

Çetin Kaya Koç
Funda Özdemir
Zeynep Ödemiş Özger

Partially Homomorphic Encryption



Springer

Partially Homomorphic Encryption

Çetin Kaya Koç • Funda Özdemir
Zeynep Ödemiş Özger

Partially Homomorphic Encryption

 Springer

Çetin Kaya Koç
Department of Computer Science
University of California
Santa Barbara, CA, USA

Funda Özdemir
Faculty of Engineering
and Natural Sciences
Istinye University
Istanbul, Turkey

Zeynep Ödemiş Özger
Faculty of Engineering and Architecture
Kâtip Çelebi University
İzmir, Turkey

ISBN 978-3-030-87628-9 ISBN 978-3-030-87629-6 (eBook)
<https://doi.org/10.1007/978-3-030-87629-6>

© Springer Nature Switzerland AG 2021

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The potential applications of homomorphic operations over encryption functions were recognized and appreciated almost about the same time as the first public-key cryptographic algorithm RSA was invented. The RSA algorithm is multiplicatively homomorphic. The ensuing 30 years have brought on several additively or multiplicatively homomorphic encryption functions with increasing algorithmic inventiveness and complicated mathematics. However, until 2009, it was not clear whether a fully homomorphic encryption algorithm, one that allows both additive and multiplicative homomorphisms, would exist. This was resolved by Craig Gentry, and followed up by several authors who have proposed fully homomorphic encryption algorithms and addressed issues related to their formulation, arithmetic, efficiency and security. While formidable efficiency barriers remain, we now have a variety of fully homomorphic encryption algorithms that can be applied to various private computation problems in healthcare, finance and national security.

This book started with the desire and motivation to describe and implement partially homomorphic encryption functions using a unified mathematical notation. We hope that we accomplished our projected goal, and you will find our exposition helpful to understand and implement these algorithms. If a highly efficient fully homomorphic encryption function for multiple applications were to become available, the usefulness of partially homomorphic encryption functions would be in question. But this lofty end is not yet in sight. Studying partially homomorphic encryption functions may help us to understand the difficulties ahead and perhaps to avoid blind alleys and dead ends. Moreover, partially homomorphic encryption algorithms may have certain perhaps limited applications for which significantly more efficient implementations can be obtained. If not, it is always enjoyable to learn interesting algorithms and their underlying beautiful mathematics.

The authors thank their families and dedicate this book to them.

Çetin Kaya Koç
Funda Özdemir
Zeynep Özdemir Özger

Contents

- 1 Introduction** 1
- 2 Mathematical Background** 3
 - 2.1 Number Theory 3
 - 2.1.1 Divisibility and Factorization 3
 - 2.1.2 Greatest Common Divisor (GCD) 4
 - 2.1.3 Modular Arithmetic 6
 - 2.1.4 Modular Exponentiation 8
 - 2.1.5 Modular Inverse 9
 - 2.1.6 Euler’s Theorem 11
 - 2.1.7 Carmichael’s Theorem 12
 - 2.1.8 Generators 13
 - 2.1.9 Chinese Remainder Theorem 14
 - 2.1.10 Quadratic Residues 15
 - 2.1.11 Higher-Order Residues 19
 - 2.1.12 Residue Classes 19
 - 2.1.13 Random Number Generators 22
 - 2.2 Group Theory 23
 - 2.2.1 Basic Axioms 23
 - 2.2.2 Multiplicative Groups 23
 - 2.2.3 Additive Groups 24
 - 2.2.4 Order of a Group 24
 - 2.2.5 Cyclic Groups and Subgroups 25
 - 2.2.6 Discrete Logarithm Problem 26
 - 2.2.7 Diffie-Hellman Problem 26
 - 2.2.8 Cosets and Lagrange’s Theorem 27
 - 2.2.9 Sylow’s Theorem 28
 - 2.2.10 Direct Products 29
 - 2.2.11 Homomorphisms 29
 - 2.3 Field Theory 30
 - 2.4 Elliptic Curves 31

2.4.1	Group Law	32
2.4.2	Elliptic Curve Point Multiplication	34
2.4.3	Elliptic Curves over Finite Fields	34
3	Rivest-Shamir-Adleman Algorithm	37
3.1	Key Generation	37
3.2	Encryption	38
3.3	Decryption	38
3.4	Homomorphic Properties	39
3.5	Security	40
3.6	Example	41
4	Goldwasser-Micali Algorithm	43
4.1	Key Generation	43
4.2	Encryption	44
4.3	Decryption	45
4.4	Homomorphic Properties	45
4.5	Security	46
4.6	Example	47
5	ElGamal Algorithm	51
5.1	Multiplicatively Homomorphic ElGamal Algorithm	51
5.1.1	Key Generation	51
5.1.2	Encryption	51
5.1.3	Decryption	52
5.1.4	Homomorphic Properties	53
5.1.5	Security	55
5.1.6	Example	56
5.2	Additively Homomorphic ElGamal	58
5.2.1	Encryption	58
5.2.2	Decryption	59
5.2.3	Homomorphic Properties	59
5.3	The Elliptic Curve ElGamal Algorithm	60
5.3.1	Key Generation	60
5.3.2	Encryption	60
5.3.3	Decryption	61
5.3.4	Homomorphic Properties	61
6	Benaloh Algorithm	63
6.1	Key Generation	63
6.2	Encryption	64
6.3	Decryption	64
6.4	Homomorphic Properties	65
6.5	Security	67
6.6	Example	67

7	Naccache-Stern Algorithm	71
7.1	The Deterministic Version	71
7.1.1	Key Generation	71
7.1.2	Encryption	72
7.1.3	Decryption	72
7.1.4	Security	74
7.2	The Probabilistic Version	74
7.2.1	Key Generation	74
7.2.2	Encryption	75
7.2.3	Decryption	75
7.2.4	Homomorphic Properties	75
7.2.5	Security	77
7.2.6	Example	77
8	Okamoto-Uchiyama Algorithm	83
8.1	Key Generation	83
8.2	Encryption	84
8.3	Decryption	84
8.4	Homomorphic Properties	87
8.5	Security	89
8.6	Example	91
9	Paillier Algorithm	95
9.1	Key Generation	95
9.2	Encryption	95
9.3	Decryption	98
9.4	Homomorphic Properties	101
9.5	Security	102
9.6	Example	103
10	Damgård-Jurik Algorithm	107
10.1	Forming the Plaintext Space	107
10.2	Key Generation	110
10.3	Encryption	110
10.4	Decryption	111
10.5	Homomorphic Properties	115
10.6	Security	116
10.7	Example	118
11	Boneh-Goh-Nissim Algorithm	123
11.1	Key Generation	125
11.2	Encryption	125
11.3	Decryption	127
11.4	Homomorphic Properties	127
11.5	Security	130
11.6	Example	131

12 Sander-Young-Yung Algorithm	135
12.1 Key Generation	135
12.2 Encryption	135
12.3 Decryption	136
12.4 Homomorphic Properties	138
12.5 Security	139
12.6 Example	139
References	141
Further Reading	142
Index	143

Chapter 1

Introduction



Homomorphic encryption is the last chapter for now in the 3000-year history of cryptography and about 50 years of public-key cryptography. Performing meaningful computations with encrypted text without decrypting it seems magical at first. Imagine a deck of cards with unknown numbers on each, facing down on the table: Can you add these numbers and obtain the total value with 100% accuracy, without peeking at any of them? Homomorphic encryption can do this for you; so can a successful illusionist using some elaborate tricks, but homomorphic encryption is honest and straightforward, and involves no tricks whatsoever; it is just beautiful mathematics.

Answers to the question of how homomorphic cryptography accomplishes this feat are found in tens of doctoral theses and in hundreds of journal, conference and electronic archival articles. This book brings together several of these algorithms with their descriptions, algorithmic steps, examples, and security analyses. We are limiting our scope however to *partially homomorphic encryption functions*, which are those that allow one or a few arithmetic operations to be performed homomorphically, rather than all arithmetic operations. We also included two *somewhat homomorphic encryption functions*; however, focused on their partial homomorphisms only. The reader should note that, in this book, we use the terms homomorphic encryption and homomorphic cryptography interchangeably.

The general concept of homomorphic encryption was introduced by Rivest, Adleman and Dertouzos in [22] using the term *privacy homomorphism*, soon after the RSA algorithm was invented [21]. The multiplicative homomorphic property of the RSA algorithm was first noted in print in this paper, as $(x^e)(y^e) = (xy)^e$. Unfortunately, the RSA algorithm does not allow additive homomorphism, and therefore homomorphically evaluating a polynomial is not possible using the RSA encryption function.

The breakthrough in the search for fully homomorphic encryption functions came with the doctoral dissertation of Craig Gentry in 2009. He introduced the first construction for a fully homomorphic encryption scheme [8]. Furthermore, since 1978, we have seen very productive research on partial homomorphisms of existing or new encryption functions.

A taxonomy of fully versus partial homomorphic encryption is however too simplistic. Basic arithmetic operations (addition and multiplication) or Boolean gates (AND, OR, NOT) are just the building blocks for evaluating polynomials, arithmetic or Boolean circuits, or other atomic computations such as search for minimum or maximum, or compare and sort operations. A particular homomorphic property of an encryption function cannot be considered separately from its application to perform private computations. There are mathematical issues as well as efficiency considerations. Particularly, circuit depth issues are significant barriers to efficient computations.

This book specifically concentrates on partially homomorphic encryption functions introduced or analyzed since the RSA algorithm. We exclude the subjects of *somewhat homomorphic*, *leveled homomorphic* and *fully homomorphic* encryption functions. This selection leaves us a handful of mathematically interesting encryption functions. Our observation is that these algorithms are scattered in journals, conference proceedings and electronic archives, and even though they have much in common and they are designed using a few mathematical building blocks, a unified approach to their descriptions and analyses is missing; each one of them seems to be a separate mountain to climb. The need to unify the notation and start with a set of well-defined building blocks to describe and analyze the encryption functions is behind our motivation to write this book.

There is a unifying way we view and present these algorithms in this book. First of all, we wanted it to be more or less self-contained, so we included a pretty extensive mathematical background. Almost everything one needs beyond a four-year computer science or mathematics education in order to understand the encryption chapters is included in Chapter 2; however, this chapter also serves a secondary and more important purpose: It unifies the notation to describe the key generation, encryption, decryption, and homomorphic operations for each of the algorithms presented. We spent a great deal of our time to provide a unified mathematical background, since this was also the only way for us to understand these partially homomorphic encryption algorithms for the purpose of implementing them.

Chapter 2

Mathematical Background



In this chapter, we present the necessary mathematical background for the encryption algorithms in the subsequent chapters.

2.1 Number Theory

The set of integers is represented as $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$.

2.1.1 Divisibility and Factorization

Given two integers $a, b \in \mathbb{Z}$, we say that a **divides** b or b is divisible by a and write $a \mid b$ if there is an integer c such that $a \times c = b$. If a does not divide b , we write $a \nmid b$.

Division Algorithm. Let $a, b \in \mathbb{Z}$. Then there exist unique integers q and r such that

$$a = q \times b + r \text{ and } 0 \leq r < b,$$

where a is the dividend, b is the divisor, q is the quotient and r is the remainder. Notice that b divides a if and only if $r = 0$.

An integer $p \geq 2$ is said to be **prime** if its only positive divisors are the trivial ones: 1 and p . Thus if $a \mid p$ and p is a prime, then either $a = 1$ or $a = p$. An integer greater than 2 that is not prime is said to be **composite**. Note that 1 is neither prime nor composite. The following is an important divisibility property of primes.

Theorem 1 Let $a, b \in \mathbb{Z}$ and p be a prime. If $p \mid (a \times b)$ then $p \mid a$ or $p \mid b$.

Prime numbers are the irreducible multiplicative building blocks of the positive integers, as stated in the following theorem.

Theorem 2 (Fundamental Theorem of Arithmetic) Let $n > 1$ be an integer. Then there exists a unique set of prime numbers $\{p_1, p_2, \dots, p_r\}$ and positive integer ex-

ponents $\{e_1, e_2, \dots, e_r\}$ such that

$$n = p_1^{e_1} \times p_2^{e_2} \times \dots \times p_r^{e_r}.$$

This unique decomposition of n into prime powers is called the **prime factorization** of n .

It is easy to compute the product of primes. However, it is very difficult to factor the resulting product of large distinct primes. The problem of determining the prime factorization of $n = p \times q$, where p and q are large distinct primes, is called the **Integer Factorization Problem** (or Factoring Problem \ Factorization Problem), denoted by **Fact** $[n]$, on which the security of many public-key cryptosystems is based. Most cryptosystems use $n = p \times q$ and some of them use $n = p^2 \times q$, where p and q are chosen large primes having the same bit length.

2.1.2 Greatest Common Divisor (GCD)

Given two nonzero integers $a, b \in \mathbb{Z}$, we say that g is a common factor or common divisor of a and b if $g \mid a$ and $g \mid b$. We call g the **greatest common divisor** of a and b and write $\gcd(\mathbf{a}, \mathbf{b})$ if g is nonnegative and all other common divisors of a and b divide g . Note that $\gcd(a, 0) = a$.

We say that a and b are **relatively prime** if $\gcd(a, b) = 1$, or equivalently that their only common divisors are ± 1 . The following theorem is easy to observe.

Theorem 3 If $g = \gcd(a, b)$, then $\frac{a}{g}$ and $\frac{b}{g}$ are relatively prime.

Proposition 1 If the integers a and b have the prime factorization

$$\begin{aligned} a &= p_1^{e_1} \times p_2^{e_2} \times \dots \times p_r^{e_r} \\ b &= p_1^{f_1} \times p_2^{f_2} \times \dots \times p_r^{f_r}, \end{aligned}$$

where each exponent is greater than or equal to 0, then

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} \times p_2^{\min(e_2, f_2)} \times \dots \times p_r^{\min(e_r, f_r)}.$$

In the above proposition, zero exponents are used to make the set of primes $\{p_1, p_2, \dots, p_r\}$ the same for both a and b .

Euclidean Algorithm. The most commonly used algorithm for computing the greatest common divisor of two integers is the Euclidean algorithm which solves the problem by successive division. This algorithm is based on the property that

$$\gcd(a, b) = \gcd(b, a - q \times b),$$

where q is the integer division $q = \lfloor a/b \rfloor$. Here the problem of finding the gcd of two given integers is reduced to that of finding the gcd of two smaller integers. This reduction rule is applied recursively until the algorithm obtains $\gcd(g, 0) = g$, which is equal to $\gcd(a, b)$ since each iteration preserves the gcd of the previous iteration.

Algorithm 1 Euclidean algorithm for GCD

```

1: function GCD( $a, b$ )
2:   Input: Nonnegative integers  $a$  and  $b$  with  $a > b$ .
3:   Output:  $g = \gcd(a, b)$ .
4:   while  $b \neq 0$  do
5:      $q := \lfloor a/b \rfloor$ 
6:      $r := a - q \times b$ 
7:      $a := b$ 
8:      $b := r$ 
9:   return  $a$ 

```

Example 1 *Let's use the Euclidean algorithm to calculate $\gcd(72, 27)$.*

$$\begin{aligned}
 \gcd(72, 27) &= \gcd(27, 72 - 2 \times 27) \\
 &= \gcd(27, 18) \\
 &= \gcd(18, 27 - 1 \times 18) \\
 &= \gcd(18, 9) \\
 &= \gcd(9, 18 - 2 \times 9) \\
 &= \gcd(9, 0) \\
 &= 9.
 \end{aligned}$$

Another well-known algorithm to find the gcd of two nonnegative integers is the **binary GCD algorithm**, which is also known as Stein's algorithm [25]. This algorithm employs shift operations instead of expensive division operations, which makes it more efficient than the Euclidean GCD algorithm.

The following well-known theorem, which is known as Bézout's Lemma, shows that the greatest common divisor of two integers can be written as a linear combination of them.

Theorem 4 (Bézout's Lemma) *Given any two nonzero integers a and b , there exist integers s and t such that $s \times a + t \times b = \gcd(a, b)$. In particular, if a and b are relatively prime, then there exist integers s and t such that $s \times a + t \times b = 1$.*

The integers s and t are called Bézout coefficients for a and b and they are unique up to integer multiples of a and b .

Extended Euclidean Algorithm. The Bézout coefficients s and t above can be computed using the extended Euclidean algorithm (EEA), which applies the back-substitution method through the remainders in the Euclidean algorithm.

Algorithm 2 Extended Euclidean Algorithm (EEA)

```

1: function EEA( $a, b$ )
2:   Input: Integers  $a$  and  $b$  not both zero.
3:   Output:  $(g, s, t)$  where  $g = \gcd(a, b) = s \times a + t \times b$ .
4:    $g_0 := a, g_1 := b$ 
5:    $s_0 := 1, s_1 := 0$ 
6:    $t_0 := 0, t_1 := 1$ 
7:   while  $g_1 \neq 0$  do
8:      $q := \lfloor g_0 / g_1 \rfloor$ 
9:      $(g_0, g_1) := (g_1, g_0 - q \times g_1)$ 
10:     $(s_0, s_1) := (s_1, s_0 - q \times s_1)$ 
11:     $(t_0, t_1) := (t_1, t_0 - q \times t_1)$ 
12:    $g := g_0, s := s_0, t := t_0$ 
13:   return  $(g, s, t)$ 

```

Example 2 Consider $a = 72, b = 27$. The above algorithm computes the following table

iteration	q	g_0	g_1	s_0	s_1	t_0	t_1
0	-	72	27	1	0	0	1
1	2	27	18	0	1	1	-2
2	1	18	9	1	-1	-2	3
3	2	9	0	-1	3	3	-5

Therefore EEA returns $g = 9, s = -1$ and $t = 3$, with the property that

$$g = s \times a + t \times b.$$

Thus we have

$$9 = (-1) \times 72 + 3 \times 27.$$

2.1.3 Modular Arithmetic

Let $a, b, n \in \mathbb{Z}$ and $n > 0$. We say that a is **congruent** to b modulo n if and only if $n \mid (a - b)$, and write $a \equiv b \pmod{n}$. Here n is called the **modulus**. If $n \nmid (a - b)$, we write $a \not\equiv b \pmod{n}$, and say that a and b are **incongruent** modulo n .

Congruence modulo n is an equivalence relation, which means a binary relation that is reflexive, symmetric and transitive. That is, for all $a, b, c \in \mathbb{Z}$,

- (i) $a \equiv a \pmod{n}$.
- (ii) If $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$.
- (iii) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.

The next theorem connects the congruence notion with the division algorithm.