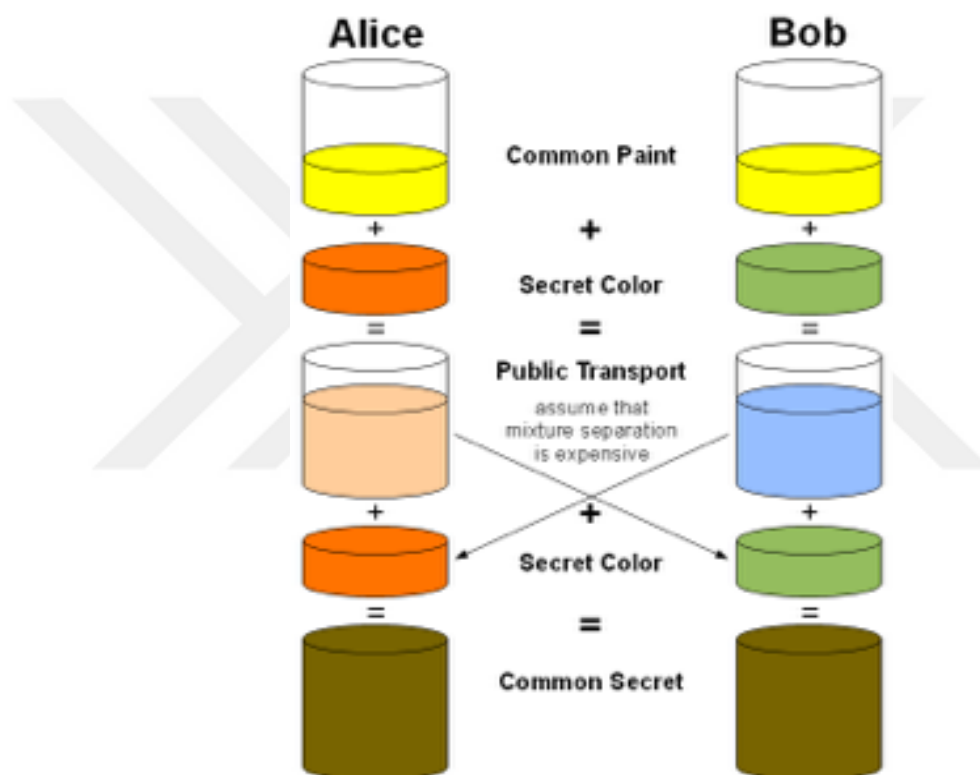


Figure 2.5: Diffie-Hellman Key Exchange Colour Analogy.

The illustration points out that communicating parties obtain the same secret in the end of the process. More detailed flowchart can be seen below: (p is a large prime and α is the primitive root modulo p . Assume we have the equation $g^k \equiv a \pmod{n}$, g is called the primitive root modulo n if $\gcd(a,n) = 1$. The number k is the discrete logarithm of a to the base g modulo n . [6])

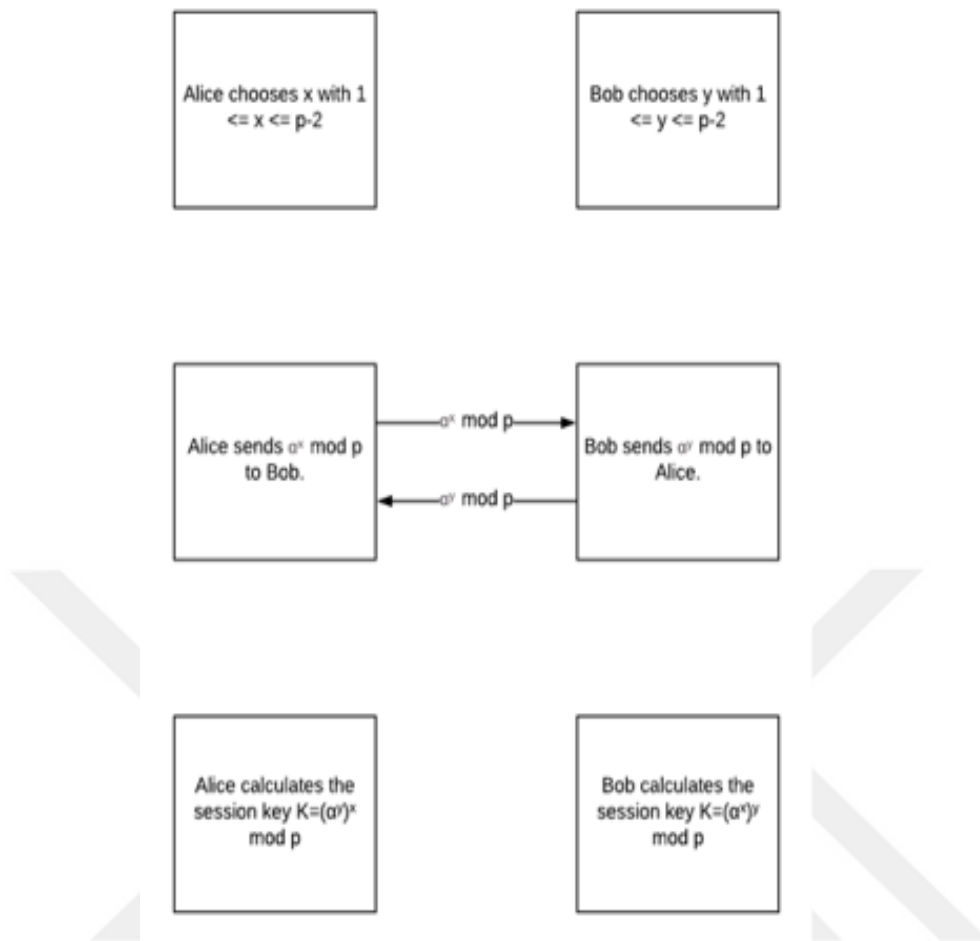


Figure 2.6: Diffie-Hellman Key Exchange Flow.

Diffie-Hellman Key Exchange is reliable, because calculating discrete logs is very difficult. If someone can easily compute discrete logs, this algorithm is no longer reliable. If the eavesdropper Darth can compute discrete logs, he can obtain K and break the system. If Darth can compute y from α^y (α is public as stated above) and x from α^x , the system is no longer useful. However, as of today, Discrete Log Problem has no solution.

2.2 RSA Cryptosystem

In 1978, Rivest, Adleman and Shamir proposed a system for secure and private electronic mail transfer. They state that the public key cryptography system invented by Diffie and Hellman lies in the heart of their newly-proposed RSA system. To

understand their system, one has to know some mathematical theorems and algorithms, namely Euler's Theorem and Euclidean Algorithm.

Euler's Theorem says that if $n = p * q$ (p, q prime) and $gcd(a, n) = 1$, then

$$a^{\phi(n)} \equiv 1 \pmod{n} \quad (2.1)$$

Note that: $\phi(n) = (p - 1) * (q - 1)$.

In order to find decrypting key d from encrypting key e , we should find the modular multiplicative inverse of e modulo $(q - 1) * (p - 1)$. The modular multiplicative inverse is found using Extended Euclidean Algorithm. We have the following equation:

$$d * e + (p - 1) * (q - 1) * y = 1 \quad (2.2)$$

When we reduce Equation 2.2 to modulo $(p - 1) * (q - 1)$, we get

$$d * e \equiv 1 \pmod{(p - 1) * (q - 1)} \quad (2.3)$$

Extended Euclidean Algorithm solves the equations of form Equation 2.3. Equation 2.3 is derived from Equation 2.2, which is our initial problem: finding decrypting (private) and encrypting (public) keys.

Using the theories and algorithms above, Rivest, Shamir and Adleman developed RSA cryptosystem in 1978. The figure below explains the algorithm visually.

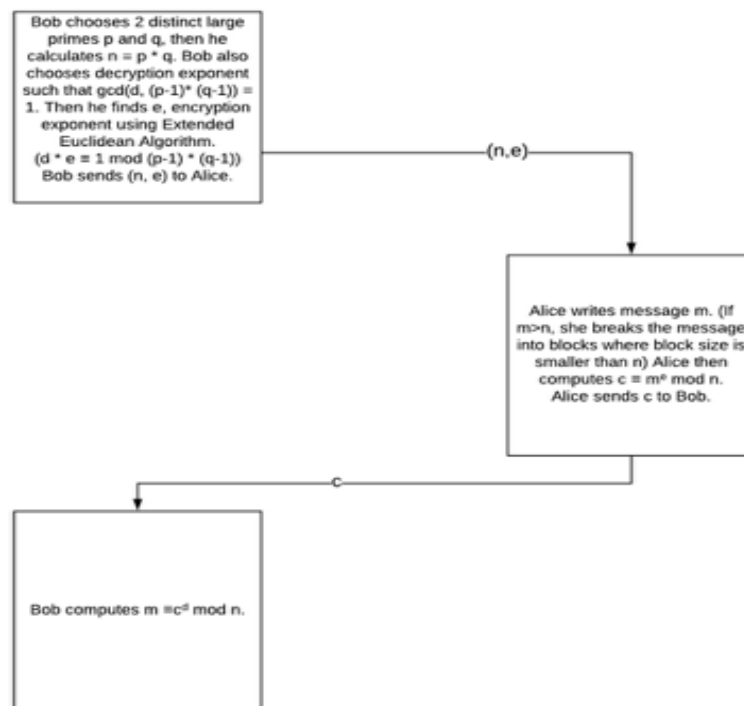


Figure 2.7: RSA Flow

In their paper “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Rivest, Shamir and Adleman explain how to use the proposed system effectively. They offer solutions to problems related to difficulties of putting the cryptosystem into practice. Since computational power has limitations, they explain the “exponentiation by repeated squaring and multiplication” [7] procedure for calculating $M^e \bmod n$ effectively. Calculating $M^e \bmod n$ requires $2 * \log_2 e$ multiplications and $2 * \log_2 e$ divisions using the procedure. Let $e_k e_{k-1} \dots e_0$ be e 's binary representation and $C = 1$. The procedure is [7]:

Step 1. Repeat steps 1a and 1b for $i = k, k - 1, \dots, 0$:

Step 1a. Set $C = C^2 \% n$.

Step 1b. If $e_i = 1$, then $C = (C \cdot M) \% n$.

Step 2. Stop. C is the encrypted form of M .

RSA cryptosystem requires two large prime numbers p and q . n , which is equal to $p * q$, has to be computationally expensive to factor, since correct factorization of n will break the system, nobody should be able to find p and q from n . In order to find a large prime of x digits, prime number theorem says that approximately $(\ln 10^x) / 2$ numbers should be tested before finding a prime number [8]. If we want a 10 digit prime number, we should test approximately $\ln(10^{10}) / 2 \approx 12$ numbers.

For primality testing, Rivest, Shamir and Adleman suggest using a probabilistic algorithm by Solovay and Strassen. This algorithm selects a random number a from a uniform distribution on $\{1, \dots, b - 1\}$, and tests the following condition [9]:

$$\gcd(a, b) = 1 \text{ and } J(a, b) \equiv a^{(b-1)/2} \bmod b \quad (2.4)$$

$J(a, b)$ is the Jacobi symbol [7].

Apart from Solovay-Strassen Primality Test, there are other ways for detecting primality: Fermat Primality Test and Miller-Rabin Primality Test. These algorithms can be used while selecting p and q , too.

For testing primality of n using Fermat Primality Test, we should choose a random integer a with $1 < a < n-1$. If the following condition holds, n is probably prime [10]:

$$a^{n-1} \equiv 1 \bmod n \quad (2.5)$$

Fermat Primality Test is highly accurate for large n . However, it guarantees compositeness, not primality.

For conducting Miller-Rabin Primality Test for n , we should select a random integer a which is greater than 1 and less than $n - 1$. We also have an odd m which satisfies $n - 1 = 2^k * m$. Below is the algorithm [10]:

Step 1. Compute $b_0 \equiv a^m \pmod n$.

Step 2. If $b_0 \equiv 1 \pmod n$ or $b_0 \equiv -1 \pmod n$, stop and declare that n is PROBABLY prime.

Step 3. Repeat steps 3a, 3b and 3c for $i=1, 2, \dots, k-1$:

Step 3a. Calculate $b_i \equiv b_{i-1}^2 \pmod n$.

Step 3b. If $b_i \equiv 1 \pmod n$, stop and declare that n is composite.

Step 3c. If $b_i \equiv -1 \pmod n$, stop and declare that n is PROBABLY prime.

Step 4. If $b_{k-1} \not\equiv -1 \pmod n$ then n is composite.

For selecting secure p and q , RSA inventors also state that p and q should have the same number of digits and both $p-1$ and $q-1$ should have very large prime factors. Also, $gcd((p - 1), (q - 1))$ should be small [7].

Another number that should be selected is the decryption exponent d . We should select d and then derive the encryption exponent e using the Extended Euclidean Algorithm explained before. The decryption exponent d must be coprime to $\phi(n) = (p - 1) * (q - 1)$, so $gcd(d, \phi(n)) = 1$ must hold. Any prime number larger than $maximum(p, q)$ is okay, the primality can be tested with one of the test described above (Solovay-Strassen, Miller-Rabin, Fermat). After choosing d , the Extended Euclidean Algorithm returns e , the encryption exponent.

After describing the flow of RSA cryptosystem in detail, we should also mention the possible security flaws and ways of breaking the system. Rivest, Shamir and Adleman handle the security under four subtitles: Factoring n , Computing $\phi(n)$ without Factoring n , Determining d Without Factoring n or Computing $\phi(n)$, Computing d in Some Other Way.

Factoring n , which is very difficult as mentioned before, breaks the system. However, since n is equal to multiplication of two large primes, it is computationally exhaustive and takes a lot of time. Some factoring algorithms exist, but they are not fast and feasible enough to conclude that RSA is not safe. At the time of publication of RSA cryptosystem, the fastest algorithm computed factors of n in $(\ln(n))^{\sqrt{\ln(n)/\ln(\ln(n))}}$ steps [7]. As of today, Shor's Algorithm, which is named after its creator Peter Shor, factorizes large numbers very quickly and efficiently, however it is a quantum algorithm, it runs on quantum computers. Peter Shor states in his paper "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer" that the quantum factoring algorithm has asymptotically $O((\log n)^2(\log \log n)(\log \log \log n))$ complexity on a quantum computer, along with a polynomial amount of post-processing complexity on a von Neumann Architecture computer which is required for conversion of output [11]. General Number Field Sieve is the fastest factorization algorithm for classical computers. Its complexity for factoring a semiprime integer n (consisting of $\lfloor \log_2 n \rfloor + 1$ bits) is $O(\exp[c * (\log n)^{1/3} * (\log \log n)^{2/3}])$ where c is a constant which can take three different values [12].

$\phi(n)$ must be kept secret because the decryption exponent d can easily be derived using Extended Euclidean Algorithm (encryption exponent e is public). As explained before, $d * e \equiv 1 \pmod{\phi(n)}$.

Determining decryption exponent d without factoring n or computing $\phi(n)$ is not easier than factoring n , therefore it is infeasible. In order to compute d in some other way, one has to find a solution to Discrete Logarithm Problem, which has no feasible solution as of today. Deriving d from $c^d \pmod n$ (c is the encrypted message, so both c and n are public) is an example of Discrete Logarithm Problem.

In his paper "Twenty Years of Attacks on the RSA Cryptosystem" Dan Boneh summarized the possible kinds of attack under five topics: Factoring Large Semiprimes, Elementary Attacks, Low Public Exponent, Low Private Exponent and Implementation Attacks [13].

Since $n = p * q$ and $d * e \equiv 1 \pmod{(p-1)(q-1)}$, (n and e are public), one can derive the private/decrypting exponent d after obtaining p and q . See the flow below

for breaking the cryptosystem using factorization of large integer $n = p * q$. Note that Darth is an eavesdropper who wants to read the messages in a communication without permissions from communicating parties.

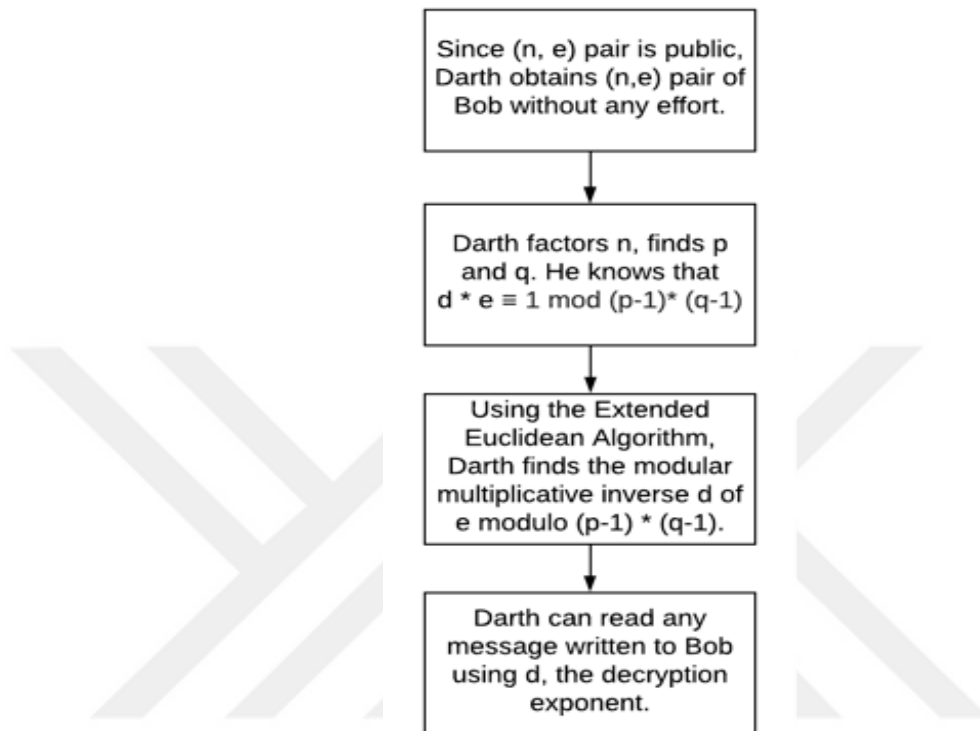


Figure 2.8: RSA Flow When an Eavesdropper Factors n .

RSA cryptosystem relies on the difficulty of factoring large integers into their prime factors. The flow shows that if an eavesdropper Darth could factor n into p and q , RSA would no longer be secure. As mentioned above, the current factorization algorithms for classical computers are not sufficient for this task and quantum computers are owned and used by limited number of institutions.

Elementary attacks on RSA cryptosystem spoils users' mistakes while using and/or establishing the system. For example, instead of generating different $n = p * q$, using a fixed n causes security flaws. Assume a central authority provides its user i with a unique pair e_i, d_i , however users share the same $n = p * q$. A message to user x $m_x = m^{e_x}$ (message is m , e_x is user x 's public key) can be read by user y , because user y can extract d_x easily since user y knows $p, q, p-1, q-1$ and e^x . For "blinding" attack,

using one-way hash to the message before signing is a solution, therefore it is no threat anymore [13].

Low private exponent d is an advantage regarding computation time, however it is a serious threat to security. Michael Wiener proved in 1988 that for $n = p * q$ with $q < p < 2q$ when $d < \frac{1}{3}n^{\frac{1}{4}}$, one can easily find d [13].

Low public exponent can be a security threat, too. However it is not as dangerous as using a low private exponent. There are many kinds of attacks, Coppersmith's Theorem is the base theorem of the most powerful attacks in case of a low public exponent [14]. The theorem is:

“Let n be an integer and $f \in \mathbb{Z}[x]$ be a monic polynomial of degree d . Set $x = n^{\frac{1}{d}-\epsilon}$ for some $\epsilon \geq 0$. Then, given (n, f) Darth can efficiently find all integers $|x_0| < x$ for which $f(x_0) = 0 \text{ mod } n$ holds. LLL algorithm used here dominates the execution time [15].”

Note that LLL is a basis reduction algorithm.

Implementation attacks focus on the vulnerabilities of the RSA cryptosystem when it is put in practice in different kinds of practical fields. Various implementations are adapted in various fields. For example, timing attacks can be used for attacking financial systems. Dan Boneh describes a timing attack as measuring the time it takes a smart card to perform a RSA decryption or signature and then derive decryption exponent d from this information. The repeated squaring algorithm is used for timing attacks. The binary representation of d is $d_n d_{n-1} \dots d_0$. The repeated squaring algorithm computes $c \equiv m^d \text{ mod } n$ in the end:

Step 1. Set $z = m, c = 1$.

Step 2. Repeat steps 2a, 2b $i=0,1,2, \dots, n$:

Step 2a. If $d_i = 1, c \equiv c * z \text{ mod } n$

Step 2b. $z \equiv z^2 \text{ mod } n$

Step 3. Return $c \equiv m^d \text{ mod } n$

Note that $d_0 = 1$ because d is a prime number greater than 2, therefore it must be an odd number.

m is the signature produced by the smartcard. Darth, who wants to break the system, asks for many signatures, measures the generation time for each signature m (T) and measures the time for the smart card to compute $m * m^2 \bmod n$ for each signature m (t). For each $d_n d_{n-1} \dots d_1$, starting from d_1 to d_n Kocher inspects the behaviours of T and t , and he concludes that when T and t are highly correlated, the value of d_i is 1, when they act independently, the value of d_i is 0 [16].

There are some ways to prevent this kind of attack. Boneh informs of two precautions: first one is to add delays which will result in constant time of modular exponentiation. Second precaution is blinding by Rivest, Boneh summarizes this method in his survey: The smart card selects a number $r \in \mathbb{Z}_n^*$ and calculates $m' \equiv m * r^e \bmod n$ before the decryption of m . After this, the smart card calculates $c' \equiv (m')^d \bmod n$ and sets $c \equiv \frac{c'}{r} \bmod n$. This results in the application of d to a random message m' which is not known by Darth [13].

Another kind of Timing Attack is called Random Faults. Random Faults Attack takes advantage of usage of CRT. CRT is used in calculation of $m^d \bmod n$ because usage of CRT speeds up the operation. CRT is used in the following way:

d_p and d_q are integers such that $d_p \equiv d \bmod (p - 1)$ and $d_q \equiv d \bmod (q - 1)$. Bob computes $c_p \equiv m^{d_p} \bmod p$ and $c_q \equiv m^{d_q} \bmod q$. Then he calculates $c \equiv T_1 * c_p + T_2 * c_q \bmod n$ where

$$T_1 = \begin{cases} 1 \bmod p \\ 0 \bmod q \end{cases} \text{ and } T_2 = \begin{cases} 0 \bmod p \\ 1 \bmod q \end{cases} \quad (2.6)$$

Boneh, DeMillo and Lipton discovered that while generating the signature c , even a simple one-bit mistake can cause Darth to compute p and q from n . Padding mechanisms are a way of preventing Random Faults attack.

Bleichenbacher's Attack on PKCS 1 is an attack which spoils a flaw in padding mechanism of the old PKCS. Bleichenbacher discovered the potential threat and now this mechanism is no longer used.

After explaining more than a decade of attacks on the RSA cryptosystem, Dan Boneh states that none of these attacks can break the cryptosystem. After years of widespread usage and countless attacks, RSA still dominates the industry.



3. INTEGER FACTORIZATION

3.1 Binary Quadratic Forms and Groups

An equation of form $f(x, y) = ax^2 + bxy + cy^2$ is called a binary quadratic form. We have the discriminant $\Delta = b^2 - 4ac$ for each binary quadratic form. The following conditions are compulsory:

- $\Delta \equiv 1 \pmod{4}$ or $\Delta \equiv 0 \pmod{4}$
- $b \equiv \Delta \pmod{2}$

If a binary quadratic form is primitive, then $\gcd(a, b, c) = 1$ [17]. Binary quadratic forms form a group which is called class group. A group $(S, *)$ consisting of $a_0, a_1, a_2, \dots, a_n$ with operation $*$ has four properties:

- $a_x * a_y \in S$ where $0 \leq x \leq n$ and $0 \leq y \leq n$
- $(a_x * a_y) * a_z = a_x * (a_y * a_z)$ where $0 \leq x \leq n$ and $0 \leq y \leq n$ and $0 \leq z \leq n$
- There exists an identity e such that $e * a_x = a_x$ and $a_x * e = a_x$ where $0 \leq x \leq n$ and $0 \leq y \leq n$
- Each element has an inverse such that $a_x * a_y = a_y * a_x = e$ where $0 \leq x \leq n$ and $0 \leq y \leq n$

The order of a group is equal to the number of elements in a group.

$(\mathbb{Z}_6, +)$ is an example of a group. This group consists of $\{0, 1, 2, 3, 4, 5\}$, its order is 6 and it satisfies the conditions above:

- For example: $1 + 3 \equiv 4 \pmod{6}$ and $4 + 2 \equiv 0 \pmod{6}$.
- For example: $(4 + 2) + 5 \equiv 5 \pmod{6}$ and $4 + (2 + 5) \equiv 5 \pmod{6}$
- The identity is equal to 0. For example: $3 + 0 \equiv 3 \pmod{6}$
- For example: $1 + 5 \equiv 0 \pmod{6}$ and $2 + 4 \equiv 0 \pmod{6}$

Let a be an element in a group G . The smallest n such that $a^n = e$ (e is identity) is called the order of a . For example, assume that we have a group $(\mathbb{Z}_{2017}^*, *)$. The order of this group is 2016. We know that $2016 = 2^5 * 3^3 * 7$. The order of 1999 is 1008 since $1999^{1008} \equiv 1 \pmod{2017}$ and $1008 = 2^4 * 3^2 * 7$. The fact that 2016 is divisible by 1008 is no coincidence. Any element in group G has order dividing the order of G . Since binary quadratic forms are groups, the features explained above hold for them, too. For my purpose, which is factoring large integers, binary quadratic forms can be used. When we select $\Delta = n$, which is a semiprime and p and q are its factors, we can obtain a Binary Quadratic Form.

3.2 Shanks' SQUFOF Algorithm and Improving SQUFOF

Daniel Shanks developed an integer factorization algorithm back in the 1970s using Binary Quadratic Forms. However, as of today, Shanks' algorithm is not used due to its non efficiency in practice. The intention is to take Shanks' algorithm a step further using the multiplier method found by Nari, Ozdemir and Yaraneri and implement the new parallel algorithm using cutting-edge technology devices. Also, since the parallel codes will run on different environments executing different codes, we will be able to see the advantages and disadvantages of each platform.

Assume we have the binary quadratic form $f(x, y) = ax^2 + bxy + cy^2$ and $\gcd(a, b, c) = 1$. The discriminant D of f is $D = b^2 - 4ac$. For any given integer m , which represents the binary quadratic form f ($m = ax^2 + bxy + cy^2$), the equation $4am = (2ax + by)^2 - Dy^2$ holds. It can be seen that when $D < 0$, m and a have the same sign. Given two binary quadratic forms f and f' :

$$f(x, y) = ax^2 + bxy + cy^2 = (x, y) \begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (3.1)$$

and

$$f'(x, y) = (a', b', c') = a'x^2 + b'xy + c'y^2 \quad (3.2)$$

If a 2x2 matrix $A = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$ with $\det(A) = 1$ satisfies the condition

$$\begin{pmatrix} a' & b'/2 \\ b'/2 & c' \end{pmatrix} = \begin{pmatrix} \alpha & \gamma \\ \beta & \delta \end{pmatrix} \begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix} \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix},$$

then f and f' are congruent. If there is a prime number s which divides the discriminant D , then binary quadratic forms which look like (s, rs, c) can be produced. This type of binary quadratic forms is called ambiguous. A reduced binary quadratic form is obtained when $0 < b < \sqrt{D}$ and $\sqrt{D} - b < 2|a| < \sqrt{D} + b$. There exists finite number of reduced forms for each D . Also, there exists a congruent reduced form for $f = (a, b, c)$ which has the same discriminant D . For negative discriminants, there is only one reduced form, however, there may be more than one reduced forms for positive discriminants. Two reduced forms $f = (a, b, c)$ and $f' = (c, b', c')$ belong to the same class, which means that they are adjacent, if $b' + b \equiv 0 \pmod{2c}$. Reduced form f has two neighbours (f' and (x', b'', a)) and its neighbours are congruent under the following matrix transformation [18]:

$$\begin{pmatrix} 0 & -1 \\ 1 & \frac{b+b'}{2c} \end{pmatrix} \quad (3.3)$$

The set of reduced forms with its neighbours form a cycle and two reduced forms are congruent if and only if they are in the same cycle. Since there are finite number of reduced forms, in the end, the first reduced form will be obtained again after moving in the same direction [18].

In order to factor the semiprime $n = p^*q$, the information above can be used. Nari, Ozdemir and Yaraneri proved that given integers c and c' , the order of prime forms (p, p, c) and (q, q, c') is equal 1 if discriminant is equal n and assuming $p < q$, (p, kp, c') is a reduced binary quadratic form which has discriminant n . The purpose of the new algorithm is to find the neighbours of $(1, b, c)$ using matrix transformation (3.3) until (p, kp, c') is found. Note that $(1, b, c)$ is the reduced form of $(1, 1, (n-1)/4)$, which is the identity element of the class. When (p, kp, c') is found, p , which is a factor of n is no more secret. The execution time to reach (p, kp, c') depends on the number of elements

in the cycle. If the cycle consists of moderate number of elements, the factorization of n can be completed in polynomial time [18].

Below is the pseudocode of the algorithm used in this thesis derived from Nari, Ozdemir and Yaraneri's algorithm. Note that we have a binary quadratic form consisting of (a, b, c) . For parallelization, MPI is used. MPI enables us to run the code with different r values on different hosts. For calculation with large integers, GMP is used. "mpz_t" type is GMP's type for handling large integers. The parallel C++ pseudocode for SQUFOF with multiplier " r " can be found below:

```
function calc_c( a, b, c, delta) {
    c = floor((b^2 - delta) / (4a))
}

function calc_step(c, delta, b, step, delta_s){
    step = floor((delta_s + b) / 2c )
}

function calc_b(step, b, c){
    b = 2 * step * c - b
}

function set_initial_delta_and_b_and_c(delta, b, c){
    tmp_4 = delta % 4
    delta_sqrt = floor(sqrt(delta))
    tmp_2 = delta_sqrt % 2
    if tmp_4 == 1
        b = tmp_2 == 0 ? delta_sqrt - 1 : delta_sqrt
    elseif tmp_4 == 0
        b = tmp_2 == 1 ? delta_sqrt - 1 : delta_sqrt
    else
        delta = 4*delta
        delta_sqrt =(floor(sqrt(delta))
        tmp_2 = delta_sqrt % 2
```

```

        b = tmp_2 == 1 ? delta_sqrt - 1 : delta_sqrt
    end
    (delta,b, (floor((b^2 - delta) / 4)))
}

int loop(logn, a, c, delta, b, step, delta_s, first_delta, r){
    res = 0
    while loop_cnt < (logn / 5)
        calc_step(step, c, delta, b, delta_s)
        a = c
        calc_b(step, c,b)
        calc_c(a, b, c, delta)
        temp = gcd(a, first_delta)
        if ((temp != 1 && temp != 2 && temp != 4 && temp != r) || a==1)
            res = 1
            break
        end
    end
    res
}

function main() {
    int size = <NUMBER OF PROCESSORS>
    int rank = <PROCESSOR ID>
    delta = <NUMBER TO BE FACTORED>
    first_delta = delta
    r = <MULTIPLIER FOR DELTA>
    while (true){
        tmp = r% 4
        if (tmp == 0 )
            break;
        r = r+ 1;
    }
}

```

```

first_r = r
int global_sum = 0;
int loop_res;
int lc = 1;
while (global_sum == 0) {
    r = first_r + 100 * rank * lc;
    delta = first_delta * r
    set_initial_delta_and_b_and_c(delta, b,c);
    a = 1
    delta_s = sqrt(delta);
    logn = log(delta, 2);
    loop_res = loop(logn, a, c, delta, b, step, delta_s, first_delta, r);
    MPI::COMM_WORLD.MPI::Comm::Allreduce(&loop_res, &global_sum,
1, MPI_INT, MPI_SUM);
    lc++;
}
return 0;
}

```

In a parallel environment using x different processors, we will have x different “ r ”s. Choosing r is very important, because a “good” r can result in factoring the number in few steps. This will be further elaborated later. After processors execute the loop in the “int loop” function for at most $\log(n)/5$ times, they return an integer. If this integer is equal 1, then we stop processing, because this means a factor is found. When the integer returned from the “int loop” function equals 0, then we find a different r and execute the “int loop” function again.

GMP library is used in the code. In its official website, GMP is described as an open source and free library for high precision arithmetic on rational numbers, integers and floating-point numbers. GMP has no limits as long as there is available memory on the host machine [19].

The fact that GMP has no precision limits (when the machine has available memory) makes it very appealing for computational engineers and cryptographers.

As stated, code above achieves parallelism through MPI which provides communication mechanisms among the CPUs. In the code above, the only communication performed is:

```
MPI::COMM_WORLD.MPI::Comm::Allreduce(&loop_res, &global_sum, 1,
MPI_INT, MPI_SUM);
```

This command is executed after all processors finish the int loop function. If a processor has the result loop_res equal 1, it means that a factor is found. The line above sums up loop_res from each processor and saves the value into global_sum variable. When global_sum is greater than or equal to 1, we can say that we find a factor.

MPI's biggest problem is communication, even if this code executes MPI::Comm::Allreduce function for not so many times, communication is a burden. Also, MPI is just for CPUs, however GPUs are very suitable for time-consuming computations. NVIDIA describes CUDA as a platform which enables concurrency using the power of GPUs. NVIDIA asserts that using CUDA speeds computation up through parallelization [20].

However, there is no proper, up-to-date GMP BigInt library available for CUDA. There are some libraries, but benchmarks show that even addition operation using them is up to 300 times slower than performing addition on CPU using GMP. Below, you can see the benchmarks of cuGMP which can be found on Github. (<https://github.com/trubus/cuGMP>).

Table 3.1: GMP cuGMP Comparison.

Operation	Iteration	OperandSizeBits	GMPMicrosecs	cuGMPMicrosecs
+	0	1024	2	289
+	1	1024	0	303
+	2	1024	0	290
+	3	1024	0	310
+	4	1024	1	300
+	5	1024	0	282
+	6	1024	0	296
+	7	1024	0	277
+	8	1024	0	277

However, a GPU code for Shanks' algorithm was written using cuGMP in order to have no doubts. The results prove that cuGMP is not favourable.

Another platform I used is Julia. Julia's popularity is rising every year due to its advantages [21]. Compared to C++ which I used with MPI, Julia is very easy to work on. It uses C language's GMP library, so there is no need to search other libraries which may not be reliable and up-to-date. Another advantage of Julia is that it is very fast. Figure 3.1 below shows a comparison of Julia with other languages in terms of speed. The figure is taken from Julia Language's official website [22]:

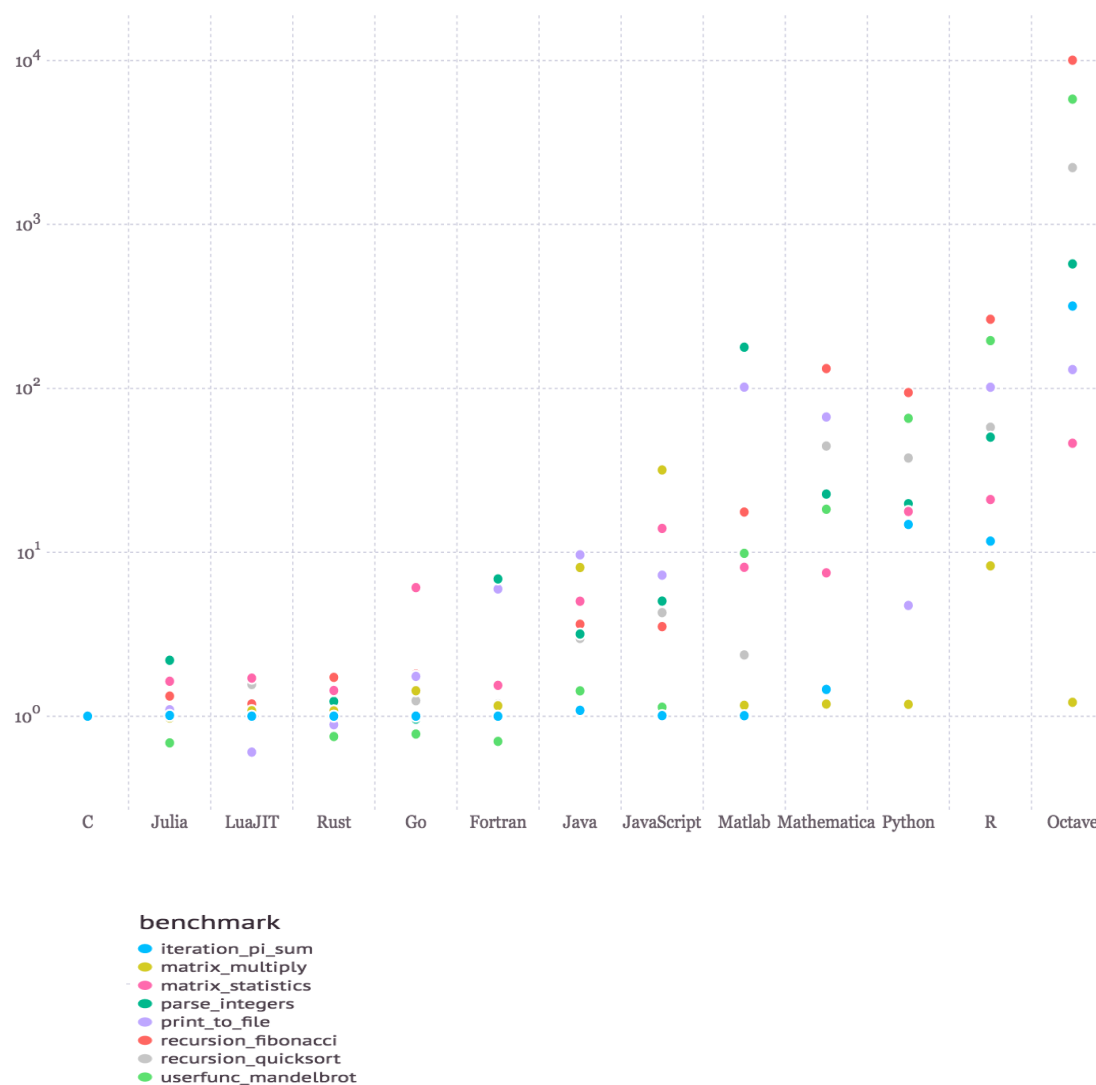


Figure 3.1: Julia Performance Comparison.

Julia also supports parallelism effectively. Three levels of parallelism offered by this language are:

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

I used Multi-Core Processing since I ran my code on multiple CPUs of UHEM's Sariyer Cluster. In contrast to MPI/C++, there is no explicit message passing in Julia's multi-core processing. I used a distributed for loop in order to achieve parallelization. There are other ways of achieving parallelism, too. The Distributed module provides different data structures and functions for parallel and distributed computations. Below, you can see the Julia pseudocode used for factoring RSA numbers. Note that `calc_c`, `calc_step`, `calc_b`, `set_initial_delta_and_b_and_c` and `loop` functions' pseudocodes are the same.

```
function main()
    first_delta = <NUMBER TO BE FACTORED>
    r = <MULTIPLIER FOR DELTA>
    global_sum = 0
    iter = 1;
    while r % 4 != 0
        r = r+1
    end
    while global_sum == 0
        global_sum = @distributed (+) for i = 1:nprocs()-1
            rr = r + 100 * (myid() - 1) * iter + 1
            (delta,b,c) = set_initial_delta_and_b_and_c(BigInt(first_delta
                * rr))
            delta_s = BigInt(floor(sqrt(delta)))
            loop(log(delta), BigInt(1), b, c, delta, delta_s,first_delta, rr)
        end
        iter = iter + 1
    end
end
end
```

The @everywhere macro is used for making functions visible to other processors other than the master processor. The setprecision(5000) function sets the precision to 5000 bits for operations with BigInt data structure. The @distributed macro is for creating a distributed loop, after each loop iteration, the results returned are added because of the (+) sign.

A programmer can easily recognise the difference in readability levels of each code. Julia is not as fast as MPI/C++, however, it is more practical to write and read Julia code. I will share the benchmarks later in this thesis.

3.3 Taking Shanks' Algorithm a Step Further

As I have stated earlier in this thesis, my main intention was to find a mechanism for factoring a large integer. Previously, Nari, Ozdemir and Yaraneri developed an extension for Shanks' Algorithm as explained before. They assert that given $n = \alpha * \beta$ ($\alpha \neq \pm 1$, α and β are integers and coprime such that $b \neq 0$) and a prime number c , there exists an integer t and interval I such that:

$$t := \alpha^2 \beta^2 (2c + b)^2 \quad (3.1)$$

and

$$I := \left(\frac{(\sqrt{t+1} - 1)^2}{\beta^{4n}}, \frac{(\sqrt{t+1} + 1)^2}{\beta^{4n}} \right) \quad (3.2)$$

Ozdemir and Yaraneri state that an integer “ r ” selected from this interval accelerates Shanks' Algorithm and enables us to factor n . We can only test this hypothesis with previously-factored integers since α and β (factors of n) are used for calculating the interval. My aim is to find a connection between successful “ r ”s and n in order to factor other semiprime numbers which were not factored. $r \% 4$ must be equal 1.

The table below shows the intervals. Note that for RSA-100 and RSA-110,

$c = 7987998710999017000000000013700000000045300003090000000247$,

for RSA-120, RSA-129, RSA-130 $c =$

7987998710991231231231390170000000001370000000045300003090000000339.

We have two intervals because we have two prime factors. Note that “Interval 1”s are calculated with the smaller factor. As the number n increases, the factors increase, prime number c should increase, too.

Table 3.2: Calculated Interval Beginning and Ends.

	Begin-1	End-1	Begin-2	End-2
R	2417405370274	2417405370274	2694774596120	269477459612
	6784185142961	6784185142961	2089875037806	020898750378
S	1434389953507	1434389953507	3443402222279	063443402222
	0304637894885	0304637894885	4606674830286	279460667483
A	3565663871101	3565663871101	3172975563977	028631729755
	6334933840197	6334933840197	3383719340914	639773383719
-	3068447111795	3068447111795	2394269467385	340914239426
	9444807568022	9444807568022	8812260929412	946738588122
1	9869023958261	9870617784977	9348788885949	609294129350
				471666885
0				
0				
R	2437264405040	2437264405040	2672817346721	267281734672
	9906260774080	9906260774080	6456311378434	164563113784
S	3552403448931	3552403448931	1548376943869	341548376943
	4044248957973	4044248957973	2516216816060	869251621681
A	3778580586875	3778580586875	9879420312172	606098794203
	5251107205576	5251107205576	6593414019462	121726593414
-	4762291675508	4762291675508	9361706586942	019462936170
	5235181041957	5235181041957	0985271670110	658694209852
1	3789064494653	3789064505089	9440858476145	716701109440
				858487073
1				
0				
R	1205274642158	1205274642158	5404878151797	540487815179
	0310855461743	0310855461743	4011051651301	740110516513
S	2030512903405	2030512903405	9145760232914	019145760232
	6675630309474	6675630309474	5458644219027	914545864421
A	1618131759927	1618131759927	2847072175699	902728470721
	7555179339636	7555179339636	6594911708125	756996594911
-	3422577230392	3422577230392	1794767022036	708125179476
	0709267863150	0709267863150	9015636379955	702203690156
1	3931925857311	3931925857311	1005756682449	363799551005
	7218046220929	7218046220929	5337112962942	756682449533
2	3738419241877	3830587216991	4764725878590	711296294249
	57	81	89	599034862344
0				1
R	2718706507480	2718706507480	2396125717281	239612571728
	6864907985358	6864907985358	6218858519923	162188585199
S	3457001718486	3457001718486	6252475319510	236252475319
	3429734718649	3429734718649	8384335019837	510838433501
A	1911897423901	1911897423901	7103323261192	983771033232
	3107477449650	3107477449650	6440795139500	611926440795
-	4958287433875	4958287433875	2424343514224	139500242434
	4976307393480	4976307393480	8426466255824	351422484264
1	2273689382970	2273689382970	2413722264565	662558242413
	9545156394058	9545156394058	9079561474222	722264565907
2	8176056183852	8176075685128	8407223361458	956147422284
	9	5	985	072251922410
9				89

R	2224501620791	22245016207	292845935441340340	292845935441
	7483972715920	91748397271	979380231528289722	340340979380
S	0539632128000	59200539632	505341507676725549	231528289722
	0159868945712	12800001598	229927505899803085	505341507676
A	8328123281819	68945712832	523592538748485336	725549229927
	7361292162491	81232818197	948226241696948946	505899803085
-	6978761547026	36129216249	999789242471802821	523592538748
	8138616592342	16978761547	270695053599292706	485336948226
1	7209238864810	02681386165	5	241696948946
	2802142045385	92342720923		999789242471
3	5937794984104	88648102802		802821270695
	01	14204538559		053615395107
0		37796387523		7
		17		

Let's try to factor the RSA numbers above using the serial versions of Julia and MPI/C++ codes above. Note that the number we select from the intervals $(r) \bmod 4$ should be equal 1.

Table 3.3: Factorization Results Using Intervals.

	Begin-1	End-1	Begin-2	End-2
RSA-100	FAIL	SUCCESS	FAIL	SUCCESS
RSA-110	FAIL	SUCCESS	FAIL	SUCCESS
RSA-120	FAIL	SUCCESS	FAIL	SUCCESS
RSA-129	FAIL	SUCCESS	FAIL	SUCCESS
RSA-130	FAIL	SUCCESS	FAIL	SUCCESS
RSA-576	FAIL	SUCCESS	FAIL	SUCCESS
RSA-220	FAIL	SUCCESS	FAIL	SUCCESS
RSA-230	FAIL	SUCCESS	FAIL	SUCCESS

The pattern above is obvious: with a number r close to interval ends, RSA numbers can be factored in less than a second. After discovering this fact, experiments were made with the averages of the interval beginnings and ends. The results show that the averages are “successful”, which means that the averages accelerate Shanks’ Algorithm in contrast to interval beginnings. Another point results prove is that the calculated interval ends are not actually the ends, numbers larger than interval ends accelerate factorization.

Experiments and the interval formulae show that, the interval length decrease as the number to be factored increases (while using the same prime number c).

In order to discover a meaningful relationship between intervals and RSA numbers, $\log(\text{RSA} - \text{NUMBER}, \text{INTERVAL} - \text{BEGIN} - \text{OR} - \text{END})$ was calculated. As the RSA-Number increases, $\log(\text{RSA} - \text{NUMBER}, \text{INTERVAL} - \text{BEGIN} - \text{OR} - \text{END})$ decreases. So, for semiprime numbers X and Y , if $Y > X$, then $\log(X, \text{INTERVAL} - \text{BEGIN} - \text{OR} - \text{END}) > \log(Y, \text{INTERVAL} - \text{BEGIN} - \text{OR} - \text{END})$. The plot below explains the situation for Interval Ends:

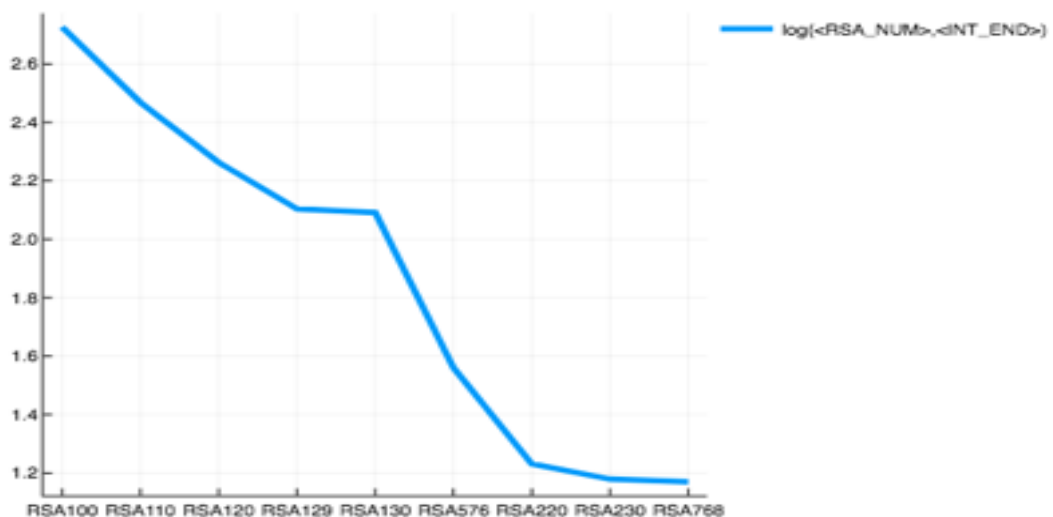


Figure 3.2: $\log(\text{RSA_NUM}, \text{INTERVAL_END})$.

The plot makes more sense when the x-axis is replaced with the number of decimal digits of each RSA number since we can see the slope more clearly.

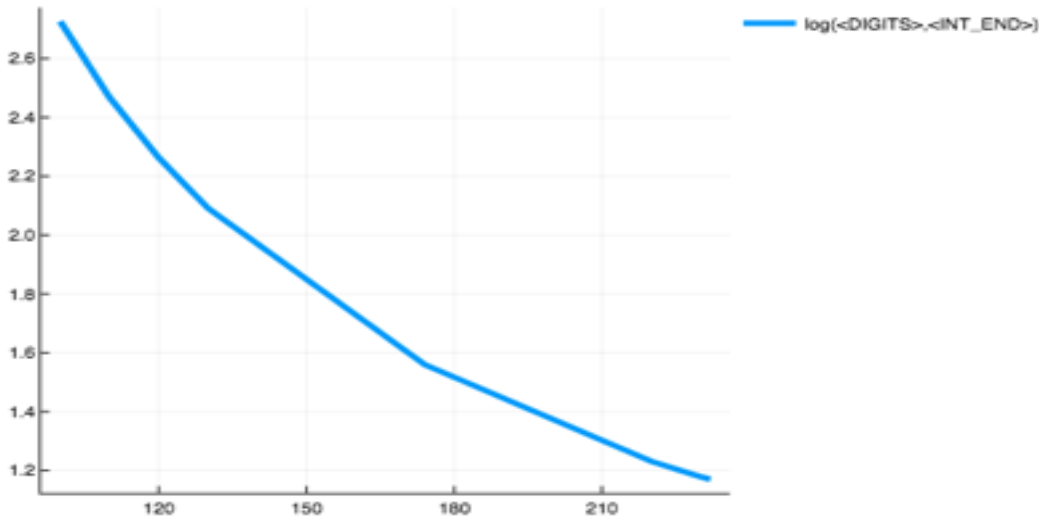


Figure 3.2: $\log(\langle \text{RSA_NUMBER_DIGITS} \rangle, \langle \text{INTERVAL_END} \rangle)$.

3.4 Factoring a Semiprime

$s=152260502792253336053561837813263742971806811497100625256164029200$
 $9551300230960365734328241360708667$ is a semiprime consisting of 100 digits. The
 prime number c is set to :

15000000000000000000000009809809809000000000000000000000000000000
 00000000000000092384203984098234293.

The semiprime subject to factorization is greater than RSA-100 and less than RSA-110. Therefore an interval of number “ r ”s which accelerates Shanks’ Algorithm exists around:

$$(s^{\log(s, \log(\text{RSA100}, \text{RSA100_INTERVAL_END}))}, s^{\log(s, \log(\text{RSA110}, \text{RSA110_INTERVAL_END}))}).$$

Another parameter that should be kept in mind is the length of intervals. For RSA-100, the interval length is around $3 * 10^{54}$, for RSA-110, the interval length is around 10^{49} . The interval length for the number to be factored is at least 10^{49} and at most $3 * 10^{54}$.

.Since $\log(\text{RSA} - 100, \text{RSA} - 100_INTERVAL_END)$,

$\log(\text{RSA} - 110, \text{RSA} - 110_INTERVAL_END)$,

$\log(\text{RSA} - 120, \text{RSA} - 120_INTERVAL_END)$,

$\log(\text{RSA} - 129, \text{RSA} - 129_INTERVAL_END)$ are known, in order to find

$\log(s, S_INTERVAL_END)$, a factorization function was used. The Newton

interpolation code written in Julia [23] was modified due to its inability to run with integers of size larger than 64 bits. According to the function obtained from Newton interpolation $s^{2.06619590888562183171}$ is close to the interval end. Since the factors of s are actually known, its interval ends can be calculated. The interval end for s is (using s 's smaller factor):

```
85242470593140523480578268674184216176646374853810143182703116602157
92177864716346011131262549601568160070715884474700560782020462320377
29545117738884163583184654239339088307731998546912879319350625834333
7.
```

The interval end guessed without using the factor is:

```
85242470593140523480578268674184216176646374855022507311494722302466
08251233720575284447736458127224393706750855009608370266869082124063
96196500894119421209520435745009193818139901413402616885379678954100
7.
```

The difference of calculated interval and guessed interval is:

```
12123641287916057003081607336900422927331647390852565623363603497053
49078094848486198036866665138315523525762633578150567010551040790286
64897375660290531197670.
```

The interval length for s is smaller than the interval length for RSA-100 and greater than the interval length for RSA-110. $3 * 10^{53}$ is smaller than the interval length for s (since s is very close to RSA-100), therefore it is used as step while trying to find the correct interval. Using 1000 processors, the factor is found after approximately $4 * 10^{100}$ iterations using the codes above.



4. BENCHMARKS

4.1 Serial Code Comparison

Serial versions of Julia and C++ codes were compared while factoring RSA-100, RSA-140 and RSA-230. Each code was run for 50 times in order to eliminate misleading results. Since Julia has a JIT compiler in contrast to C++, the compilation time was subtracted from Julia benchmarks. RSA-100 was factored after 700, RSA-140 was factored after 4300, RSA-230 was factored after 4400 iterations. The RSA-230-2 column shows the results when RSA-230 is factored after 147 iterations. The RSA-230-3 column shows the result after 14668 iterations.

Table 4.1: Julia vs C++ Serial Code Results.

	RSA-100	RSA-140	RSA-230	RSA-230-2	RSA-230-3
Julia	0.712	5.0136	8.0248	0.461	25.792
	seconds	seconds	seconds	seconds	seconds
C++	0.539	5.1442	12.044	0.402	40.147
	seconds	seconds	seconds	seconds	seconds

The results show that when the number of loop iterations increase, Julia's performance gets better. Julia is a dynamic language and uses LLVM. JIT compiler and LLVM together results in more optimized code.

4.2 Parallel Code Comparison

The graph below shows the factorization of RSA-100 with 1, 5, 10, 20 processors with each programming language.

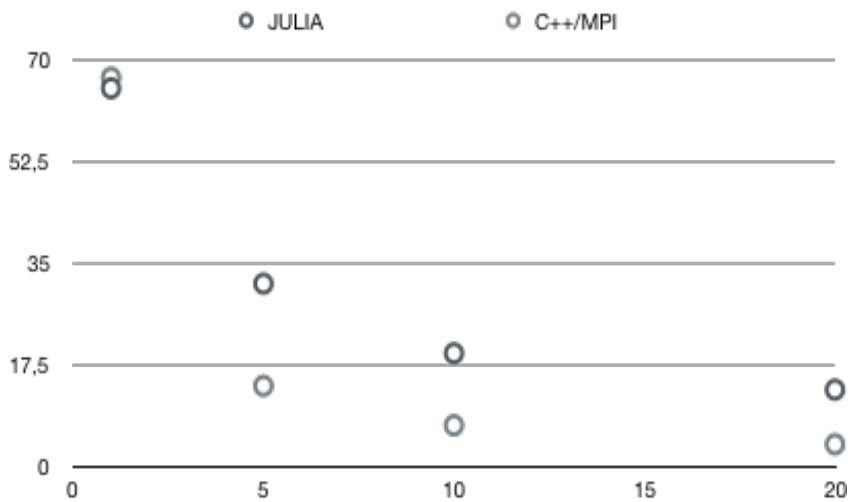


Figure 4.1: Julia vs C++ Parallel Code Results.

The x-axis shows the number of processors while the y-axis shows the execution time. It can be seen that Julia is faster than C++ if the serial code is run, however, C++ and MPI together result in a perfect efficient and speedup in contrast to Julia. For measuring the performance of parallel code, Efficiency (E) and Speedup (S) are two key concepts. The graphs below show the efficiency and speedup. Efficiency (E) is equal $E = \frac{S}{N}$ while Speedup (S) is equal $S = \frac{1}{(1-p) + \frac{p}{s}}$ (N is the number of processors, p is the portion of execution time which is affected by more number of processors and s shows how many times p is faster with more processors).

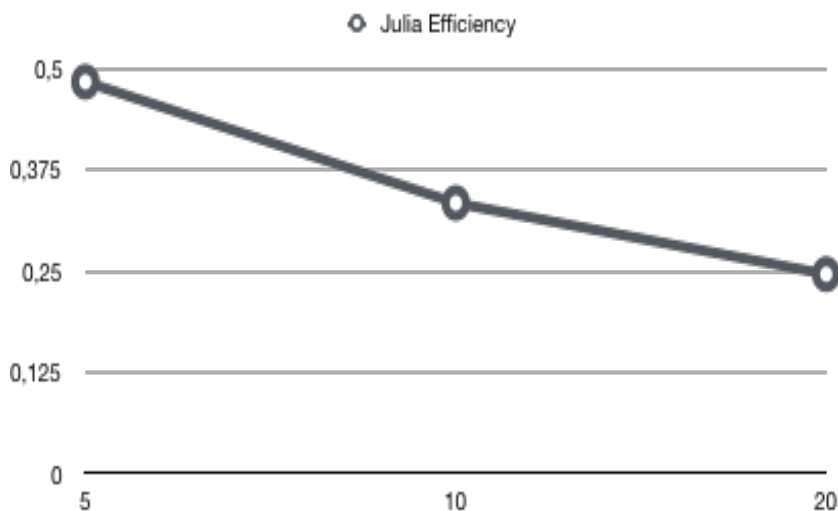


Figure 4.2: Julia Efficiency.

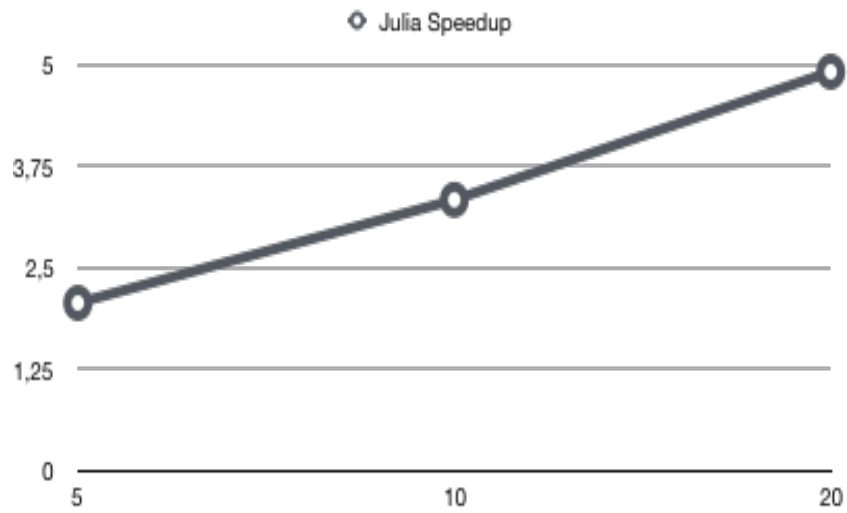


Figure 4.3: Julia Speedup.

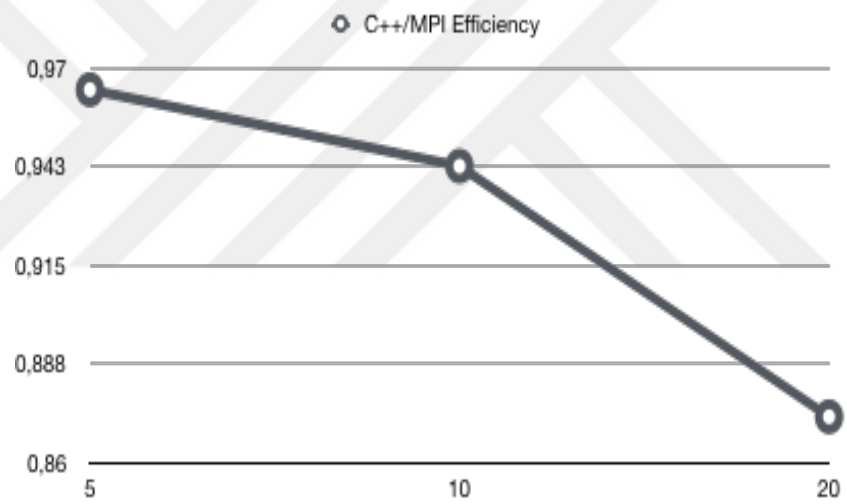


Figure 4.4: C++/MPI Efficiency.

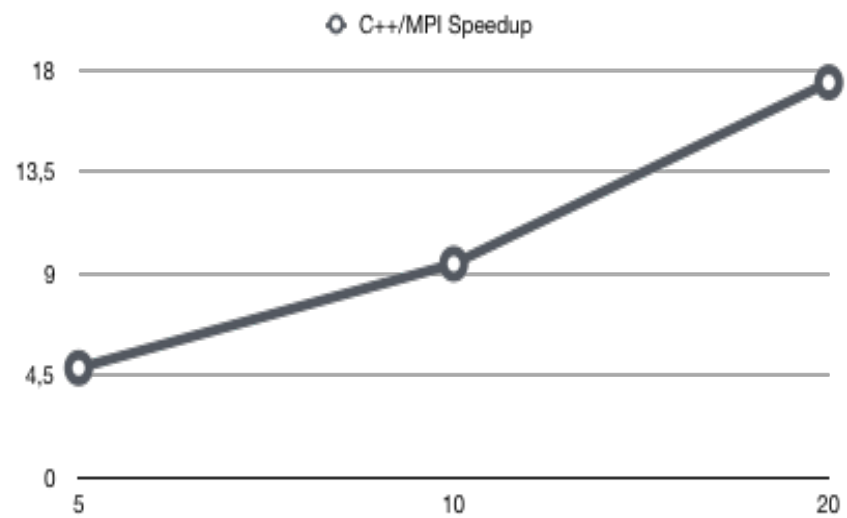


Figure 2.5: C++/MPI Speedup.

For both parallel platforms, it is observed that efficiency drops as number of processors increase. The main reason for this decrease in efficiency is the lag due to network. As the number of processors increase, the number and size of data transfers increase, too. In MPI, when x processors are used, the number of data transfers is equal $2x - 2$ since MPI_Allreduce performs a reduction in the root processor ($x-1$ data transfers) and then broadcasts the result to other non-root processors ($x-1$ data transfers). Julia's efficiency is very poor when compared to MPI's efficiency. With 5 processors, MPI's efficiency is equal 0.96 which is very close to maximum efficiency of 1. Also, MPI achieves nearly linear speedup which shows that each CPU's utilization rate is nearly 100%.



5. CONCLUSION AND FUTURE WORK

The computation results show that another method for integer factorization can be used. If intervals are guessed correctly, factorization of a semiprime can be accomplished in less than a second. However, the formula does not calculate the interval end and interval beginning precisely. The beginning is larger than the calculated beginning and the end is larger than the calculated interval end.

Tests were run using multiple programming languages. The results show that Julia can be an alternative for programmers in the future due to its simplicity and speed. However, using Julia's distributed for loop is not a good alternative for programmers who demand efficiency and speedup. The refinement of the Distributed.jl library is my next aim. A CUDA code was also tested, however since GMP libraries for GPUs do not have good performance, the results were disappointing. The future work includes refactoring and enhancing the GMP libraries for CUDA. Also, finding a better approximation method for intervals of semiprimes whose factors are not known is another goal.



REFERENCES

- [1] **Url-1** <<https://access.redhat.com/blogs/766093/posts/1976023>>, date retrieved 02.05.2019.
- [2] **Url-2** <<https://www.wired.com/2014/06/d-wave-quantum-speedup>>, date retrieved 02.05.2019.
- [3] **Diffie, W. & Hellman, M.** (1976), November. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22, 644-654.
- [4] **Stallings, W.** (2010). *Cryptography and Network Security*. New York, NY: Prentice Hall.
- [5] **Url-3** <https://commons.wikimedia.org/wiki/File:Diffie-Hellman_Key_Exchange.png>, date retrieved 02.05.2019.
- [6] **Url-4** <<http://mathworld.wolfram.com/PrimitiveRoot.html>>, date retrieved 02.05.2019.
- [7] **Rivest, R & Shamir, A. & Adleman, L.** (1977), April. Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 26, 96-99.
- [8] **Niven, I. & Zuckermann, H. & Montgomery, H.** (1960). *An Introduction to the Theory of Numbers*. Hoboken, NJ: John Wiley & Sons.
- [9] **Solovay, R. & Strassen, V.** (1977). A Fast Monte Carlo Test for Primality. *SIAM Journal on Computing*, 6, 84–85.
- [10] **Washington, L. & Trappe, W.** (2006). *Introduction to Cryptography with Coding Theory*. Upper Saddle River, NJ: Pearson Prentice Hall.
- [11] **Shor, P.** (1995), August. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26, 1484–1509.
- [12] **Url-5** <<http://mathworld.wolfram.com/NumberFieldSieve.html>>, date retrieved 02.05.2019.
- [13] **Boneh, D.** (2006). Twenty Years of Attacks on the RSA Cryptosystem. *American Mathematical Society*, 46, 203-213.

- [14] **Wiener, M.** (1989). Cryptanalysis of Short RSA Secret Exponents. *IEEE Transactions on Information Theory*, 36, 553-558.
- [15] **Coppersmith, D.** (1997). Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10, 233-260.
- [16] **Kocher, P.** (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *CRYPTO '96 Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, 104-113.
- [17] **Buell, D.** (1989). Binary Quadratic Forms Classical Theory and Modern Computations. New York, NY: Springer Verlag.
- [18] **Nari, K., Özdemir E., Yaraneri, E.** (2017), October. İkili Kuadratik Formlar ile Çarpanlara Ayırma. *Journal of Engineering Technology and Applied Sciences*, 2, 101-111.
- [19] **Url-6** <<http://gmpylib.org>>, date retrieved 02.05.2019.
- [20] **Url-7** <<https://developer.nvidia.com/cuda-zone>>, date retrieved 02.05.2019.
- [21] **Url-8** <<http://pypl.github.io/PYPL.html>>, date retrieved 02.05.2019.
- [22] **Url-9** <<https://julialang.org/benchmarks>>, date retrieved 02.05.2019.
- [23] **Url-10** <<http://homepages.math.uic.edu/~jan/mcs471/divdifpol.pdf>>, date retrieved 02.05.2019.

CURRICULUM VITAE

Name Surname :Deniz Kırıldıođ
Date and Place of Birth :20.09.1991 - Australia
Address :ITU Ayazaga Kampusu Koru Yolu ARI3 Binası
No:1001
E-Mail :deniz.kirlidog@gmail.com
B.Sc :Bilkent University, Department of Computer
Engineering

PRESENTATIONS ON THE THESIS:

- **Kırıldıođ, Deniz**, 2019: Computational Methods for Integer Factorization. *International Congress of Energy, Economy and Security Presentation*, April 6-7 2019, Istanbul, Turkey.