MASTER

Kleptography
cryptography with backdoors

Antheunisse, M.

Award date:
2015

Link to publication
Master Thesis

Kleptography
Cryptography with Backdoors

Author:
Milou Antheunisse
(student number: 780190)

Supervisor:
Prof. Dr. Tanja Lange

Eindhoven University of Technology
Department of Mathematics and Computer Science

October 26, 2015
Abstract

In a black-box cryptosystem, only the input and output of the system are accessible for the user. This means that the user does not have any knowledge of the internals of the device. This principle has been used regularly, for example in the Capstone project, launched by the US government in 1993. In 1996, Adam Young and Moti Yung showed that black-box cryptography should not be trusted. They designed an attack for black-box cryptosystems: SETUP (Secretly Embedded Trapdoor with Universal Protection). The SETUP mechanism enables the attacker to exfiltrate users’ private information without the users noticing. This field of study is denoted by the term “kleptography”.

First, a general introduction to cryptography is presented. This chapter serves as a stepping stone towards the next chapter on kleptography. Formal definitions of SETUP are given, followed by a distinction between weak, regular and strong SETUP. Furthermore, the notion of leakage bandwidth is introduced. SETUP implementations in RSA key generation, ElGamal key generation and signature scheme and Diffie-Hellman key exchange are shown and explained in detail. Finally, some measures are given to protect against a SETUP attack.
Contents

List of Algorithms ........................................ IV
List of Examples ........................................... V
List of Figures ............................................. VII

1 Introduction ............................................. 1
  1.1 Motivation ........................................... 1
  1.2 Thesis Outline ....................................... 1

2 An Introduction to Cryptography ...................... 3
  2.1 Terminology .......................................... 3
    2.1.1 Encryption and Decryption .................... 3
    2.1.2 Symmetric vs. Asymmetric Cryptography ... 4
  2.2 Number Theory ....................................... 4
    2.2.1 Euclidean Algorithm ......................... 5
    2.2.2 Prime Numbers ................................ 7
    2.2.3 Congruences ................................... 9
    2.2.4 Discrete Logarithm and Integer Factorization . 16
  2.3 Classical Cryptosystems ............................. 16
    2.3.1 Caesar Cipher ................................ 16
    2.3.2 Vigenère Cryptosystem ....................... 17
    2.3.3 Vernam Cipher ................................ 18
  2.4 Modern Cryptosystems ............................... 18
    2.4.1 Cryptographic Primitives ...................... 19
    2.4.2 Diffie-Hellman Key Exchange .................. 22
    2.4.3 ElGamal ...................................... 23
    2.4.4 RSA .......................................... 27
  2.5 Applications of Cryptography ..................... 29
    2.5.1 Bank Cards and Credit Cards ................. 29
    2.5.2 Online Banking ............................... 30
    2.5.3 Pretty Good Privacy ......................... 30

3 Kleptography ............................................. 33
  3.1 Background Information ............................ 33
    3.1.1 Subliminal Channels ......................... 33
    3.1.2 Key Escrow .................................. 37
  3.2 A History of Kleptography ......................... 39
  3.3 SETUP ............................................... 40
    3.3.1 Secretly Embedded Trapdoor with Universal Protection . 41
    3.3.2 Definition of SETUP ........................ 41
    3.3.3 Weak SETUP ................................. 42
    3.3.4 Strong SETUP ............................... 43
    3.3.5 Weak, Regular or Strong SETUP? ............. 43
## CONTENTS

3.3.6 Leakage Bandwidth ........................................... 43
3.4 SETUP in RSA .................................................. 44
  3.4.1 SETUP in RSA Key Generation I ............................ 44
  3.4.2 SETUP in RSA Key Generation II (PAP) ..................... 46
3.5 SETUP in ElGamal ............................................... 53
  3.5.1 SETUP in ElGamal Key Generation I ........................ 53
  3.5.2 SETUP in ElGamal Key Generation II ....................... 58
  3.5.3 SETUP in ElGamal Signature Scheme ....................... 63
3.6 SETUP in Diffie-Hellman ....................................... 71
  3.6.1 SETUP in Diffie-Hellman Key Exchange .................... 72

4 Conclusions .................................................... 83
  4.1 SETUP Attacks ................................................. 83
  4.2 Protection against SETUP Attacks ............................ 84
  4.3 Further Research ............................................. 84
  4.4 Final Words .................................................. 85

A SETUP in RSA ................................................... 87
  A.1 SETUP in RSA Key Generation I ............................... 87
  A.2 SETUP in RSA Key Generation II (PAP) ....................... 88

B SETUP in ElGamal ............................................... 91
  B.1 SETUP in ElGamal Key Generation I .......................... 91
  B.2 SETUP in ElGamal Key Generation II ........................ 93
  B.3 SETUP in ElGamal Signature Scheme ........................ 95

C SETUP in Diffie-Hellman ....................................... 97
  C.1 SETUP in Diffie-Hellman Key Exchange .................... 97

D Safe Primes ..................................................... 99
## List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Euclidean algorithm</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Extended Euclidean algorithm</td>
<td>6</td>
</tr>
<tr>
<td>A.1</td>
<td>SETUP in RSA key generation I</td>
<td>87</td>
</tr>
<tr>
<td>A.2</td>
<td>SETUP in RSA key generation I - Modulus factoring by attacker</td>
<td>87</td>
</tr>
<tr>
<td>A.3</td>
<td>SETUP in RSA key generation II (PAP)</td>
<td>88</td>
</tr>
<tr>
<td>A.4</td>
<td>SETUP in RSA key generation II (PAP) - Modulus factoring by attacker</td>
<td>89</td>
</tr>
<tr>
<td>B.1</td>
<td>SETUP in ElGamal key generation I</td>
<td>91</td>
</tr>
<tr>
<td>B.2</td>
<td>SETUP in ElGamal key generation I - Recovery of private key by attacker</td>
<td>92</td>
</tr>
<tr>
<td>B.3</td>
<td>SETUP in ElGamal key generation II</td>
<td>93</td>
</tr>
<tr>
<td>B.4</td>
<td>SETUP in ElGamal key generation II - Recovery of private key by attacker</td>
<td>94</td>
</tr>
<tr>
<td>B.5</td>
<td>SETUP in ElGamal signature scheme I - First signature</td>
<td>95</td>
</tr>
<tr>
<td>B.6</td>
<td>SETUP in ElGamal signature scheme I - Second signature</td>
<td>95</td>
</tr>
<tr>
<td>B.7</td>
<td>SETUP in ElGamal signature scheme I - Recovery of private key by attacker</td>
<td>95</td>
</tr>
<tr>
<td>B.8</td>
<td>SETUP in ElGamal signature scheme II - First signature</td>
<td>96</td>
</tr>
<tr>
<td>B.9</td>
<td>SETUP in ElGamal signature scheme II - Second signature</td>
<td>96</td>
</tr>
<tr>
<td>C.1</td>
<td>SETUP in Diffie-Hellman key pair generation - First key pair generation</td>
<td>97</td>
</tr>
<tr>
<td>C.2</td>
<td>SETUP in Diffie-Hellman key pair generation - Second key pair generation</td>
<td>97</td>
</tr>
<tr>
<td>C.3</td>
<td>SETUP in Diffie-Hellman key pair generation - Recovery of private key by attacker</td>
<td>97</td>
</tr>
</tbody>
</table>
List of Examples

2.1 Euclidean algorithm ................................................. 6
2.2 Extended Euclidean algorithm .................................... 7
2.3 Residue classes, complete residue system and reduced residue system ............ 9
2.4 Modular inverse using extended Euclidean algorithm ................................. 10
2.5 Chinese remainder theorem .......................................... 15
2.6 Caesar cipher .......................................................... 17
2.7 Vigenère cryptosystem .................................................. 17
2.8 Kasiski’s method ......................................................... 18
2.9 Vernam cipher ............................................................ 18
2.10 The Diffie-Hellman key exchange method ........................................... 23
2.11 Key pair generation, encryption and decryption with ElGamal .......................... 25
2.12 Authentication with ElGamal signature scheme ....................................... 26
2.13 Key pair generation, encryption and decryption with RSA .............................. 29

3.1 Authentication with ElGamal signature scheme with subliminal channel .............. 36
3.2 SETUP in RSA key generation I ........................................ 45
3.3 SETUP in RSA key generation II - $U$ and $U + 1$ .................................... 48
3.4 SETUP in RSA key generation II (PAP) ........................................... 49
3.5 SETUP in ElGamal key generation I ......................................... 55
3.6 SETUP in ElGamal key generation II .......................................... 60
3.7 SETUP in the ElGamal signature scheme .......................................... 66
3.8 SETUP in Diffie-Hellman key exchange .............................................. 75
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The conventional cryptosystem.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Data Encryption Standard.</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Triple DES.</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Advanced Encryption Standard.</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Diffie-Hellman key exchange.</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Key pair generation, encryption and decryption with ElGamal encryption.</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Authentication with ElGamal signature scheme.</td>
<td>26</td>
</tr>
<tr>
<td>2.8</td>
<td>Key pair generation, encryption and decryption with RSA.</td>
<td>28</td>
</tr>
<tr>
<td>2.9</td>
<td>Connection details ING.</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Signature generation in ElGamal signature scheme with subliminal channel.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Key escrow cryptosystem.</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Structure of LEAF.</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>RSA key pair generation with SETUP mechanism.</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>PAP - Probabilities for key size $k$ and bound $B_2$, rounded to two decimal digits.</td>
<td>52</td>
</tr>
<tr>
<td>3.6</td>
<td>SETUP in ElGamal key generation I - Probabilities for key size $k$ and bound $B_2$, rounded to two decimal digits.</td>
<td>58</td>
</tr>
<tr>
<td>3.7</td>
<td>SETUP in ElGamal key generation II - Probabilities for key size $k$ and bound $B_1$, rounded to two decimal digits.</td>
<td>62</td>
</tr>
<tr>
<td>3.8</td>
<td>Diffie-Hellman key pair generation with SETUP mechanism.</td>
<td>73</td>
</tr>
<tr>
<td>3.9</td>
<td>Fraction $f$ quadratic residue or quadratic non-residue for odd or even $X$ and $x_2$.</td>
<td>74</td>
</tr>
<tr>
<td>3.10</td>
<td>“ElGamal variables” in Diffie-Hellman key pair generation with SETUP mechanism.</td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of presented SETUP attacks.</td>
<td>84</td>
</tr>
<tr>
<td>D.1</td>
<td>Safe primes $p$ (such that $p = 2q + 1$ with $q$ prime) with corresponding probabilities $P$, rounded to six decimal digits, and prime $q$.</td>
<td>99</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 1996, Adam Young and Moti Yung wrote an article in which they introduce the \textit{SETUP} attack \cite{36}. They developed this attack in order to show that users should not trust a \textit{black-box} cryptosystem. This is a cryptosystem of which only the input and the output are accessible for the user. Young and Yung implemented this attack in several cryptosystems. They showed that an attacker could easily use the public output of the system to recover users’ private information secretly. This means that the users simply leak private information via the public output without knowing it. A year later, Young and Yung introduced the term \textit{kleptography} to denote this attack.

1.1 Motivation

For this master thesis, the origin and early development of kleptography have been researched, based on articles written by Young and Yung. On the early development and the first SETUP attacks, not much information is available. It seems that only years later this topic was noticed by other cryptographers, since from approximately 2006 onwards more articles on this topic can be found. However, the number of articles is still not very large and the articles that are written by other authors, do not cover the early development and the first SETUP attacks. This means that the only sources available are the articles written by Young and Yung themselves. Unfortunately, these articles do not explain all details and some important issues are simply omitted.

Therefore, the goal of this master thesis is to give a solid, understandable explanation of the early development of kleptography, with an emphasis on the first SETUP attacks. In this thesis, possible SETUP attacks on several well-known cryptosystems have been studied extensively and are described and explained in detail. Furthermore, to provide the reader all necessary cryptographic background information, an introduction to cryptography in general is written as well.

1.2 Thesis Outline

In chapter 2, an introduction to cryptography is provided. Important terms are introduced, followed by an explanation on the difference between symmetric and asymmetric cryptography. The next section covers the most relevant subjects in number theory. Topics that are discussed here are the Euclidean algorithm, properties of prime numbers, congruences, the discrete logarithm problem and integer factorizations. Furthermore, several important theorems are presented, such as the fundamental theorem of arithmetic and Fermat’s little theorem. This section is followed by a section on classical cryptosystems and a section on modern cryptosystems. The three classical cryptosystems that are presented are Caesar cipher, Vigenère cryptosystem and Vernam cipher. In the section on modern cryptosystems, a few symmetric primitives are introduced, followed by the explanation of the Diffie-Hellman key exchange protocol, ElGamal encryption, ElGamal signature scheme and RSA. The chapter ends with a section on some applications of cryptography. To clarify the explanations of definitions, theorems, cryptosystems, etc., many worked-out examples are included in this chapter. Short examples are included in the text and large examples are displayed in a box.
Chapter 3 is on kleptography and SETUP attacks. First, two more advanced cryptographic topics are introduced: subliminal channels and key escrow. Next, a very brief history of kleptography is given, together with a list of the most significant articles by Young and Yung in this area. The third section is on SETUP: formal definitions are given and a distinction between weak, regular and strong SETUP is made. Furthermore, the notion of leakage bandwidth is presented. In the fourth section, two different SETUP attacks on RSA are explained. The next section covers three SETUP attacks on ElGamal: two on ElGamal key generation and one on the ElGamal signature scheme. In the final section of this chapter, the SETUP attack on Diffie-Hellman key exchange is explained. All SETUP attacks are described in detail. Furthermore, worked-out examples are included. In the appendices, a description of each algorithm in pseudocode can be found.

The last chapter concludes the master thesis by summarizing the most significant results. Furthermore, some practical issues are mentioned that could be considered in order to protect against a SETUP attack. Finally, a few recommendations are given for possible further research.
Chapter 2

An Introduction to Cryptography

In this chapter an introduction to cryptography is given to provide the reader the necessary information to be able to understand the main part of this thesis. In section 2.1, the reader is familiarised with common cryptography terms. Besides, the difference between symmetric and asymmetric cryptography is explained. In section 2.2, some significant topics in number theory are introduced, such as the Euler $\phi$-function and the discrete logarithm problem. Next, in section 2.3, three classical (symmetric) cryptosystems are presented. In section 2.4, a few cryptographic primitives are presented, followed by three modern cryptosystems. The final section of this chapter, section 2.5, contains some examples of modern day applications of cryptography.

2.1 Terminology

Alice and Bob are famous characters in cryptography. These placeholder names are frequently used to represent two parties that want to communicate confidentially with one another. Unfortunately, there is also a third party involved, named Eve. Eve tries to manipulate and spy on the communication between Alice and Bob. She could, for example, read and record the message that Alice sends to Bob (eavesdropping), pretend to be Alice and send a message to Bob (masquerade) or secretly change the message that Alice sends to Bob (modification). To protect their communication, Alice and Bob can use a cryptosystem. Cryptography is the science of creating cryptosystems, such that a potential eavesdropper cannot interfere. Cryptanalysis is the science of breaking cryptosystems. Cryptology is the term that covers both cryptography and cryptanalysis.

2.1.1 Encryption and Decryption

The message that Alice wants to send to Bob is called the plaintext $m$. A cryptosystem encrypts this message into a (unreadable) ciphertext $c$, using a key $k$, such that it is protected against eavesdroppers. This is denoted by $E_k(m) = c$. Now the ciphertext can be sent to Bob. After decryption by the same key, Bob is able to read Alice’s message ($D_k(c) = m$). This conventional cryptosystem, shown in figure 2.1 (p. 4), was described by Shannon in 1949 [29]. Note that the channels from the key source to encryption and decryption are secure.

There are several reasons why one would use a cryptosystem. Alice and Bob do not want Eve to read and understand their messages (confidentiality), Bob wants to make sure that the message is really from Alice (authenticity) and he wants to make sure that the message from Alice is not changed by Eve (integrity) [33]. If Alice encrypts the message, Eve cannot read or understand it, so encryption of the plaintext provides confidentiality. To assure Bob that Alice really wrote the message, she can sign it. This digital signature maintains some authenticity. Finally, hash functions, and more generally checksums, are used to ensure integrity. A hash function is a function that maps data of any size to a value (the hash value) of fixed size. A checksum is a kind of hash function that checks the integrity of data. Some checksums are even able to correct errors [12]. More on hash function can be read in section 2.4.1.

In a cryptographic protocol the exact steps are described that Alice and Bob should perform to generate a secure communication channel, using cryptography. Such a protocol describes, for example, how Alice
and Bob can agree upon a key, how they should encrypt or decrypt and how they can authenticate messages.

Usually, the design of a cryptosystem is not kept secret. The idea behind this is based on a principle described by the Dutch cryptologist Kerckhoffs in the 19th century. He stated that a cryptosystem should be able to fall into the hands of the enemy without inconvenience [15]. These days, this is interpreted as follows: the security of a cryptosystem should only rely on the secrecy of the key and not on the secrecy of the system itself.

The message that Alice wants to send to Bob is usually a text with letters. However, messages consisting of numbers are much more convenient for cryptosystems. This means that for most cryptosystems, the message should be represented by numbers. One could, for example, replace every letter by a number that represents its position in the alphabet (A=01, B=02, ..., Y=25, Z=26). One could also use richer character encoding methods, such as ASCII.

### 2.1.2 Symmetric vs. Asymmetric Cryptography

In the conventional cryptosystem described before, the key for encryption and decryption is the same. Cryptosystems in which this is the case, are covered by the term symmetric cryptography. The key is a shared secret between all involved parties. In contrast, there also exist asymmetric (or public-key) cryptosystems, in which the keys for encryption and decryption are not the same. This can be imagined by a padlock: it does not need a key to be closed, but it does need a key to be opened. The so-called public key is used to encrypt the plaintext and the private key is used to decrypt the ciphertext. This means that anyone can encrypt a message, but only the owner of a private key can decrypt and read these messages.

Asymmetric cryptography is relatively new compared to symmetric cryptography. The oldest symmetric cryptosystems date from BC (for example Caesar cipher, section 2.3.1, p. 16), while asymmetric cryptography was introduced in 1976 for the first time (Diffie-Hellman key exchange, section 2.4.2, p. 22). In general, symmetric cryptosystems are mathematically less complicated than asymmetric cryptosystems. Encryption and decryption of data is relatively easy, and therefore usually faster, with symmetric cryptosystems. However, a disadvantage of symmetric cryptography is the sharing of the key. One needs a secure channel to transmit the key from the sender to the receiver(s) of the message. In asymmetric cryptography, this problem does not arise. The public key can be sent to anyone and is not required to remain a secret, so there is no need for a secure channel. The private key is already in hands of the legitimate person. Another huge advantage of public-key cryptosystems, is that it is computationally hard to derive the private key from the public key. However, since asymmetric cryptography is more complicated, it works slower than symmetric cryptography [34].

### 2.2 Number Theory

Since public-key cryptography is based on number theory, it is impossible to study cryptography without some understanding of number theory. In this section, some important subjects of number theory are explained. First, two important sets are introduced.
Definition 2.1. The set of natural numbers is denoted by:

\[ \mathbb{N} = \{0, 1, 2, 3, \ldots \} \]

Definition 2.2. The set of integers is denoted by:

\[ \mathbb{Z} = \{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \} \]

The set of integers is closed under addition, subtraction and multiplication (this will not be proven in this thesis). However, it is not closed under division. For example, let \( a = 10 \) and \( b = 4 \), thus \( a, b \in \mathbb{Z} \). Then \( a/b = 10/4 = 2.5 \notin \mathbb{Z} \).

Definition 2.3. Let \( a, b \in \mathbb{Z} \) with \( b \neq 0 \) and \( \frac{a}{b} = q \). If \( q \in \mathbb{Z} \), then \( a \) is divisible by \( b \) and \( b \) is called a factor of \( a \) \((b \mid a)\). If \( q \notin \mathbb{Z} \), then \( b \) does not divide \( a \) \((b \nmid a)\).

For example, \( 432/27 = 16 \), so 432 is divisible by 27 \((27 \mid 432)\), but 28 does not divide 432 \((28 \nmid 432)\). In the latter case, one can apply division with remainder. If one subtracts 28 from 432 repeatedly until the smallest positive integer is reached, then this smallest positive integer is called the remainder. This means that dividing 432 by 28 gives 432 = 15 \( \cdot \) 28 + 12, where 15 is called the quotient and 12 is the remainder.

2.2.1 Euclidean Algorithm

As shown before, 27 is a divisor of 432. But 27 is also a divisor of 189 and 270. If two integers are given, it is possible to compute the largest integer that divides both numbers. For 432 and 189, this would be 27. This number is called the greatest common divisor.

Definition 2.4. Let \( a, b \in \mathbb{Z} \). The greatest common divisor (gcd) of \( a \) and \( b \) is the largest positive integer \( d \), such that \( d \mid a \) and \( d \mid b \).

Note that gcd\((0,a) = a\), since the largest factor of \( a \) is \( a \) itself and 0 is also divisible by \( a \).

Definition 2.5. Let \( a, b \in \mathbb{Z} \). The least common multiple (lcm) of \( a \) and \( b \) is the smallest positive integer \( m \), such that \( a \mid m \) and \( b \mid m \).

For example, gcd\((7,21) = 7 \) and lcm\((7,21) = 21 \), gcd\((12,40) = 4 \) and lcm\((12,40) = 120 \). The gcd can be computed by using the Euclidean algorithm, named after the Greek mathematician Euclid.

Algorithm 2.1 Euclidean algorithm

Input: \( a, b \in \mathbb{N} \), with \( a \geq b \)

1. \( a' \leftarrow a, b' \leftarrow b \)
2. \( \text{while } b' > 0 \) do
3. \( r \leftarrow \text{remainder of } a' \text{ after division by } b' \)
4. \( a' \leftarrow b' \)
5. \( b' \leftarrow r \)
6. \( \text{end while} \)

Output: gcd\((a,b) = a' \)

To show that the Euclidean algorithm indeed gives the greatest common divisor, one needs the following result of division with remainder. Earlier on this page, division with remainder was introduced and 432 = 15 \( \cdot \) 28 + 12 was used as an example (dividing 432 by 28 gives 12 as a remainder). Note that in this example gcd\((432,28) = \text{gcd}(28,12) = 4 \). As a matter of fact, this is not a coincidence.

Lemma 2.6. Let \( a, b, q, r \in \mathbb{Z} \) with \( b \neq 0 \), \( a = b \cdot q + r \) and \( 0 \leq r < b \). Then: gcd\((a,b) = \text{gcd}(b,r) \).

Proof. Let gcd\((a,b) = d > 0 \), so \( d \) is a divisor of \( a \) and a divisor of \( b \). This implies that \( d \) is a divisor of \( a - b \cdot q \). Since \( a - b \cdot q = r \), \( d \) is also a divisor of \( r \) and hence of gcd\((b,r)\). So gcd\((a,b)] \text{gcd}(b,r)\).

Let gcd\((b,r) = d' > 0 \), so \( d' \) is a divisor of \( b \) and a divisor of \( r \). Since \( a = b \cdot q + r \), \( d' \) is also a divisor of \( a \). If \( d' \) is a divisor of \( a \) and \( b \), it also divides gcd\((a,b)\), so gcd\((b,r)] \text{gcd}(a,b)\).

Finally, if gcd\((a,b)] \text{gcd}(b,r)\) and gcd\((b,r)] \text{gcd}(a,b)\) both hold, then gcd\((a,b) = \text{gcd}(b,r)\). \(\square\)
As can be seen in algorithm 2.1, the Euclidean algorithm uses division with remainder. In the algorithm \( \gcd(a, b) \) is replaced by \( \gcd(a', b') \), with \( a' = b \) and \( b' \) the remainder after dividing \( a \) by \( b \). Then, according to lemma 2.6, \( \gcd(a, b) = \gcd(a', b') \). The algorithm terminates if \( b' = 0 \), so at \( \gcd(a', 0) \).

In the following example the \( \gcd \) of 2983 and 570 is computed using the Euclidean algorithm.

**Example 2.1: Euclidean algorithm**

Let \( a = 2983 \) and \( b = 570 \), then \( \gcd(a, b) \) is computed as follows:

<table>
<thead>
<tr>
<th>( r )</th>
<th>( a' )</th>
<th>( b' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2983</td>
<td>570</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>570</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

This gives \( a' = 19 \) and \( b' = 0 \), so the algorithm terminates and \( \gcd(2983, 570) = 19 \). It is also possible to write it down as follows:

\[
\gcd(2983, 570) = \gcd(570, 133) = \gcd(133, 38) = \gcd(38, 19) = \gcd(19, 0) = 19.
\]

So \( \gcd(2983, 570) = 19 \).

With the Euclidean algorithm, one can compute the \( \gcd \) of two integers. It is possible to compute not only the \( \gcd \), but also the integer coefficients \( u \) and \( v \) such that \( \gcd(a, b) = u \cdot a + v \cdot b \). These coefficients can be computed using the extended Euclidean algorithm.

**Algorithm 2.2 Extended Euclidean algorithm**

**Input:** \( a, b \in \mathbb{N} \), with \( a \geq b \)

1. \( a' \leftarrow a \), \( b' \leftarrow b \)
   - \( u_0 \leftarrow 1 \), \( u_1 \leftarrow 0 \)
   - \( v_0 \leftarrow 0 \), \( v_1 \leftarrow 1 \)
2. **while** \( b' > 0 \) **do**
3. \( q \leftarrow a' \ \text{div} \ b' \)
4. \( r \leftarrow a' - q \cdot b' \)
5. \( (a', b') \leftarrow (b', r) \)
6. \( (u_0, u_1) \leftarrow (u_1, u_0 - q \cdot u_1) \)
7. \( (v_0, v_1) \leftarrow (v_1, v_0 - q \cdot v_1) \)
8. **end while**

**Output:** \( \gcd(a, b) = a' = u_0 \cdot a + v_0 \cdot b \)

To show that indeed \( a' = u_0 \cdot a + v_0 \cdot b \), one should use the intermediate steps of the algorithm. By induction, it is possible to show that the following two equations are true in every loop of the algorithm:

\[
a' = u_0 \cdot a + v_0 \cdot b \quad \text{and} \quad b' = u_1 \cdot a + v_1 \cdot b.
\]  

(2.1)

Initially, \( a' = 1 \cdot a + 0 \cdot b \) and \( b' = 0 \cdot a + 1 \cdot b \), which are obviously true. Assume now that the equations in 2.1 hold for the previous rounds. In the next loop, \( a', u_0 \) and \( v_0 \) are replaced by \( b', u_1 \) and \( v_1 \) respectively, which shows that the first equation still holds. Besides, \( b' \) is replaced by \( a' - q \cdot b' \), \( u_1 \) by \( u_0 - q \cdot u_1 \) and \( v_1 \) by \( v_0 - q \cdot v_1 \). This means that the following equation should hold:

\[
a' - q \cdot b' = (u_0 - q \cdot u_1)a + (v_0 - q \cdot v_1)b.
\]
Using equations (2.1) gives:

\[ a' - q \cdot b' = u_0 \cdot a + v_0 \cdot b - q(u_1 \cdot a + v_1 \cdot b) \]
\[ = (u_0 - q \cdot u_1)a + (v_0 - q \cdot v_1)b. \]

This shows that the two equations are indeed true in every loop of the algorithm, such that eventually \( \gcd(a, b) = a' = u_0 \cdot a + v_0 \cdot b. \)

In the following example the \( \gcd \) of 2983 and 570 is computed using the extended Euclidean algorithm.

\textbf{Example 2.2: Extended Euclidean algorithm}

Let \( a = 2983 \) and \( b = 570 \), then \( \gcd(a, b) \) is computed as follows:

<table>
<thead>
<tr>
<th>( q )</th>
<th>( r )</th>
<th>( a' )</th>
<th>( b' )</th>
<th>( u_0 )</th>
<th>( u_1 )</th>
<th>( v_0 )</th>
<th>( v_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>133</td>
<td>2983</td>
<td>570</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>133</td>
<td>38</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-5</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>38</td>
<td>19</td>
<td>-4</td>
<td>13</td>
<td>21</td>
<td>-68</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>13</td>
<td>-30</td>
<td>-68</td>
<td>157</td>
</tr>
</tbody>
</table>

This gives \( a' = 19 \) and \( b' = 0 \), so the algorithm terminates and the output is:

\[ \gcd(2983, 570) = 19 = 13 \cdot 2983 - 68 \cdot 570. \]

\textbf{2.2.2 Prime Numbers}

A positive integer \( a > 1 \) has at least two positive divisors: 1 and \( a \) itself. There are positive integers, such as 17 and 23, that have only these two divisors. These numbers are called \textit{prime numbers}.

\textbf{Definition 2.7.} A positive integer \( p > 1 \) is called a \textit{prime number} if the only positive divisors are 1 and \( p \) itself.

\textbf{Definition 2.8.} Let \( a, b \in \mathbb{Z} \). If \( \gcd(a, b) = 1 \), then \( a \) and \( b \) are called \textit{co-prime} or \textit{relatively prime}.

An integer \( a > 1 \) that is not a prime, is divisible by another integer \( b \), with \( 1 < b < a \). For example, \( 45 \) is divisible by \( 9 \), so \( 45 = 9 \cdot 5 \). In this case, 5 is a prime and 9 is not, which means that 9 is again divisible by another integer. Indeed, \( 9 = 3 \cdot 3 \), such that eventually \( 45 = 3^2 \cdot 5 \).

\textbf{Lemma 2.9.} Every integer \( n > 1 \) has at least one prime divisor.

\textit{Proof.} Assume there exist positive integers that are not divisible by a prime. Let \( S = \{ n \in \mathbb{Z} : n > 1, n \text{ not divisible by a prime} \} \). Then, according to the well-ordering principle, this set contains a smallest member. Let \( s \) be this smallest member. Since \( s \in S \), it is not divisible by a prime, so \( s \) cannot be a prime itself. This implies that \( s \) is divisible by a positive integer \( k \), with \( 1 < k < s \). Since \( s \in S \), \( k \) cannot be a prime and \( k \) cannot have a prime divisor. So, \( k > 1 \) and \( k \) does not have any prime divisors. This means that \( k \in S \). However, \( k < s \) and \( s \) was the smallest member of \( S \). This is a contradiction, so the initial statement is not true and every integer \( n > 1 \) has at least one prime divisor.

Euclid proved that if a prime \( p \) divides the product of two integers, then \( p \) must divide one of these integers.

\textbf{Theorem 2.10} (Euclid I). Let \( a, b \in \mathbb{Z} \) and let \( p \) be a prime such that \( p | a \cdot b \). Then \( p | a \) or \( p | b \).

\textit{Proof.} If \( p \nmid b \), then \( p | a \) should hold. From the fact that \( p \) is a prime and \( p \nmid b \) it follows that \( \gcd(p, b) = 1 \). This implies that there exist \( x, y \in \mathbb{Z} \) such that:

\[ \gcd(p, b) = 1 = x \cdot p + y \cdot b. \]
Multiplication by $a$ gives then:

$$a = a(x \cdot p + y \cdot b) = a \cdot x \cdot p + y \cdot a \cdot b.$$ 

Obviously, $p | p$ and therefore $p | a \cdot x \cdot p$. Since $p | a \cdot b$, also $p | y \cdot a \cdot b$. So both terms on the righthand side are divisible by $p$. Finally, this implies that $p | a$.

The proof for the case $p \nmid a$ and $p | b$ goes likewise.

The first ten primes are 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29. The largest known prime counts more than 17 million digits. In the 3rd century BC, Euclid was already able to prove that the number of primes is infinite.

**Theorem 2.11** (Euclid II). *The number of prime numbers is infinite.*

**Proof.** Assume that the number of prime numbers is finite. Let $\{p_1, p_2, p_3, \ldots, p_n\}$ denote these prime numbers. Let $P = p_1 \cdot p_2 \cdot p_3 \cdots p_n + 1$. According to lemma 2.9, $P$ has at least one prime divisor. Let $p_i$ be the prime divisor of $P$. Since $p_i \in \{p_1, p_2, p_3, \ldots, p_n\}$, it is also a divisor of $p_1 \cdot p_2 \cdot p_3 \cdots p_n$. If it is a divisor of $P$ and a divisor of the product of all primes, it must also be a divisor of the difference $P - p_1 \cdot p_2 \cdot p_3 \cdots p_n = 1$. However, $p_i$ is a prime number greater than 1, so it cannot be a divisor of 1. Therefore the initial assumption that the number of prime numbers is finite, is not true.

As shown before, the integer 45 can be written as $3^2 \cdot 5$. Since 3 and 5 are primes, this is called the prime factorization of 45. Also other integers can be factored into primes, for example 1176 = $2^3 \cdot 3 \cdot 7^2$. In fact, every integer has a unique prime factorization.

**Theorem 2.12** (Fundamental theorem of arithmetic). *Let $a \in \mathbb{N}^*$, then $a$ has exactly one prime factorization, apart from rearrangements of the prime factors. So $a = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, with $p_i$ distinct primes such that $p_1 < p_2 < \ldots < p_k$ and $e_i$ positive integers.*

**Proof.** The proof of the fundamental theorem of arithmetic consists of two parts: the existence and the uniqueness of a prime factorization [34]. By definition, $a = 1$ has the empty product as prime factorization. Therefore, let $a > 1$.

- **Existence.** The existence of the factorization in theorem 2.12 can be proven using induction. Let $a = 2$. Since 2 is a prime number, it has a trivial prime factorization. Next, let $a \geq 3$ and assume that a prime factorization exists for all integers between 1 and $a$. If $a$ is a prime, there is nothing left to prove, since a prime is equal to its own prime factorization. If $a$ is not a prime, there exist integers $a_1$ and $a_2$ such that $a = a_1 \cdot a_2$ with $a_1, a_2 < a$. Then, according to the induction hypothesis, there exists a prime factorization for $a_1$ and a prime factorization for $a_2$. So:

$$a_1 = p_1^{e_1} \cdot p_2^{e_2} \cdots p_i^{e_i} \quad \text{and} \quad a_2 = q_1^{f_1} \cdot q_2^{f_2} \cdots q_j^{f_j},$$

such that:

$$a = p_1^{e_1} \cdot p_2^{e_2} \cdots p_i^{e_i} \cdot q_1^{f_1} \cdot q_2^{f_2} \cdots q_j^{f_j}$$

is a prime factorization of $a$.

- **Uniqueness.** Assume that $a$ itself has two different prime factorizations, namely:

$$a = p_1^{e_1} \cdot p_2^{e_2} \cdots p_m^{e_m} \quad \text{and} \quad a = q_1^{f_1} \cdot q_2^{f_2} \cdots q_n^{f_n}. \quad (2.2)$$

To show the uniqueness, one needs to show that:

$$p_1^{e_1} \cdot p_2^{e_2} \cdots p_m^{e_m} = q_1^{f_1} \cdot q_2^{f_2} \cdots q_n^{f_n}$$

implies that $m = n$, $p_i = q_j$ for some $i$ and $j$, and that $e_i = f_j$. Since $p_1$ is a divisor of $a$, $p_1$ is also a divisor of $q_1^{f_1} \cdot q_2^{f_2} \cdots q_n^{f_n}$. Then, according to theorem 2.10 (Euclid I), there must exist an $i$ such that $p_1 | q_i^{f_i}$. Note that $q_i^{f_i} = q_i \cdot q_i \cdots q_i$ ($f_i$ times). Using theorem 2.10 again, $p_1$ must also be a divisor of $q_i$. Without loss of generality, let $p_1$ divide $q_1$. Since $q_1$ is a prime, this means that $p_1 = q_1$, such that:

$$p_1^{e_1} \cdot p_2^{e_2} \cdots p_m^{e_m} = p_1^{f_1} \cdot q_2^{f_2} \cdots q_n^{f_n}.$$
2.2. NUMBER THEORY

Assume that \( e_1 > f_1 \) and divide both sides by \( p_1^{f_1} \), then:

\[
p_1^{e_1-f_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_m^{e_m} = q_2^{f_2} \cdot \ldots \cdot q_n^{f_n}.
\]

This implies that the left-hand side is divisible by prime \( p_1 \), but the right-hand side is not, which is not possible. Conversely, assume that \( e_1 < f_1 \) and divide both sides by \( p_1^{f_1} \). Then the left-hand side is not divisible by \( p_1 \), but the right-hand side is. This means that \( e_1 \) must be equal to \( f_1 \), such that:

\[
p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_m^{e_m} = q_1^{f_1} \cdot q_2^{f_2} \cdot \ldots \cdot q_n^{f_n}.
\]

Both sides can be divided by \( p_1^{f_1} \) to obtain:

\[
p_2^{e_2} \cdot \ldots \cdot p_m^{e_m} = q_2^{f_2} \cdot \ldots \cdot q_n^{f_n}.
\]

The same approach can be repeated, such that eventually:

\[
p_1^{e_1} = q_1^{f_1},
\]

\[
p_2^{e_2} = q_2^{f_2},
\]

\[
\vdots
\]

\[
p_m^{e_m} = q_n^{f_n}.
\]

This shows that equation 2.2 is indeed true, which means that \( a \) has a unique prime factorization. The combination of these two parts proves that there exists one unique prime factorization for every integer \( a > 0 \). □

2.2.3 Congruences

**Definition 2.13.** Let \( n > 0 \) be a given positive integer. The integers \( a \) and \( b \) are called congruent modulo \( n \) if \( n | (a - b) \). This is written as:

\[
a \equiv b \pmod{n}.
\]

For example, let \( a = 432 \) and \( n = 28 \). This gives \( 432 \equiv 12 \pmod{28} \). Note the similarity with division with remainder \((432 = 15 \cdot 28 + 12)\).

**Definition 2.14.** The residue class of \( a \) modulo \( n \) is the set of \( x \in \mathbb{Z} \) such that \( x \equiv a \pmod{n} \). This is denoted by \([a]_n\).

**Definition 2.15.** A complete residue system modulo \( n \) contains exactly one element from each residue class (the so-called representatives), such that every integer is congruent modulo \( n \) to exactly one element of the complete residue system. Usually the set \([0, 1, 2, \ldots, n - 1]\) is used as complete residue system mod \( n \).

**Definition 2.16.** A reduced residue system modulo \( n \) contains only the representatives of the residue classes that are relatively prime to \( n \).

**Example 2.3: Residue classes, complete residue system and reduced residue system**

The six residue classes of integers modulo 6 are:

- \([0]_6 = \{\ldots, -12, -6, 0, 6, 12, \ldots\}\);
- \([1]_6 = \{\ldots, -11, -5, 1, 7, 13, \ldots\}\);
- \([2]_6 = \{\ldots, -10, -4, 2, 8, 14, \ldots\}\);
- \([3]_6 = \{\ldots, -9, -3, 3, 9, 15, \ldots\}\);
- \([4]_6 = \{\ldots, -8, -2, 4, 10, 16, \ldots\}\);
- \([5]_6 = \{\ldots, -7, -1, 5, 11, 17, \ldots\}\).

A complete residue system modulo 6 is formed by the set of integers \(\{0, 1, 2, 3, 4, 5\}\). Another complete residue system modulo 6 is \(\{-2, -1, 0, 1, 2, 3\}\). A reduced residue system modulo 6 is \(\{1, 5\}\).
Note that if $n$ is prime, the reduced residue system is equal to the complete residue system, except for the zero.

**Definition 2.17.** Let $a, b \in \mathbb{Z}$. If $a \cdot b \equiv 1 \pmod{n}$, then $b$ is the modular inverse of $a$ modulo $n$. This is denoted by $b \equiv a^{-1} \pmod{n}$.

Note that if $b \equiv a^{-1} \pmod{n}$, then also $b + n \equiv a^{-1} \pmod{n}$ and $b + i \cdot n \equiv a^{-1} \pmod{n}$, with $i \in \mathbb{Z}$. In fact, all elements of the residue class of $b$ modulo $n$ are inverses of $a$ modulo $n$. Usually, $a^{-1}$ is chosen such that $1 \leq a^{-1} < n$ and then one speaks of the class of $a^{-1}$ (denoted by $[a^{-1}]_n$). Note that each element of this class is a modular inverse of $a$ modulo $n$.

If $a$ and $n$ are co-prime, the extended Euclidean algorithm will give:

$$\gcd(a, n) = 1 = u_0 \cdot a + v_0 \cdot n.$$  

If this is reduced mod $n$, this gives:

$$u_0 \cdot a \equiv 1 \pmod{n},$$

so that $u_0$ is the modular inverse of $a$. This implies that if $a$ and $n$ are co-prime, the modular inverse of $a$ can be computed using the extended Euclidean algorithm.

**Theorem 2.18.** If $\gcd(a, n) = 1$, there exists exactly one inverse modulo $n$ of $a$, with $1 \leq a^{-1} < n$.

**Proof.** To prove this theorem, one needs to prove that the inverse exists and that this inverse is unique.

- **Existence.** Let $\gcd(a, n) = 1$. With the extended Euclidean algorithm, one can find $x, y \in \mathbb{Z}$ such that $\gcd(a, n) = 1 = x \cdot a + y \cdot n$. Reducing mod $n$ gives then: $x \cdot a \equiv 1 \pmod{n}$, which shows that $x$ is the inverse modulo $n$ of $a$.

- **Uniqueness.** Assume that $a$ has to two inverses modulo $n$, namely $x_1$ and $x_2$, with $1 \leq x_1, x_2 < n$. Then:

$$x_1 \cdot a \equiv x_2 \cdot a \equiv 1 \pmod{n}$$

$$x_1 \cdot a \equiv x_2 \cdot a \pmod{n}$$

$$(x_1 - x_2)a \equiv 0 \pmod{n}.$$  

This means that $(x_1 - x_2)a$ is a multiple of $n$, so $(x_1 - x_2)a = k \cdot n$ for $k \in \mathbb{Z}$. Since $a$ and $n$ are relatively prime, $(x_1 - x_2)$ must be an integer multiple of $n$. However, since $1 \leq x_1, x_2 < n$, $(x_1 - x_2)$ cannot be an integer multiple of $n$. This implies that $x_1$ must be equal to $x_2$, such that the inverse of $a$ modulo $n$ is unique.

This proves that if $\gcd(a, n) = 1$, there exists one unique inverse of $a$ modulo $n$, with $1 \leq a^{-1} < n$.

**Theorem 2.19.** If $\gcd(a, n) > 1$, there does not exist an inverse modulo $n$ of $a$.

**Proof.** Assume that $\gcd(a, n) > 1$ and that there exists an inverse of $a$ (mod $n$) such that $x \cdot a \equiv 1 \pmod{n}$. Let $\gcd(a, n) = d > 1$. If $x \cdot a \equiv 1 \pmod{n}$, then there exists a $y$ such that $x \cdot a = y \cdot n + 1$ and $x \cdot a - y \cdot n = 1$. Since $\gcd(a, n) = d$, the integer $d$ divides $a$, $n$ and any linear combination of the two. So $d$ is also a divisor of $x \cdot a - y \cdot n$. This implies that the integer $d$ should also divide 1, so $d$ must be equal to 1. However, since $d > 1$ initially, this gives a contradiction. Conclusion, $a$ does not have an inverse modulo $n$ if $\gcd(a, n) > 1$.

---

**Example 2.4: Modular inverse using extended Euclidean algorithm**

Let $a = 2983$ and $n = 577$, compute the modular inverse of $a$ (mod $n$). The extended Euclidean algorithm gives the following output:

$$\gcd(2983, 577) = 1 = 53 \cdot 2983 - 274 \cdot 577.$$  

This gives:

$$53 \cdot 2983 - 274 \cdot 577 = 1$$

$$53 \cdot 2983 \equiv 1 \pmod{577}.$$  

This implies that $2983^{-1} \equiv 53 \pmod{577}$. Indeed, $2983 \cdot 577 = 158099 \equiv 1 \pmod{577}$.
In definition 2.16, the reduced residue system was introduced. This system can also be denoted differently, using the notion of a multiplicative group.

**Definition 2.20.** Let $K$ be a ring. Then $K^*$ denotes the multiplicative group of $K$.

Note that $K^*$ only contains all invertible elements of $K$. This means that $0 \notin K^*$, since $0$ does not have an inverse. Using this, the reduced residue system can be denoted as follows.

**Definition 2.21.** Let $n$ be a positive integer. Then $\mathbb{Z}_n^* = \{ i : \gcd(i, n) = 1 \text{ and } 0 < i < n \}$ is a called the multiplicative group modulo $n$.

**Definition 2.22.** Let $g \in \mathbb{Z}_n^*$. If $\mathbb{Z}_n^* = \{ g^1, g^2, \ldots, g^{n-1} \}$, then $\mathbb{Z}_n^*$ is a cyclic group and $g$ is a generator of this group.

Note that this means that the elements $\{ g^1, g^2, \ldots, g^{n-1} \}$ must be unique. This also implies that each element in $\mathbb{Z}_n^*$ is some power of $g$. Let $n = 7$, then $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$. Since $\langle 3 \rangle = \{3, 2, 6, 4, 5, 1\} = \mathbb{Z}_7^*$, $\mathbb{Z}_7^*$ is cyclic and $3$ is a generator of the group. Note that $2$ is not a generator of this group, since $\langle 2 \rangle = \{2, 4, 1\} \neq \mathbb{Z}_7^*$. In this example, the group $\mathbb{Z}_7^*$ is cyclic, because it is generated by one of its elements. This is not the case for any choice of $n$. For example, $\mathbb{Z}_6^* = \{1, 3, 5, 7\}$ is not cyclic, because this group does not have a generator.

**Theorem 2.23.** Let $\mathbb{Z}_n^*$ be the multiplicative group modulo $n$. If $n$ is prime, then $\mathbb{Z}_n^*$ is cyclic.

A proof of this theorem can be found in [32].

The number of elements of a group is called the order of the group. According to definition 2.21, the order of the multiplicative group modulo $n$ is equal to the number of integers between 0 and $n$ that are relatively prime to $n$. This number is denoted by Euler’s totient function.

**Definition 2.24.** Euler’s totient function, $\varphi(n)$, represents the number of positive integers $i \leq n$ such that $\gcd(i, n) = 1$ ($i$ and $n$ are relatively prime). By definition, $\varphi(0) = 1$ and $\varphi(1) = 1$.

Note that $\gcd(n, n) = n$, so Euler’s totient function could also represent the number of positive integers $i < n$ (instead of $i \leq n$) such that $\gcd(i, n) = 1$. If $n$ is prime, $n$ is relatively prime to any positive integer $<n$ except $0$. This means that $\varphi(n) = n - 1$. This was also the case in $\mathbb{Z}_7^*$. Since $7$ is a prime, the order of this group is equal to $7 - 1 = 6$.

As shown before, $3$ is a generator of $\mathbb{Z}_7^*$, but $2$ is not. One can check whether this group has more generators:

- $\langle 1 \rangle = \{1\}$
- $\langle 2 \rangle = \{2, 4, 1\}$
- $\langle 3 \rangle = \{3, 2, 6, 4, 5, 1\}$
- $\langle 4 \rangle = \{4, 2, 1\}$
- $\langle 5 \rangle = \{5, 4, 6, 2, 3, 1\}$
- $\langle 6 \rangle = \{6, 1\}$

This shows that $\mathbb{Z}_7^*$ has two generators, namely $3$ and $5$. Actually, there exists a theorem that states how many generators a cyclic group has. Before introducing this theorem, two properties of a generator of a group are presented first.

**Theorem 2.25.** Let $G$ be a cyclic group of order $n - 1$. The element $g \in G$ generates $G$ if and only if $n - 1$ is the smallest positive integer such that:

$$g^{n-1} = 1.$$  

**Proof.** The proof consists of two parts:

$\Rightarrow$ Assume that $g$ generates $G$, so:

$$G = \langle g \rangle = \{ g^1, g^2, \ldots, g^{n-1} \}.$$  

Note that, by definition, all elements of $\langle g \rangle$ are unique integers modulo $n$. If this is not the case, $g$ could never generate all unique elements of $G$. So $g^i \neq g^j$ (mod $n$) for $i \neq j$ and $1 \leq i, j < n$. Since $g$ generates $G$, there must exist an integer $k$, with $1 < k \leq n - 1$, such that:

$$g^k = 1.$$
Now assume that $k < n - 1$. Then $k + 1 \leq n - 1$. This implies that:

$$g^{k+1} = g^k g = g.$$  

This means that:

$$g^{k+1} = g.$$  

So if $k < n - 1$, then $g^{k+1} = g$. However, this is in contradiction with the definition of $\langle g \rangle$, so $k$ must be equal to $n - 1$.

$\Rightarrow$ Assume $n - 1$ is the smallest positive integer such that:

$$g^{n-1} = 1.$$  \hfill (2.3)

If $g$ generates $G$, all integers $g^1, g^2, \ldots, g^{n-1}$ should be unique integers. Assume that this is not the case, so there exist integers $i$ and $j$ such that $g^i = g^j$, with $1 \leq i, j \leq n - 1$ and $i \neq j$. Assume, without loss of generality, that $i > j$. Then $i = j + x$, for some integer $x$ with $1 \leq x < n - 1$. Then:

$$g^i = g^j,$$

$$g^{i+x} = g^j,$$

$$g^j g^x = g^j,$$

$$g^x = 1.$$  

This implies that there exists an integer $x < n - 1$, such that $g^x = 1$. However, this is in contradiction with equation 2.3. This shows that all integers $g^1, g^2, \ldots, g^{n-1}$ should indeed be unique integers modulo $n$. This means that:

$$\langle g \rangle = \{1, 2, \ldots, n-1\} = G,$$

and that $g$ generates $G$.  \hfill $\Box$

Indeed, for $\mathbb{Z}_7$: 3 is a generator of this group and $3^6 \equiv 1 \pmod{7}$. This theorem can be used to prove the following theorem on the number of generators of a cyclic group.

**Theorem 2.26.** Let $G$ be a cyclic group of order $n - 1$, generated by $g \in G$. Then $g^k$ generates $G$ as well if and only if $\gcd(k, n-1) = 1$.

**Proof.** The proof consists of two parts:

$\Rightarrow$ Assume that $g^k$ generates $G$, with $1 \leq k < n - 1$. Then it must be shown that $k$ is relatively prime to $n - 1$. Note that, since $g^k$ generates $G$, generator $g$ must also be a power of $g^k$, so:

$$g = (g^k)^i = g^{ki},$$

for some $1 \leq i < n - 1$. This implies that:

$$k \cdot i \equiv 1 \pmod{n-1}.$$  

Note that this means that $i$ is the modular inverse of $k$ modulo $n - 1$. According to theorem 2.18, this inverse only exists if $\gcd(k, n - 1) = 1$. So, indeed, if $g^k$ generates $G$, then $\gcd(k, n-1) = 1$.

$\Leftarrow$ Now assume that $\gcd(k, n-1) = 1$. Then it must be shown that $g^k$ generates $G$. If $\gcd(k, n-1) = 1$, then the inverse of $k$ modulo $n - 1$ exists (2.18). Let $i$ denote the modular inverse of $k$ modulo $n - 1$, so:

$$k \cdot i \equiv 1 \pmod{n-1}.$$  

This implies that:

$$g = g^1 = g^{ki} = (g^k)^i.$$  

Let $a$ be an random element of $G$, then there exists an integer $j$ such that:

$$a = g^j \quad \text{and} \quad a = (g^k)^{i \cdot j}.$$  

Since $a$ is a random element of $G$, this shows that any element $a \in G$ can be written as a power of $g^k$. Finally, this means that $g^k$ is also a generator of the group if $\gcd(k, n-1) = 1$.  \hfill $\Box$
Finally, using theorem 2.26, the following theorem on the number of generators will be proved.

**Theorem 2.27.** Let $G$ be a cyclic group of order $n − 1$. Then the number of generators of $G$ is equal to $\varphi(n − 1)$.

**Proof.** This theorem follows from theorem 2.26. Let $g$ be a generator of $G$, then $g^{k}$ is a generator as well for all $1 \leq k < n − 1$ such that $\gcd(k, n − 1) = 1$. By the definition of Euler's totient function (definition 2.24), there are exactly $\varphi(n − 1)$ integers $1 \leq k < n − 1$ that are relatively prime to $n − 1$. So, indeed, there are exactly $\varphi(n − 1)$ generators of $G$. \hfill $\square$

Note that since $n$ is prime, the number of generators of $\mathbb{Z}_{n}^{\ast}$ is equal to $\varphi(n) = n − 1$. This implies that $\mathbb{Z}_{p}^{\ast}$ has $\varphi(7) = \varphi(6) = 2$ generators, which is indeed true.

**Theorem 2.28.** Let $p, q$ be two different prime numbers and let $n = p \cdot q$. Then $\varphi(n) = (p − 1)(q − 1)$.

**Proof.** Here $\varphi(n)$ is the number of integers in $\{1, 2, \ldots, n − 1\}$ that are relatively prime to $n$. Instead of counting the co-primes, one can also count the integers that are not relatively prime to $n$. Since $n$ is the product of two primes, the only divisors of $n$ are $1, p, q$ and $n$ itself. The integers in $\{1, 2, \ldots, n − 1\}$ that are not relatively prime to $n$ must therefore be multiples of $p$ or $q$. This means that the following integers in $\{1, 2, \ldots, n − 1\}$ are not relatively prime to $n$:

- $p, 2p, 3p, \ldots, (q − 1)p$;
- $q, 2q, 3q, \ldots, (p − 1)q$.

Note that these two sets are disjoint. If an element of the first set is a multiple of $q$, it would be too big for the set, since $k \cdot q > (q − 1)p$ for $k \geq 1$. The same holds for multiples of $p$ in the second set. The number of integers that are not relatively prime to $n$ is then equal to $(q − 1) + (p − 1) = p + q − 2$. In total, there are $n − 1 = p \cdot q − 1$ integers $0 < i < n$. Finally, the number of integers relatively prime to $n$ is then:

$$p \cdot q − 1 − (p + q − 2) = p \cdot q − p − q + 1 = (p − 1)(q − 1).$$ \hfill $\square$

**Definition 2.29.** Let $p > 2$ be prime $x \in \mathbb{Z}_{p}$. If there exists an element $a \in \mathbb{Z}_{p}$ such that:

$$a^{2} \equiv x \pmod{p},$$

then $x$ is called a **quadratic residue** modulo $p$. Otherwise, $x$ is called a **quadratic non-residue** modulo $p$.

Recall for $\mathbb{Z}_{7}$ that:

- $0^{2} \equiv 0 \pmod{7}$;
- $1^{2} \equiv 1 \pmod{7}$;
- $2^{2} \equiv 4 \pmod{7}$;
- $3^{2} \equiv 2 \pmod{7}$;
- $4^{2} \equiv 2 \pmod{7}$;
- $5^{2} \equiv 4 \pmod{7}$;
- $6^{2} \equiv 1 \pmod{7}$.

This means that 0, 1, 2, and 4 are the quadratic residues modulo 7 and 3, 5 and 6 are the quadratic non-residues modulo 7.

Fermat’s little theorem (theorem 2.30) shows an important property of integers modulo a prime. Theorem 2.32 can be seen as an extension of Fermat’s little theorem to general moduli.

**Theorem 2.30** (Fermat’s little theorem). Let $a, p \in \mathbb{Z}$ with $p$ prime and $p \nmid a$, then:

$$a^{p−1} \equiv 1 \pmod{p}.$$

**Proof.** Let $G = \mathbb{Z}_{p}^{\ast}$ be a multiplicative group of order $p − 1$ and assume that $1 \leq a \leq p − 1$. Since $p$ is prime, $G$ is cyclic. Assume that $g \in G$ generates $G$. This implies that $a$ can be written as a power of $g$:

$$a \equiv g^{i} \pmod{p},$$

for some $i \in \{1, 2, \ldots, p − 1\}$. Since $g$ is a generator:

$$g^{p−1} \equiv 1 \pmod{p}.$$
Using this, gives:
\[ a^{p-1} \equiv (g^i)^{p-1} \equiv (g^{p-1})^i \equiv 1^i \equiv 1 \pmod{p}. \]
So indeed:
\[ a^{p-1} \equiv 1 \pmod{p}. \]  
(2.4)

Multiplying both sides of the equation in 2.4 by \( a \) gives:
\[ a^p \equiv a \pmod{p}, \]
which is also a common statement of Fermat’s little theorem.

To prove theorem 2.32, one can use Lagrange’s theorem:

**Theorem 2.31** (Lagrange’s theorem). The order of a subgroup of a finite group divides the order of the group.

A proof of this theorem can be found in multiple algebra textbooks, for example [11], [17] and [35].

**Theorem 2.32** (Fermat-Euler). Let \( a, n \in \mathbb{Z} \) with \( \gcd(a, n) = 1 \). Then:
\[ a^{\varphi(n)} \equiv 1 \pmod{n}. \]

**Proof.** Let \( G = \mathbb{Z}_n^* \) be a multiplicative group of order \( \varphi(n) \), assume that \( 1 \leq a \leq n - 1 \) and let \( H \) be the multiplicative subgroup of order \( k \) generated by \( a \), so \( H = \{1, a, a^2, \ldots, a^{k-1}\} \). Again, this means that \( a^k \equiv 1 \pmod{n} \). Since \( \gcd(a, n) = 1 \), \( a \in G \). Then, according to Lagrange’s theorem (theorem 2.31), the order of \( H \) divides the order of \( G \). This means that \( k|\varphi(n) \), so there exists an integer \( x \) such that \( k \cdot x = \varphi(n) \). This implies that:
\[ a^{\varphi(n)} \equiv a^k x \equiv (a^k)^x \equiv 1^x \equiv 1 \pmod{n}. \]

Indeed, \( a^{\varphi(n)} \equiv 1 \pmod{n} \). □

The next theorem is the Chinese remainder theorem.

**Theorem 2.33** (Chinese remainder theorem). Let \( a_1, a_2, \ldots, a_k \) be \( k \) integers and let \( m_1, m_2, \ldots, m_k \) be \( k \) pairwise co-prime integers, so \( \gcd(m_i, m_j) = 1 \) for all \( 1 \leq i, j \leq k \) with \( i \neq j \). Then the system of \( k \) congruence relations:
\[
\begin{align*}
x &\equiv a_1 \pmod{m_1} \\
x &\equiv a_2 \pmod{m_2} \\
\vdots \\
x &\equiv a_k \pmod{m_k}
\end{align*}
\]
has one unique solution \( x \) modulo \( \prod_{i=1}^k m_i \).

**Proof.** To prove the Chinese remainder theorem, one needs to prove the existence and the uniqueness of the solution:

- **Existence.** The system of \( k \) congruence relations given in theorem 2.33, can be constructed as follows:
  - Compute the new modulus \( M = \prod_{i=1}^k m_i \).
  - Compute \( M_i = M/m_i \) for all \( i \).
  - Compute \( n_i = M_i^{-1} \pmod{m_i} \) for all \( i \) (extended Euclidean algorithm).
  - Then \( x \equiv \sum_{i=1}^k a_i \cdot M_i \cdot n_i \pmod{M} \).
Since the moduli \( m_i \) are pairwise co-prime integers, \( \gcd(M_i, m_i) = 1 \) for all \( 1 \leq i \leq k \). Then, according to theorem 2.18, the inverse of \( M_i \) modulo \( m_i \) exists, which proves the existence of \( n_i \).

Next, it must be shown that \( x \) satisfies the system of congruence relations. Take, for example, the \( j \)-th congruence relation (\( 1 \leq j \leq k \)):

\[
x \equiv a_j \pmod{m_j}.
\]

Then:

\[
\sum_{i=1}^{k} a_i \cdot M_i \cdot n_i \equiv a_j \pmod{m_j}
\]

must hold. All terms of the summation are congruent to zero modulo \( m_j \), except for the \( j \)-th term. This is because \( m_j \) is a divisor of \( M_i \) if \( i \neq j \). Then it must be shown that \( a_j \cdot M_j \cdot n_j \equiv a_j \pmod{m_j} \). Since \( m_j \) is not a divisor of \( M_j \), this term is also congruent to zero modulo \( m_j \).

- **Uniqueness.** To prove the uniqueness of the solution, assume that there are two different solutions, \( x_1 \) and \( x_2 \). This implies that:

\[
x_1 \equiv x_2 \pmod{m_i},
\]

for all \( 1 \leq i \leq k \). This means that \( (x_1 - x_2) \) must be divisible by all \( m_i \)'s and, since the \( m_i \)'s are co-prime, therefore also by the product of all \( m_i \)'s, namely \( M \). So \( M | (x_1 - x_2) \), which implies that \( x_1 \equiv x_2 \pmod{M} \).

The proof of the existence and the proof of the uniqueness of the solution together show that the Chinese remainder theorem is indeed true.

---

**Example 2.5: Chinese remainder theorem**

Solve the following system of congruence relations for \( x \):

\[
\begin{align*}
x &\equiv 6 \pmod{11} \\
x &\equiv 11 \pmod{13} \\
x &\equiv 16 \pmod{17}.
\end{align*}
\]

- First, it is verified that 11, 13 and 17 are pairwise relatively prime.
- Then, the new modulus is computed: \( M = 11 \cdot 13 \cdot 17 = 2431 \).
- Next:
  - \( M_1 = 2431/11 = 221 \);
  - \( M_2 = 2431/13 = 187 \);
  - \( M_3 = 2431/17 = 143 \).
- Then:
  - \( n_1 \equiv 221^{-1} \equiv 1 \pmod{11} \);
  - \( n_2 \equiv 187^{-1} \equiv 8 \pmod{13} \);
  - \( n_3 \equiv 143^{-1} \equiv 5 \pmod{17} \).
- Finally:

\[
x \equiv 6 \cdot 221 \cdot 1 + 11 \cdot 187 \cdot 8 + 16 \cdot 143 \cdot 5 \equiv 50 \pmod{2431}.
\]

Indeed, 50 \( \equiv 6 \pmod{11} \), 50 \( \equiv 11 \pmod{13} \) and 50 \( \equiv 16 \pmod{17} \).

The last theorem of this section is a result of the Chinese remainder theorem.

**Theorem 2.34.** Let \( \gcd(p, q) = 1 \). If \( a \equiv b \pmod{p} \) and \( a \equiv b \pmod{q} \), then \( a \equiv b \pmod{p \cdot q} \).

**Proof.** If \( a \equiv b \pmod{p} \), then \( p | (a - b) \). If \( a \equiv b \pmod{q} \), then \( q | (a - b) \). Since \( p \) and \( q \) are relatively prime, this implies that \( p \cdot q | (a - b) \) and therefore \( a \equiv b \pmod{p \cdot q} \).
2.2.4 Discrete Logarithm and Integer Factorization

In theory, a cryptosystem would be perfectly secure if Eve cannot break the system with the information she retrieves from the communication channels. Even if she would have a computer with infinite capacity and power, she would be unable to break the system. Despite the existence of such a system (Vernam cipher in symmetric cryptography, section 2.3.3, p. 18), it turns out that in practice it is very difficult to design a useful, perfectly secure cryptosystem using public-key cryptography. Even the existence of one-way functions is not known (a one-way function is easy to compute when given an input, but it is hard to compute the inverse when given an output). Instead, there are many public-key cryptosystems that rely on hard mathematical problems.

The known algorithms for solving these hard mathematical problems take super-polynomial time and can only be run for relatively small parameters. For large problems, the capacity of a computer fails dramatically and the algorithm would simply take too much time to solve the problem. Even the capacity of the biggest computers in the world together would not suffice, even when run for a very long time. This means that as long as these problems cannot be solved within a reasonable amount of time, the cryptosystems can be considered secure. In this section, two hard mathematical problems are introduced: the discrete logarithm problem and the integer factorization problem.

Discrete Logarithm Problem

The discrete logarithm problem is first defined for a group in general and then for the multiplicative group of integers modulo a prime.

**Definition 2.35 A.** Let $G$ be a group and let $g$ be the generator of a cyclic subgroup of $G$. The discrete logarithm problem is defined as follows. Given $h \in \langle g \rangle$, find the smallest positive integer $x$ such that $g^x = h$.

**Definition 2.35 B.** Let $G$ be the multiplicative group $\mathbb{Z}_p^*$ of a finite field with $p$ a prime. Let $g \in \mathbb{Z}_p^*$ be the generator of this group. Given $h \in \mathbb{Z}_p^*$, find the discrete logarithm $x$ such that $g^x \equiv h \pmod{p}$.

There exist multiple methods to find the discrete logarithm. Generic methods work in both cases. These are, for example, Pohlig-Hellman, Baby-Step Giant-Step and Pollard-$\rho$. A method that only works for the multiplicative group modulo a prime, is Index Calculus. Cryptosystems that are based on the discrete logarithm problem are, among other systems, the Diffie-Hellman key exchange method (section 2.4.2, p. 22) and ElGamal (section 2.4.3, p. 23).

Integer Factorization Problem

**Definition 2.36.** Let $a \in \mathbb{N}^*$. The integer factorization problem is the problem of finding the prime factorization of $a$, such that $a = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, with $p_i$ distinct primes and $e_i$ positive integers.

According to the fundamental theorem of arithmetic (theorem 2.12), there exists such a prime factorization for each integer. A few methods to find the prime factorization are: trial division, Pollard $p - 1$, Pollard-$\rho$ for factorization, quadratic sieve and the number field sieve.

2.3 Classical Cryptosystems

In this section three classical cryptosystems are presented. The first system is Caesar cipher, followed by Vigènere cryptosystem and Vernam cipher. These three cryptosystems are all examples of symmetric cryptosystems.

2.3.1 Caesar Cipher

One of the oldest cryptosystems was used by the Romans during Julius Caesar’s time. This cryptosystem, named after the emperor, uses a cyclic permutation of the alphabet over $k$ places, where $k$ serves as the key of this cryptosystem.
2.3. CLASSICAL CRYPTOSYSTEMS

Example 2.6: Caesar cipher
If \( k = 4 \), then

\[
\begin{array}{cccccccc}
W & X & Y & Z & A & B & C & D \\
H & I & J & K & L & M & N & O \\
P & Q & R & S & T & U & V & W \\
X & Y & Z & A & B & C & D & E \\
F & G & H & I & J & K & L & M \\
N & O & P & Q & R & S & T & U \\
V & W & X & Y & Z & A & B & C \\
D & E & F & G & H & I & J & K
\end{array}
\]

This means that the plaintext “THIS IS SECRET” is encrypted as “XLMV MW WIGVIX”.

Even if one does not have the key, it is very easy to break the system and decrypt the ciphertext. An effective approach to break the system is frequency analysis. For every language, one can make a letter frequency table. In English, for example, the letter ‘e’ has the highest and the letter ‘z’ has the lowest frequency (relative frequency of respectively 12.51 % and 0.09 %). If in the ciphertext the letter ‘m’ has the highest frequency, it is very plausible that \( k = 8 \).

It is also possible to apply a brute force attack. Since the alphabet has only 26 letters, there are 25 possible keys (or 26, if one counts \( k = 0 \) as key). If one tries all 25 cyclic permutations, one of them will give a plaintext that makes sense.

2.3.2 Vigenère Cryptosystem
The Caesar cipher uses the same shift for each letter of the plaintext. During the 16th century, Frenchman Blaise de Vigenère wrote about an improvement of this already existing cryptosystem. The improved cryptosystem, which was named after Vigenère, uses a combination of multiple Caesar ciphers. The key is a word and this word determines which shift is applied on each letter of the plaintext. For example, if the first letter of the key is “H” and the first letter of the plaintext is “A”, then “A” is mapped to “H” (i.e. \( k = 7 \) in the Caesar cipher).

Example 2.7: Vigenère cryptosystem
Let \( k = HELLO \). The plaintext “THIS IS SECRET” is then encrypted as follows:

<table>
<thead>
<tr>
<th>plaintext</th>
<th>T</th>
<th>H</th>
<th>I</th>
<th>S</th>
<th>I</th>
<th>S</th>
<th>S</th>
<th>E</th>
<th>C</th>
<th>R</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>H</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>O</td>
<td>H</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>O</td>
<td>H</td>
<td>E</td>
</tr>
<tr>
<td>ciphertext</td>
<td>A</td>
<td>L</td>
<td>T</td>
<td>D</td>
<td>W</td>
<td>Z</td>
<td>W</td>
<td>P</td>
<td>N</td>
<td>F</td>
<td>L</td>
<td>X</td>
</tr>
</tbody>
</table>

This system has long been considered safe. It was only in the 19th century that the Prussian army officer Kasiski broke the system. The most difficult part of breaking this system is the determination of the key length \( n \). If \( n \) is very long compared to the length of the plaintext, the system could indeed be unbreakable. However, if short keys are used, the system can be broken.

Kasiski used the following observation. If the ciphertext contains identical fragments of three or more letters, it is likely that these fragments have been encrypted with the same part of the key and that these fragments correspond to identical fragments of the plaintext as well. The distance between these fragments is then a multiple of the key length.
Example 2.8: Kasiski’s method
Consider the following plaintext:

ROSEMARYTOLDCARLOTHATTHEMOVETHYWATCHEDWASVERYSCARY

The following ciphertext was created using the Vigenère cryptosystem and \( k = \text{FILM} \):

WWDQRICKYWWPHICXTBSQRYBSQDFHPPPBIDHIHZJJEICK

In this ciphertext, the following fragments appear repeatedly:
- ICK, starting at positions 6 and 50;
- HIC, starting at positions 13 and 49;
- BSQ, starting at positions 22 and 30.

This means that the key length must be a divisor of 44, 36 and 8, so \( n = 4 \) or \( n = 2 \). Indeed, \( n = 4 \).

Assume that using this analysis one concludes that \( n = i \). This means that the letters \( \{1, 1 + i, 1 + 2i, \ldots\} \) of the plaintext are all encrypted with the same letter (the first letter of the key). The same holds for \( \{2, 2 + i, 2 + 2i, \ldots\} \), etc. Each subset of the plaintext is encrypted with Caesar cipher, so can be broken using frequency analysis.

2.3.3 Vernam Cipher
To improve the security of the Vigenère cryptosystem, one could use a keyword of the same length as the plaintext. If this key is randomly generated and used only once, the system is proved to be secure. This means that if someone only has a ciphertext, that it is impossible to recover the plaintext. This system is called one-time pad or Vernam cipher, after the American Gilbert Sandford Vernam, who invented this cryptosystem in the early 20th century. The ciphertext is generated exactly the same way as in the Vigenère cryptosystem, only with a key as long as the plaintext.

Example 2.9: Vernam cipher
Let \( k = \text{BPJJILHYCYCV} \). The plaintext “THISISSECRET” is then encrypted as follows:

<table>
<thead>
<tr>
<th>plaintext</th>
<th>I</th>
<th>S</th>
<th>E</th>
<th>C</th>
<th>R</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>B</td>
<td>P</td>
<td>J</td>
<td>J</td>
<td>I</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>ciphertext</td>
<td>U</td>
<td>W</td>
<td>R</td>
<td>B</td>
<td>Q</td>
<td>D</td>
<td>Z</td>
</tr>
</tbody>
</table>

This cryptosystem is the only perfectly secure system known. It is used for the so-called hot line between Washington and Moscow, which was developed during the Cold War. Although this system is perfectly secure, it has some major disadvantages. The key is as long as the message itself and the key can be used only once. This means that a new, completely random key must be generated and transmitted each time the system is used.

2.4 Modern Cryptosystems
In this section three modern cryptosystems are presented: the Diffie-Hellman key exchange protocol, ElGamal and RSA. These cryptosystem are all based on public-key cryptography and they are all widely used today. But before moving to these well-known cryptosystems, some other results in cryptography are presented. These are used as building blocks in the public-key cryptosystems and to build bigger protocols.
2.4. MODERN CRYPTOSYSTEMS

2.4.1 Crypto-graphic Primitives

First of all, the notion of a cryptographic hash function is explained. Next, information on the block ciphers DES and Triple-DES is given, followed by a brief explanation of the symmetric block cipher AES. The section ends with definitions of a trapdoor one-way functions and keyed randomization functions.

Hash Functions

In section 2.1.1 it was mentioned briefly that hash functions are used to ensure integrity. As explained previously, a hash function maps data of any size to a hash value of fixed size. Usually, this is a mapping from a binary string of any size to a binary string of fixed size. The fixed size often ranges between 64 and 256 bits. Furthermore, it is required that it is easy to evaluate the hash function [33].

A special class of hash functions are the so-called cryptographic hash functions [33].

Definition 2.37. Let \( H : \{0,1\}^* \rightarrow \{0,1\}^k \) denote a hash function, so \( H(x) \) is easy to compute for any \( x \in \{0,1\}^* \). Then \( H \) is called a cryptographic hash function if all of the following properties hold:

- **Preimage resistance or onewayness.** Given hash value \( y \in \{0,1\}^k \) of \( H \), it is computationally infeasible to find a preimage \( x \in \{0,1\}^* \) such that \( H(x) = y \).

- **Second preimage resistance or weak collision resistance.** Given the preimage \( x \in \{0,1\}^* \), it is computationally infeasible to find a second preimage \( x' \in \{0,1\}^* \) with \( x' \neq x \), such that \( H(x) = H(x') \).

- **Collision resistance or strong collision resistance.** It is computationally infeasible to find a collision \((x, x')\) with \( x, x' \in \{0,1\}^* \) and \( x \neq x' \), such that \( H(x) = H(x') \).

A hash value can be compared to a fingerprint of a message: a relatively small and unique tag that encapsulates the complete message. Well-known examples of hash functions are: MD4, MD5, SHA-0, SHA-1 and SHA-256. An application of SHA-1 can be found in figure 2.9 on p. 30.

Hash functions are often used in combination with a private key that is shared between the sender and receiver of the message. In this case, the hash function is called a Message Authentication Code (MAC). A MAC can be used to ensure both integrity and authenticity. Furthermore, hash functions can be used as randomization functions. These will be discussed briefly on p. 22. In practice, a hash function and a randomization function are basically the same. Both function map an input of arbitrary size to a binary output of fixed sized. More information on hash functions and randomization functions can be found in [14].

DES and Triple DES

In 1977, the Data Encryption Standard (DES) was published by the American International Business Machines Corporation (IBM). They developed this symmetric-key cryptosystem for data encryption as a response to a request of the National Bureau of Standards (NBS). Three years earlier, the NBS had challenged American industry to develop a cryptosystem that could be used for the protection of sensitive data of the government.

DES is a so-called block cipher.

Definition 2.38. A block cipher is an algorithm that encrypts a fixed number of bits from the plaintext to obtain the ciphertext, using a key. The same key is used to decrypt the ciphertext.

DES uses a block size of the plaintext of 64 bits. Also the key consists of 64 bits. However, only 56 bits of these are effective: 8 bits are used to accomplish an odd parity in each block of 8 bits. More details on DES can be found in [33].
For twenty years the standard had been used successfully. However, one of the weaknesses of this system was the length of key (the keys consists of 64 bits, but the effective key size is only 56 bits). Already when DES was proposed it was suspected that the standard could be broken. Indeed, in 1998 DES was broken by a brute-force attack. In the meantime, the standard had been modified. This improved version of DES was called triple DES, which refers to the three DES implementations of the standard.

In figure 2.3 on p. 20 can be seen how triple DES uses three executions of the standard. The second implementation is a decryption instead of an encryption, denoted by DES$^{-1}$. By using two different keys, the effective key size is doubled compared to DES (112 bits instead of 56 bits). Note that an implementation of triple DES can also be used to handle normal DES by setting key 2 equal to key 1.

![Figure 2.3: Triple DES.](image)

In 1997, the American National Institute of Standards and Technology (NIST) decided to start a worldwide competition to find a safe successor of DES, named the Advanced Encryption Standard (AES) [20]. Several algorithms from different countries were submitted. The five finalist of this competition were: RIJNDAEL (Belgium), SERPENT (UK, Israel and Norway), TWOFISH (USA), RC6 (USA) and MARS (USA) [24]. In 2001, the winner of the competition was announced: the RIJNDAEL algorithm, designed by Joan Daemen and Vincent Rijmen, became the successor of DES [21].

**AES**

AES is a symmetric block cipher that encrypts and decrypts data using a key, just as its predecessor. It processes data in blocks of 128 bits and uses keys of 128, 192 or 256 bits. In this section, AES is explained very briefly. A more thorough explanation can be found in [21].

As can be seen in figure 2.4, AES can be divided into three parts: *initial round*, *rounds* and *final round*. First, the plaintext is represented as a $4 \times 4$ matrix, of which each entry represents 8 bits (or 1 byte). This matrix is called the *state* matrix. In each part, the algorithm uses a so-called *round key*. Each round, this key is different. The round keys are derived from the initial key. The algorithm works as follows:

- **Initial round.** The algorithm computes the XOR of the state matrix with the first round key.
- **Rounds.** Next, the algorithm continues for several rounds. The total number of rounds depends on the size of the key. If the key is 128, 192 or 256 bits, the program continues for respectively 9, 11 or 13 rounds. Each round consists of the following transformations:
  - **SubBytes.** Each byte of the state matrix is substituted using a *substitution* or *s-box*.
  - **ShiftRows.** The second row of the state matrix is rotated over one byte, the third row over two bytes and the fourth row over three bytes.
  - **MixColumns.** Each column is multiplied by a fixed matrix.
  - **AddRoundKey.** The state matrix is encrypted by computing the XOR of each column with the corresponding column of the round key.

- **Final round.** In the final round, all transformations are repeated once again, except for **MixColumns**.

Nowadays, AES is the most used algorithm for encryption on the Internet. An illustration of AES can be found in figure 2.9 on p. 30.
2.4. MODERN CRYPTOSYSTEMS

plaintext as state matrix

Initial round

- AddRoundKey
- RoundKey

Rounds

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

Final round

- SubBytes
- ShiftRows
- AddRoundKey

ciphertext

Figure 2.4: Advanced Encryption Standard.
CHAPTER 2. AN INTRODUCTION TO CRYPTOGRAPHY

Trapdoor One-Way Functions and Keyed Randomization Functions

As mentioned before, hash functions can be used as randomization functions. First, the definition of a trapdoor one-way function is given.

**Definition 2.39.** Let \( f : A \rightarrow B \) be a function, so \( f(a) \in B \) for all \( a \in A \). Let \( f^{-1} \) be the inverse of this function, such that \( f^{-1}(b) \in A \) for all \( b \in B \). Then \( f \) is called a trapdoor one-way function if:

- \( f(a) \) is easy to compute for all \( a \in A \);
- \( f^{-1}(b) \) is difficult to compute for most \( b \in B \);
- \( f^{-1}(b) \) is easy to compute for all \( b \in B \) given some specific extra information.

Note that a cryptographic hash function that is completely collision resistance, is actually a one-way function without a trapdoor. By definition, it is easy to evaluate the hash function (first condition of definition 2.39). Furthermore, it is computationally infeasible to find the preimage of a hash value (second condition of definition 2.39).

Trapdoor one-way functions are often used to describe public-key cryptosystems. Since the public key is known to everybody, it is easy to encrypt a message. However, it is difficult to decrypt the message, except for someone that knows the private key.

The definition of a trapdoor one-way function can be used to build a keyed randomization function. A keyed randomization function is a trapdoor one-way function that outputs a random value of fixed size when given an input. Random means that the output of the function is uniformly distributed. The key can be used as trapdoor to compute the inverse of the function [16]. Note that this function is symmetric: the key for encryption and the key for decryption are the same.

Keyed randomization functions are used in some of the described SETUP attacks. An important property of these functions is that the attacker can invert them using a key. One might think that the use of these functions is not secure, since a keyed randomization function is a symmetric system. However, it is important to note that in SETUP these functions are used in combination with public-key cryptography.

2.4.2 Diffie-Hellman Key Exchange

The very first publication of a key exchange protocol with both public and secret keys appeared in 1976 by Whitfield Diffie and Martin Hellman [8]. In addition, also Ralph Merkle made significant contributions to the development of this method. The idea behind this method is that two parties can agree upon a shared secret (the key), while the communication between these parties can be public. This means that Alice and Bob can communicate with each other to agree upon a secret key, but that Eve cannot construct this key from the information she retrieves from the insecure communication channel. This shows immediately an advantage of an asymmetric cryptosystem: one does not need a secure channel to transfer a secret key.

The security of the Diffie-Hellman key exchange method is closely related to the discrete logarithm problem. The method can be divided into four parts: defining the system parameters and computing the key pairs for the actual key exchange to obtain a shared secret key. The four parts are defined as follows:

- **System parameters.** Alice and Bob initialize the system by sharing a prime number \( p \) and a generator \( g \) of \( \mathbb{Z}_p^* \). These system parameters are public.
- **Key pair generation.** Both Alice and Bob generate two keys. First, the private keys, \( x_A \) and \( x_B \), are chosen from \( \{1, 2, \ldots, p-1\} \). Secondly, the public keys are generated:

  \[
  y_A \equiv g^{x_A} \pmod{p} \quad \text{and} \quad y_B \equiv g^{x_B} \pmod{p}.
  \]

- **Key exchange.** Alice and Bob send each other their public key.
- **Shared secret key.** Alice and Bob both compute the shared secret key. Alice computes:

  \[
  s_A \equiv y_B^{x_A} \pmod{p}
  \]

  and Bob computes:

  \[
  s_B \equiv y_A^{x_B} \pmod{p}.
  \]

Then \( s_A = s_B = s \), the shared secret key.
In figure 2.5 a visual representation of the Diffie-Hellman key exchange can be found.

\[
\begin{align*}
\text{system parameters} & \quad p, g & \quad p, g & \quad p, g \\
\text{key pair generation} & \quad x_A & \quad y_A \equiv g^{x_A} \pmod{p} & \quad y_B \equiv g^{x_B} \pmod{p} \\
\text{key exchange} & \quad y_B & \quad y_A & \quad s_A \equiv y_B^{x_A} \pmod{p} \\
\text{shared secret key} & \quad s_B \equiv y_A^{x_B} \pmod{p}
\end{align*}
\]

Figure 2.5: Diffie-Hellman key exchange.

Since \(x_A \cdot x_B = x_B \cdot x_A\), the shared secret keys computed by Alice and Bob are indeed the same:

- \(s_A \equiv y_B^{x_A} \equiv (g^{x_B})^{x_A} \equiv g^{x_B \cdot x_A} \pmod{p}\);
- \(s_B \equiv y_A^{x_B} \equiv (g^{x_A})^{x_B} \equiv g^{x_A \cdot x_B} \pmod{p}\).

Eve only knows \(p, g, y_A \equiv g^{x_A}\) and \(y_B \equiv g^{x_B}\). If she is able to solve the discrete logarithm problem, she can compute \(x_A\) and \(x_B\) and therefore the shared secret key. However, she does not even have to know \(x_A\) and \(x_B\) separately, \(g^{x_A \cdot x_B}\) would already suffice. This variant of the discrete logarithm problem is called the computational Diffie-Hellman problem.

Definition 2.40. Let \(G\) be a group and let \(g\) be the generator of a cyclic subgroup of \(G\). The computational Diffie-Hellman problem is defined as follows. Given \(g^x, g^y \in (g)\), compute \(g^{x \cdot y}\).

This problem is no harder than the discrete logarithm problem, since if Eve can solve the discrete logarithm problem, she can also solve the Diffie-Hellman problem. However, if she knows how to solve the Diffie-Hellman problem, she can still not solve the discrete logarithm problem.

Example 2.10: The Diffie-Hellman key exchange method

The steps are executed as follows:

- **System parameters.** Let \(p = 151\) and \(g = 6\).
- **Key pair generation.** Assume Alice chooses \(x_A = 123\) and Bob chooses \(x_B = 8\). Then:
  \(y_A \equiv g^{x_A} \equiv 6^{123} \equiv 83 \pmod{151}\) and \(y_B \equiv g^{x_B} \equiv 6^8 \equiv 43 \pmod{151}\).
- **Key exchange.** Alice receives \(y_B = 43\) from Bob and Bob receives \(y_A = 83\) from Alice.
- **Shared secret key.** Alice computes:
  \(s_A \equiv y_B^{x_A} \equiv 43^{123} \equiv 44 \pmod{151}\)
  and Bob computes:
  \(s_B \equiv y_A^{x_B} \equiv 83^8 \equiv 44 \pmod{151}\).

Indeed, \(s_A = s_B\).

2.4.3 ElGamal

The Diffie-Hellman key exchange method is closely related to the discrete logarithm problem. Other cryptosystems that are related to this problem, are the two systems designed by Tahar ElGamal in 1985 [9]. He designed a system for asymmetric key encryption and a digital signature scheme.
ElGamal Encryption

The ElGamal encryption cryptosystem consists of four parts: the definition of the system parameters, the key pair generation, the encryption of the plaintext and the decryption of the ciphertext:

- **System parameters.** First, the system parameters must be chosen: a prime number \( p \) and a generator \( g \) of \( \mathbb{Z}_p^* \). These system parameters are known to everybody.
- **Key pair generation.** Every user of the system generates a key pair. User \( U \) generates a private key \( x_U \in \{1, 2, \ldots, p-1\} \) and computes the corresponding public key \( y_U \) as follows:
  \[
y_U \equiv g^{x_U} \pmod{p}.
  \]

If Bob wants to be able to receive encrypted messages from Alice, Alice should use Bob’s public key for encryption. On the contrary, if Alice wants to receive messages from Bob, Bob should use Alice’s public key to encrypt. So both Alice and Bob publish their public key, such that they can receive encrypted messages from each other.

- **Encryption.** Assume that Alice wants to send a message to Bob. To do so, she first chooses a random number \( k \in \{1, 2, \ldots, p-1\} \). Next, she computes:
  \[
r \equiv g^k \pmod{p},
  \]
  using Bob’s public key \( y_B \). Alice sends the pair \((r, s)\) to Bob.
- **Decryption.** Bob receives the pair \((r, s)\) from Alice and decrypts the message as follows:
  \[
m \equiv \frac{s}{r^x_B} \pmod{p},
  \]
  using his own private key \( x_B \).

Figure 2.6 shows a visual representation of the ElGamal encryption method. This decryption recovers indeed the correct message:

\[
s \equiv m \cdot y_B^k \equiv m \cdot (g^{x_B})^k \equiv m \cdot g^{x_B k} \equiv m \pmod{p}.
\]

Eve knows the two system parameters \( p \) and \( g \), the public key \( y_B \) and she can intercept the message \((r, s)\). To be able to decrypt the message, she needs the private key \( x_B \) or the random number \( k \). For the first
case, she knows that \( y_B \equiv g^{x_B} \pmod{p} \), where \( x_B \) is the only unknown integer for her. However, this is again the discrete logarithm problem. This means that Eve must solve the discrete logarithm problem first and then she can break the system. If Eve could find the integer \( k \), she would be able to compute \( y_B^k \) and recover the message \( m \) from \( s \). However, if she wants to compute \( k \) from \( r \equiv g^k \pmod{p} \), she has to solve the discrete logarithm problem again.

**Example 2.11: Key pair generation, encryption and decryption with ElGamal**

The steps are executed as follows:

- **System parameters.** Let \( p = 2311 \) and \( g = 3 \).
- **Key pair generation.** Bob chooses \( x_B = 175 \) as private key and computes the public key:
  \[ y_B \equiv g^{x_B} \equiv 3^{175} \equiv 2089 \pmod{2311} \]
- **Encryption.** First, Alice chooses \( k = 144 \). Then she computes:
  \[ r \equiv g^k \equiv 3^{144} \equiv 1203 \pmod{2311} \]
  Alice wants to send the message “HI”. She encodes this as \( m = 0809 \) and encrypts:
  \[ s \equiv m \cdot y_B^k \equiv 809 \cdot 2089^{144} \equiv 408 \pmod{2311} \]
  She sends \((r, s) = (1203, 408)\) to Bob.

- **Decryption.** Bob receives \((1203, 408)\) and computes:
  \[ m \equiv s \cdot r^{x_B} \equiv 408 \cdot 1203^{175} \equiv 809 \pmod{2311} \]
  which he decodes as “HI”.

**ElGamal Signature Scheme**

The ElGamal signature scheme consists of four parts: the definition of the system parameters, the key pair generation, the signature generation and the signature verification:

- **System parameters.** First, the system parameters must be chosen: a prime number \( p \) and a generator \( g \) of \( \mathbb{Z}_p^* \). These system parameters are shared between Alice and Bob and known to everybody.
- **Key pair generation.** Just as in ElGamal encryption, every user \( U \) generates a private and a public key. Alice can use Bob’s public key for signature verification and vice versa.
- **Signature generation.** Assume that Alice wants to sign a message (that can be verified by Bob). Just as in the ElGamal encryption system, Alice represents her message as an integer \( m \), such that \( 0 \leq m \leq p - 1 \). Next, she chooses a random integer \( k \in \{1, 2, \ldots, p - 2\} \) such that \( \gcd(k, p - 1) = 1 \) and she computes:
  \[ r \equiv g^k \pmod{p} \]
  The integer \( k \) and \( p - 1 \) should be relatively prime. This is because the protocol needs the inverse of \( k \) modulo \( p - 1 \). This inverse does not exist if \( \gcd(k, p - 1) \neq 1 \) (theorems 2.18 and 2.19). Therefore \( k \in \{1, 2, \ldots, p - 2\} \) instead of \( k \in \{1, 2, \ldots, p - 1\} \). If \( k = p - 1 \), then \( \gcd(p - 1, p - 1) = p - 1 \neq 1 \) and the inverse does not exist. After the computation of \( r \), Alice uses her private key \( x_A \) to compute \( s \) such that:
  \[ m \equiv x_A \cdot r + k \cdot s \pmod{p - 1} \]
  So:
  \[ s \equiv k^{-1}(m - x_A \cdot r) \pmod{p - 1} \]
  She can do this using the extended Euclidean Algorithm. Finally, Alice sends \((m, r, s)\) to Bob.
- **Signature verification.** Bob receives \((m, r, s)\) from Alice and verifies that:
  \[ g^m \equiv y_A^r \cdot r^s \pmod{p} \]
In this description, it is mentioned that the message $m$ should be smaller than $p - 1$. However, in practice it is not uncommon that $m$ is longer than $p - 1$. Therefore in practice $H(m)$ is used instead of $m$, where $H$ denotes a cryptographic hash function. The cryptographic hash function $H$ shortens the message and makes sure that $H(m)$ is smaller than $p - 1$. This means that the hash value of the message is usually shorter than the message itself, which makes signature generation work faster. A visual representation of the ElGamal signature scheme can be found in figure 2.7 on p. 26.

![Figure 2.7: Authentication with ElGamal signature scheme.](image)

The verification by Bob is correct, since:

$$g^m \equiv g^{x_A \cdot r + k \cdot s} \equiv g^{x_A \cdot r} \cdot g^{k \cdot s} \equiv (g^{x_A})^r \cdot (g^k)^s \equiv y_A^r \cdot r^s \pmod{p}.$$  

**Example 2.12: Authentication with ElGamal signature scheme**

The steps are executed as follows:

- **System parameters.** Let $p = 2311$ and $g = 3$.
- **Key pair generation.** Alice chooses $x_A = 175$ as private key and computes the public key:

  $$y_A \equiv g^{x_A} \equiv 3^{175} \equiv 2089 \pmod{2311}.$$  

- **Signature generation.** First, Alice chooses $k = 149$. Then she computes:

  $$r \equiv g^k \equiv 3^{149} \equiv 1143 \pmod{2311}.$$  

Alice wants to sign the message “HI”. She encodes this as $m = 0809$. Using the extended Euclidean algorithm, she finds that:

$$\gcd(k, p - 1) = 1 = \gcd(149, 2310) = -31 \cdot 149 + 2 \cdot 2310,$$

such that:

$$k^{-1} = 149^{-1} \equiv -31 \equiv 2279 \pmod{2310}.$$  

She encrypts:

$$s \equiv k^{-1}(m - x_A \cdot r) \equiv 2279(809 - 175 \cdot 1143) \equiv 1066 \pmod{2310}.$$  

She sends $(m, s, r) = (809, 1143, 1066)$ to Bob.
2.4. Signature verification. Bob receives \((809, 1143, 1066)\) and computes:

\[ y_A^r \cdot r^s \equiv 2089^{1203} \cdot 1203^{1066} \equiv 138 \pmod{2311}, \]

which is equal to:

\[ g^m \equiv 3^{809} \equiv 138 \pmod{2311}. \]

So Bob concludes that the message is indeed from Alice.

2.4.4 RSA

In 1978, Ronald Rivest, Adi Shamir and Leonard Adleman introduced a new asymmetric cryptosystem [23]. This system became known as RSA, named after its inventors. RSA can be divided into three different parts: the key pair generation, encryption of the plaintext and decryption of the ciphertext.

- **Key pair generation.** Every user \(U\) of the system generates a private and a public key. User \(U\) performs the following steps:
  - First, \(U\) generates two different random primes \(p_U\) and \(q_U\). These primes need to remain secret and should be destroyed as soon as the key pair generation is complete.
  - The next step is the computation of the modulus \(n_U = p_U \cdot q_U\). This modulus \(n_U\) does not remain secret. The modulus \(n_U\) will be part of both \(U\)’s public and private key.
  - \(U\) computes:
    \[ \varphi(n_U) = (p_U - 1)(q_U - 1). \]
    Also this integer should be destroyed immediately after the key pair generation, together with the primes \(p_U\) and \(q_U\).
  - Next, \(U\) chooses an integer \(e_U\) (encryption exponent), such that \(1 < e_U < \varphi(n_U)\) and \(\gcd(e_U, \varphi(n_U)) = 1\). This encryption exponent will be part of the public key.
  - Then, \(U\) needs to compute the inverse of \(e_U \pmod{\varphi(n_U)}\), so:
    \[ d_U \equiv e_U^{-1} \pmod{\varphi(n_U)}. \]
    \(U\) uses the extended Euclidean algorithm to do so (as in example 2.4). This \(d_U\) (decryption exponent) is a part of the private key.
  - \(U\)’s public key consists of the pair \((e_U, n_U)\) and her private key consists of the pair \((d_U, n_U)\). Alice and Bob both publish their public key. If Alice wants to send an encrypted message to Bob, she should use his public key. Bob will use his own private key to decrypt this message. The other way round, if Bob wants to send a message, he uses Alice’s public key for encryption and Alice uses her own private key for decryption.

- **Encryption.** Assume Alice wants to send a message to Bob. She uses Bob’s public key \((e_B, n_B)\) to encrypt her message. First she represents her message as an integer \(m\), such that \(0 \leq m < n_B\). Then this message \(m\) is encrypted as follows:

  \[ c = E(m) = m^{e_B} \pmod{n_B}. \]

  If she wishes to send a longer message, she can break his message into multiple parts. Each part will then be encrypted separately.

- **Decryption.** Bob receives \(c\) from Alice and decrypts this using his private key \((d_B, n_B)\):

  \[ D(c) = c^{d_B} \pmod{n_B}. \]

Note that it is possible to use \(m = 0\) or \(m = 1\). However, this is not very wise to do so, since then the ciphertext will be equal to the plaintext. A visual representation of RSA is shown in figure 2.8 on p. 28.

To show that decrypting the ciphertext indeed gives the correct plaintext, one needs to show that \(D(E(m)) = m\). First, consider the case where \(m\) is relatively prime to \(n_B\), so \(\gcd(m, n_B) = 1\). Then, according to the Fermat-Euler theorem (theorem 2.32):

\[ m^{\varphi(n_B)} \equiv 1 \pmod{n_B}. \]


Since \(d_B\) is the inverse of \(c_B\), \(e_B \cdot d_B \equiv 1 \mod \varphi(n_B)\). This implies that the product of the two equals one plus a multiple of \(\varphi(n_B)\), so \(e_B \cdot d_B = 1 + k \cdot \varphi(n_B)\) with \(k \in \mathbb{Z}\). Then decryption gives:

\[
c_B^{d_B} \equiv (m^{e_B})^{d_B} \equiv m^{e_B \cdot d_B} \equiv m^{1+k \cdot \varphi(n_B)} \equiv m^1 \cdot m^{k \cdot \varphi(n_B)} \equiv m \cdot (m^{\varphi(n_B)})^k \equiv m \cdot 1^k \equiv m \mod n_B.
\]

So indeed, \(c_B^{d_B} \equiv m \mod n_B\) if \(\gcd(m, n_B) = 1\).

To prove that decrypting the ciphertext gives the correct plaintext for \(\gcd(m, n_B) > 1\), it needs to be shown that:

\[
c_B^{d_B} \equiv m^{e_B \cdot d_B} \equiv m \mod n_B.
\]

Since \(n_B = p_B \cdot q_B\), this can be rewritten as:

\[
m^{e_B \cdot d_B} \equiv m \mod p_B \cdot q_B.
\]

Then, according to the Chinese remainder theorem (theorem 2.34), it suffices to show that:

\[
m^{e_B \cdot d_B} \equiv m \mod p_B
\]

and

\[
m^{e_B \cdot d_B} \equiv m \mod q_B.
\]

Note that \(0 \leq m < n_B\), so \(0 \leq m < p_B \cdot q_B\). Then there are two possibilities:

- \(p_B | m\) and \(q_B \nmid m\);
- \(p_B \nmid m\) and \(q_B | m\).

Note that if \(p_B \nmid m\) and \(q_B \nmid m\), then the message \(m\) is relatively prime to \(n_B\). Since \(p_B\) and \(q_B\) are both primes and \(m < p_B \cdot q_B\), it is not possible that both \(p_B\) and \(q_B\) divide \(m\).

Assume that \(p | m\) and \(q \nmid m\). Since \(p|m\):

\[
m^{e_B \cdot d_B} \equiv 0^{e_B \cdot d_B} \equiv 0 \equiv m \mod p,
\]

which satisfies equation 2.5. Since \(q \nmid m\), Fermat’s little theorem (theorem 2.30) can be used:

\[
m^{\varphi(q_B)} - 1 \equiv 1 \mod q_B,
\]

such that:

\[
m^{e_B \cdot d_B} \equiv m^{e_B \cdot d_B - 1} m \equiv m^{k(p_B - 1)(q_B - 1)} m \equiv (m^{\varphi(q_B)} - 1) k(p_B - 1) m \equiv 1^{k(p_B - 1)} m \equiv m \mod q_B,
\]

which states equation 2.6. Again, both equation 2.5 and equation 2.6 are true, which proves that decryption works.
• Assume that $p \nmid m$ and $q | m$. The proof of this case goes the same as in the previous case, but then the other way around. This shows that in all possible cases decryption indeed provides the correct plaintext, so:
\[c^{d_B} \equiv m^{e_B \cdot d_B} \equiv m \pmod{n_B},\]
if $\gcd(m, n_B) = 1$ and if $\gcd(m, n_B) > 1$.

Note that for the security of the system the best case would be that $m$ is relatively prime to $n_B$. If this is not the case, one of the primes is a divisor of the message. Assume that $p_B$ divides $m$. This means that $p_B$ divides both $m$ and $n_B$. Then one could compute the gcd of $m$ and $n_B$ to find prime $p_B$. However, note that Alice does not know Bob’s prime factors. She writes her own message and she does not know whether her message is relatively prime to Bob’s modulus or not. It is possible to compute the probability that Alice’s message is relatively prime the modulus, but this will not be discussed in this thesis.

Eve knows Bob’s public key $(e_B, n_B)$ and the encrypted message $c$. She also knows that $d_B \equiv e_B^{-1} \pmod{\varphi(n_B)}$. This means that if Eve can compute $\varphi(n_B)$, she can compute the inverse of $e_B$ to find $d_B$ and decrypt the message. If $n_B$ would be a small number, she could easily compute $\varphi(n_B)$. However, in practice $n_B$ is too big to be able to compute $\varphi(n_B)$, so Eve needs another approach. Since Eve knows that $n_B = p_B \cdot q_B$, she could also try to find the prime factorization of $n_B$, but this is, just as the discrete logarithm problem, a hard problem, known as the integer factorization problem (definition 2.36).

Example 2.13: Key pair generation, encryption and decryption with RSA

The steps are executed as follows:

- **Key pair generation.** Bob generates the two key pairs as follows:
  - $p_B = 367$ and $q_B = 983$;
  - $n_B = p_B q_B = 367 \cdot 983 = 360761$;
  - $\varphi(n_B) = (p_B - 1)(q_B - 1) = 366 \cdot 982 = 359412$;
  - $e_B = 65537$;
  - $d_B = 190145$;
  - public key $(65537, 360761)$;
  - private key $(190146, 360761)$.

- **Encryption.** Alice wants to send the message “CAT”. She encodes this as $m = 030119$ and encrypts:
  \[c = m^{e_B} = 30119^{65537} \equiv 357592 \pmod{360761}.

- **Decryption.** Bob receives $c = 357592$ from Alice and decodes this using his private key:
  \[m = c^{d_B} = 357592^{190145} \equiv 30119 \pmod{360761}.

Finally, he decodes this as “CAT”.

2.5 Applications of Cryptography

The main goals of cryptography are confidentiality, authentication and integrity. The first cryptosystems were designed to protect the communication between multiple parties. Caesar, for example, would encrypt his message and send a messenger with the secret message on a horse to someone else to share his battle tactics. Although nowadays one would probably not send a messenger on a horse anymore, the idea is still the same. If Alice sends a confidential e-mail to Bob, she wants to be sure Eve cannot read it as well. Some examples of applications of cryptography in daily life are: bank cards and credit cards, online banking and PGP.

2.5.1 Bank Cards and Credit Cards

Most people own nowadays one or more bank or credit cards. These cards can be used to, among other things, pay in stores and withdraw money from an ATM. The first bank cards made use of a
magnetic stripe. This magnetic stripe turned out to be vulnerable to *skimming* and therefore Europay, MasterCard and Visa introduced the EMV-standard as safer alternative. Instead of a magnetic stripe, a chip is integrated in the card.

The EMV-standard uses multiple cryptographic methods to ensure the security of the system. For example, if someone uses a card to pay in a store, the entered *personal identification number* (PIN) is verified by the terminal and the card using asymmetric cryptography. The chip in the bank card contains a public key pair, consisting of a public and a private key. The public key is used by the terminal to encrypt the PIN and the private key is used by the card itself to decrypt the encrypted PIN. For the encryption and decryption of the PIN, RSA is used [10].

### 2.5.2 Online Banking

Since the end of the 20th century, it has become very common to bank online. Through any internet connection, one can connect to the bank to, for example, check the balance of the account, make money transactions and pay bills. Especially with the rise of smartphones and tablets it is nowadays very easy to connect to the internet anywhere in the world to be able to bank online. It is evident that the connection between the user and the bank should be secure. This means that all sorts of safety measures are applied to prevent the interception of a third party.

Many banks use SSL (*Secure Sockets Layer*) or its successor TLS (*Transport Layer Security*) to secure the connection between the user and the server. These protocols make use of so-called *certificates* to authenticate the user. The secure channel is used to transfer the password and the data. Such a secure channel is not only used for online banking, but also for other websites that require login [4].

![Connection details ING.](image)

As soon as a website uses a secure channel, this is visible by the padlock in the browser. If one clicks the padlock, one can see the details of the secure channel. In figure 2.9 the details of the connection between a user and the Dutch bank ING are shown. Here one can see that the connection uses indeed TLS. Furthermore, the connection is encrypted with AES, SHA-1 is used for message authentication and finally RSA is used as the public key encryption mechanism.

### 2.5.3 Pretty Good Privacy

In 1991, the American Phil Zimmermann created a computer program for data encryption and decryption, named *Pretty Good Privacy* (PGP). He forwarded the program to a friend, who uploaded it on the internet so that everyone could use it. The program gained popularity in and outside the US. This was a development the US Government was not very content with. The US Government had included cryptography on the US Munitions List, which meant that exporting cryptographic programs was actually illegal. They started an investigation of him. Furthermore, Zimmermann had also some issues with RSA Data Security since he used the RSA algorithm in his program. Eventually, all problems were solved and nowadays PGP is used all over the world [18].

PGP uses both symmetric and asymmetric cryptography. Assume Alice wants to send a message to
Bob. First, the data is compressed before encryption. This saves both time and space, but also improves the security. Then, the program generates a so-called session key, which is used only once. This session key is used to symmetrically encrypt the compressed message. Next, the session key is encrypted, using Bob’s public key. Alice sends the encrypted message and the encrypted session key to Bob. Finally, Bob can use his private key to recover the session key and decrypt Alice’s message with this session key [44].
Chapter 3

Kleptography

In 1996, Adam Young and Moti Yung published an article in which they warn about the danger of black-box cryptography [36]. This term is used to denote cryptosystems of which only the input and output are accessible. This means that the user cannot access the algorithm and memory of the system [40]. Since the user does not have access to these internal workings, he is exposed to several threats. Young and Yung introduced their so-called SETUP mechanism (Secretly Embedded Trapdoor with Universal Protection), that gives the manufacturer of the black-box cryptosystem the possibility to exfiltrate secret key information without the notice of its users. Besides, this mechanism protects against other attackers and against reverse-engineering (i.e. the recovery of the black-box design by others). The term kleptography is used to denote this attack [37].

In this chapter, the articles by Young and Yung and their SETUP mechanism take center stage. First, more cryptographic background information is provided, which will be essential for the sections on SETUP. In the second section, a brief history of kleptography is given, together with a timeline of the most significant articles. This section is followed by a section on SETUP itself. The name “SETUP” is explained and several formal definitions are given. Young and Yung have shown how SETUP can be embedded in several cryptosystems. In the final three sections of this chapter, embeddings in RSA, ElGamal key generation and signature scheme and Diffie-Hellman key exchange are explained.

The descriptions of each SETUP attack have approximately the same structure. Each description contains at least the following information. For each attack, the algorithm that should be implemented in a device is presented. It is also explained how the attacker can exfiltrate user’s private information. After this explanation, a worked-out example is included to show that the attack indeed works. It is important to note that for these examples simplified versions of the algorithm are used. Several systems loop over random choices till a suitable one is found. In the examples these loops are omitted and only final suitable choices are shown. Furthermore, relatively small integers (in decimal representation) are used. The main purpose of these examples is to show that the attack works and does not interfere with the common procedures. Decimal representations are used, so that the reader can easily verify the computations. Finally, for each attack a brief summary is given. For some attacks, other information is included as well.

3.1 Background Information

The cryptographic information provided in the first chapter is basically standard information that can be found in any introduction to cryptography. In this section, more advanced topics are discussed.

3.1.1 Subliminal Channels

Subliminal channels were introduced for the first time by Simmons in 1984 as a solution to the following prisoners’ problem [30]. Assume Alice and Bob are arrested for committing a crime together. They are brought to a prison and placed in separate cells, such that they cannot communicate with one another. However, the warden of the prison, Eve, allows Alice and Bob to communicate, as long as she can monitor all the messages. Alice and Bob accept the proposal, since they like to plan an escape. Nonetheless, this
CHAPTER 3. KLEPTOGRAPHY

is not without risk: Eve could mislead Alice and Bob by creating messages herself or by modifying their messages. Despite this risk, Alice and Bob still want to communicate to discuss their plans. They have to create a secret communication channel using Eve, without her getting suspicious. This channel is called the subliminal channel. To prevent that Eve passes fraudulent messages, Alice and Bob insist that they are allowed to use authentication by signing the messages.

Authentication Without Secrecy

Simmons provides a solution to this prisoners’ problem by describing a way to create a subliminal channel. First, he considers the communication between Alice and Bob without the exchange of secret information, the so-called authentication without secrecy channel. If Alice and Bob want to communicate with each other, Alice can add some prearranged redundant information to the message. If Bob decrypts the message, he notices the prearranged redundant information and knows the message is really from Alice. They can, for example, end every message with a specific sequence. If Bob cannot find this sequence, he knows that the message is not legitimate. Note that the redundant information should be encrypted, otherwise Eve would be able to subtract and use the redundant information easily.

Assume that the message and the redundant information are encrypted as a block cipher. Bob will only accept the message if he can recognise the redundant information. Let \( H_r \) be the number of bits of the redundant information for authentication. Eve could try to guess the exact content of this information. For example, if \( H_r = 3 \), there are only eight different possibilities, namely 000, 001, 010, 011, 100, 101, 110 and 111. The probability that Eve will guess the correct redundant information is then \( 2^{-3} \). In general this means that the probability that Eve guesses the correct redundant information (denoted by \( P_A \)) equals \( 2^{-H_r} \), so \( P_A = 2^{-H_r} \).

For encryption and decryption, there are two possibilities: a single key or a two key algorithm (i.e. symmetric or asymmetric cryptography). If Alice and Bob use symmetric cryptography for encryption and authentication, Alice must be able to send the key and a message to Bob before she shares the key with Eve. If Eve would have the key beforehand, she knows how to encrypt messages, since the keys for encryption and decryption are the same. This means that Eve could send fraudulent messages to Alice and Bob. If Alice and Bob use asymmetric cryptography for authentication, Alice and Bob can immediately share their public keys with Eve. Eve can check the signatures using these public keys. However, since she does not know the corresponding private keys, Eve cannot impersonate Alice and/or Bob by creating false signatures. Therefore, the advantage of the use of asymmetric cryptography is that Alice and Bob can share their public keys with Eve immediately, without the risk of being impersonated.

Subliminal Channel

Let \( H_m \) be the number of bits of the message Alice wants to send to Bob. Then in total \( H_m + H_r \) bits need to be send. The idea of a subliminal channel is that Alice and Bob take some bits of the redundant information, \( H_s \) bits, so that they can use these bits to communicate secretly. Eve does not know the redundant information and will not notice this. Note that it does mean that less bits remain for authentication, namely \( H_r - H_s \), and that the probability that Eve guesses the remaining redundant information correctly increases. However, the need for secret communication between Alice and Bob is more important than the increase of this probability, of which Eve is not even aware.

A year after his first publication on subliminal channels, Simmons implemented the subliminal channels in two digital signature schemes: the Ong-Schnor-Shamir signature scheme and the ElGamal signature scheme [31]. The solution to the prisoners’ problem is also described by Young and Yung [40]. Assume that Alice wants to send a message to Bob and assume that they agreed upon Alice’s signing private key \( x \) before Eve put them in prison. Young and Yung describe the solution as follows:

- Alice writes a normal, harmless message that needs to be signed. If Eve reads this message, she will not suspect that Alice and Bob are about to mislead her.
- Alice signs this message with the signing private key \( x \) and a modified signing algorithm such that the subliminal message is encoded within the signature.
- Eve receives the signed message from Alice.
- Eve authenticates the signature, reads the message, concludes that nothing is wrong and forwards the signed message to Bob.
- Bob authenticates the signature and reads the message.
• Bob uses the signing private key $x$ to decode the signature and extract the subliminal message (see below for details of a subliminal channel in ElGamal signature scheme).

Young and Yung emphasize that even if Eve knows about the subliminal channel, she is unable to reveal it or read the subliminal message, since she does not know the private signing key $x$. However, as soon as Eve suspects that Alice and Bob are misleading her, she could always refuse to pass messages, which means that Alice and Bob cannot communicate at all. Young and Yung define a subliminal channel as follows.

**Definition 3.1.** A subliminal channel is a channel that exists within a cryptographic protocol, authentication system, digital signature algorithm, and so on, that transmits additional messages to a (special) receiver that is hidden, such that the messages cannot be read by other receivers.

**Subliminal Channel in ElGamal Signature Scheme**

As mentioned before, in 1985 Simmons implemented a subliminal channel in the Ong-Schnor-Shamir signature scheme and the ElGamal signature scheme [31]. The latter of these two is explained in this section. In section 2.4.3 (p. 23) the original ElGamal signature scheme can be found.

To generate a subliminal channel, Bob needs to know Alice’s private key $x_A$. In the prisoners’ problem this would mean that Alice tells Bob her secret key before they are put in prison. Let $m$ be the normal message and $m'$ the subliminal, secret message. To generate a subliminal channel, the signature generation of the original ElGamal scheme is adapted and the decoding of the subliminal message is added to the total scheme. These steps are performed as follows:

- **Signature generation.** Alice computes:

  $$c_1 \equiv g^{m'} \pmod{p}.$$  

  Then, she uses her private key $x_A$ to compute $c_2$ such that:

  $$m \equiv x_A \cdot c_1 + m' \cdot c_2 \pmod{p-1}.$$  

  So:

  $$c_2 \equiv (m')^{-1}(m - x_A \cdot c_1) \pmod{p}.$$  

  Alice sends the triple $(m, c_1, c_2)$ to Bob.

- **Decoding.** Bob receives $(m, c_1, c_2)$ from Alice. After verification of the signature, he retrieves the subliminal message as follows:

  $$m' \equiv c_2^{-1}(m - x_A \cdot c_1) \pmod{p-1}.$$  

  Note that to be able to decode the subliminal message, $\gcd(c_2, p - 1) = 1$ must hold, otherwise $c_2^{-1}$ does not exist.

In figure 3.1 the signature generation of the original ElGamal signature scheme is compared to the signature scheme with the subliminal channel. In the original scheme, Alices chooses a random $k$, whereas in the scheme with the subliminal channel, she uses the subliminal message $m'$. Note that also $\gcd(m', p - 1) = 1$ must hold, otherwise $(m')^{-1}$ does not exist.

<table>
<thead>
<tr>
<th>ElGamal signature scheme</th>
<th>ElGamal signature scheme with subliminal channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$c_1 \equiv g^{m'} \pmod{p}$</td>
</tr>
<tr>
<td>$c_1 \equiv g^k \pmod{p}$</td>
<td>$c_2 \equiv (m')^{-1}(m - x_A \cdot c_1) \pmod{p - 1}$</td>
</tr>
<tr>
<td>$c_2 \equiv k^{-1}(m - x_A \cdot c_1) \pmod{p - 1}$</td>
<td>$(m, c_1, c_2)$</td>
</tr>
</tbody>
</table>

Figure 3.1: Signature generation in ElGamal signature scheme with subliminal channel.
CHAPTER 3. KLEPTOGRAPHY

The signature verification by Bob and Eve still works, since:

\[ g^m \equiv g^{x_A c_1 + m' c_2} \equiv g^{x_A c_1} \cdot g^{m' c_2} \equiv (g^{x_A})^{c_1} \cdot (g^{m'})^{c_2} \equiv y_A^{c_1} \cdot c_2^2 \pmod{p}. \]

Also the decoding of the subliminal message recovers the subliminal message correctly, since \( c_2 \) was defined as follows:

\[ c_2 \equiv (m')^{-1}(m - x_A \cdot c_1) \pmod{p - 1}. \]

This implies that:

\[ m' \equiv c_2^{-1}(m - x_A \cdot c_1) \pmod{p - 1}, \]

which is exactly what Bob computes to recover the subliminal message.

---

Example 3.1: Authentication with ElGamal signature scheme with subliminal channel

The steps are executed as follows:

- **System parameters.** Let \( p = 2311 \) and \( g = 3 \).

- **Key pair generation.** Alice chooses \( x_A = 175 \) as private key and computes the public key:

\[ y_A \equiv g^{x_A} \equiv 3^{175} \equiv 2089 \pmod{2311}. \]

- **Signature generation.** Alice wants to send as subliminal message the time she wants to meet Bob to escape prison, namely 17 minutes past noon, she encodes this as \( m' = 1217 \). Then she computes:

\[ c_1 \equiv g^{m'} \equiv 3^{1217} \equiv 1470 \pmod{2311}. \]

Alice wants to send as normal message “HI”. She encodes this as \( m = 0809 \). Using the extended Euclidean algorithm, she finds that:

\[ \gcd(m', p - 1) = \gcd(1217, 2310) = 503 \cdot 1511 - 265 \cdot 2310, \]

so that:

\[ (m')^{-1} = 1217^{-1} \equiv 503 \pmod{2310}. \]

She encrypts:

\[ c_2 \equiv (m')^{-1}(m - x_A \cdot c_1) \equiv 503(809 - 175 \cdot 1470) \equiv 577 \pmod{2310}. \]

She sends \( (m, c_1, c_2) = (809, 1470, 577) \) to Eve.

- **Signature verification.** Eve receives \( (809, 1470, 577) \) and computes:

\[ g_A^{c_1} \cdot c_2^2 \equiv 2089^{1470} \cdot 1470^{577} \equiv 138 \pmod{2311}, \]

which is equal to:

\[ g^m \equiv 3^{809} \equiv 138 \pmod{2311}. \]

So Eve concludes that the message is indeed from Alice and passes it on to Bob.

- **Decoding.** Bob authenticates the signature the same way Eve did and also concludes that the message is genuine. Finally, he decodes the subliminal message. Using the extended Euclidean algorithm, he finds that:

\[ \gcd(c_2, p - 1) = \gcd(577, 2310) = 1 = 1153 \cdot 577 - 288 \cdot 2310, \]

so that:

\[ c_2^{-1} \equiv 1153 \pmod{2310}. \]

Then he computes:

\[ m' \equiv c_2^{-1}(m - x_A \cdot c_1) \equiv 1153(809 - 175 \cdot 1470) \equiv 1217 \pmod{2310}. \]

So Bob knows that Alice wants to meet at 12:17 to escape.
3.1.2 Key Escrow

In the cryptosystems discussed before, the keys for encryption and decryption were only known to the sender and/or receiver of the message. A problem arises if the decryption key is lost: the data encrypted with the corresponding encryption key becomes unusable. A solution to this is that a third party stores information on the decryption key, so that the data can be recovered if necessary. This is called key escrow.

**Definition 3.2.** A key escrow encryption system is a cryptosystem in which information on the decryption key is held by a third party, such that authorized persons can recover the key and use it to recover the plaintext.

In the first place, the users of the system are authorized to recover their own keys. However, also other persons might be authorized, such as the officers of an organization or the government [7].

A key escrow encryption system consists of three preeminent parts [7]:

- **User Security Component.** In this part the encryption and decryption of data takes place and additional information is provided that can be used to recover the key(s). A possibility to do this is the attachment of a so-called data recovery field (DRF) to the encrypted data.

- **Key Escrow Component.** In this part, which is operated by key escrow agents (KEA), the data recovery keys (DRK) are stored. It is possible that only one agent has a data recovery key, but it is also possible that the data recovery key is split and distributed over multiple agents. This means that all agents must participate in order to recover the data.

- **Data Recovery Component.** Finally, the DRF and the data recovery keys can be used to recover the keys and data.

In figure 3.2 a key escrow encryption system with the three parts is shown [7].

![Diagram of Key Escrow Cryptosystem](image)

Figure 3.2: Key escrow cryptosystem.

Note that if the additional third party can be trusted completely, this is a secure system. However, the holder of the escrow key can decrypt all messages and also other parties might be authorized to recover the key. Therefore, one could wonder whether key-escrow can really be trusted.

**Key Escrow in RSA**

In this section, it is shown how key escrow can be implemented in RSA with multiple key escrow agents [28]. To recover the data, all key escrow agents need to participate. The advantage of this system is that the security of the data recovery does not depend on only one agent. Besides, none of the agents has direct access to the private key.
Assume that Alice uses RSA with key escrow. The main idea of this approach is to split the private key $d$ into integers $d_1, d_2, \ldots, d_k$ such that:

$$d \equiv \sum_{i=1}^{k} d_i \pmod{\varphi(n)},$$

with $n$ the corresponding modulus. Each key escrow agent $i$ (KEAi) has its own public and private key. Alice assigns each $d_i$ to one of the agents and uses their public keys to encrypt their part of the private key. So Alice encrypts the private key share $d_i$ with the public key of KEAi and hands it over to KEAi. Each agent can decrypt his share of the private key and together all agents can recover the private key.

Note that the actual system is much more complicated than described here. There are more parties involved in this process, such as a registration authority and an investigation agency. This simplified example is only provided to give the reader an idea of how key escrow could be implemented in RSA. The complete information can be found in [28].

Clipper Chip

In 1993, the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) launched a project called the Capstone project. This project provided cryptographic standards that would be used by the government. Besides, the standards were also published for public use. For data encryption, the algorithm Skipjack was used, which was implemented in the Clipper chip [27].

The US government stated that the idea behind this project was to make strong encryption available for all Americans, without putting the security of the country and its citizens at risk. Obviously, the government recognized the huge advantages of strong cryptography. However, they argued that they also feared the featuring risks. For example, not only harmless citizens would use the strong encryption products, but also criminals would have access to these products. Using this argument, the government decided to design a technology that would provide strong encryption, but that would also provide them the possibility to intercept communication if necessary (and with lawful authorization). For this purpose, they designed the Escrowed Encryption Standard (EES) and implemented key escrow in the used Clipper chip. The mechanism that was used for key escrow was named Law Enforcement Access Field (LEAF). This LEAF was used to store a copy of the session key [6].

In 1994, Matt Blaze published an article in which he explained all the things that were wrong with the Clipper chip [3]. He mentioned that: “By far the most controversial aspect of the EES system, however, is key escrow” [3, p.59]. The LEAF in the mechanism was used by the government to gain access to data that had been encrypted using the Clipper chip. In this field, a copy of the session key was stored. The session key was encrypted using Skipjack and a so-called unit key and copies of the unit keys were held in escrow by two federal agencies. Under certain conditions and with lawful authorization, the government could recover the unit keys, access the LEAF, decrypt the session key and decrypt data. Blaze showed that it was possible to use the Clipper chip in such a way that the storage of the session key in the LEAF could be circumvented. This means that a user can prevent that the government can recover the session key and decrypt and read data.
3.2. A HISTORY OF KLEPTOGRAPHY

The LEAF is constructed as follows (see figure 3.3):

- Each time the Clipper chip is used to send an encrypted message, a new session key is generated, denoted by $k_s$. The size of this key is 80 bits. The session key is shared between the sender and receiver of the message.
- The session key is encrypted with Skipjack using the unit key ($k_u$). This key is unique for each device and assigned during manufacture. The output is still 80 bits.
- Next, the program generates a 16 bits checksum, with the session key and so-called initialization vectors ($IV$) with some extra information as input.
- The unit identifier (unique for each device), the encrypted session key and the checksum are combined and encrypted with Skipjack and a family key ($k_f$) to obtain the LEAF. The family key is a fixed key shared between all EES devices.

The session key can be recovered by the government as follows. First, the LEAF and the data itself must be obtained using wiretapping technology. The LEAF is then decrypted using the family key to get the unit identifier, the encrypted session key and the checksum. With lawful authorization, the escrow agencies will each provide their share of the unit key that belongs to the unit identifier. By combining these two shares the original unit key can be reconstructed. To recover the session key, the unit key is used to decrypt the encrypted session key. Finally, the session key can be used to decrypt the data.

This approach shows that it must be possible to obtain the LEAF and the data via wiretapping. To enable this, the program is forced to send the LEAF and the data on the same channel. If a message is sent from the sender to the receiver, it will only be decrypted if the valid LEAF is sent with the data. This means that as long as the LEAF does not pass the integrity test, the data will not be decrypted.

Blaze discovered that the integrity of the LEAF depends entirely on the checksum of 16 bits. The receiving EES device does not know the unit ID or unit key of the other device, so it can only verify the LEAF by the checksum (and thus the session key and initialization vectors). The LEAF is 128 bits, of which 16 bits are the checksum. Since the integrity only depends on the checksum, it does not matter what the other 112 bits are. This implies that if a random 128-bit string is generated, the probability that it is accepted as a valid LEAF is equal to $1/2^{16}$.

Using this, Blaze suggested the following attack. The idea was to send a LEAF that would be accepted by the receiving device, but that could not be used to recover the session key. In important detail of the attack is that the attacker can generate a random 128-bit string and check with his own EES device whether this string is a valid LEAF. Since the probability that such a string is valid is equal to $1/2^{16}$, the expected average number of trials equals $2^{16}$. As soon as a valid LEAF is found, the attacker can send the message and the valid-looking LEAF to the receiver. If the LEAF is intercepted, it can be decrypted with the family key. The checksum then will turn out to be valid, but the unit identifier and encrypted session will turn out to be replaced by random bits. More details on this attack and other possible attacks can be found in [3].

With his findings, Blaze definitely encouraged the discussion on the Clipper chip. Eventually, in 1996, the government decided to reject the project.

3.2 A History of Kleptography

In the 1990s, Young and Yung started to focus on the risks of black-box cryptography. This kind of cryptography was often used by, for example, the US government. As mentioned before, in 1993 the Capstone project was launched by the NSA and NIST. The introduction of the new standards went not very smoothly. In March 1994, for example, The Wall Street Journal reported that: “Critics fear that the government will use the NSA technology, designed in secret, to spy on Americans” [5, p. B1]. The potential users did not trust the standards and eventually the government had to terminate the project.

According to Young and Yung, the suspicions of the critics were completely legitimate. The proposed standards would be used as a black-box and users would not know whether the used algorithm would be safe or whether the algorithm would leak private key information to the government. Furthermore, the users did not know if they would be at risk if, at some point, someone would be able to reverse-engineer the black-box. To prove that such a black-box mechanism is indeed not secure, Young and Yung presented the SETUP mechanism [36].

Since 1996 Young and Yung have written many articles on kleptography. A few examples are:
Young and Yung introduced the SETUP mechanism for the first time. With this mechanism, Young and Yung show that black-box cryptography offered by powerful entities cannot be trusted. SETUP attacks in RSA, ElGamal, DSA (digital signature algorithm) and Kerberos are described. All these mechanisms make use of a so-called subliminal channel (see section 3.1.1). The article is concluded with some recommendations to prevent or reduce the risks of the SETUP attack.

In 1997, Young and Yung redefine the SETUP attack by distinguishing between weak, regular and strong SETUP and the notion of leakage bandwidth is introduced. In addition, a strong attack based on the discrete logarithm problem is given with an application for the Diffie-Hellman key exchange protocol. They also provide an improved version of the SETUP attack in RSA, using the probabilistic bias removal method.

Previously, Young and Yung presented a strong attack based on the discrete logarithm problem. In this article, they continue with this attack and they show how the attack on the Diffie-Hellman key exchange can be embedded in the following systems: ElGamal, DSA, the Schnorr signature algorithm and the Menezes-Vanstone PKCS. Note that the latter is an elliptic curve cryptosystem. Furthermore, they introduce the term kleptogram to denote an encryption of a hidden value that is incorporated in public information.

Several articles on kleptographic attacks had been written and in this article Young and Yung answer some remaining open questions. They design new attack techniques, in which the attacker only needs one piece of information to be able to exfiltrate private information, using a so-called Newton channel. Furthermore, they show how this attack can be implemented in the ElGamal signature scheme and the show how another attack can be implemented in DSA.

In this article, Young and Yung adapt the notion of a strong SETUP attack to two different games, involving a designer, an eavesdropper and an agent. They use this model to revisit the SETUP attack on RSA.

In several articles, SETUP attacks on RSA have been discussed. In this article, the use of elliptic curve cryptography for use in kleptography is introduced, which provides many advantages. An application is shown using an attack on RSA.

In order to achieve certain goals within kleptographic attacks, Young and Yung introduce the covert key exchange. Using this new introduced primitive, previously designed SETUP attacks can be improved. Another implementation of SETUP in RSA is shown, again using elliptic curve cryptography.

3.3 SETUP

Young and Yung designed the SETUP (Secretly Embedded Trapdoor with Universal Protection) mechanism to be able to perform a kleptographic attack. The most important property of this mechanism is that the attacker can receive the user's private key information in such a way that the user cannot notice the leakage of information and that the attacker cannot be caught. The mechanism does not leak private information directly to the attacker, but incorporates the private key in public parameters or public output of the system. This means that the attacker can use the public parameters/output to recover the private key. The mechanism is constructed such that only the attacker, who sets up the system, can have access to the key and other attackers cannot.

In this section, the formal definition of (regular) SETUP is given, followed by the definitions of weak
3.3. SETUP

and strong SETUP. Finally, the notion of leakage bandwidth is introduced to denote how much information the attacker needs to be able to exfiltrate the user’s private key. The more practical side of the SETUP attack will be covered in the next three sections, where it is shown how SETUP can be implemented in RSA, ElGamal key generation and signature scheme and Diffie-Hellman key exchange.

3.3.1 Secretly Embedded Trapdoor with Universal Protection

The first SETUP mechanism, designed in 1996, makes use of a subliminal channel. As encryption algorithm, a PKCS function is used, denoted by the letter E. PKCS stands for public-key cryptography standards and represents a group of standards that can be used for asymmetric cryptography [26]. This PKCS function E is a trapdoor one-way function (see definition 2.39). This trapdoor one-way function is embedded secretly by an attacker within the cryptosystem, which gives a secretly embedded trapdoor. Finally, the information that is extracted secretly from the cryptosystem is universally protected, since the leaked information can only be decrypted by the attacker and not by other parties. This gives then: Secretly Embedded Trapdoor with Universal Protection [36].

3.3.2 Definition of SETUP

The first formal definition of a SETUP mechanism was given by Young and Yung in 1996 [36]:

Definition 3.3. Let C be a publicly known cryptosystem. A SETUP mechanism is an algorithmic modification made to C to get C’ such that:
1. The input of C’ agrees with the public specification of the input of C.
2. C’ computes using the attacker’s public encryption function E (and possibly other functions as well), contained within C’.
3. The attacker’s private decryption function D is not contained within C’ and is known only by the attacker.
4. The output C’ agrees with the public specifications of the output of C. At the same time, it contains published bits (of the user’s secret key) which are easily derivable by the attacker but otherwise hidden. (The output can be generated during key-generation or during system operation).
5. Furthermore, the outputs of C and C’ are polynomially indistinguishable to everyone (including those who have access to the code of C’) except the attacker.

The first condition ensures that the user should be able to use system C’ the same way as system C. If the input should be different, the user will probably get suspicious. Next, the second condition makes sure that the computation time of C’ does not differ significantly from the computation time of C. If, for example, system C’ works twice as slow as system C, the user will probably notice it. Since only the attacker knows the decryption function D, no other parties except the attacker, can decrypt information, according to the third condition. The fourth condition tells that the output of C’ should look like the output of C. If the user expects an output of, for example, 512 bits, the output of C’ should also have a size of 512 bits. This output contains hidden information on the user’s key that can be exfiltrated by the attacker.

Finally, the fifth condition sharpens the previous condition: the output C’ should not only agree with the public specifications of the output of C, but the output of both mechanisms should be polynomially indistinguishable to everyone except the attacker. This means that, for example, the user can not distinguish between the output of C and the output of C’ within polynomial time. It is important to note that the remark between brackets (“including those who have access to the code of C’”) does not need to be a requirement. In [36], Young and Yung improved their definition of SETUP and introduce another condition to distinguish between users of the system and persons (including the user) that have access to the code (definition 3.5).

Young and Yung provided a definition for a cryptosystem that is modified to a SETUP mechanism:

Definition 3.4. Let C be a publicly known cryptosystem. A contaminated cryptosystem C’ is a modified version of C that contains a SETUP mechanism.

In 1997, Young and Yung adjusted their first definition of a SETUP mechanism [37]. This definition defines regular SETUP:
CHAPTER 3. KLEPTOGRAPHY

Definition 3.5. Assume that $C$ is a black-box cryptosystem with a publicly known specification. A (regular) SETUP mechanism is an algorithmic modification made to $C$ to get $C'$ such that:

1. The input of $C'$ agrees with the public specifications of the input of $C$.
2. $C'$ computes efficiently using the attacker's public encryption function $E$ (and possibly other functions as well), contained within $C'$.
3. The attacker's private decryption function $D$ is not contained within $C'$ and is known only by the attacker.
4. The output of $C'$ agrees with the public specifications of the output of $C$. At the same time, it contains published bits (of the user's secret key) which are easily derivable by the attacker (the output can be generated during key-generation of during system operation like message sending).
5. Furthermore, the outputs of $C$ and $C'$ are polynomially indistinguishable to everyone except the attacker.
6. After the discovery of the specifics of the SETUP algorithm and after discovering its presence in the implementation (e.g., reverse-engineering of hardware tamper-proof device), users (except the attacker) cannot determine past (or future) keys.

As can be seen here, the first four conditions are still the same. However, the fifth condition was slightly changed and a sixth condition was added. In the fifth condition, the remark on persons that have access to the code was deleted. This remark is now part of the sixth condition. The sixth condition states that if the user knows that the system is contaminated and is able to reverse-engineer it, it is still impossible to (re)compute past or future (private) keys of the user using the information retrieved from the contaminated system.

As mentioned before, the leaked information can only be decrypted by the attacker, since only the attacker knows how to do this (with the decryption function $D$). Since this information is not incorporated in the system, the user still cannot (re)compute keys, despite the fact that the user knows the specifics of the system. This condition is actually not a requirement for regular SETUP. In regular SETUP it is assumed that the used system is a black-box cryptosystem: the user of the contaminated cryptosystem does not have access to the internals of the device and therefore cannot take a look at the exact implementations. However, condition 6 is more important for the definition of strong SETUP (section 3.3.4, p. 43). 

Note that the SETUP attack is a so-called Trojan horse. One speaks of a Trojan horse if a computer program is contaminated with another program, that allows the attacker to access the victim's computer. The Trojan horse only works if the user executes the contaminated program. The attacker could use such an attack, for example, to subtract, modify or delete data, or even to crash the computer. A Trojan horse differs from a virus by the fact that it does not replicate itself. In the case of the SETUP attack, the Trojan horse acts as a backdoor: it passes on information on the key secretly to the attacker. The name is derived from the mythical Trojan horse that provided the Greeks a backdoor into the city of Troy [19].

3.3.3 Weak SETUP

Young and Yung defined weak SETUP as follows [37]:

Definition 3.6. A weak SETUP is a regular SETUP except that the outputs of $C$ and $C'$ are polynomially indistinguishable to everyone except the attacker and the owner of the device who is in control (knowledge) of his or her own private key (i.e., condition 5 of definition 3.5 is changed).

This means that the owner/user of the device would also, just as the attacker, be able to distinguish between output from $C$ and output from $C'$ in polynomial time. Such a weak SETUP mechanism might seem to be cryptographically insecure. However, in practice this is not immediately the case. First of all, the user must suspect that the system might be contaminated. As long as the user is not familiar with such an attack, he or she will not suspect it. Secondly, if the user thinks that the system is contaminated, he or she must know how it can be detected. Taking these two steps into account, a weak SETUP mechanism can be used (with some caution). In fact, there is actually a useful application of weak SETUP. Weak SETUP can be used if Alice and Bob want to communicate subliminally with each other. They can create a contaminated cryptosystem themselves and use that to organize a subliminal channel. This application is explained in section 3.5.3 on SETUP in the ElGamal signature scheme (p. 63).
3.3.4 Strong SETUP

In regular SETUP, it is assumed that the system is a black-box cryptosystem (except for condition 6). According to Young and Yung, this is actually a necessary assumption for polynomial indistinguishability: as long as the user does not know the algorithm, the outputs of $C$ and $C'$ are polynomially indistinguishable. For the definition of strong SETUP, it is assumed that sometimes the contaminated device uses SETUP and sometimes it uses the normal algorithm. Then [37]:

**Definition 3.7.** A strong SETUP is a regular SETUP, but in addition we assume that the users are able to hold and fully reverse-engineer the device after its past usage and before its future usage. They are able to analyze the actual implementation of $C'$ and deploy the device. However, the users still cannot steal previously generated/future generated keys, and if the SETUP is not always applied to future keys, then SETUP-free keys and SETUP keys remain polynomially indistinguishable.

This means that the user can completely reverse-engineer the device and investigate the internals before and after each time the device is used, but does not know Eve’s keys and random choices. The user knows the functions that are used in the system and knows, for example, which fixed values are used. Using this information, it is still impossible to (re)compute keys. Besides, since the SETUP mechanism is not used each time, the user does not know which keys contain hidden secret information and which keys do not contain such information. Note that it is assumed that the device does not contain any information on when the SETUP mechanism is used and when not. This version of SETUP is more secure than the regular or weak SETUP, since it does not require anymore that the system is a black-box cryptosystem.

3.3.5 Weak, Regular or Strong SETUP?

To summarize the information on weak, regular and strong SETUP, here are the most important properties listed:

- If the user of the device, and only the user, is able to distinguish between outputs of $C$ and outputs of $C'$ within polynomial time with the knowledge of the private key, the attack constitutes a weak SETUP attack.
- If the user of the device cannot distinguish between outputs of $C$ and outputs of $C'$ within polynomial time, even when knowing the private key, the attack constitutes a regular SETUP attack.
- The attack constitutes a strong SETUP attack if the user, having complete knowledge of the system and access to the user’s current keys, is unable to (re)compute past or future keys and cannot distinguish between outputs of $C$ and outputs of $C'$.

Obviously, this is a measure for the security of the system. In each SETUP attack that is described in coming sections, it is denoted whether the attack is weak, regular or strong.

3.3.6 Leakage Bandwidth

As mentioned before, the attacker uses public output of a contaminated system to exfiltrate the user’s private key. In the next three sections, implementations of the SETUP mechanism are explained. In most of these systems, the attacker only needs one piece of public information, for example one public key, to exfiltrate one private key. However, in another system, the attacker needs two consecutive public signatures to exfiltrate one private key. To denote how much public information the attacker needs to be able to exfiltrate private key information, Young and Yung introduced the notion of **leakage bandwidth** [37]:

**Definition 3.8.** A $(m,n)$-leakage scheme is a SETUP mechanism that leaks $m$ keys/secret messages over $n$ keys/messages that are output by the cryptographic device ($m \leq n$).

This means that if the attacker only needs one public key to exfiltrate one private key, this system is a $(1,1)$-leakage scheme. The system that needs two consecutive signatures to exfiltrate one private key is then a $(1,2)$-leakage scheme.
3.4 SETUP in RSA

As explained in section 2.4.4 (p. 27), RSA makes use of a public key \((e, n)\) and a private key \((d, n)\). For simplicity’s sake, the subscripts are omitted from now on. In this case, \((e, n)\) and \((d, n)\) are the keys generated by the receiver of a message. Assume Alice and Bob are using RSA to communicate securely (Bob wants to send a message to Alice) and Eve is the attacker that wants to contaminate the RSA system.

3.4.1 SETUP in RSA Key Generation I

A relatively simple SETUP mechanism can be generated as follows. Let \((D, N)\) and \((E, N)\) be Eve’s private and public RSA keys respectively. The aim of the attack is to exfiltrate Bob’s private key \(d\). The mechanism is based on the idea that \(e\) need not to be chosen randomly, but can be made dependant on the attacker’s public key.

Algorithm

In RSA, the encryption exponent \(e\) is generated randomly such that \(1 < e < \phi(n)\) and \(\gcd(e, \phi(n)) = 1\). Assume that \(e\) is generated from \(\{0, 1\}^k\), so \(e\) is an integer consisting of \(k\) bits (binary number). Also primes \(p\) and \(q\) are generated randomly from \(\{0, 1\}^k\). Instead of the random generation of the encryption exponent, \(e\) is set to be \(p^E \pmod{N}\). If \(\gcd(e, \phi(n)) \neq 1\), the system generates a new value for \(p\). Next, the decryption exponent \(d\) is computed as usual, so \(d \equiv e^{-1} \pmod{\phi(n)}\). See figure 3.4 for a comparison of normal RSA and RSA with SETUP. In appendix A.1 (p. 87), a description of the algorithm in pseudocode can be found.

![RSA key pair generation with SETUP](image)

**Modulus Factoring by Attacker**

Eve can factor the modulus \(n\) by computing:

\[ p \equiv e^D \pmod{N}. \]

In section 2.4.4 it is shown already that this decryption works. Note that it works if \(\gcd(p, N) = 1\) and if \(\gcd(p, N) > 1\). Eve knows \(n\), so she can also compute prime \(q\) such that \(n = p \cdot q\). This makes it very easy to compute \(\phi(n)\), namely \(\phi(n) = (p - 1)(q - 1)\). As explained in section 2.4.4, Eve can compute the private decryption exponent \(d\) if she knows \(\phi(n)\). This means that now she can easily compute \(d \equiv e^{-1} \pmod{\phi(n)}\) and decrypt all the messages that Alice sends to Bob. Note that this attack has a \((1, 1)\)-leakage scheme, since Eve only needs one encryption exponent \(e\) to exfiltrate prime \(p\).

In example 3.2, a worked-out example of SETUP in RSA can be found. Several aspects of the attack are shown here. First of all, the attacker’s keys are generated, followed by the generation of Alice’s key pairs by the program. Next, it is shown how Bob encrypts a message using Alice’s parameters and how
Alice decrypts Bob’s message. Finally, Eve uses Alice’s encryption exponent $e$ to exfiltrate prime $p$, which she can use to compute Alice’s private decryption exponent $d$.

**Example 3.2: SETUP in RSA key generation I**

The steps are executed as follows:

- **Key pair generation by Eve.** Eve generates the two key pairs as follows:
  - $P = 683$ and $Q = 863$;
  - $N = P \cdot Q = 683 \cdot 863 = 589429$;
  - $\varphi(N) = (P - 1)(Q - 1) = 682 \cdot 862 = 578884$;
  - $E = 39451$;
  - $D \equiv E^{-1} \equiv 319163 \pmod{578884}$;
  - public key $(E, N) = (39451, 589429)$;
  - private key $(D, N) = (319163, 589429)$.

- **Key pair generation by Alice.** The contaminated program generates the two key pairs as follows:
  - $p = 367$ and $q = 983$;
  - $n = pq = 367 \cdot 983 = 360761$;
  - $\varphi(n) = (p - 1)(q - 1) = 366 \cdot 982 = 359412$;
  - $e \equiv pE \equiv 367^{39451} \equiv 319379 \pmod{589429}$;
  - $d \equiv e^{-1} \equiv 88127 \pmod{359412}$;
  - public key $(e, n) = (319379, 360761)$;
  - private key $(d, n) = (88127, 360761)$.

- **Encryption by Bob.** Bob wants to send the message “CAT”. He encodes this as $m = 030119$ and encrypts, using Alice’s public key pair $(e, n)$:
  
  $$c = m^e = 30119^{319379} \equiv 89390 \pmod{360761}.$$  

- **Decryption by Alice.** Alice receives $c = 89390$ from Bob and decodes this using her private key pair $(d, n)$:
  
  $$m = c^d = 89390^{88127} \equiv 30119 \pmod{360761}.$$  

She decodes this as “CAT”.

- **Attack by Eve.** Eve computes prime $p$ as follows, using Alice’s public key $(e, n)$ and her own private key pair $(D, N)$:
  - First, Eve computes prime $p$:
    
    $$p \equiv e^D \equiv 319379^{319163} \equiv 367 \pmod{589429}.$$  

  - Next, she uses $n$ and $p$ to compute the other prime factor $q$:
    
    $$q = \frac{n}{p} = \frac{360761}{367} = 983.$$  

  - With $p$ and $q$, Eve can compute:
    
    $$\varphi(n) = (p - 1)(q - 1) = 366 \cdot 982 = 359412.$$  

  - Finally, she computes:
    
    $$d \equiv e^{-1} \equiv 88127 \pmod{359412},$$  

  and Eve can decrypt the message the same way Alice did.

**Security**

In practice, this SETUP mechanism is not effective. Young and Yung point out that programs such as PGP use relatively small exponents (on the order of 5 bits), since a small encryption exponent simplifies the encryption. However, the SETUP mechanism computes $e \equiv p^E \pmod{N}$, which generates a relatively
large encryption exponent. In example 3.2, \( e = 319379 \), which is large compared to the public modulus \( n \) (360761). Bob expects a small encryption exponent and he will definitely notice it if \( e \) is suddenly greater than usual. Therefore, Young and Yung conclude that this SETUP mechanism will probably be noticed by the users. However, if the attacker manages to discourage the use of small exponents, this attack could definitely be used.

**Summary**

This first attack describes a SETUP mechanism for RSA key pair generation. The contaminated system incorporates prime \( p \) in the encryption exponent \( e \), which is part of the public key. Eve can factor the user’s modulus \( n \) by decrypting \( e \) with her own secret decryption exponent \( D \). If she manages to factor the user’s modulus, she can easily compute \( \varphi(n) \) and the user’s private key \( d \). Finally, Eve can use this private key to decrypt messages that Alice receives. Since Eve only needs one encryption exponent \( e \) to recover the private key, this attack has a \((1,1)\)-leakage scheme. However, since it is not hard to distinguish between output of the contaminated system and the normal system, this is not even a weak SETUP attack. If the encryption exponent \( e \) need not to be small, then this would be a strong SETUP attack, since Alice cannot (re)compute past of future keys if she has access to the system. This is because \( p \) and \( q \) are generated randomly and cannot be predicted.

### 3.4.2 SETUP in RSA Key Generation II (PAP)

In the SETUP mechanism described before, Eve’s public key exponent \( E \) is incorporated in the system’s public key exponent \( e \). Since this requires \( e \) to be large, Young and Yung suggest to hide \( E \) in \( n \) instead. This SETUP mechanism is described by the program *Pretty Awful Privacy (PAP)*. PAP hides the secret information in the upper order bits of the public modulus \( n \). The upper order bits are the most significant or the leftmost bits of a binary number.

**Algorithm**

Let \((E, N)\) and \((D, N)\) be the attacker’s public and private key, with \( N \) the \( k \)-bit attacker’s modulus. The program PAP uses a keyed randomization function twice (see section 2.4.1). PAP executes the following steps, with predefined fixed values for \( K \), \( B_1 \) and \( B_2 \):

- First, PAP generates a random \( k \)-bit prime \( p \).
- Then, it uses the keyed randomization function \( R_1 \) to randomize \( p \). The function maps a \( k \)-bit prime onto a \( k \)-bit binary integer. PAP uses \( K + i \) as key with \( i = 0 \) as initial value. The outcome is denoted by \( p' \), so:

\[
p' = R_1(p).
\]

- The following step has two possibilities:
  - If \( p' \geq N \), then \( i = i + 1 \) and the previous step is executed again. This process continues at most \( B_1 \) times. This means that as soon as \( i = B_1 \), the program restarts and generates a new prime \( p \). Note that if the program restarts, then \( i = 0 \) again.
  - If \( p' < N \), then \( p' \) is encrypted as follows:

\[
p'' \equiv (p')^E \pmod N.
\]

Note that \( p'' \) is a \( k \)-bit integer.

- Then, PAP uses the keyed randomization function \( R_2 \) to randomize \( p'' \) with \( K + j \) as key (initially \( j = 0 \)). The result is a \( k \)-bit integer \( p''' \):

\[
p''' = R_2(p'').
\]

- In order to get a \( 2k \)-bit key, \( p''' \) is concatenated with a randomly chosen \( k \)-bit string, generated from \( \{0, 1\}^k \):

\[
X = p'''|\{0, 1\}^k.
\]

- PAP computes \( X/p \) to find the quotient \( q \). Then there are two possibilities:
- If \( q \) is not a prime, then \( j = j + 1 \) and \( p'' \) is recomputed. This process continues at most \( B_2 \) times, so as soon as \( j = B_2 \), the program restarts and generates a new prime \( p \) (with \( i = 0 \) and \( j = 0 \)).
- If \( q \) is a prime, then \( p, q \) are the prime factors and the steps of RSA are executed as follows:
  * \( n = p \cdot q \);
  * \( \varphi(n) = (p - 1)(q - 1) \);
  * \( e = 17 \);
  * check whether \( \gcd(e, \varphi(n)) = 1 \); if not, \( e = e + 2 \) until \( \gcd(e, \varphi(n)) = 1 \);
  * \( d \equiv e^{-1} \pmod{\varphi(n)} \).

- Finally, the program outputs the public key \((e, n)\) and the private key \((d, n)\).

Note that \( q \) is found by computing \( X/p \). It is possible that \( X \) is divisible by \( p \), then \( n = X \). However, it is also possible that there is remainder left after division. In this case \( n \neq X \). In appendix A.2 (p. 88), a description of the algorithm in pseudocode can be found.

Note that the attacker’s modulus \( N \) is a \( k \)-bit integer, whereas the user’s modulus \( n \) is a \( 2k \)-bit integer. This implies that the RSA keys of the user are significantly more secure than the keys of the attacker. This means that if it is possible to break an RSA system using a \( k \)-bit modulus, the attacker’s private key can be found and this could be used to exfiltrate private keys of other contaminated systems. However, since the user’s modulus is twice as large, these RSA parameters will still be secure.

Usually, RSA generates two random primes \( p \) and \( q \) and computes the modulus \( n = p \cdot q \). PAP generates only one random prime \( p \) and uses this prime to compute the other prime \( q \) and the modulus \( n \). Since the program outputs \( p, q \) and \( n \), it is of the utmost importance that \( p \) and \( q \) look like they are completely random to the user. The first step to achieve this “look-a-like randomness”, is the use of the keyed randomization function \( R_1 \). To be able to perform the encryption with the attacker’s encryption exponent \( E \), \( p \) should be smaller than the attacker’s modulus \( N \), otherwise the attacker cannot recover \( p \). If PAP would generate a random \( p \) that is always smaller than \( N \), this step would not be necessary. However, this reduces the randomness of the output of the program. Thus, \( R_1 \) is used to generate a random integer \( p' \) < \( N \).

After the encryption of \( p' \) into \( p'' \), the keyed randomization function \( R_2 \) is used to generate the random integer \( p''' \). The main reason for the use of \( R_2 \), is to reduce the computational complexity. If \( q \) turns out not to be prime, \( X \) is changed so that \( q \) will be a prime. It is most logical to change \( p''' \) (the first \( k \) bits of \( X \)) and not the \( k \)-bit random string. This is because both \( p \) and the random string have a length of \( k \) bits. This implies that if one changes the random string instead of \( p''' \), it is possible that \( q \) does not change at all. However, if \( p''' \) is modified, then \( q \) will certainly change. So the probability that the program finds a prime \( q \) is greater if \( p''' \) is changed than if the random string is changed. So \( R_2 \) generates another random \( p''' \) to change \( X \), such that, hopefully, \( q \) is prime. It would also be possible to remove \( R_2 \) from the algorithm, but then \( p' \) should be recomputed and encrypted again to generate a different \( X \), which increases the computational complexity. In other words, the use of \( R_2 \) avoids “the overhead of excessive public key encryptions” [36, p. 93].

### Modulus Factoring by Attacker

To factor the user’s modulus, Eve performs the following steps:

- Let \( U \) represent the \( k \) upper order bits of the public modulus \( n \).
- Eve decrypts \( U \) using \( K + j \) and the inverse of the keyed randomization function \( R_2 \), so
  \[
  p'' = R_2^{-1}(U).
  \]

Since Eve does not know how often the program recomputed \( p'' = R_2(p'') \), she needs to decrypt \( U \) with all possible keys, so \( 0 \leq j \leq B_2 - 1 \). This gives \( B_2 \) different possible values of \( p' \). However, Eve does know that \( p' < N \), since \( p' \) was encrypted using \( E \) and \( N \). This means that she could exclude all possible values of \( p'' \) that are larger than \( N \).
- Then, Eve uses her secret decryption exponent \( D \) to decrypt the possible values of \( p'' \):
  \[
  p' \equiv (p'')^D \pmod{N},
  \]
- The next step is the decryption of the \( B_2 \) possible values of \( p' \) using the inverse of the keyed
randomization function $R_1$ and the key $K + i$:

$$p = R_1^{-1}(p').$$

Again, since Eve does not know how often the program recomputed $p' = R_1(p)$, she needs to use $B_1$ different possible keys for each possible value of $p'$.

For the last step, there are three possibilities:

- If one of the possible values of $p$ divides $n$, so $p|n$, then Eve succeeded to factor the public modulus.
- If none of the possible values of $p$ divides $n$, then all these steps are repeated for $U + 1$.
- If $U + 1$ does not succeed either, then Eve must conclude that the contaminated program was not used to generate the keys.

If $U$ does not give a factorization of $n$, Eve should try $U + 1$. The reason for this is the following. At one point, PAP divides $X$ by $p$ to compute the quotient $q$. It is possible that:

$$X = p \cdot q,$$

such that $X = n$ and $U = p''$. However, it is also possible that:

$$X = p \cdot q + r,$$

with $r$ the remainder after division. Then $X \neq p \cdot q$ and possibly $U \neq p'''$ (but not necessarily; see example 3.3). Actually:

$$n = p'''|\{0, 1\}^k + r.$$

Since $p$ is $k$ bits long, the remainder $r$ is at most $k$ bits. This implies that if $r$ is added to $p'''|\{0, 1\}$, then it can only modify the $k$ upper order bits ($p'''$) by at most 1. Therefore, if $U$ does not give a proper factorization, $U + 1$ will give one if PAP is used. Finally, note that Eve only needs one public modulus to exfiltrate prime $p$. This means that this attack has a $(1, 1)$-leakage scheme.

### Example 3.3: SETUP in RSA key generation II - $U$ and $U + 1$

Assume Bob used PAP to generate the key pairs. For decryption, there are three possibilities:

- $X = p \cdot q$ and $U = p'''$;
- $X = p \cdot q + r$, but $U = p'''$ still holds;
- $X = p \cdot q + r$ and $U \neq p'''$ (and then $U + 1 = p'''$).

Of all three cases an example is given with a modulus of 6 bits. Note that the integers in (parentheses) are the decimal representations.

**• $X = p \cdot q$ and $U = p'''$:**

Assume $p = 101$ (5) and assume that PAP computed $X = 100011$ (35), with $p''' = 100$ (4). Then:

$$X = p \cdot q = 101 \cdot 111 \quad (5 \cdot 7).$$

This means that:

$$n = p \cdot q = 100011 \quad (35) = X,$$

such that $U = 100$ (4) and indeed $U = p'''$.

**• $X = p \cdot q + r$ and $U = p'''$:**

Assume $p = 101$ (5) and assume that PAP computed $X = 100111$ (39), with $p''' = 100$ (4). Then:

$$X = p \cdot q + r = 101 \cdot 111 + 100 \quad (5 \cdot 7 + 4).$$

This means that:

$$n = p \cdot q = 101 \cdot 111 = 100011 \quad (35) \neq X,$$

such that $U = 100$ (4) and indeed $U = p'''$ still holds. This means that Eve will find the correct factorization with $U$, although $X \neq n$. 

48
3.4. SETUP IN RSA

- \( X = p \cdot q + r \) and \( U \neq p''' \):
  Assume \( p = 101 \) (5) and assume that PAP computed \( X = 111010 \) (58), with \( p''' = 111 \) (7). Then:

\[
X = p \cdot q + r = 0101 \cdot 1011 + 0011 \quad (5 \cdot 11 + 3).
\]

This means that

\[
n = p \cdot q = 101 \cdot 1011 = 110111 \quad (55) \neq X,
\]

such that \( U = 110 \) (4) and indeed \( U \neq p''' \). This means that Eve will not find a correct factorization with \( U \). However, if she uses \( U + 1 \), then \( U + 1 = 111 = p''' \) and this will give a correct factorization.

In example 3.4, a worked-out example of SETUP in RSA II (PAP) can be found. The attacker’s keys are generated, followed by the generation of Alice’s key using the contaminated cryptosystem. Next, Bob encrypts a message with Alice’s public parameters and Alice can decrypt this message. Furthermore, it is shown how Eve can factor Alice’s modulus \( n \). Note that this example uses decimal representation. However, sometimes also the binary representation is given, since the program actually needs binary representations.

Example 3.4: SETUP in RSA key generation II (PAP)

The steps are executed as follows:

- **Key pair generation by Eve.** Eve generates the two key pairs as follows:
  - \( P = 13 \) and \( Q = 11 \);
  - \( N = P \cdot Q = 13 \cdot 11 = 143 \);
  - \( \varphi(N) = (P - 1)(Q - 1) = 12 \cdot 10 = 120 \);
  - \( E = 97 \);
  - \( D \equiv E^{-1} \equiv 97^{-1} \equiv 73 \) (mod 120);
  - public key \((E, N) = (97, 143)\);
  - private key \((D, N) = (73, 143)\).

- **Key pair generation by Alice.** The contaminated program generates the two key pairs as follows:
  - First, the program generates prime \( p \) randomly:
    \[
p = 59.
    \]
  - Then, the program uses the first keyed randomization function \( R_1 \) to get \( p' \). Assume that:
    \[
p' = R_1(p) = R_1(59) = 102.
    \]
  - Since \( p' < N \), the program encrypts \( p' \) using Eve’s encryption exponent \( E \):
    \[
p'' \equiv (p')^E \equiv 102^{97} \equiv 141 \) (mod 143).
  - Next, the program uses the second keyed randomization function \( R_2 \) to get \( p''' \). Assume that:
    \[
p''' = R_2(p'') = R_2(141) = 113.
    \]
  - Note that 113 is equal to 01110001 in binary representation (in 8 bits). Using this binary representation, \( p''' \) is concatenated with a randomly chosen \( k \)-bit string:
    \[
    X \equiv 0111000100110001
    \]
  - Note that \( X \) is equal to 28977 in decimal representation.
  - The program computes \( X/p \) to find \( q \):
    \[
    q = \left\lfloor \frac{X}{p} \right\rfloor = \left\lfloor \frac{28977}{59} \right\rfloor = 491.
    \]
    Since \( q \) is prime, the program uses \( p = 59 \) and \( q = 491 \) to generate Alice’s key pair:
n = p \cdot q = 59 \cdot 491 = 28969;
\phi(n) = (p - 1)(q - 1) = 58 \cdot 490 = 28420;
e = 17;
d \equiv e^{-1} \equiv 17^{-1} \equiv 21733 \pmod{28420}.

\text{public key } (e, n) = (17, 28969);
\text{private key } (d, n) = (21733, 28969).

\text{Encryption by Bob.} Bob wants to send the message “HI”. He encodes this as } m = 0809 \text{ and encrypts, using Alice’s public key pair } (e, n):

\begin{align*}
c &= m^e \\ &= 809^{17} \\ &\equiv 16216 \pmod{28969}.
\end{align*}

\text{Decryption by Alice.} Alice receives } c = 16216 \text{ from Bob and decodes this using her private key } (d, n):

\begin{align*}
m &= c^d \\ &= 16216^{21733} \\ &\equiv 809 \pmod{28969}.
\end{align*}

Finally, she decodes this as “HI”.

\text{Attack by Eve.} Eve uses Alice’s public key } (e, n) \text{ to exfiltrate prime } p \text{ and therefore the private decryption exponent } d. \text{ She knows that } (e, n) = (17, 28969).

– The binary representation of } n \text{ equals } 0111000100101001. \text{ Eve takes the 8 upper order bits, so:}

\begin{align*}
U &= 01110010.
\end{align*}

The decimal representation of } U \text{ equals 113.

– Eve uses the inverse of the second keyed randomization function, } R_2, \text{ to get } p’:

\begin{align*}
p’ &= R_2^{-1}(U) \equiv R_2^{-1}(113) = 141.
\end{align*}

– Next, she decrypts } p’ \text{ with her private decryption exponent } D:

\begin{align*}
p &\equiv (p’)^D \equiv 141^{73} \equiv 102 \pmod{143}.
\end{align*}

– Then Eve uses the inverse of the first keyed randomization function, } R_1, \text{ to get } p:

\begin{align*}
p &= R_1^{-1}(p’) = R_1^{-1}(102) = 59.
\end{align*}

– Since } p \text{ divides } n, \text{ Eve knows that } p \text{ is a prime factor of } n. \text{ She computes:

\begin{align*}
q &= \frac{n}{p} = \frac{28969}{59} = 491.
\end{align*}

– Then:

\begin{align*}
\phi(n) = (p - 1)(q - 1) = 58 \cdot 490 = 28420.
\end{align*}

– Finally, she computes:

\begin{align*}
d \equiv e^{-1} \equiv 17^{-1} \equiv 73 \pmod{28240},
\end{align*}

and Eve can decrypt the message the same way Bob did.

\text{Bounds } B_1 \text{ and } B_2

Whether this method indeed generates primes } p \text{ and } q \text{ depends on the fixed values for } B_1 \text{ and } B_2. \text{ Young and Yung state briefly that it can be shown that if } B_1 \text{ and } B_2 \text{ are chosen properly, the probability that PAP finds a } p \text{ and } q \text{ is acceptable. They chose } B_1 = 16 \text{ and } B_2 = 512 \text{ and argue that it can be shown using the prime number theorem that these values are sufficient. However, they do not show how this can be done. Therefore, in this thesis it is shown, using the prime number theorem, that these values are indeed acceptable.}

Initially, a random prime } p \text{ is generated. The keyed randomization function } R_1 \text{ generates } p’. \text{ This
$k$-bit $p'$ should be smaller than the $k$-bit attacker’s modulus $N$. Note that $p', N \in \{0,1\}^k$, which implies that there are $2^k$ possibilities for both integers. Then:

$$P(p' < N) = P(p' \geq N) = \frac{1}{2}.$$ 

If the computation of $p'$ is executed $B_1$ times, then:

$$P(p' \geq N \text{ after } B_1 \text{ times}) = \left(\frac{1}{2}\right)^{B_1}.$$ 

This implies that:

$$P(p' < N \text{ after } \leq B_1 \text{ times}) = 1 - \left(\frac{1}{2}\right)^{B_1}.$$ 

With $B_1 = 16$, this means that:

$$P(p' < N \text{ after } \leq 16 \text{ times}) = 1 - \left(\frac{1}{2}\right)^{16} \approx 1.00,$$

which is indeed very acceptable. Of course, Eve could also choose her primes $P$ and $Q$ such that her modulus $N$ is very close to $2^k$. Then the probability that $p' \geq N$ decreases and the probability that $p' < N$ after at most $B_1$ times increases, which makes it even more likely that the program generates a sufficient integer $p'$.

For $B_2$, one needs the prime number theorem [1].

**Definition 3.9.** The prime-counting function, $\pi(x)$, represents the number of primes smaller than or equal to $x$.

**Theorem 3.10.** The prime-counting function $\pi(x)$ is asymptotic to $x / \log(x)$. This is denoted as follows:

$$\pi(x) \asymp \frac{x}{\log(x)}.$$ 

This means that:

$$\lim_{{x \to \infty}} \frac{\pi(x)}{x / \log(x)} = 1.$$ 

A proof of this theorem can be found in number theory books, such as [1] and [13].

According to this theorem, if someone generates a random integer $y$ smaller than or equal to $x$, then:

$$P(y = \text{prime}) = \frac{x / \log(x)}{x} = \frac{1}{\log(x)}.$$ 

The keyed randomization function $R_2$ generates the $k$-bit quantity $p''$, which is concatenated with a randomly chosen $k$-bit string from $\{0,1\}^k$ to generate the $2k$-bit quantity $X$. Next, $X/p$ is computed to find $q$. Note that since $p$ is a $k$-bit prime, $q$ will be a $k$-bit integer as well. Since $q$ is a $k$-bit integer, there are in total $2^k$ possibilities for $q$. Then, according to the prime number theorem (theorem 3.10), approximately $1 / \log(2^k)$ of these possibilities are indeed prime, such that:

$$P(q = \text{prime}) \approx \frac{1}{\log(2^k)},$$

and

$$P(q \neq \text{prime}) \approx 1 - \frac{1}{\log(2^k)}.$$ 

If necessary, $G$ generates a new value for $p''$ at most $B_2$ times. Then:

$$P(q \neq \text{prime after } B_2 \text{ times}) \approx \left(1 - \frac{1}{\log(2^k)}\right)^{B_2}.$$
Finally:

$$P(q = \text{prime after} \leq B_2 \text{ times}) \approx 1 - \left(1 - \frac{1}{\log(2^k)}\right)^{B_2}.$$

Young and Yung implemented PAP with a target key size of 512 bits. This means that the primes should have a length of 256 bits. They used $B_2 = 512$, which gives:

$$P(q = \text{prime after} \leq 512 \text{ times}) \approx 1 - \left(1 - \frac{1}{\log(2^{256})}\right)^{512} \approx 0.94,$$

which is also acceptable. This probability does rely on the target key size.

For $B_1$, the probability that $p'$ is smaller than $N$ does not depend on the target key size $k$. This means that $B_1 = 16$ will suffice for all possible key sizes. For $B_2$, the probability that the program generates a prime $q$ does depend on the key size $k$. The table in figure 3.5 shows the probabilities that the program generates a prime $q$ given a key size $k$ and a bound $B_2$. For example, if $k = 1024$, then $B_2$ should be equal to 2048 to obtain the same probability as for $k = 256$ and $B_2 = 512$ (proposed by Young and Yung). In general, it is advisable to use a bound at least twice as large as the target key size.

![Figure 3.5: PAP - Probabilities for key size $k$ and bound $B_2$, rounded to two decimal digits.](image)

**Security**

The last step is to show that PAP generates two primes $p$ and $q$ that both seem to be random to the user. The prime $p$ is chosen uniformly at random from $\{0, 1\}^k$, so this one is definitely random. Now assume that the range of the keyed randomization functions $R_1$ and $R_2$ are indistinguishable from a truly random choice, then $p'$, $p''$ and $p'''$ are random as well. This means that the $k$ upper order bits of $X$ are also random. Finally, since $p$ and $X$ are random, prime $q$ will be random as well in the set of $k$-bit primes. Then:

**Theorem 3.11.** PAP is a contaminated cryptosystem based on the RSA cryptosystem.

**Application: Auto-Escrowing-Keys in Hardware**

A possible application of the program PAP is to use it as a hardware key-escrow mechanism. If PAP is implemented in the user’s device, it can subtract private keys without the knowledge of the user. These private keys will be stored using two or more escrow agents. To enhance the security of PAP, each device will use another public key $E$ (the public key of the attacker). If necessary, the agents can combine their shared keys and reconstruct the user’s private keys. An advantage compared to “normal” key-escrow described in section 3.1.2, is that users can generate their keys at any time, without the intervention of other parties, such as registration authorities or investigation agencies. The private keys can be exfiltrated from the public output of the cryptosystem and there is no need for a connection between each device and the other parties.

**Summary**

PAP is a SETUP mechanism for RSA key pair generation. It incorporates an encryption of prime $p$ in the user’s modulus $n$. Eve uses her private decryption exponent $D$ to exfiltrate $p$ from $n$. Next, she
3.5 SETUP IN ELGAMAL

can factor $n$, compute Alice’s private key $d$ and decrypt messages that Alice receives. Eve only needs
one modulus to be able to exfiltrate the private key, so this attack has a $(1,1)$-leakage scheme. In this
attack, the output from the contaminated system is polynomially indistinguishable from the output from
the normal system as long as Alice does not know the exact internals of the device. This means that
this SETUP mechanism is a regular SETUP attack. It is not a strong SETUP attack, because of the
following reason.

Alice knows her own public modulus $n$ and her public and private exponent $e$ and $d$. She can use
these to factor $n$ and recover the primes $p$ and $q$. If Alice knows that her system is contaminated and
she figures out how the system works, she can easily encrypt the prime $p$ and perform all the other steps
of the system. Then she will definitely notice that $p$ is incorporated in the $k$ upper order bits of her
modulus $n$. In this way, she can distinguish output from the contaminated system from the output of a
normal system within polynomial time. Therefore, this is not a strong SETUP attack.

3.5 SETUP in ElGamal

As explained in section 2.4.3, ElGamal designed two systems based on the discrete logarithm problem:
one system for encryption and a digital signature scheme. Young and Yung designed SETUP mechanisms
for the ElGamal key generation, and a SETUP mechanism for the signature scheme [36].

Recall that first the system parameters are chosen: prime $p$ and generator $g$ of $\mathbb{Z}_p^*$. Note that usually
these system parameters are shared immediately between all users. Next, the private key $x$ is chosen
randomly from $\{1,2,\ldots,p-1\}$ and the public key $y$ is set equal to $g^x \pmod{p}$. Young and Yung describe
two different versions of SETUP in ElGamal key generation. The first version is very similar to PAP:
the attacker’s keys are still RSA keys and RSA encryption is used. In the second version, referred to as
a “pure ElGamal system”, the attacker’s keys are generated using ElGamal and ElGamal encryption is
used.

It is important to note that the attacks described in this section are less powerful than the attack on
RSA (PAP), because of the following. As mentioned before, prime $p$ and generator $g$ are usually shared
immediately between all parties and are kept fixed. In the first attack on ElGamal key generation, only
$p$ is shared and fixed and $g$ varies. In practice, this is not very likely. The same reasoning applies for the
second attack on ElGamal key generation, where both $p$ and $g$ are recomputed each time a new public
key is generated. On the contrary, the attack on the ElGamal signature scheme is more realistic, since
in this attack $p$ and $g$ are kept fixed.

3.5.1 SETUP in ElGamal Key Generation I

In this first version of SETUP in ElGamal key generation, the program incorporates the user’s private
key $x$ in the generator $g$, which is a public parameter.

Algorithm

This version is explained very briefly by Young and Yung as follows [36]. Assume that the $k$-bit prime
$p$ is shared between two or more parties and that private key $x$ and generator $g$ will be generated by an
ElGamal key generation program. In this case, the private key $x$ will be used to compute the generator $g$,
such that the attacker can recover the private key using $g$. Note that this generator $g$ will also be
published by Alice, so that each party that wants to communicate with Alice will use her parameters.
Let $(E,N)$ and $(D,N)$ be the attacker’s public and private key, generated with RSA, with $N$ the $k$-bit
attacker’s modulus. Then the following steps are executed:

- First, $x$ is generated randomly from $\{1,2,\ldots,p-1\}$.
- Next, $x$ is encrypted with the attacker’s public key $(E,N)$ and randomized with a keyed random-
  ization function to get $x'$.
- If $x' < p$ and $x'$ is a generator of $\mathbb{Z}_p$, then $x'$ is set to be the generator $g$. Otherwise, a new $x$ is
  generated and the previous step is repeated.

This description is not complete. For example, $x$ is encrypted using $(E,N)$, but in this case it is possible
that $x > N$ (if $N < p$). In order to decrypt, $x$ should be smaller than $N$. Furthermore, if $x'$ does
CHAPTER 3. KLEPTOGRAPHY

not satisfy the program, the program restarts immediately. However, this is increases the computational complexity of the program.

The first issue can be solved in three different ways. An easy fix would be to generate \( x \) randomly from \( \{1, 2, \ldots, N-1\} \) instead of \( \{1, 2, \ldots, p-1\} \). However, if \( N \ll p \), the outputs might become polynomially distinguishable. The second possibility is the following. Assume that \( N < p < 2N \). If it turns out that \( N < x < p \), the program could compute with \( x' \equiv x \pmod{N} \). The decryption then gives an \( x' \), so that \( x = x' + N \). Eve checks both values. In PAP, a keyed randomization function was inserted to generate a random integer smaller than the attacker’s modulus \( N \), so this approach can be used for SETUP in ElGamal as well. This fix is used in this thesis. To avoid the restart of the program, the program could return to a keyed randomization function (just as in PAP). With both steps added to the description, the program looks as follows:

- First, \( x \) is generated randomly from \( \{1, 2, \ldots, p-1\} \).
- Next, the program uses a keyed randomization function \( R_1 \) with key \( K+i \) to randomize \( x \) (initially \( i = 0 \)):

\[
x' = R_1(x),
\]

with \( R_1 : \{0, 1\}^k \rightarrow \{0, 1\}^k \).

- The following step has two possibilities:
  - If \( x' \geq N \), then \( i = i+1 \) and the previous step is executed again. Just as in PAP, this process continues at most \( B_1 \) times. If \( i = B_1 \), then \( i \) is set to 0 again and the program restarts with a new value for \( x \).
  - If \( x' < N \), \( x' \) can be encrypted with the attacker’s public key:

\[
x'' \equiv (x')^E \pmod{N},
\]

- Then, \( x'' \) is randomized with keyed randomization function \( R_2 \) and key \( K+j \) (initially \( j = 0 \)):

\[
x''' = R_2(x''),
\]

with \( R_2 : \{0, 1\}^k \rightarrow \{0, 1\}^k \).

- At this point, there are two possibilities:
  - If \( x''' \equiv p \) and/or \( x''' \) is not a generator of \( \mathbb{Z}_p^* \), then \( j = j+1 \) and \( x''' \) is randomized again. This continues at most \( B_2 \) times. If \( j = B_2 \), the program restarts and generates a new \( x \) (with \( i = 0 \) and \( j = 0 \)).
  - If \( x''' \equiv p \) and \( x''' \) is a generator of \( \mathbb{Z}_p^* \), then:

\[
g = x'''.
\]

- Finally, the program outputs \( p \) and \( g \) as system parameters, \( x \) as private key and \( y = g^x \pmod{p} \) as public key.

In appendix B.1 (p. 91), a description of the algorithm in pseudocode can be found.

Recall that in PAP the attacker’s modulus is a \( k \)-bit integer and the user’s modulus is a 2\( k \)-bit integer. This means that the RSA keys of the user are more secure than the RSA keys of the attacker. In the SETUP attack on ElGamal key generation, the keys of the user and attacker are approximately the same size, which means that the user’s ElGamal system and the attacker’s RSA system have approximately the same security. This implies that this attack is more secure than PAP for the attacker.

**Recovery of Private Key by Attacker**

The attacker can compute the user’s private key \( x \) using the public generator \( g \). Note that \( g = x''' \). Eve performs the following steps:

- First, she uses the inverse of the keyed randomization function \( R_2 \) and the key \( K+j \) to get \( x''' \):

\[
x''' = R_2^{-1}(g).
\]

Just as in PAP, Eve does not know if and how often the program recomputed \( x''' \). This means that she has to do this step for all possible keys, so for \( 0 \leq j \leq B_2 - 1 \). Eve will get \( B_2 \) different possible values of \( x''' \). However, Eve does know that \( x''' \equiv N \), since \( x' \) was encrypted using \( E \) and \( N \). This means that she could exclude all possible values of \( x''' \) that are larger than \( N \).
• Next, she decrypts all possible values of \( x'' \) using her private decryption exponent \( D \):
\[
x' \equiv (x'')^D \pmod{N}.
\]
• Eve uses the inverse of the keyed randomization function \( R_1 \) with key \( K + i \) to compute \( x \):
\[
x = R_1^{-1}(x').
\]
Again, since Eve does not know the value of \( i \), she needs to do this for all possible values of \( i \), so \( 0 \leq i \leq B_1 \). But she also knows that \( x < p \), so all possible values for \( x \) that are larger than \( p \), can be excluded immediately
• Now Eve has at most \( B_1 \cdot B_2 \) possible private keys. If this program is used to generate the user’s parameters and keys, then for one of the possible keys the following equation should hold:
\[
g^x \equiv y \pmod{p}.
\]
If this equation is satisfied, then Eve knows that she found Alice’s private key.
Since Eve only needs one public generator \( g \) to exfiltrate the user’s private key \( x \), this attack has a \((1,1)\)-leakage scheme.

In example 3.5, a worked out example of SETUP in ElGamal key generation can be found. The following aspects of the attack are shown. First, the attacker’s key pairs are generated. Next, the program generates prime \( p \), Alice’s private key \( x \), generator \( g \) and Alice’s public key \( y \). Then, Bob uses Alice’s public key to encrypt a message, which is decrypted by Alice, using her own private key \( x \). Finally, it is shown how Eve exfiltrates Alice’s private key \( x \) from generator \( g \). Note that in this example, the possible loops have been omitted. It is assumed that \( x' \) and \( x''' \) are immediately usable and that there is no need to recompute these values.

Example 3.5: SETUP in ElGamal key generation I

The steps are executed as follows:

- **Key pair generation by Eve.** Eve generates the two keys as follows:
  - \( P = 683 \) and \( Q = 863 \);
  - \( N = P \cdot Q = 683 \cdot 863 = 589429 \);
  - \( \varphi(N) = (P - 1)(Q - 1) = 682 \cdot 862 = 578884 \);
  - \( E = 39451 \);
  - \( D \equiv E^{-1} \equiv 319163 \pmod{578884} \);
  - public key \( (E, N) = (39451, 589429) \);
  - private key \( (D, N) = (319163, 589429) \).
- **Key generation by Alice.** The contaminated program computes generator \( g \), prime \( p \), private key \( x \) and public key \( y \) as follows:
  - The program generates a random prime \( p \):
    
    \[
p = 42307.
    \]
  - Then the private key \( x \) is generated randomly:
    
    \[
x = 19243.
    \]
  - The program computes \( x' \) using the first keyed randomization function, \( R_1 \). Assume that:
    
    \[
x' = R_1(x) = R_1(19243) = 139222.
    \]
  - Next, the program encrypts \( x' \) with the attacker’s encryption exponent \( E \):
    
    \[
x'' \equiv (x')^E \equiv 139222^{319163} \equiv 185758 \pmod{589429}.
    \]
  - Using the second keyed randomization, \( R_2 \), the program computes \( x''' \). Assume that:
    
    \[
x''' = R_2(x'') = 11957.
    \]
– Since \( x''' < p \) and \( x''' \) is a generator of \( \mathbb{Z}_p^* \), \( x''' \) becomes the generator \( g \). So:
\[
g = x''' = 11957.
\]

– Finally, the program computes Alice’s public key:
\[
y \equiv g^x \equiv 11957^{19243} \equiv 1862 \pmod{42307}.
\]

• Encryption by Bob. Bob wants to send the message “HI” to Alice. He encodes this as \( m = 0809 \), chooses \( k = 1487 \) and encrypts, using Alice’s public key and the public parameters:
\[
c_1 \equiv g^k \equiv 11957^{1487} \equiv 12259 \pmod{42307},
\]
and:
\[
c_2 \equiv m \cdot y^k \equiv 809 \cdot 1862^{1487} \equiv 28040 \pmod{42307}.
\]
Bob sends \((c_1, c_2) = (12259, 28040)\) to Alice.

• Decryption by Alice. Alice receives \((12259, 28040)\) from Bob and decrypts this using her private key:
\[
m \equiv c_2 \equiv \frac{28040}{12249^{19243}} \equiv 809 \pmod{42307}.
\]
Finally, she decodes this as “HI”.

• Attack by Eve. Eve computes private key \( x \) as follows.
  – First, Eve computes \( x'' \) using generator \( g \) and the inverse of the second keyed randomization function, \( R_2 \):
    \[
x'' = R_2^{-1}(g) = R_2^{-1}(11957) = 185758.
    \]
  – Then, she decrypts this using her own decryption exponent \( D \):
    \[
x' \equiv (x'')^D \equiv 185758^{319163} \equiv 139222 \pmod{589429}.
    \]
  – Finally, Eve computes, using the inverse of the first keyed randomization function, \( R_1 \):
    \[
x = R_1^{-1}(x') = R_1^{-1}(139222) = 19243,
    \]
and Eve can decrypt the message the same way Alice did.

Bounds \( B_1 \) and \( B_2 \)

Just as in PAP, \( B_1 \) and \( B_2 \) should be chosen such that it is very likely that the program generates a generator. For PAP, Young and Yung chose \( B_1 = 16 \) and \( B_2 = 512 \). For the first attack on ElGamal key generation, they do not give any information on these bounds.

\( B_1 \) can be chosen the same as in PAP. For \( B_1 \), \( x' \) should be smaller than \( N \) in order to be able to encrypt. Note that \( x', N \in \{0, 1\}^k \), such that:
\[
P(x' < N) = P(p' \geq N) = \frac{1}{2}.
\]
Then:
\[
P(x' \geq N \text{ after } B_1 \text{ times}) = \left(\frac{1}{2}\right)^{B_1}.
\]
This implies that:
\[
P(x' < N \text{ after } \leq B_1 \text{ times}) = 1 - \left(\frac{1}{2}\right)^{B_1}.
\]
3.5. SETUP IN ELGAMAL

With $B_1 = 16$, this means that:

$$ P(x' < N \text{ after } \leq 16 \text{ times}) = 1 - \left( \frac{1}{2} \right)^{16} \approx 1.00, $$

which is very acceptable. Note that this is the same computation as for PAP. Again, Eve could make sure that her public modulus $N$ is close to $2^k$, such that the probability that a proper $x'$ is generated increases.

For $B_2$, one needs the probability that $x''' < p$ and that $x'''$ is a generator of $\mathbb{Z}_p^*$. Since both $x'''$ and $p$ are $k$-bit integers:

$$ P(x''' < p) = P(x''' \geq p) = \frac{1}{2}. \quad (3.1) $$

Assuming that $x''' < p$, one needs the probability that $x'''$ is a generator of $\mathbb{Z}_p^*$. Theorem 2.27 is used to compute this probability. In this case, $G = \mathbb{Z}_p^*$. The order of this group is $p - 1$, which means that there are $\varphi(p - 1)$ elements that generate $G$. Next, one needs a lower bound for Euler's totient function $\varphi$ (see section 2.2.3 for Euler's totient function). The following bound is from [2]:

**Theorem 3.12.** Let $\varphi$ denote Euler's totient function. Then, for $n \geq 3$:

$$ \varphi(n) > \frac{n}{e^\gamma \log \log n + \frac{3}{\log \log n}}, $$

where $\gamma$ denotes the Euler-Mascheroni constant ($\gamma \approx 0.577$).

A proof of this theorem can be found in [25]. This means that from the $p - 1$ elements of $\mathbb{Z}_p^*$, at least:

$$ \frac{p - 1}{e^\gamma \log \log(p - 1) + \frac{3}{\log \log(p - 1)}} $$

are a generator of this group. Using this theorem, one gets, assuming that $x$ is smaller than $p$ and that there are $2^k$ possibilities for $p$:

$$ P(x''' \text{ is a generator of } \mathbb{Z}_p^*) \approx \frac{e^\gamma \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}}{2^k - 1}. $$

Combining 3.1 and 3.2 gives:

$$ P(x''' < p \text{ and } x''' \text{ is a generator of } \mathbb{Z}_p^*) \approx \frac{1}{2 \left( e^\gamma \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)} \right)}. $$

Then:

$$ P(x''' \geq p \text{ and/or } x''' \text{ is not a generator of } \mathbb{Z}_p^*) \approx 1 - \frac{1}{2 \left( e^\gamma \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)} \right)}. $$

If $x''' \geq p$ or $x''' < p$, but $x'''$ is not a generator, the program generates a new value for $x'''$ at most $B_2$ times. So:

$$ P(x''' \geq p \text{ and/or } x''' \text{ is not a generator of } \mathbb{Z}_p^* \text{ after } B_2 \text{ times}) \approx \left( 1 - \frac{1}{2 \left( e^\gamma \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)} \right)} \right)^{B_2}. $$
Finally, the probability that \( x'' < p \) and \( x'' \) is a generator after at most \( B_2 \) times becomes:

\[
P(x'' < p \text{ and } x'' \text{ is a generator after } \leq B_2 \text{ times}) = 1 - \left( 1 - \frac{1}{2 \left( e^\gamma \log \log (2^k - 1) + \frac{3}{\log \log (2^k - 1)} \right)^{B_2}} \right).
\]

In the table in figure 3.6, the probabilities can be found for different key sizes \( k \) and different bounds \( B_2 \).

<table>
<thead>
<tr>
<th>( B_2 )</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 256 )</td>
<td>0.57</td>
<td>0.81</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>( k = 512 )</td>
<td>0.53</td>
<td>0.77</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>( k = 1024 )</td>
<td>0.50</td>
<td>0.74</td>
<td>0.93</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>( k = 2048 )</td>
<td>0.46</td>
<td>0.71</td>
<td>0.91</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>( k = 4096 )</td>
<td>0.43</td>
<td>0.67</td>
<td>0.89</td>
<td>0.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 3.6: SETUP in ElGamal key generation I - Probabilities for key size \( k \) and bound \( B_2 \), rounded to two decimal digits.

As can be seen in this figure, \( B_2 = 128 \) suffices for all possible key sizes between 256 and 4096 bits.

**Summary**

This attack is similar to PAP. It describes a SETUP mechanism for ElGamal key generation. Note that the public and private keys of the attacker are generated using RSA. The user’s private key \( x \) is incorporated in the user’s public parameter \( g \). Eve can exfiltrate \( x \) by decrypting the \( g \) with her own decryption exponent \( D \). Since Eve only needs the public parameter \( g \) once, this attack has a \((1, 1)\)-leakage scheme.

Alice knows the public parameter \( g \) and her private key \( x \). If she suspects that the system is contaminated and she is able to reverse-engineer it, she can use her own private key and follow all the steps of the algorithm. In this case, it is easy to notice whether \( x \) is incorporated in the public parameter \( g \). Therefore, this attack is not a strong SETUP attack. It is actually a regular SETUP attack, since the output of the system is polynomially indistinguishable from the output of a normal ElGamal key generation system, even for Alice knowing her private key. Finally, as mentioned before, this attack is not very likely to be used, since usually \( p \) and \( g \) are shared and fixed and in this attack \( g \) is variable.

**Variant to Version I**

Young and Yung also mention a variant to this first version. The difference is that the generator \( g \) is shared between two or more parties, instead of prime \( p \), and that \( x \) and \( p \) are generated using an ElGamal key generation program. Here \( x \) is used to generate \( p \), such that the attacker can recover \( x \) using \( p \). However, since this is even less likely to occur, this variant is not discussed in this thesis.

### 3.5.2 SETUP in ElGamal Key Generation II

In the previous version of SETUP in ElGamal key generation, the attacker’s private and public key, \((D, N)\) and \((E, N)\), are generated using RSA key generation. In this version of SETUP, the attacker’s keys are generated using ElGamal with prime \( P \) and generator \( G \). The program generates the user’s system parameters \( p \) and \( g \), the private key \( x \) and the public key \( y \) using the attacker’s parameters and the attacker’s public key \( Y \).
3.5. SETUP IN ELGAMAL

Algorithm

Let \( P \) be the attacker’s k-bit prime, \( X \in \{1, 2, \ldots, P - 1\} \) the attacker’s private key, \( G \) the generator of \( \mathbb{Z}_P^* \) and \( Y \equiv G^X \mod P \) the attacker’s public key. Note that all parameters and keys are k-bit integers.

- First, the user’s private key \( x \) is generated randomly from \( \{1, 2, \ldots, P - 1\} \).
- Next, integer \( k \) is generated randomly.
- Then \( x \) is encrypted using ElGamal encryption with the attacker’s parameters, i.e. prime \( P \) and public key \( Y \):
  \[ c_2 \equiv x \cdot Y^k \mod P. \]
- Next, \( c_2 \) is randomized using a keyed randomization function \( R_1 \) with key \( K + i \) (\( i \) initially equal to 0):
  \[ c_2' = R_1(c_2), \]
  with \( R_1 : \{0, 1\}^k \rightarrow \{0, 1\}^k \). This value will be used to generate prime \( p \) for the user.
- Then there are again two possibilities:
  - If \( c_2' \) is not prime and/or \( c_2' \leq x \), then \( i = i + 1 \) and \( c_2 \) is randomized again (at most \( B_2 \) times).
  - If \( c_2' \) is prime and \( c_2' > x \), then the program continues with:
    \[ c_1 \equiv G^{k'} \mod P. \]

The integer \( c_1 \) will be used to generate the generator \( g \).
- Next, \( c_1 \) is randomized with keyed randomization function \( R_2 \) and key \( K + j \) (\( j = 0 \) initially):
  \[ c_1' = R_2(c_1), \]
  with \( R_2 : \{0, 1\}^k \rightarrow \{0, 1\}^k \).
- After the randomization of \( c_1 \), there are again two possibilities:
  - If \( c_1' \geq c_2' \) and/or \( c_1' \) is not a generator of \( \mathbb{Z}^{*}_{c_2} \), then \( j = j + 1 \) and the program randomizes \( c_1 \) with key \( K + j \) (at most \( B_2 \) times).
  - If \( c_1' < c_2' \) and \( c_1' \) generates \( \mathbb{Z}^{*}_{c_2} \), then:
    \[
    p = c_2' \\
    g = c_1' \\
    y \equiv g^x \mod p,
    \]
    and the program outputs system parameters \( p \) and \( g \) and keys \( x \) and \( y \).

In appendix B.2 (p. 93), a description of the algorithm in pseudocode can be found.

Note that just as in the first attack on ElGamal key generation, the keys of the attacker have approximately the same size as the keys of the user. This implies that the system for the attacker has approximately the same security as the system for the user.

The original program described by Young and Yung in [36] is slightly different. Young and Yung use the extra condition that \( k \) and \( P - 1 \) should be relatively prime. However, there does not arise a problem if \( k \) and \( P - 1 \) are not relatively prime, so this condition is removed from the algorithm. Furthermore, in the original program the second keyed randomization function does not appear. The random integer \( k \) was regenerated if the two conditions for \( c_1 \) were not satisfied. However, this increases the computational complexity. To avoid the restart of the program, the keyed randomization function \( R_2 \) is added, so that \( c_1' \) can be randomized again if it does not satisfy the conditions.

The keyed randomization function \( R_1 \) is used for the same reason as in PAP: it makes sure that prime \( p \) looks random to the user. Furthermore, if \( c_2' \) does not satisfy the two conditions, \( c_2 \) can be randomized again. Without the randomization, another \( k \) should be generated to compute a different \( c_2 \). However, this requires an encryption, which is computationally more complex than a randomization.

The integer \( c_2' \) should satisfy two conditions. First, it should be a prime, since \( c_2' \) becomes the prime of the system parameters of the user. Second, it should be greater than the user’s private key \( x \). This is because eventually \( x \) should be an element of \( \{1, 2, \ldots, p - 1\} \), so \( x \in \{1, 2, \ldots, c_2' - 1\} \). This implies that \( p > x \) and thus \( c_2' > x \). The integer \( c_1' \) should also satisfy two conditions: \( c_1' \) is assigned to be the generator of \( \mathbb{Z}_p \), so \( c_1' \) should be smaller than \( p \) (\( = c_2' \)) and it should generate the group.
Recovery of Private Key by Attacker

Eve uses the user’s system parameters $p$ and $g$ to recover the user’s private key $x$. Note that $p = c'_2$ and $g = c'_1$. She performs the following steps:

- First, Eve uses the inverse of the keyed randomization function $R_2$ with key $K + j$ to get $c_1$:
  \[ c_1 = R_2^{-1}(g). \]

  Since there are $j$ possible keys, this gives $j$ possible values for $c_1$. Eve can immediately exclude all the values that are equal to or larger than $P$, since $c_1 < P$.

- Next, she uses the inverse of the keyed randomization function $R_1$ with key $K + i$ to get $c_2$:
  \[ c_2 = R_1^{-1}(p). \]

  This gives $i$ possible values for $c_2$. Again, Eve can exclude all values equal to or larger than $P$.

- Next, Eve uses her own private key $X$ to recover the user’s private key $x$:
  \[ x = \frac{c_2}{c_1^X}. \]

- Finally, Eve has at most $i \cdot j$ possible private keys. Eve can use Alice’s public key $y$ to check whether one of these possible private keys satisfies:
  \[ g^x \equiv y \pmod{p}. \]

Eve indeed recovers the user’s private key, since:

\[ \frac{c_2}{c_1^X} = \frac{x \cdot Y^k}{(G^k)^X} = \frac{x \cdot (G^X)^k}{G^k} = \frac{x \cdot G^{X \cdot k}}{G^{X \cdot k}} = x. \]

Note that this attack has a (1,1)-leakage scheme, because Eve only needs the pair $p$ and $g$ once to exfiltrate the private key $x$.

In example 3.6, a worked-out example of SETUP in ElGamal key generation II can be found. The following aspects of the attack are shown. First, the attacker’s private and public key are generated using ElGamal. Next, the program generates Alice’s private key $x$, prime $p$, generator $g$ and Alice’s public key $y$. Then, Bob uses Alice’s public key to encrypt a message, which is decrypted by Alice, using her own private key $x$. Finally, it is shown how Eve exfiltrates Alice’s private key $x$ from prime $p$ and generator $g$. Note that in this example, the possible loops have been omitted. It is assumed that $c'_1$ and $c'_2$ are immediately usable and there is no need to recompute these values.

Example 3.6: SETUP in ElGamal key generation II

The steps are executed as follows:

- **Key pair generation by Eve.** Eve generates her private and public key as follows:
  - First she chooses her system parameters, let $P = 23993$ and $G = 15765$, such that $G$ generates $\mathbb{Z}_P$.
  - Eve chooses $X = 9237$ as private key and computes the public key as follows:
    \[ Y \equiv G^X \equiv 15765^{9237} \equiv 6211 \pmod{23993}. \]

- **Key generation by Alice.** The contaminated program computes generator $g$, prime $p$, private key $x$ and public key $y$ as follows:
  - The program generates Alice’s private key randomly: $x = 19243$.
  - Then the program generates $k = 7661$ randomly.
  - Using the private key and the attacker’s $Y$ and $P$:
    \[ c_2 \equiv x \cdot Y^k \equiv 19243 \cdot 6211^{7661} \equiv 21843 \pmod{23993}. \]
3.5. SETUP IN ELGAMAL

The program uses the first randomization function to compute:
\[ c_2' = R_1(c_2) = R_1(21843) = 27337. \]

Note that \( c_2' \) is prime and \( c_2' > x \).

Next, the program computes:
\[ c_1 \equiv g^k \equiv 15765^{7661} \equiv 7495 \pmod{23993}. \]

Then, using the second randomization function:
\[ c_1' = R_2(c_1) = R_2(7495) = 10023. \]

Finally, since \( c_1' < c_2' \) and \( c_1' \) is a generator of \( \mathbb{Z}_{c_2'}^* \), the program outputs these parameters and Alice’s public key \( y \):
\[ p = c_2' = 27337 \]
\[ g = c_1' = 10023 \]
\[ y \equiv g^x \equiv 10023^{19423} \equiv 13027 \pmod{27337}. \]

- **Encryption by Bob.** Bob wants to send the message “HI” to Alice. He encodes this as \( m = 0809 \), chooses \( k = 1487 \) and encrypts, using Alice’s public key and the public parameters:
\[ r \equiv g^k \equiv 10023^{1487} \equiv 16434 \pmod{27337}, \]
and:
\[ s \equiv m \cdot y^k \equiv 809 \cdot 13027^{1487} \equiv 17176 \pmod{27337}. \]

Bob sends \((r, s) = (16434, 17176)\) to Alice.

- **Decryption by Alice.** Alice receives \((16434, 17176)\) from Bob and decrypts this using her private key:
\[ m \equiv s \cdot r^{-x} \equiv 17176 \cdot 16434^{19412} \equiv 809 \pmod{27337}. \]

Finally, she decodes this as “HI”.

- **Attack by Eve.** Eve computes private key \( x \) as follows.
  - First, Eve computes \( c_1 \) using generator \( g \) and the inverse of the second keyed randomization function, \( R_2 \):
\[ c_1 = R_2^{-1}(g) = R_2^{-1}(10023) = 7495. \]
  - Then, she computes, using the inverse of the first keyed randomization function, \( R_1 \), and prime \( p \):
\[ c_2 = R_1^{-1}(p) = R_1^{-1}(27337) = 21843, \]
  - Finally, Eve decrypts this using her own private key \( X \):
\[ x \equiv \frac{c_2}{c_1^X} \equiv \frac{21843}{7495^{9237}} \equiv 19423 \pmod{23993}. \]

and Eve can decrypt the message the same way Alice did.

**Bounds \( B_1 \) and \( B_2 \)**

As mentioned before, in the original program described by Young and Yung in [36], only the first keyed randomization function appears. However, they do not give any information on the corresponding bound \( B_1 \). In this thesis, acceptable values for both \( B_1 \) and \( B_2 \) are computed.

The \( k \)-bit integer \( c_2' \) has to satisfy two conditions: it should be prime and it should be greater than the
user’s private key \( x \). This means that \( B_1 \) should be chosen such that there is an acceptable probability that the program generates a \( c'_2 \) that satisfies these conditions. The integer \( c'_2 \) has a length of \( k \) bits. Using theorem 3.10, this implies that:

\[
P(c'_2 = \text{prime}) \approx \frac{1}{\log(2^k)}.
\]  

(3.3)

The private key \( x \) also has a length of \( k \) bits. This means that:

\[
P(c'_2 > x) = P(c'_2 \leq x) = \frac{1}{2}.
\]  

(3.4)

Combining 3.3 and 3.4 gives:

\[
P(c'_2 = \text{prime and } c'_2 > x) \approx \frac{1}{2\log(2^k)}.
\]

Then:

\[
P(c'_2 \neq \text{prime and/or } c'_2 \leq x) \approx 1 - \frac{1}{2\log(2^k)}.
\]

The program tries to generate a proper \( c'_2 \) at most \( B_1 \) times, so:

\[
P(c'_2 \neq \text{prime and/or } c'_2 \leq x \text{ after } B_1 \text{ times}) \approx \left(1 - \frac{1}{2\log(2^k)}\right)^{B_1}.
\]

Finally:

\[
P(c'_2 = \text{prime and } c'_2 > x \text{ after } \leq B_1 \text{ times}) \approx 1 - \left(1 - \frac{1}{2\log(2^k)}\right)^{B_1}.
\]

Note that this is the same computation as for \( B_1 \) in PAP, except for the factor 2. This factor appears, since for PAP the only condition was that \( q \) should be prime, whereas here \( c'_2 \) should be prime and it should be larger than \( x \). In the table in figure 3.7, the probabilities can be found for different key sizes \( k \) and different bounds \( B_1 \).

<table>
<thead>
<tr>
<th>( B_1 )</th>
<th>0.76</th>
<th>0.94</th>
<th>1.00</th>
<th>1.00</th>
<th>1.00</th>
<th>1.00</th>
<th>1.00</th>
<th>1.00</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.51</td>
<td>0.76</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1024</td>
<td>0.30</td>
<td>0.51</td>
<td>0.76</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2048</td>
<td>0.17</td>
<td>0.30</td>
<td>0.51</td>
<td>0.76</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4096</td>
<td>0.09</td>
<td>0.17</td>
<td>0.30</td>
<td>0.51</td>
<td>0.76</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 3.7: SETUP in ElGamal key generation II - Probabilities for key size \( k \) and bound \( B_1 \), rounded to two decimal digits.

As can be seen in this figure, it is advisable to take \( B_1 \) at least four times as large as the target size \( k \). However, one could question whether this is still practical. Obviously, \( B_1 \) should be that large, otherwise the probability of success is simply too low. But a relatively large computation time could result into a suspicious user. The supplier of the device could tell the user that prime generation just takes a lot of time. However, since \( p \) and \( q \) are usually fixed, this is not ideal.

For \( c'_1 \), there are also two conditions: \( c'_1 \) should be smaller than \( c'_2 \) and it should generate \( \mathbb{Z}^*_p \). The bound \( B_2 \) should be chosen such that the program will probably find a sufficient \( c'_1 \). Since \( c'_1 \) and \( c'_2 \) are both \( k \)-bit integers:

\[
P(c'_1 < c'_2) = P(c'_1 \geq c'_2) = \frac{1}{2}.
\]  

(3.5)
3.5. SETUP IN ELGAMAL

Then, using theorems 2.27 and 3.12:

\[
P(c'_1 \text{ is a generator of } \mathbb{Z}_{c'_2}^*) \approx \frac{1}{e^{\gamma} \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}}.
\]  

(3.6)

Combining 3.5 and 3.6 gives:

\[
P(c'_1 < c'_2 \text{ and } c'_1 \text{ is a generator of } \mathbb{Z}_{c'_2}^*) \approx \frac{1}{2 \left(e^{\gamma} \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}\right)}.
\]

Then:

\[
P(c'_1 \geq c'_2 \text{ and/or } c'_1 \text{ is not a generator of } \mathbb{Z}_{c'_2}^*) \approx 1 - \frac{1}{2 \left(e^{\gamma} \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}\right)}.
\]

Such that:

\[
P(c'_1 \geq c'_2 \text{ and/or } c'_1 \text{ is not a generator of } \mathbb{Z}_{c'_2}^* \text{ after } B_2 \text{ times}) \approx \left(1 - \frac{1}{2 \left(e^{\gamma} \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}\right)}\right)^{B_2}.
\]

Finally, the probability that \(c'_1 < c'_2\) and \(c'_1\) is a generator after at most \(B_2\) times becomes:

\[
P(c'_1 < c'_2 \text{ and } c'_1 \text{ is a generator of } \mathbb{Z}_p^* \text{ after } \leq B_2 \text{ times}) = 1 - \left(1 - \frac{1}{2 \left(e^{\gamma} \log \log(2^k - 1) + \frac{3}{\log \log(2^k - 1)}\right)}\right)^{B_2}.
\]

This is the same computation as for the first version of SETUP in ElGamal key generation. So this will give exactly the same table as in figure 3.6, which showed that \(B_2 = 128\) suffices for \(256 \leq k \leq 4096\). If \(k = 2048\), a commonly used size for ElGamal key generation at the moment, then \(B_1 = 8192\) and \(B_2 = 128\) will suffice.

Summary

Just as the previous attack, this attack describes a SETUP mechanism for ElGamal key generation. A difference with the previous attack is that now the attacker’s private and public key are generated using ElGamal key generation. Eve uses the public prime \(p\) and generator \(g\) to exfiltrate the user’s private key \(x\). Since she only needs these parameters once, this attack has a \((1, 1)\)-leakage scheme.

Just as in the previous attack, Alice knows the public parameter \(g\) and her private key \(x\). However, in this attack, the program uses ElGamal encryption instead of RSA encryption. In ElGamal encryption, randomness is added with the integer \(k\). Alice cannot predict future keys given her present keys. This means that this attack is a strong SETUP attack.

3.5.3 SETUP in ElGamal Signature Scheme

Young and Yung also proposed a SETUP mechanism for the ElGamal signature scheme. The attacker’s keys are again generated using ElGamal key generation. In this program, the public parameters of the attacker are also used as the user’s public parameters. This means that the public parameters of the user remain fixed, whereas in the previous version of SETUP these parameters were recomputed each time the device was used. Therefore this version of SETUP is much more realistic.
Algorithm I

Let $p$ be the attacker’s and user’s prime and $g$ a generator of $\mathbb{Z}_p^*$. Let $X \in \{1, 2, \ldots, p - 1\}$ be the attacker’s private key and $Y \equiv g^X \pmod{p}$ the corresponding public key and let $x$ be the user’s private key.

Recall from section 2.4.3 that Alice sent the triple $(m, r, s)$ to Bob for authentication. Note that $m$ is the hash of the message and $r$ and $s$ form the actual signature. In this attack, Eve must get at least two consecutive signatures from Alice, so $(m_i, r_i, s_i)$ and $(m_{i+1}, r_{i+1}, s_{i+1})$. The program computes these signatures as follows:

- A random integer $k_i$ is generated randomly from $\{1, 2, \ldots, p - 2\}$ such that $\gcd(k_i, p - 1) = 1$. This is the usual condition for the ElGamal signature scheme.
- If the condition in the previous step is satisfied, the signature is computed as follows:

$$r_i \equiv g^{k_i} \pmod{p}$$

$$s_i \equiv k_i^{-1}(m_i - x \cdot r_i) \pmod{p - 1},$$

with $x$ the user’s private key and $m_i$ the hash of the user’s message. The program outputs signature $(m_i, r_i, s_i)$.

- The subsequent signature is generated as follows. First, $c \in \{1, 2, \ldots, p - 2\}$ is considered as an integer ($c \mod{p}$) and $(c, p, -1, x)$. Note that $c$ needs the inverse of $k_i$ modulo $p$ if both conditions are satisfied, to compute $k_{i+1}$.

If this condition is satisfied, the inverse of $c$ modulo $p - 1$ exists (theorem 2.18). Let $c^{-1} \in \{1, 2, \ldots, p - 2\}$ be the modular inverse of $c$ modulo $p - 1$. If the previous condition and:

$$\gcd(g^{c^{-1}} \pmod{p}, p - 1) = 1,$$

are satisfied, then the integer $k_{i+1}$ is not generated randomly, just as $k_i$, but:

$$k_{i+1} = c^{-1}.$$

If conditions 3.7 and 3.8 are not satisfied, the program generates a random $k_{i+1}$, such that $k_{i+1} \in \{1, 2, \ldots, p - 2\}$ and $\gcd(k_{i+1}, p - 1) = 1$, and continues as the normal ElGamal signature scheme. This means that the program outputs a proper signature for the user, but this signature cannot be used by the attacker to recover the private key.

- The signature is then computed as usual:

$$r_{i+1} \equiv g^{k_{i+1}} \pmod{p}$$

$$s_{i+1} \equiv k_{i+1}^{-1}(m_{i+1} - x \cdot r_{i+1}) \pmod{p - 1}.$$

The program outputs the signature $(m_{i+1}, r_{i+1}, s_{i+1})$.

In appendix B.3 (p. 95), a description of the algorithm in pseudocode can be found.

In the first step, $k_i$ and $p - 1$ should be relatively prime. This is because in the next step the program needs the inverse of $k_i$ modulo $p - 1$. This inverse does not exist if $\gcd(k_i, p - 1) \neq 1$ (theorems 2.18 and 2.19). Note that $c$ is considered as an integer ($c \in \{1, 2, \ldots, p - 1\}$). Condition 3.7 should be satisfied, since in the program needs the inverse of $c$ modulo $p - 1$. This inverse is used in condition 3.8 and, if both conditions are satisfied, to compute $k_{i+1}$. This means that if $c$ is used to compute $k_{i+1}$, then $c \in \{1, 2, \ldots, p - 2\}$ (if $c = p - 1$, then $\gcd(p - 1, p - 1) = p - 1 \neq 1$). Why condition 3.8 should be satisfied, is explained in the section on the key recovery by the attacker.
3.5. SETUP IN ELGAMAL

Recovery of Private Key by Attacker

The attacker uses two signatures \((m_i, r_i, s_i)\) and \((m_{i+1}, r_{i+1}, s_{i+1})\) to recover the user’s private key \(x\). First, Eve computes:

\[
r_i^X \pmod{p} \equiv (g^{k_i})^X \\
\equiv g^{k_i \cdot X} \\
\equiv (g^X)^{k_i} \\
\equiv Y^{k_i} \\
\equiv c \pmod{p}.
\]

Recall that \(c\) is an integer in \(\{1, 2, \ldots, p-2\}\). Then, the private key \(x\) can be found by computing:

\[
r_{i+1}^{-1} \left( m_{i+1} - \frac{s_{i+1}}{c} \right) \pmod{p-1}.
\]

(3.9)

The attacker needs the inverse of \(r_{i+1} \pmod{p-1}\). For this inverse to exist, \(\gcd(r_{i+1}, p-1)\) must be equal to one. Recall that, if \(k_{i+1} \equiv c^{-1} \pmod{p-1}\), then:

\[
r_{i+1} \equiv g^{k_{i+1}} \\
\equiv g^{c^{-1}} \pmod{p}.
\]

This implies that:

\[
\gcd(g^{c^{-1}} \pmod{p}, p-1) = 1
\]

must hold, which explains condition 3.8. If \(\gcd(r_{i+1}, p-1) \neq 1\) and \(k_{i+1} \neq c^{-1}\) and Eve knows that she cannot use this pair of signatures to recover the private key.

To show that 3.9 indeed gives the private key \(x\), note that:

\[
s_{i+1} \equiv k_{i+1}^{-1}(m_{i+1} - x \cdot r_{i+1}) \\
\equiv c(m_{i+1} - x \cdot r_{i+1}) \pmod{p-1}.
\]

(3.10)

Using 3.10:

\[
\frac{s_{i+1}}{c} \equiv \frac{c(m_{i+1} - x \cdot r_{i+1})}{c} \\
\equiv (m_{i+1} - x \cdot r_{i+1}) \pmod{p-1}.
\]

(3.11)

Then using 3.11:

\[
m_{i+1} - \frac{s_{i+1}}{c} \equiv m_{i+1} - (m_{i+1} - x \cdot r_{i+1}) \\
\equiv x \cdot r_{i+1} \pmod{p-1}.
\]

(3.12)

Finally, inserting 3.12 in 3.9 gives:

\[
r_{i+1}^{-1} \left( m_{i+1} - \frac{s_{i+1}}{c} \right) \equiv r_{i+1}^{-1} (x \cdot r_{i+1}) \\
\equiv x \pmod{p-1}.
\]

So indeed, 3.9 gives the attacker the correct user’s private key \(x\). Note that this attack has a \((1, 2)\)-leakage scheme, since Eve needs two consecutive signatures to exfiltrate one private key \(x\).

It is important to note that Eve cannot use all pairs of consecutive signatures. The program generates a random \(k_i\) and computes \(k_{i+1}\) using \(k_i\). Next, \(k_{i+2}\) is generated randomly again and \(k_{i+3}\) will be computed using \(k_{i+2}\). This means that if Eve gets the consecutive signatures with \(k_{i+1}\) and \(k_{i+2}\), she cannot use these signatures, since \(k_{i+2}\) does not depend on \(k_{i+1}\).

In example 3.7, a worked-out example of SETUP in ElGamal signature scheme can be found. First, it is shown how Eve’s and Alice’s key pairs are generated and how Alice computes her first signature \((m_1, r_1, s_1)\). Next, this signature is verified by Bob. Then Alice computes the second signature, \((m_2, r_2, s_2)\), which is also verified by Bob. Finally, it is shown how Eve can exfiltrate Alice’s private key \(x\) using both signatures.
Example 3.7: SETUP in the ElGamal signature scheme

The steps are executed as follows:

- **Key pair generation by Eve.** Eve generates her private and public key as follows:
  - First, she chooses her system parameters, let \( p = 23993 \) and \( g = 15765 \), such that \( g \) generates \( \mathbb{Z}_p^* \).
  - Eve chooses \( X = 9237 \) as private key and computes the public key as follows:
    \[
    Y \equiv g^X \equiv 6211 \pmod{23993}.
    \]

- **Key pair generation by Alice.** Alice’s contaminated device generates Alice’s private and public key as follows:
  - First, the program generates the private key \( x \) randomly:
    \[ x = 19423. \]
  - Then, the public key is computed as follows:
    \[
    y \equiv g^x \equiv 15765^{19423} \equiv 14274 \pmod{23993}.
    \]

- **First signature generation by Alice.** The contaminated program computes the signature \((m_1, r_1, s_1)\) as follows:
  - The program generates \( k_1 = 12993 \), such that \( \gcd(k_1, p - 1) = 1 \).
  - Alice wants to use the message “HI” and encodes this as \( m_1 = 0809 \).
  - Then the program computes:
    \[
    r_1 \equiv g^{k_1} \equiv 15765^{12993} \equiv 2400 \pmod{23993}
    \]
    \[
    s_1 \equiv k_1^{-1}(m_1 - x \cdot r_1) \equiv 4873(809 - 19423 \cdot 2400) \equiv 10097 \pmod{23992}.
    \]
  Alice sends \((m_1, r_1, s_1) = (809, 2400, 10097)\) to Bob.

- **First signature verification by Bob.** Bob receives \((809, 2400, 10097)\) and computes:
  \[
  y^{r_1} \cdot r_1^{-s_1} \equiv 14274^{2400} \cdot 2400^{10097} \equiv 1858 \pmod{23993},
  \]
  which is equal to:
  \[
  g^{m_1} \equiv 15765^{809} \equiv 1858 \pmod{23993}.
  \]
  So Bob concludes that the message is indeed from Alice.

- **Second signature generation by Alice.** The contaminated program computes the signature \((m_2, r_2, s_2)\) as follows:
  - The program computes:
    \[
    c \equiv Y^{k_1} \equiv 6211^{12993} \equiv 4413 \pmod{23993}.
    \]
    Since \( \gcd(c, p - 1) = 1 \) and \( \gcd(g^{c^{-1}} \pmod{p}, p - 1) = 1 \):
    \[
    k_2 = c^{-1} = 8541.
    \]
  - Alice wants to use the message “OK” and encodes this as \( m_1 = 1511 \).
  - Then the program computes:
    \[
    r_2 \equiv g^{k_2} \equiv 15765^{20379} \equiv 8541 \pmod{23993}
    \]
    \[
    s_2 \equiv k_2^{-1}(m_2 - x \cdot r_2) \equiv 4413(1511 - 19423 \cdot 8541) \equiv 16578 \pmod{23992}.
    \]
  Alice sends \((m_2, r_2, s_2) = (1511, 8541, 16578)\) to Bob.
3.5. SETUP IN ELGAMAL

- Second signature verification by Bob. Bob receives \((1511, 8541, 16578)\) and computes:
\[
y^2 \cdot r_2^s \equiv 14274^{8541} \cdot 8541^{16578} \equiv 22880 \pmod{23993},
\]
which is equal to:
\[
g^{m_2} \equiv 15765^{1511} \equiv 22880 \pmod{23993}.
\]
So Bob concludes that the message is indeed from Alice.

- Attack by Eve. Eve knows:
\[
(m_1, r_1, s_1) = (809, 2400, 10097) \quad \text{and} \quad (m_2, r_2, s_2) = (1511, 6819, 16578).
\]
First, she computes:
\[
c \equiv r_1^X \pmod{p} \equiv 2400^{9237} \equiv 4413 \pmod{23992}.
\]
Then Eve computes:
\[
x \equiv r_2^{-1}(m_2 - \frac{s_2}{c}) \equiv 6819^{-1}(1511 - \frac{16578 \cdot 8541}{4413}) \equiv 1147(1511 - 16578 \cdot 8541) \equiv 19423 \pmod{23992},
\]
which indeed gives her Alice’s private key.

**Integers** \(k_i\) **and** \(k_{i+1}\)

The recovery of the private key described in the previous section only works if valid values for \(k_i\) and \(k_{i+1}\) are used, so if conditions 3.7 and 3.8 are satisfied. Assume Eve has two consecutive signatures. Then the probability that these signatures are useful to exfiltrate the user’s private key depends on the probability that conditions 3.7 and 3.8 are satisfied (note that these computations are not included in the article by Young and Yung). It is assumed that \(c\) and \(p - 1\) are both random integers and this is used to compute an estimate of the chance of success. Later, a more precise analysis will be considered, where it is taken into account that \(p\) is prime and therefore \(p - 1\) is even (p. 69).

To compute this probability, using that \(c\) and \(p - 1\) are random, positive integers, one needs the following theorem:

**Theorem 3.13.** Let \(a\) and \(b\) be random, positive integers. Then:
\[
P(\gcd(a, b) = 1) = \frac{6}{\pi^2}.
\]

Two different proofs of this theorem can be found in [22]. Assume that \(c\) and \(p - 1\) are random, positive integers, then:
\[
P(\gcd(c, p - 1) = 1) = \frac{6}{\pi^2}.
\]
Note that \(p\) is prime and that \(p\) is often chosen such that \(p - 1\) contains a large prime factor. This means that the probability that \(c\) and \(p - 1\) are relatively prime increases and that the probabilities computed here can be considered to be a lower bound in this case. As mentioned before, on p. 69, another approach is explained and there it is shown that it is indeed wise to choose \(p\) such that \(p - 1\) contains a large prime factor.

Similarly, assume that also \(g^{c^{-1}} \pmod{p}\) is a random, positive integer, then:
\[
P(\gcd(g^{c^{-1}} \pmod{p}, p - 1) = 1) = \frac{6}{\pi^2}.
\]
Combining the two gives:

\[ P(\gcd(c, p - 1) = 1 \text{ and } \gcd(g^{c-1} \mod p, p - 1) = 1) = \left(\frac{6}{\pi^2}\right)^2 \approx 0.37. \tag{3.13} \]

This means that the probability that Eve has useful consecutive signatures, is equal to 0.37, which is low. Note that:

\[ P(\gcd(c, p - 1) \neq 1 \text{ and/or } \gcd(g^{c-1} \mod p, p - 1) \neq 1) = 1 - \left(\frac{6}{\pi^2}\right)^2. \]

Now assume that Eve received 4 pairs of consecutive signatures (so 8 consecutive signatures in total). It is possible that none of the pairs can be used to exfiltrate the private key. Then:

\[ P(\text{all 4 pairs of signatures not useful}) = \left(1 - \left(\frac{6}{\pi^2}\right)^2\right)^4. \]

This means that the probability that at least one signature is useful is given by:

\[ P(\text{at least 1 out of 4 pairs of signatures useful}) = 1 - \left(1 - \left(\frac{6}{\pi^2}\right)^2\right)^4 \approx 0.84. \]

**Algorithm II**

The program can be improved by increasing this probability. In the program, \(k_{i+1}\) is a function of \(k_i\). Similarly, \(k_{i+2}\) can be chosen such that it is a function of \(k_{i+1}\), \(k_{i+3}\) can be chosen such that it is a function of \(k_{i+2}\), and so on. This means that if Eve receives 8 consecutive signatures, that there are 7 possible pairs. Then:

\[ P(\text{at least 1 out of 7 pairs of signatures useful}) = 1 - \left(1 - \left(\frac{6}{\pi^2}\right)^2\right)^7 \approx 0.96. \]

In the original program, \(k_{i+1}\) was generated randomly if \(k_i\) did not satisfy two conditions (conditions 3.7 and 3.8). This means that if Eve has two consecutive signatures, it is possible that for each signature \(k_i\) was generated randomly. These two consecutive signatures could not be used by Eve to exfiltrate the user’s private key. The program can be improved by inserting a loop that keeps regenerating \(k_i\) until both conditions are satisfied. This means that for the next signature, \(k_{i+1}\) will always depend on \(k_i\). Then the program looks as follows:

- First, the program sets \(j = 0\).
- Then, a random integer \(k_i\) is generated randomly from \(\{1, 2, \ldots, p - 2\}\) such that \(\gcd(k_i, p - 1) = 1\). This is the usual condition for the ElGamal signature scheme.
- Next, the integer \(c\) is introduced and defined as follows:

\[ c \equiv Y^{k_i} \mod p. \]

- The following two conditions should be satisfied:

\[ \gcd(c, p - 1) = 1 \tag{3.14} \]

and:

\[ \gcd(g^{c-1} \mod p, p - 1) = 1. \tag{3.15} \]

This gives three possibilities:

- If these conditions are not satisfied, then \(j = j + 1\) and the program generates a new integer \(k_i\). This process continuous at most \(B\) times.
- If these conditions are not satisfied and \(j = B\), then \(k_i\) is used and \(k_{i+1}\) is generated randomly from \(\{1, 2, \ldots, p - 2\}\).
If these conditions are satisfied, the first signature is computed as follows:

\[ r_i \equiv g^{k_i} \pmod{p} \]
\[ s_i \equiv k_i^{-1}(m_i - x \cdot r_i) \pmod{p - 1}, \]

with \( x \) the user’s private key and \( m_i \) the hash of the user’s message. Besides, the integer \( k_{i+1} \) is not generated randomly, but:

\[ k_{i+1} \equiv c^{-1} \pmod{p - 1}. \]

The program outputs the signature \((m_i, r_i, s_i)\).

The subsequent signature is then computed as usual:

\[ r_{i+1} \equiv g^{k_{i+1}} \pmod{p} \]
\[ s_{i+1} \equiv k_{i+1}^{-1}(m_{i+1} - x \cdot r_{i+1}) \pmod{p - 1}. \]

The program outputs the signature \((m_{i+1}, r_{i+1}, s_{i+1})\).

In appendix B.3 (p. 95), a description of the algorithm in pseudocode can be found.

The final step is then to choose \( B \) such that it is very likely that the program generates two consecutive signatures that can be used to exfiltrate the private key. Note that, as before:

\[ P(\gcd(c, p - 1) = 1 \text{ and } \gcd(g^{-1} \pmod{p}, p - 1) = 1) = \left(\frac{6}{\pi^2}\right)^2. \]

Then:

\[ P(\gcd(c, p - 1) \neq 1 \text{ and/or } \gcd(g^{-1} \pmod{p}, p - 1) \neq 1) = 1 - \left(\frac{6}{\pi^2}\right)^2, \]

and:

\[ P(\gcd(c, p - 1) \neq 1 \text{ and/or } \gcd(g^{-1} \pmod{p}, p - 1) \neq 1 \text{ after } B \text{ times}) = \left(1 - \left(\frac{6}{\pi^2}\right)^2\right)^B. \]

Finally this gives:

\[ P(\gcd(c, p - 1) = 1 \text{ and } \gcd(g^{-1} \pmod{p}, p - 1) = 1 \text{ after } \leq B \text{ times}) = 1 - \left(1 - \left(\frac{6}{\pi^2}\right)^2\right)^B. \]

For \( B = 8 \), this gives:

\[ P(\gcd(c, p - 1) = 1 \text{ and } \gcd(g^{-1} \pmod{p}, p - 1) = 1 \text{ after } \leq 8 \text{ times}) \approx 0.98, \]

which is very acceptable.

**Safe Primes**

Previously, the probability that two consecutive signatures can be used by Eve to exfiltrate the user’s private key was computed (p. 67). Here it was assumed that \( p - 1, c \) and \( g^{c-1} \pmod{p} \) were random, positive integers. However, there is another approach to compute this probability.

If \( c \) is smaller than \( p - 1 \), one could use Euler’s totient function (definition 2.24). According to this definition, the number of integers smaller than \( p - 1 \) that are relatively prime to \( p - 1 \), is equal to \( \varphi(p - 1) \). Instead of using that \( c \) and \( p - 1 \) are two random integers, one could keep \( p - 1 \) fixed and then compute the probability that \( c \) is smaller than \( p - 1 \) and that it is relatively prime to \( p - 1 \). Note that \( c \in \{1, 2, \ldots, p - 1\} \). Then:

\[ P(\gcd(c, p - 1) = 1) = \frac{\varphi(p - 1)}{p - 1}. \]
The same approach for \( g^{c-1} \pmod{p} \) and \( p-1 \) gives (assuming that \( g^{c-1} \pmod{p} \) can take any value in \( \{1,2,\ldots,p-1\} \)):
\[
P(\gcd(g^{c-1} \pmod{p}, p-1) = 1) = \frac{\varphi(p-1)}{p-1}.
\]
Combining the two gives:
\[
P(\gcd(c,p-1) = 1 \text{ and } \gcd(g^{c-1} \pmod{p}, p-1) = 1) = \left( \frac{\varphi(p-1)}{p-1} \right)^2. \tag{3.16}
\]
This implies that it depends on the choice of prime \( p \) whether two consecutive signatures will be useful. For example, if \( p = 359 \), this probability is equal to 0.25, but for \( p = 43891 \), this probability is equal to 0.04 (both rounded to two decimal digits). Note that since \( p \) is generated first and then \( p-1 \) is considered to remain fixed, \( c \) and \( g^{c-1} \pmod{p} \) are now bounded. This implies that the probabilities computed here will be lower than the probability in 3.13, where \( p-1 \), \( c \) and \( g^{c-1} \pmod{p} \) were all considered to be random, positive integers without an upper bound. Since \( c \) and \( g^{c-1} \pmod{p} \) are bounded now, there are simply less integers that are relatively prime to \( p-1 \).

In order to get a high probability in 3.16, \( \varphi(p-1) \) should be as large as possible. This implies that \( p-1 \) should have a large prime factor, because of the following. Recall from theorem 2.28 that if \( n \) is a product of two different prime numbers \( p \) and \( q \), then \( \varphi(n) = (p-1)(q-1) \). Using this, \( \varphi(p-1) \) is as large as possible if \( p-1 \) is a product of two primes, with one of these primes very large. The largest prime factor \( p-1 \) could have, is prime \( q \) such that \( p-1 = 2q \). This means that \( p = 2q+1 \) and therefore \( \varphi(p-1) = \varphi(2q) = q-1 \). A prime that satisfies this equation is a so-called safe prime.

**Definition 3.14.** A prime number \( p \) is called a safe prime if \( p = 2q+1 \) with \( q \) prime.

In appendix D, all safe primes \(< 2000\) are listed, together with the corresponding probabilities in 3.16 and the largest prime factor of \( p-1 \). As can be seen here, the larger the safe prime, the closer the probability gets to 0.25. This is because of the following. Note that, with \( p = 2q+1 \):
\[
\frac{\varphi(p-1)}{p-1} = \frac{q-1}{2q}.
\]
Then:
\[
\lim_{q \to \infty} \frac{q-1}{2q} = \frac{1}{2},
\]
using L'Hôpital's rule. Finally:
\[
\lim_{p \to \infty} \left( P(\gcd(c,p-1) = 1 \text{ and } \gcd(g^{c-1} \pmod{p}, p-1) = 1) \right) = \lim_{p \to \infty} \left( \frac{\varphi(p-1)}{p-1} \right)^2
= \lim_{q \to \infty} \left( \frac{q-1}{2q} \right)^2
= \left( \frac{1}{2} \right)^2
= \frac{1}{4}.
\]
Note that the limit in this case indicates that how larger the prime \( p \) (and thus \( q \)) gets, the more the probability approaches 1/4. To conclude, in order to obtain a high probability that Eve can use two consecutive signatures, it is recommended to choose \( p \) such that it is a safe prime.

Although the probability approaches 0.25 for large safe primes, it will never be equal to 0.25 for large primes. When computing the probability for all primes, it turned out that there were exactly five primes that did have a corresponding probability of exactly 0.25, namely 3,5,17,257 and 65537. These are actually a special kind of primes: they are so-called Fermat primes.

**Definition 3.15.** A prime number \( p \) is called a Fermat prime if \( p = 2^{2^n} + 1 \) for some integer \( n > 0 \).

This means that Fermat primes would give a (negligible) higher probability than safe primes. However, the five primes listed are the only known Fermat primes. The primes that are used for the ElGamal signature scheme are usually much larger (1024 bits). Unfortunately, Fermat primes are not useful for this purpose.
3.6 SETUP in Diffie-Hellman

Application: Simmons’ Prisoners’ Problem

In the original prisoners’ problem described by Simmons (see section 3.1.1), both Alice and Bob are in prison. In order to be able to communicate subliminally, Alice and Bob need to know each other’s public keys before they were put in prison. Young and Yung describe an application of SETUP in ElGamal signature scheme for a slightly different prisoners’ problem. Assume that Alice is in prison and Bob, who is not in prison, wants to help Alice to escape. Instead of agreeing upon a private key $x$ beforehand, they can use SETUP in ElGamal signature scheme to communicate this key.

In practice, this works as follows. Both Alice and Bob know about the SETUP attack. Alice looks up Bob’s public key and uses that key to create a contaminated cryptosystem. Next, she creates two consecutive signatures using the SETUP algorithm and sends these two signatures to Bob, so that he can use the signatures to recover Alice’s private key. Finally, Alice and Bob can use this shared private key $x$ for a subliminal channel. As mentioned before, a huge advantage of this application is that Alice and Bob do not need to agree upon a key beforehand.

Summary

In the SETUP attack for the ElGamal signature scheme, Eve needs (at least) two consecutive signatures to be able to exfiltrate the user’s private key $x$. The first signature is computed as usual, but the second signature is computed differently. Usually, the integer $k_{i+1}$ is generated randomly, but in the attack it depends on the previously randomly generated integer $k_i$. In order to improve the algorithm, prime $p$ should be chosen such that it is a safe prime. Since Eve needs two consecutive signatures, this attack has a $(1,2)$-leakage scheme. Note that if not only $k_{i+1}$ depends on $k_i$, but also $k_{i+2}$ depends on $k_{i+1}$, $k_{i+3}$ on $k_{i+2}$, and so on, this attack has actually a $(m,m+1)$-leakage scheme, since each possible pair could be used to exfiltrate the private key.

Recall that:

\[ c \equiv Y^{k_i} \pmod{p} \]

and:

\[ k_{i+1} = c^{-1}, \]

with $c^{-1}$ the modular inverse of $c$ modulo $p - 1$. This implies that:

\[ Y \equiv (k_{i+1}^{-1})^{k_i^{-1}}. \]

This means that the (fixed) attacker’s public key $Y$ can be recovered using $k_i$ and $k_{i+1}$. Alice knows the public parameters $p$ and $g$, her signatures and her own private key $x$. This means that she can recover the values of $k$ each time a signature is generated. Assume that Alice manages to recover $k_i$ and $k_{i+1}$. Then she would also be able to compute $Y$. If she does this for several random pairs $(k_i, k_{i+1})$, she would notice that each time the computation gives the same outcome, namely $Y$. If $k_i$ and $k_{i+1}$ would both be random, this would definitely not be the case. This means that Alice could easily distinguish between the output of the normal system and the output of the contaminated system within polynomial time without the knowledge of the internals of the system if she has access to $k_i$ and $k_{i+1}$. It is unclear how to transfer the notion of weak and regular SETUP to signatures. If weak implies that the user gets access to $k_i$ and $k_{i+1}$, then this attack constitutes a weak SETUP attack. However, as mentioned before, this attack could be very useful to agree upon a shared private key for subliminal communication.

3.6 SETUP in Diffie-Hellman

In the previous versions of SETUP, the main concern was to produce an output that the user would not find suspicious: the output of the normal system and the contaminated system should by polynomially indistinguishable for the user. For this purpose, SETUP created a subliminal channel that could be used to embed private information in the normal output of the device. For SETUP in Diffie-Hellman key exchange, another approach is used, namely a SETUP attack on the discrete log.

Recall that in the Diffie-Hellman key exchange protocol $p$ and $g$ are the system parameters with $p$ a prime and $g$ a generator of $\mathbb{Z}_p$. Both Alice and Bob generate a random private key from $\{1, 2, \ldots, p-1\}$, $x_A$ and $x_B$ respectively. The public keys are generated as follows: $y_A \equiv g^{x_A} \pmod{p}$ and $y_B \equiv g^{x_B} \pmod{p}$. 
Finally, they both can compute the shared private key, denoted by $k$, using each other’s public key:

$$k \equiv y_B^x \equiv y_A^x \pmod{p}.$$

Note that in this example application it is assumed that for each connection Alice generates a fresh key pair. So if Alice also wants to communicate confidentially with Carol, she uses another private key $x_A$. In the attack described below, the attacker needs two different key exchanges involving Alice, so, for example, the key exchange between Alice and Bob and the key exchange between Alice and Carol.

Usually, black-box algorithms do not give the user access to the private key $x$. The argument for this is that if the user does not know the private key, then the key cannot be stored somewhere where it could be stolen by someone. In this SETUP attack, it is also assumed that the user does not know the private key.

### 3.6.1 SETUP in Diffie-Hellman Key Exchange

In the attack described here, it is assumed that Alice uses a device that generates her private and public key $x_A$ and $y_A$. From now on, the subscript $A$ will be omitted. In this attack, $x_{\{1,2\}}$ and $y_{\{1,2\}}$ denote resp. Alice’s private and public keys.

**Algorithm**

Assume that Alice and Bob want to communicate confidentially and that they each use a Diffie-Hellman device to generate their own private and public key. Assume that the parameters $p$ and $g$ remain fixed. The device only outputs a public key, the private key is kept secret, also for the user of the device. Assume that Alice’s device is contaminated. The attacker, Eve, designed Alice’s device and inserted the following values: Eve’s public key $Y$, fixed integers $a, b$ and $W$ and a cryptographically strong hash function $H$. Furthermore, assume that $H$ only generates hash values in $\{1, 2, \ldots, p-1\}$. As described by Young and Yung, Alice’s device performs the following steps [37]:

- The first time the device outputs a public key, it performs the following steps:
  - First, the device generates a random integer $x_1$, such that $x_1 \in \{1, 2, \ldots, p-1\}$. This is Alice’s private key.
  - Next, the public key, $y_1$, is computed:
    $$y_1 \equiv g^{x_1} \pmod{p}.$$  \hfill (3.17)
  - The public key is the output of the device. Furthermore, the value of $x_1$ is stored for the next time the device is used (and only for the next time).

- The second time the device is used, the private key $x_2$ is not generated randomly, but depends on several other integers:
  - First, integer $t$ is chosen at random from $\{0,1\}$.
  - Next, $z$ is computed as follows:
    $$z \equiv g^{x_1-W\cdot t}Y^{-a}\cdot x_1^{-b} \pmod{p},$$  \hfill (3.18)
    with $a, b$ and $W$ fixed integers.
  - Then, the cryptographic hash function $H$ is used to compute Alice’s private key, $x_2$:
    $$x_2 = H(z).$$ \hfill (3.19)
  - This private key is used to compute Alice’s public key, $y_2$:
    $$y_2 \equiv g^{x_2} \pmod{p}.$$ \hfill (3.20)

In figure 3.8 normal Diffie-Hellman key pair generation is compared to Diffie-Hellman key pair generation with SETUP. In appendix C.1 (p. 97), a description of the algorithm in pseudocode can be found. Why integers $a, b$ and $W$ are used, is explained in the next section.
Diffie-Hellman key pair generation | Diffie-Hellman key pair generation with SETUP
---|---
\( p, g \) | \( p, g \)
\( x_1 \in \{1, 2, \ldots, p - 1\} \) | \( x_1 \in \{1, 2, \ldots, p - 1\} \)
\( y_1 \equiv g^{x_1} \pmod{p} \) | \( y_1 \equiv g^{x_1} \pmod{p} \)
\( x_2 \in \{1, 2, \ldots, p - 1\} \) | \( t \in \{0, 1\} \)
\( y_2 \equiv g^{x_2} \pmod{p} \) | \( z \equiv g^{x_1-W \cdot Y^{-a \cdot x_1-b}} \pmod{p} \)

\( x_2 = H(z) \)
\( y_2 \equiv g^{x_2} \pmod{p} \)

Figure 3.8: Diffie-Hellman key pair generation with SETUP mechanism.

As soon as Alice’s device outputs her public key \( y_1 \) (and Bob’s device has computed his public key), Alice and Bob exchange their public keys. Now they both can compute the shared secret key that will be used to decrypt each other’s messages. Maybe Alice and Bob like to change their shared secret key or maybe Alice likes to communicate securely with someone else, say Carol. Without loss of generality, assume the latter. Then the device uses the previously generated private key to compute a new private key (\( x_1 \) is incorporated in \( x_2 \)). Next, this new private key is used to compute the second public key to communicate with Carol. The attacker can use both public keys to recover Alice’s second private key. How she does this, is explained from p. 74 onwards.

**Fixed Integers \( a, b \) and \( W \)**

In the algorithm, there appear three fixed integers: \( a, b \) and \( W \). These are inserted as a precaution in the (exceptional) case that Alice finds out that \( H \) is invertible. Assume that Alice uses the device as a black-box cryptosystem and that she suspects that her system is contaminated. She knows how the SETUP attack works and uses this knowledge to determine whether her black-box system is contaminated or not. Furthermore, assume that \( a, b \) and \( W \) are not used (or \( a = 1 \) and \( b = 0 \)), then:

\[
z \equiv g^{x_1} Y^{-x_1} \pmod{p}.
\] (3.21)

First, Alice uses the device twice to generate a private and public key. This means that she has two private keys, \( x_1 \) and \( x_2 \). Since she knows the inverse of \( H \), she can compute \( z \) as follows:

\[
z = H^{-1}(x_2).
\]

Next, she computes the following fraction:

\[
f \equiv \frac{g^{x_1}}{z} \equiv \frac{g^{x_1}}{g^{x_1} Y^{-x_1}} \equiv Y^{x_1} \pmod{p}.
\] (3.22)

Alice knows that \( Y \equiv g^X \pmod{p} \), but she does not know the values for \( Y \) or \( X \). Obviously, \( X \) is odd or even. This gives the following possibilities (see definition 2.29 on p. 13 for the definition of quadratic residues):

- If \( X \) is odd, the following two possibilities need the be considered: \( x_2 \) is even or odd.
  - If \( x_2 \) is even, then equation 3.22 can be rewritten as follows:
    \[
f \equiv Y^{x_1} \equiv Y^{2x_1} \equiv (Y^{x_1})^2 \pmod{p},
    \]
    with \( x_1 = 2x_1' \). This implies that \( f \) is a quadratic residue modulo \( p \).
  - If \( x_2 \) is odd, then \( f \) will be a quadratic non-residue modulo \( p \).

- If \( X \) is even, then 3.22 can be rewritten as follows:
  \[
f \equiv Y^{x_1} \equiv (g^X)^{x_1} \equiv (g^{2X'})^{x_1} \equiv (g^{X'})^{2x_1} \equiv (g^{X'})^2 \pmod{p},
  \]
  with \( X = 2X' \). This implies that \( f \) is a quadratic residue modulo \( p \), no matter what \( x_2 \) is.
The four possibilities are summarised in figure 3.9.

<table>
<thead>
<tr>
<th>$x_2$ odd</th>
<th>$X$ odd</th>
<th>$X$ even</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>quadratic non-residue</td>
<td>quadratic residue</td>
</tr>
<tr>
<td>$x_2$ even</td>
<td>quadratic residue</td>
<td>quadratic residue</td>
</tr>
</tbody>
</table>

Figure 3.9: Fraction $f$ quadratic residue or quadratic non-residue for odd or even $X$ and $x_2$.

This figure clearly shows that if Alice’s system is contaminated, the probability that $f$ is a quadratic non-residue is smaller than the probability that $f$ is a quadratic residue, respectively 0.25 and 0.75. If the system is not contaminated, these probabilities should be equal (0.5 and 0.5). This means that Alice can determine whether her system is contaminated or not as follows. She can compute as many private keys as she wants. If she gathers all pairs $(x_1, x_2)$ and computes $f$ for these pairs, she can check whether the number of quadratic residues is approximately equal to the number of quadratic non-residues. If there are significantly more quadratic residues than quadratic non-residues, Alice can conclude that her system is contaminated.

To conclude, if the integers $a$, $b$ and $W$ are not used (or $a = 1$, $b = 0$ and $W = 0$), then Alice can use the inverse of $H$ to compute the fraction:

$$f \equiv g^{y_1} \equiv g^{y_1 z} (\mod p).$$

Next, she can check whether this gives a quadratic residue or not. This means that the integers $a$, $b$ and $W$ are included in the algorithm to prevent that Alice can distinguish between outputs of the normal system and outputs of the contaminated system.

**Recovery of Private Key by Attacker**

With Alice’s public keys $y_1$ and $y_2$, Eve can recover the private key $x_2$. Note that the parameters $p$ and $g$ are public and that Eve knows the integers $a$, $b$ and $W$ (since she designed Alice’s device and chose these fixed integers herself). Furthermore, Eve is the only one that knows her own private key $X$. She performs the following steps:

- First, Eve computes:
  $$r \equiv y_1^a g^b (\mod p).$$
  (3.23)

- Then, she uses $r$, Alice’s public key $y_1$ and her own private key $X$ to compute:
  $$z_1 \equiv y_1 r^X (\mod p).$$
  (3.24)

- If:
  $$y_2 \equiv g^{H(z_1)} (\mod p),$$
  (3.25)
  then $H(z_1) = x_2$ and Eve is done.

- If the previous step failed, Eve computes:
  $$z_2 \equiv \frac{z_1}{g^W}.$$  
  (3.26)

  In this case, the following equation should hold:
  $$y_2 \equiv g^{H(z_2)}.$$  
  (3.27)

And then $H(z_2) = x_2$.

Since Eve can easily obtain Carol’s public key, she can use Alice’s second private key and Carol’s public key to compute the shared secret key for the connection between Alice and Carol. Finally, she can use this key to decrypt messages that are sent between Alice and Carol.
To show that this indeed gives the correct private key \( x_2 \), equations 3.17 for \( y_1 \) and 3.23 for \( r \) are inserted into equation 3.24. Furthermore, use that \( Y \equiv g^X \pmod{p} \):

\[
\begin{align*}
z_1 & \equiv \frac{y_1}{X} \\
& \equiv g^{x_1} \\
& \equiv (g^{a} g^{b})^{x_1} \\
& \equiv ((g^{a})^{x_1} g^{b})^{x_1} \\
& \equiv \frac{g^{x_1}}{(g^{x_1} + b)^{x_1}} \\
& \equiv g^{x_1} (g^{a} - x_1 - b)^{-X} \\
& \equiv g^{x_1} (g^{a} - x_1 - b)^{-X} \\
& \equiv g^{x_1} Y^{a-x_1-b} \pmod{p}, \\
\end{align*}
\]

This equals equation 3.18 for \( t = 0 \). So if the device used \( t = 0 \), then \( z_1 = z \) and therefore \( x_2 = H(z) = H(z_1) \) (equation 3.20). Note that:

\[
y_2 \equiv g^{x_2} \equiv g^{H(z)} \equiv g^{H(z_1)} \pmod{p},
\]

which explains condition 3.25. If \( t = 1 \) is used by the device, then equation 3.24 (or equation 3.28) should be inserted into equation 3.26:

\[
\begin{align*}
z_2 & \equiv \frac{z_1}{g^{W}} \pmod{p} \\
& \equiv g^{x_2} Y^{a-x_1-b} \pmod{p} \\
& \equiv g^{x_1} Y^{a-x_1-b} , g^{-W} \pmod{p} \\
& \equiv g^{x_1} Y^{a-x_1-b} \pmod{p},
\end{align*}
\]

which indeed equals 3.18 for \( t = 1 \). This means that if the device used \( t = 1 \), then \( z_2 = z \) and \( x_2 = H(z) = H(z_2) \) (equation 3.20). Then also:

\[
y_2 \equiv g^{x_2} \equiv g^{H(z)} \equiv g^{H(z_2)} \pmod{p}.
\]

This shows that discrete log attack against Diffie-Hellman has indeed a \( (1,2) \)-leakage scheme: Eve needs two public keys to be able to exfiltrate one private key.

In example 3.8, a worked-out example of SETUP in Diffie-Hellman key exchange can be found. The following aspects are shown. First, Eve’s key pair is generated, followed by the first key pair generation by Alice \((x_1, y_1)\). Alice uses these to construct a secure channel using Diffie-Hellman key exchange with Bob. Bob’s key pair is generated as well and both Alice and Bob compute the shared secret key. Next, Alice generates a second key pair \((x_2, y_2)\), for a secure connection with Carol. Also Carol computes a key pair and Alice and Carol both compute their shared secret key. Eve uses both Alice’s public keys to exfiltrate Alice’s second private key \( x_2 \). Finally, using Carol’s public key, Eve can compute the shared secret key of Alice and Carol.

**Example 3.8: SETUP in Diffie-Hellman key exchange**

The steps are executed as follows:

- **Key generation by Eve.** Eve generates the her private and public key as follows:
  - First, she chooses her system parameters, let:

\[
p = 23993 \quad \text{and} \quad g = 15765,
\]

such that \( g \) generates \( \mathbb{Z}^*_p \).
– Eve chooses: \( X = 9237 \)

as private key and computes the public key as follows:

\[ Y \equiv g^X \equiv 6211 \pmod{23993}. \]

• **First key pair generation by Alice.** Alice’s device generates Alice’s first private and public key as follows:
  – First, the device generates Alice’s private key \( x_1 \) randomly:
    \[ x_1 = 19423. \]
  – Then, the public key is computed as follows:
    \[ y_1 \equiv g^{x_1} \equiv 15765^{19423} \equiv 14274 \pmod{23993}. \]

Alice sends her public key to Bob.

• **Key pair generation by Bob.** Bob’s device generates Bob’s private and public key as follows:
  – First, the device generates his private key \( x_B \) randomly:
    \[ x_B = 7021. \]
  – Then, the public key is computed as follows:
    \[ y_B \equiv g^{x_B} \equiv 15765^{7021} \equiv 5619 \pmod{23993}. \]

Bob sends his public key to Alice.

• **Shared secret key.** Now both Alice and Bob can compute the shared secret key. Alice computes:

\[ s_1 \equiv y_B^{x_1} \equiv 5619^{19423} \equiv 23990 \pmod{23993}. \]

Bob computes:

\[ s_1 \equiv y_1^{x_B} \equiv 14274^{7021} \equiv 23990 \pmod{23993}. \]

• **Second key pair generation by Alice.** Assume that \( a = 1, b = 4 \) and \( W = 2437 \). Alice’s device generates Alice’s second private and public key as follows:
  – First, the program chooses \( t = 0 \).
  – Then, the program computes:
    \[
    z \equiv g^{x_1-W \cdot x_B-a \cdot x_1-b} \\
    \equiv 15765^{19423-2437 \cdot 6211-1^{19423-4}} \\
    \equiv 15765^{19423-6211-1^{19427}} \\
    \equiv 14274 \cdot 18280 \\
    \equiv 4845 \pmod{23993}.
    \]

  – The second private key is computed using function \( H \). Assume that:
    \[ x_2 = H(z) = H(4845) = 7003. \]
  – Finally, the program computes Alice’s second public key:
    \[ y_2 \equiv g^{x_2} \equiv 15765^{7003} \equiv 21811 \pmod{23993}. \]

Alice sends her public key to Carol.

• **Key pair generation by Carol.** Carol’s device generates Carol’s private and public key as follows:
3.6. SETUP IN DIFFIE-HELLMAN

First, the device generates his private key \( x_C \) randomly:

\[
x_C = 12932.
\]

Then, the public key is computed as follows:

\[
y_C \equiv g^{x_C} \equiv 15765^{12932} \equiv 8968 \pmod{23993}.
\]

Carol sends her public key to Alice.

**Shared secret key.** Now both Alice and Carol can compute the shared secret key. Alice computes:

\[
s_2 \equiv y_C^{x_2} \equiv 8968^{7003} \equiv 18287 \pmod{23993}.
\]

Carol computes:

\[
s_2 \equiv y_C^{x_2} \equiv 21811^{8968} \equiv 18287 \pmod{23993}.
\]

**Attack by Eve.** Eve knows Alice’s public keys \( y_1 \) and \( y_2 \). She recovers Alice’s second private key \( x_2 \) as follows:

1. First, Eve computes:

\[
r \equiv y_1^a \cdot g^b \equiv 14274^1 \cdot 15765^4 \equiv 13490 \pmod{23993}.
\]

2. Then, she computes:

\[
z_1 \equiv \frac{y_1}{r^x} \equiv \frac{14274}{13490^{2397}} \equiv 14274 \cdot 18280 \equiv 4845 \pmod{23993}.
\]

3. Using \( z_1 \), Eve computes:

\[
g^{H(z_1)} \equiv 15765^{H(4845)} \equiv 15765^{7003} \equiv 21811 \pmod{23993}.
\]

Since this is equal to \( y_2 \), Eve knows that \( z_1 = z \). Then:

\[
x_2 = H(z) = H(z_1) = H(4845) = 7003.
\]

Finally, Eve can use Carol’s public key \( y_C \) to compute the shared secret key of Alice and Carol:

\[
s_2 \equiv y_C^{x_2} \equiv 8968^{7003} \equiv 18287 \pmod{23993}.
\]

Eve can use this shared secret key to decrypt messages that are sent between Alice and Carol.

---

**Use of ElGamal Encryption**

This attack uses the same principle as in ElGamal encryption. Recall that in ElGamal encryption, Alice computes:

\[
c_1 \equiv g^k \pmod{p} \quad \text{and} \quad c_2 \equiv m \cdot y^k \pmod{p},
\]

with \( c_2 \) the ElGamal encryption of message \( m \). Then Bob computes:

\[
\frac{c_2}{c_1^z} = \frac{m \cdot y^k}{(g^k)^z} = \frac{m \cdot (g^x)^k}{g^{z \cdot x}} = \frac{m \cdot g^{x \cdot k}}{g^{z \cdot k}} = m.
\]

In the attack, these components are computed in a different way. At first, the ElGamal encryption of “something” is computed, followed by the computation of that “something”, namely \( y_1 \) and \( z \) respectively. In normal ElGamal encryption, this would mean that first \( c_2 \) is computed, followed by the computation of the message \( m \). So \( m \) is computed such that the ElGamal encryption of \( m \) is equal to \( c_2 \). In this attack this means that first \( y_1 \) is computed, followed by the computation of \( z \), such that the ElGamal encryption of \( z \) equals \( y_1 \).
In figure 3.10 is shown which ElGamal variable corresponds to which variable in the Diffie-Hellman key pair generation with SETUP.

<table>
<thead>
<tr>
<th>Variables in ElGamal encryption</th>
<th>Equivalent variables in SETUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p, g$</td>
<td>$p, g$</td>
</tr>
<tr>
<td>$k \in {1, 2, \ldots, p - 1}$</td>
<td>$x_1 \in {1, 2, \ldots, p - 1}$</td>
</tr>
<tr>
<td>$x \in {1, 2, \ldots, p - 1}$</td>
<td>$X \in {1, 2, \ldots, p - 1}$</td>
</tr>
<tr>
<td>$y \equiv g^x \pmod{p}$</td>
<td>$Y \equiv g^X \pmod{p}$</td>
</tr>
<tr>
<td>$m \in {0, 1, \ldots, p - 1}$</td>
<td>$z \equiv g^{x_1-W} \cdot Y^{-a} \cdot x_1^{-b} \pmod{p}$</td>
</tr>
<tr>
<td>$c_1 \equiv g^k \pmod{p}$</td>
<td>$r \equiv x_1^2 g^b \pmod{p}$</td>
</tr>
<tr>
<td>$c_2 \equiv m \cdot y^k \pmod{p}$</td>
<td>$y_1 \equiv g^{x_1} \pmod{p}$</td>
</tr>
<tr>
<td>$m = \frac{c_2}{c_1}$</td>
<td>$z = \frac{y_1}{r^X}$</td>
</tr>
</tbody>
</table>

Figure 3.10: “ElGamal variables” in Diffie-Hellman key pair generation with SETUP mechanism.

Note that, for example, $y_1$ is the first public key of the user, but also operates as the ElGamal encryption $c_2$. The variables are chosen such that the attacker can recover the message $z$ using ElGamal decryption.

If the system used $t = 0$, then:

$$z \equiv g^{x_1} \cdot Y^{-a} \cdot x_1^{-b} \pmod{p}.$$  

Using ElGamal decryption and equation 3.28:

$$\frac{c_2}{c_1^2} \rightarrow \frac{y_1}{r^X} \equiv z_1 \equiv z \pmod{p}.$$  

So for $t = 0$, Eve can recover the message $z$ using ElGamal decryption.

If the system used $t = 1$, then:

$$z \equiv g^{x_1-W} \cdot Y^{-a} \cdot x_1^{-b} \pmod{p}.$$  

Note that this equals $g^{-W} \cdot z_1$. In this case, the message $z$ can be recovered by computing:

$$\frac{c_2}{c_1} \cdot g^{-W} \rightarrow \frac{y_1}{r^X} \cdot g^{-W} \equiv z_2 \equiv z \pmod{p}.$$  

So also for $t = 1$, Eve can recover the message $z$ using ElGamal decryption (with the extra factor $g^{-W}$).

In both cases, the final step is then to compute the hash value of $z$ to get the private key $x_2$.

**Security**

There are two issues that need to be considered involving the security of the system. First of all, the user of the system should not get suspicious from the output of the system. Secondly, as soon as the user suspects that the system is contaminated, it should not be able to recover the private key $x_2$ using the internals of the system. These two cases will be considered here.

In normal Diffie-Hellman key exchange, private keys $x_1$ and $x_2$ would both be generated randomly from $\{1, 2, \ldots, p - 1\}$. In the discrete log attack, $x_2$ is not generated randomly, but depends on several other inputs. However, in order to prevent that anyone notices that $x_2$ is not generated randomly, $x_2$ should be able to take on all values in $\{1, 2, \ldots, p - 1\}$. Since $x_2 = H(z)$, this implies that $z$ should be uniformly distributed in $\mathbb{Z}_p^*$ and given this condition, that the output of $H$ should be uniformly distributed. First it is shown that $z$ is uniformly distributed. Young and Yung call $z$ a “*hidden field element* with respect to $Y^*$” [37]. Note that $z \in \mathbb{Z}_p^*$. As shown in the previous section, this element can only be recovered by someone who knows the corresponding private key $X$.  

78
In order to be able to prove that \( z \) is indeed uniformly distributed in \( \mathbb{Z}_p \), Young and Yung assume that:

\[
\begin{align*}
g_1 &\equiv g^{-X-b-W} \pmod{p}, \quad g_2 \equiv g^{-X-b} \pmod{p} \quad \text{and} \quad g_3 \equiv g^{1-X-a} \pmod{p}
\end{align*}
\]

are all generators of \( \mathbb{Z}_p^* \). Obviously, \( g_1, g_2 \) and \( g_3 \) are not generators for each possible value of \( W, a \) and \( b \). Furthermore, \( W, a \) and \( b \) are fixed values, so as soon as Eve implemented these values and Alice has the device, they cannot be changed anymore. The only value that could influence the probability that \( g_1, g_2 \) and \( g_3 \) are all generators, is the prime \( p \). Prime \( p \) should be chosen such that the cyclic group \( \mathbb{Z}_p^* \) contains many generators, such that the probability that \( g_1, g_2 \) and \( g_3 \) all generate the group increases.

Recall from theorem 2.27 that the number of generators of a cyclic group of order \( n - 1 \), with \( n \) prime, is equal to \( \varphi(n-1) \). This implies that prime \( p \) should be chosen such that \( \varphi(p-1) \) is relatively large; these are the so-called safe primes (definition 3.14). This means that it is more likely that \( g_1, g_2 \) and \( g_3 \) are generators if \( p \) is a safe prime (so \( p = 2q + 1 \), with \( q \) prime).

**Theorem 3.16.** Assume that \( g_1, g_2 \) and \( g_3 \) are generators of \( \mathbb{Z}_p^* \). Then the hidden field element \( z \) is uniformly distributed in \( \mathbb{Z}_p^* \).

**Proof.** Recall from equation 3.18:

\[
z \equiv g^{x_1-W^tY^{-a}X_1-b} \pmod{p}.
\]

This can be rewritten as follows, using that \( Y \equiv g^X \pmod{p} \):

\[
z \equiv g^{x_1-W^tY^{-a}X_1-b} \\
\equiv g^{x_1-W^t(g^X)^{-a}X_1-b} \\
\equiv g^{x_1-W^t}g^{-Xa}X_1-X^b \\
\equiv g^{x_1}g^{-W^t}g^{-Xa}X_1g^{-X^b} \\
\equiv g^{-Xb-W^t}g^{x_1}XaX_1 \\
\equiv g^{-Xb-W^t}g^{(1-Xa)X_1} \\
\equiv g_1 \cdot g_3^{x_1} \pmod{p},
\]

(3.29)

with \( i \in \{1, 2\} \). Note that if \( t = 0 \), then \( i = 2 \) and if \( t = 1 \), then \( i = 1 \). Since \( g_1, g_2 \) and \( g_3 \) are generators modulo \( p \):

\[
g_i \equiv g_i^u \pmod{p},
\]

for some integer \( u \in \{1, 2, \ldots, p-1\} \). Substituting this in equation 3.29 gives:

\[
z \equiv g_3^{u} \cdot g_3^{x_1} \pmod{p} \\
\equiv g_3^{u+x_1}.
\]

Finally, since \( x_1 \) is chosen randomly from \( \{1, 2, \ldots, p-1\} \), \( z \) must be uniformly distributed in \( \mathbb{Z}_p \). \( \square \)

This shows that \( z \) is indeed uniformly distributed. To maintain this uniform distribution, function \( H \) should be chosen such that the output of \( H \) is also uniformly distributed. Note that is actually only necessary that \( g_3 \) is a generator and that \( g_1 \) and \( g_2 \) can be written as a power of \( g_3 \).

The second issue is that it must be hard for the user (or someone else) to recover the private key \( x_2 \), even though the user has access to the device. This means that the user knows which steps are taken in the algorithm and knows the values of the attacker’s public key \( Y \) and the fixed integers \( a, b \) and \( W \). However, it is assumed here that the user does not know the private keys \( x_1 \) and \( x_2 \). The device only outputs the public keys \( y_1 \) and \( y_2 \) and since \( x_1 \) is chosen randomly, it is not stored in the device permanently just as the fixed integers \( a, b \) and \( W \). Young and Yung claim that the system is secure if and only if the computational Diffie-Hellman problem is secure (see definition 2.40, p. 23).

**Theorem 3.17.** The SETUP attack on Diffie-Hellman key exchange is secure if and only if the computational Diffie-Hellman problem is secure.

**Proof.** The proof consists of two parts:
CHAPTER 3. KLEPTOGRAPHY

⇒ Assume that there exists an oracle $A$ that can break the SETUP attack on Diffie-Hellman key exchange. This means that given the attacker’s public key $Y$ and the user’s public key $y_1$, the oracle outputs $z_1$ for $t = 0$ and $z_2$ for $t = 1$. This is denoted by:

$$A(Y, y_1) = (z_1, z_2).$$

This means that someone can use the oracle to get $z_1$ and $z_2$, which only need to be inserted in $H$ to get $x_2$. Note that:

$$(z_1, z_2) = (g^{x_1} Y^{-a - x_1 - b}, g^{x_1 - W} Y^{-a - x_1 - b}),$$

and that $y_1 \equiv g^{x_1} \pmod{p}$. Inserting this gives:

$$A(Y, g^{x_1}) = (g^{x_1} Y^{-a - x_1 - b}, g^{x_1 - W} Y^{-a - x_1 - b}).$$

This oracle can be used to solve the computational Diffie-Hellman problem as follows. Given $g^x, g^y \in \langle g \rangle$, one wants to find $g^{x \cdot y}$. First, $g^x$ and $g^y$ are inserted in the oracle. This gives the following output (note the reverse order of $g^x$ and $g^y$):

$$A(g^y, g^x) = (g^x (g^y)^{-a - x - b}, g^{x - W} (g^y)^{-a - x - b}).$$

The first part of the output can be used to compute $g^{x \cdot y}$. This first part can be rewritten as follows:

$$g^x (g^y)^{-a - x - b} = g^x g^{-(x \cdot y) - a - y - b} = g^x (g^{x \cdot y})^{-a} (g^y)^{-b}.$$

Then, let:

$$f = \frac{g^x (g^y)^{-a - x - b}}{g^x (g^y)^{-b}} = \frac{g^x (g^{x \cdot y})^{-a} (g^y)^{-b}}{g^x (g^y)^{-b}} = (g^{x \cdot y})^{-a}.$$

Finally:

$$g^{x \cdot y} = f^{-1}.$$

This implies that if the SETUP attack on Diffie-Hellman key exchange can be broken, also the computational Diffie-Hellman problem can be solved.

⇐ Assume that there exists an oracle $B$ that can solve the computational Diffie-Hellman problem. So given $g^x, g^y \in \langle g \rangle$, the oracle outputs $g^{x \cdot y}$. This is denoted by:

$$B(g^x, g^y) = g^{x \cdot y}.$$

Now assume that the user uses this oracle and inserts $y_1^a g^b$ and $Y$. Note that $y_1, g, a, b$ and $Y$ are all known to the user. Then the oracle computes:

$$B(y_1^a g^b, Y) = B((g^{x_1})^a g^b, g^X) = B(g^{a \cdot x_1 + b}, g^X) = g^{(a \cdot x_1 + b) X} = (g^X)^{a \cdot x_1 + b} = Y^{a \cdot x_1 + b}.$$

Next, let:

$$f = \frac{g^{x_1}}{B(y_1^a g^b, Y)} = \frac{g^{x_1}}{Y^{a \cdot x_1 + b}} = g^{x_1} Y^{-a \cdot x_1 - b}.$$

If the program used $t = 0$, then $f = z$. Otherwise, $f / g^W = z$. This means that the user can compute $z$ using the oracle and use $z$ and function $H$ to recover private key $x_2$. This implies that if the computational Diffie-Hellman problem can be solved, then also the SETUP attack on Diffie-Hellman key exchange can be broken. $\square$

This shows that, although the user has full access to the internals of the system, it is not possible to recompute the user’s private key $x_2$. Note that this corresponds with the definition of strong SETUP (definition 3.7, p. 43).
3.6. SETUP IN DIFFIE-HELLMAN

Extension to \((m,m + 1)\)-leakage Scheme

The SETUP attack on Diffie-Hellman key exchange has a \((1,2)\)-leakage scheme: Eve needs two Diffie-Hellman key exchange connections to exfiltrate the user’s private key \(x_2\). Eve can only use this key for the connection between Alice and Carol and only until they generate a new shared secret key. Therefore, it is very useful to extend this scheme to a \((m,m + 1)\)-leakage scheme. As soon as Alice makes a third connection (for example with Dave), her private key \(x_3\) is not generated randomly, but computed as follows:

\[ x_3 = H(z), \]

with:

\[ z \equiv g^{x_2 - W^t Y - a x_2 - b} \pmod{p}. \]

The corresponding public key is then:

\[ y_3 \equiv g^{x_3} \pmod{p}. \]

This process can be repeated with new connections, so if Alice also performs key exchanges with Frank, George, Harry, etc. After \(m\) times, \(x_1\) should be chosen randomly again to ensure the randomness of the system. This implies that this attack has a \((m,m + 1)\)-leakage scheme.

Summary

In the SETUP attack that can be implemented in a device that uses Diffie-Hellman key exchange, Eve needs (at least) two connections that are generated using the Diffie-Hellman key exchange, both involving Alice. These are, for example, the connection between Alice and Bob and the connection between Alice and Carol. Usually, the private key of the user is generated randomly each time a connection is made. In this attack, Alice’s private key of the second connection depends on her private key of the first connection. Using Alice’s public keys of both connections, Eve can recover Alice’s private key of the second connection. Finally, using the public key of Carol, Eve can compute the shared secret key of the connection between Alice and Carol.

Since Eve needs two connections, this attack has a \((1,2)\)-leakage scheme. This is extended to a \((m,m + 1)\)-leakage scheme by iterating the process. If the user is able to reverse-engineer the device, it is still impossible to (re)compute past or future keys. This means that the attack constitutes a strong SETUP attack.
Chapter 4

Conclusions

In the last chapter of this master thesis, some important details of the presented SETUP attacks are repeated very briefly. Furthermore, it is mentioned which attacks are realistic and which are less realistic to be used. In the next section, a few suggestions are given in order to prevent against a SETUP attack, followed by a section on recommendations for further research in this area. Finally, the chapter ends with some final words.

4.1 SETUP Attacks

In this master thesis, six possible SETUP attacks are explained in detail. In each attack, the user thinks that the system is secure. However, the attacker can actually exfiltrate user’s private information from the public output of the system:

- **SETUP in RSA key generation I.**
  The mechanism incorporates prime $p$ in the user’s encryption exponent $e$, which is a public parameter. By decrypting $e$ with her own private decryption exponent $D$, Eve can exfiltrate $p$, compute the other prime $q$, factor the modulus $n$, compute $\varphi(n)$ and finally recover Alice’s decryption exponent $d$.

- **SETUP in RSA key generation II (PAP).**
  The mechanism incorporates prime $p$ in the user’s modulus $n$, which is a public parameter. Eve decrypts $n$ with her own private decryption exponent $D$, recovers $p$, compute the other prime $q$, factor $n$, compute $\varphi(n)$ and finally recover Alice’s decryption exponent $d$.

- **SETUP in ElGamal key generation I.**
  The mechanism incorporates Alice’s private key $x$ in generator $g$, which is a public parameter. If Eve decrypts $g$ with her own private decryption exponent $D$, she recovers Alice’s private key $x$.

- **SETUP in ElGamal key generation II.**
  The mechanism incorporates Alice’s private key $x$ in prime $p$, which is a public parameter. Eve uses $p$, generator $g$ and her own private key $X$ to decrypt and recovers Alice’s private key $x$.

- **SETUP in ElGamal signature scheme.**
  The mechanism incorporates the random integer $k_i$ of the first signature in the integer $k_{i+1}$ of the second signature. Eve can use two consecutive signatures and her own private key $X$ to recover Alice’s private key $x$.

- **SETUP in Diffie-Hellman key exchange.**
  The mechanism incorporates the user’s first private key $x_1$ in the user’s second private key $x_2$. In order to recover $x_2$, Eve needs the two public keys $y_1$ and $y_2$ from the two connections that are generated using Diffie-Hellman key exchange, both involving Alice.

In figure 4.1 on the next page it is shown for each attack whether it constitutes a weak, regular or strong attack, together with the corresponding leakage bandwidth. The most secure attack that is presented in this thesis is without doubt the SETUP attack on Diffie-Hellman key exchange, followed by the second attack on RSA (PAP). Note that the attack on Diffie-Hellman key exchange is the only strong attack. The second attack on RSA is a regular attack, but its implementation is very realistic. Despite the fact that the two attacks on ElGamal key generation are also regular SETUP attacks, these are not very realistic.
### CHAPTER 4. CONCLUSIONS

<table>
<thead>
<tr>
<th>SETUP in ...</th>
<th>Weak, regular or strong:</th>
<th>Leakage bandwidth:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA key generation I</td>
<td>-/strong</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>RSA key generation II (PAP)</td>
<td>regular</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>ElGamal key generation I</td>
<td>regular</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>ElGamal key generation II</td>
<td>strong</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>ElGamal signature scheme</td>
<td>weak</td>
<td>(1, 2)/(m, m + 1)</td>
</tr>
<tr>
<td>Diffie-Hellman key exchange</td>
<td>strong</td>
<td>(1, 2)/(m, m + 1)</td>
</tr>
</tbody>
</table>

Figure 4.1: Summary of presented SETUP attacks.

This is because in both attacks the parameters \( p \) and \( g \) are not kept fixed, while usually in ElGamal these parameters are kept fixed. Although the attack on the ElGamal signature scheme is “only” weak, this attack is much more realistic, since \( p \) and \( g \) are fixed. However, this attack is more suitable to construct a subliminal channel between two parties that both know that the system is contaminated and make good use of that.

#### 4.2 Protection against SETUP Attacks

In [36], Young and Yung end with some measures that could be taken in order to prevent against a SETUP attack. Note that a SETUP attack could only be implemented by parties that are responsible for the manufacturing of the device or parties that have direct access to the device. Mainly powerful entities, such as the US government, have these possibilities. However, these kind of parties should actually prevent against infiltration instead of implementing attacks to retrieve private information. Therefore, Young and Yung state that, after showing how a SETUP attack can be implemented, it is definitely necessary to watch these parties closely. They give the following recommendations to prevent and minimize the effects of a SETUP attack and to increase the trust of the users:

- As shown in several attacks, it is very important that the outcome of the device looked random to the user. Users did not have any influence in how these “random” parameters were chosen. To avoid that powerful entities can implement a SETUP attack, manufacturers of the device should create more openness towards the users. First of all, the algorithms should be available for the users (so not a black-box system). Secondly, the algorithm should allow the user to choose random parameters. These two measures together give the user the opportunity to compare one system with another trusted system and to check whether the outputs of both systems are the same.

- Before (key generating) software is installed, the user should always be certain that the software can be trusted. The user can check this by running an **integrity check**. Such an integrity check makes sure that the software is original and unmodified.

- Instead of using one cryptosystem that performs all the steps, it is better to use several systems that work in succession. Note that each system should be designed by different and independent manufacturers. If something odd happens in one system, another system could notice this. Not one, but multiple systems are now responsible for the process, which decreases the probability that an attack is implemented.

- The fourth recommendation sharpens the previous measure. The source that supplies the random parameters, the key generator and the user are three components of the system that should work separately. The user should make sure that each component is authenticated, a component cannot be omitted and that the connections between the components are completely confidential.

#### 4.3 Further Research

As mentioned before, Young and Yung have written many articles on kleptography (see section 3.2, p. 39). These articles contain lots of information that could be added to this master thesis. For example,
in [36] it is shown how SETUP can be implemented in Kerberos; in [37], the probabilistic bias removal method is introduced and used to create a strong SETUP attack in RSA key generation; in [41], the SETUP attack on Diffie-Hellman is used to design a strong attack in ElGamal encryption and a regular attack for the ElGamal signature scheme and in [42], elliptic curve cryptography is introduced and used to design a SETUP attack in RSA. Obviously, this list could be much longer.

Another possibility for further research is the following. The research for this master thesis has been purely theoretical. The research could be extended by actually implementing the algorithms of the attacks in programs such as Python or Sage. If also the normal algorithms are implemented, it would be possible to compare the running times of the normal systems and the contaminated systems.

To conclude, there are many possibilities to continue this research in kleptography. This master thesis only covers the very beginning of kleptography and since the first publication on SETUP in 1996, much progress has been made.

4.4 Final Words

The goal of Young and Yung in 1996 was to show that black-box cryptosystems should not be trusted and that a user should always be suspicious if the internals of a system are secret. By designing their SETUP attack, it definitely can be said that they managed to achieve their goal. They showed how the SETUP attack can be implemented and how this attack provides the attacker the possibility to exfiltrate users’ private key information without the notice of the user. Therefore, the user should always be suspicious when it comes to private key generation. According to Young and Yung, there is actually only one way to generate a private key that is completely secure [40, p. 291-292]:

“The following is a fail-safe way to generate a public key and certify it without worrying about backdoors. You will need a coin, pencil, paper, flashlight, and hard hat. It is of course necessary that you be a spelunker and that you leave all electronic devices behind, save for the flashlight.
1. Climb down into the deepest crevice in the earth’s crust.
2. Make sure no one followed you down there.
3. Flip the coin and perform Neumann unbiasing on the results to generate unbiased coin tosses.
4. Use a pencil to write down the resulting private key.
5. Compute the corresponding public key on the paper.
6. Memorize the key pair.
7. Burn the paper, pencil, and flashlight.
8. Climb back to the surface and submit the public key to a certificate authority.
This may be justly called the Caveman key generation algorithm. Provided that no one invades your brain using extrasensory powers and provided that you use your private key securely, your private key will be known only by you.”

Appendix A

SETUP in RSA

A.1 SETUP in RSA Key Generation I

Algorithm A.1 SETUP in RSA key generation I

Require: key size $k$, attacker’s public key $(E, N)$
1: prime $q \in_R \{0, 1\}^k$
2: prime $p \in_R \{0, 1\}^k$
3: $n \leftarrow p \cdot q$
4: $\varphi(n) \leftarrow (p - 1)(q - 1)$
5: $e \leftarrow p^E \pmod{N}$
6: if $\gcd(e, \varphi(n)) \neq 1$ then
7: return to step 2
8: else
9: $d \leftarrow e^{-1} \pmod{\varphi(n)}$
10: end if
Output: public key $(e, n)$ and private key $(d, n)$

Algorithm A.2 SETUP in RSA key generation I - Modulus factoring by attacker

Require: user’s public key $(e, n)$ and attacker’s private key $(D, N)$
1: $p \leftarrow e^D \pmod{N}$
2: $q \leftarrow n/p$
Output: primes $p$ and $q$
A.2 SETUP in RSA Key Generation II (PAP)

Algorithm A.3 SETUP in RSA key generation II (PAP)

Require: key size $k$, integers $B_1$ and $B_2$, attacker’s public key $(E, N)$, key $K$ and keyed randomization functions $R_1$ and $R_2$

1: prime $p \in R \{0, 1\}^k$
2: $i = 0$
3: while $i < B_1$ do
4: $p' \leftarrow R_1(p)$ with key $K + i$
5: if $p' \geq N$ then
6: $i + +$
7: else
8: go to step 12
9: end if
10: end while
11: return to step 1
12: $p'' \leftarrow (p')^E \pmod N$
13: $j = 0$
14: while $j < B_2$ do
15: $p''' \leftarrow R_2(p'')$ with key $K + j$
16: $X = p'''\{0, 1\}^k$
17: $q \leftarrow X \div p'''$
18: if $q$ is not prime then
19: $j + +$
20: else
21: go to step 25
22: end if
23: end while
24: return to step 1
25: $n \leftarrow p \cdot q$
26: $\phi(n) \leftarrow (p - 1)(q - 1)$
27: $e \leftarrow 17$
28: while $\gcd(e, \phi(n)) \neq 1$ do
29: $e \leftarrow e + 2$
30: end while
31: $d \leftarrow e^{-1} \pmod n$

Output: public key $(e, n)$ and private key $(d, n)$
Algorithm A.4 SETUP in RSA key generation II (PAP) - Modulus factoring by attacker

Require: key size \( k \), public modulus \( n \), integers \( B_1 \) and \( B_2 \), attacker’s private key \( (D,N) \), key \( K \) and keyed randomization functions \( R_1 \) and \( R_2 \)

1: \( U \leftarrow k \) upper order bits of \( n \)
2: \( l = 0 \)
3: while \( l < 2 \) do
4: \( j = 0 \)
5: while \( j < B_2 \) do
6: \( p'_j \leftarrow R^{-1}_2(U) \) with key \( K + j \)
7: \( p'_j \leftarrow (p''_j)^D \) (mod \( N \))
8: \( i = 0 \)
9: while \( i < B_1 \) do
10: \( p_{ij} \leftarrow R^{-1}_1(p''_j) \) with key \( K + i \)
11: if \( p_{ij} \mid n \) then
12: go to step 23
13: else
14: \( i ++ \)
15: end if
16: end while
17: \( j ++ \)
18: end while
19: \( U ++ \)
20: \( l ++ \)
21: end while
22: STOP and output “SETUP not used”
23: \( p \leftarrow p_{ij} \)
24: \( q \leftarrow n \mid p \)

Output: primes \( p \) and \( q \)
Appendix B

SETUP in ElGamal

B.1 SETUP in ElGamal Key Generation I

Algorithm B.1 SETUP in ElGamal key generation I

Require: key size $k$, prime $p$, integers $B_1$ and $B_2$, attacker’s public key $(E, N)$, key $K$ and keyed randomization functions $R_1$ and $R_2$

1: prime $p \in R \{0, 1\}^k$
2: integer $x \in R \{1, 2, \ldots, p-1\}$
3: $i = 0$
4: while $i < B_1$ do
5:    $x' \leftarrow R_1(x)$ with key $K + i$
6:    if $x' \geq N$ then
7:        $i++$
8:    else
9:        go to step 13
10:   end if
11: end while
12: return to step 2
13: $x'' \leftarrow (x')^E \pmod{N}$
14: $j = 0$
15: while $j < B_2$ do
16:    $x''' \leftarrow R_2(x'')$ with key $K + j$
17:    if $x''' < p$ and $x'''$ generates $\mathbb{Z}_p^*$ then
18:        $g \leftarrow x'''$
19:    else
20:        $j++$
21: end if
22: end while
23: $y \leftarrow g^x \pmod{p}$

Output: prime $p$, generator $g$, private key $x$ and public key $y$
Algorithm B.2 SETUP in ElGamal key generation I - Recovery of private key by attacker

Require: generator $g$, prime $p$, user’s public key $y$, integers $B_1$ and $B_2$, attacker’s private key $(D, N)$, key $K$ and keyed randomization functions $R_1$ and $R_2$

1: $j = 0$
2: while $j < B_2$ do
3:   $x''_j \leftarrow R_2^{-1}(g)$ with key $K + j$
4:   $x'_j \leftarrow (x''_j)^D \pmod{N}$
5:   $i = 0$
6:   while $i < B_1$ do
7:     $x_{ij} \leftarrow R_1^{-1}(x'_j)$ with key $K + i$
8:     if $g^{x_{ij}} \equiv y \pmod{p}$ then
9:       go to step 17
10:      else
11:         $i + +$
12:     end if
13:   end while
14:   $j + +$
15: end while
16: STOP and output “SETUP not used”
17: $x \leftarrow x_{ij}$

Output: private key $x$
B.2 SETUP in ElGamal Key Generation II

Algorithm B.3 SETUP in ElGamal key generation II

Require: integers $B_1$ and $B_2$, attacker’s generator $G$, key $K$, attacker’s prime $P$, attacker’s public key $Y$ and keyed randomization functions $R_1$ and $R_2$

1: integer $x \in_R \{1, 2, \ldots, P - 1\}$
2: integer $k \in_R \{1, 2, \ldots, P - 1\}$
3: $c_2 \leftarrow x \cdot Y^k \pmod{P}$
4: $i = 0$
5: while $i < B_1$ do
6:  $c_2' \leftarrow R_1(c_2)$ with key $K + i$
7:  if $c_2'$ is prime and $c_2' > x$ then
8:    $c_1 \leftarrow G^k \pmod{P}$
9:  else
10:     $i++$ and return to step 5
11: end if
12: end while
13: $j = 0$
14: while $j < B_2$ do
15:  $c_1' \leftarrow R_2(c_1)$ with key $K + j$
16:  if $c_1' \geq c_2'$ and $c_1'$ generates $\mathbb{Z}_p^*$ then
17:    $p \leftarrow c_2'$
18:    $g \leftarrow c_1'$
19:    $y \leftarrow g^x \pmod{p}$
20:  else
21:     $j++$ and return to step 14
22: end if
23: end while

Output: prime $p$, generator $g$, private key $x$ and public key $y$
Algorithm B.4 SETUP in ElGamal key generation II - Recovery of private key by attacker

Require: generator $g$, prime $p$, user's public key $y$, integers $B_1$ and $B_2$, key $K$ and keyed randomization functions $R_1$ and $R_2$

1: $j = 0$
2: while $j < B_2$ do
3:   $(c_1)_j \leftarrow R_2^{-1}(g)$ with key $K + j$
4:   $i = 0$
5:   while $i < B_1$ do
6:     $(c_2)_i \leftarrow R_1^{-1}(p)$ with key $K + i$
7:     $x_{ij} \leftarrow (c_2)_i (c_1)_j$
8:     if $g^{x_{ij}} \equiv y \pmod{p}$ then
9:       go to step 17
10:      else
11:        $i + +$
12:     end if
13:   end while
14:   $j + +$
15: end while
16: STOP and output “SETUP not used”
17: $x \leftarrow x_{ij}$

Output: private key $x$
B.3 SETUP in ElGamal Signature Scheme

Algorithm B.5 SETUP in ElGamal signature scheme I - First signature

Require: generator $g$, message $m_i$, prime $p$, private key $x$
1: $k_i \in_R \{1, 2, \ldots, p - 2\}$ such that gcd$(k_i, p - 1) = 1$
2: $r_i \leftarrow g^{k_i} \pmod{p}$
3: $s_i \leftarrow k_i^{-1}(m_i - x \cdot r_i) \pmod{p - 1}$

Output: signature $(m_i, r_i, s_i)$

Algorithm B.6 SETUP in ElGamal signature scheme I - Second signature

Require: generator $g$, integer $k_i$, message $m_{i+1}$, prime $p$, private key $x$, public key $Y$
1: $c \leftarrow Y^{k_i} \pmod{p}$
2: $k' \leftarrow c^{-1} \pmod{p - 1}$
3: if gcd$(c, p - 1) = 1$ and gcd$(g^{k'} \pmod{p}, p - 1) = 1$ then
4: $k_{i+1} \leftarrow c^{-1} \pmod{p - 1}$
5: else
6: $k_{i+1} \in_R \{1, 2, \ldots, p - 2\}$ such that gcd$(k_{i+1}, p - 1) = 1$
7: end if
8: $r_{i+1} \leftarrow g^{k_{i+1}} \pmod{p}$
9: $s_{i+1} \leftarrow k_{i+1}^{-1}(m_{i+1} - x \cdot r_{i+1}) \pmod{p}$

Output: signature $(m_{i+1}, r_{i+1}, s_{i+1})$

Algorithm B.7 SETUP in ElGamal signature scheme I - Recovery of private key by attacker

Require: signature $r_i$, signature $(m_{i+1}, r_{i+1}, s_{i+1})$, prime $p$, private key $X$
1: if gcd$(r_{i+1}, p - 1) \neq 1$ then
2: STOP and output “SETUP not used”
3: else
4: $c \equiv r_{i+1}^{-1} X \pmod{p}$
5: $x \leftarrow r_{i+1}^{-1}(m_{i+1} - s_{i+1} c) \pmod{p - 1}$
6: end if

Output: private key $x$
Algorithm B.8 SETUP in ElGamal signature scheme II - First signature

Require: generator $g$, message $m_i$, prime $p$, private key $x$

1: $j = 0$
2: while $j < B$ do
3:   $k_i \in_R \{1, 2, \ldots, p - 2\}$ such that $\gcd(k_i, p - 1) = 1$
4:   $c \leftarrow Y^{k_i} \pmod{p}$
5:   $k' \leftarrow c^{-1} \pmod{p}$
6:   if $\gcd(c, p - 1) = 1$ and $\gcd(g^{k'} \pmod{p}, p - 1) = 1$ then
7:     go to step 12
8:   else
9:     $j++$
10: end if
11: end while
12: $r_i \leftarrow g^{k_i} \pmod{p}$
13: $s_i \leftarrow k_i^{-1}(m_i - x \cdot r_i) \pmod{p - 1}$

Output: signature $(m_i, r_i, s_i)$

Algorithm B.9 SETUP in ElGamal signature scheme II - Second signature

Require: generator $g$, integer $k_i$, message $m_{i+1}$, prime $p$, private key $x$, public key $Y$

1: $k_{i+1} \leftarrow c^{-1} \pmod{p - 1}$
2: $r_{i+1} \leftarrow g^{k_{i+1}} \pmod{p}$
3: $s_{i+1} \leftarrow k_{i+1}^{-1}(m_{i+1} - x \cdot r_{i+1}) \pmod{p}$

Output: signature $(m_{i+1}, r_{i+1}, s_{i+1})$
Appendix C

SETUP in Diffie-Hellman

C.1 SETUP in Diffie-Hellman Key Exchange

Algorithm C.1 SETUP in Diffie-Hellman key pair generation - First key pair generation

Require: \( p, g \)
1: integer \( x_1 \in R \{1, 2, \ldots, p-1\} \)
2: \( y_1 \equiv g^{x_1} \pmod{p} \)
Output: public key \( y_1 \)

Algorithm C.2 SETUP in Diffie-Hellman key pair generation - Second key pair generation

Require: \( p, g, Y, W, a, b, x_1 \)
1: integer \( t \in R \{0, 1\} \)
2: \( z \leftarrow g^{x_1-W-tY-a-x_1-b} \pmod{p} \)
3: \( y_2 \leftarrow H(z) \)
Output: public key \( y_2 \)

Algorithm C.3 SETUP in Diffie-Hellman key pair generation - Recovery of private key by attacker

Require: \( p, g, y_1, y_2, a, b, W \)
1: \( r \leftarrow y_1^ag^b \pmod{p} \)
2: \( z_1 \leftarrow \frac{y_1}{r} \pmod{p} \)
3: if \( y_2 \equiv g^{H(z_1)} \pmod{p} \) then
4: \( x_2 \leftarrow H(z_1) \)
5: else
6: \( z_2 \leftarrow \frac{z_1}{g^b} \)
7: if \( y_2 \equiv g^{H(z_2)} \) then
8: \( x_2 \leftarrow H(z_2) \)
9: else
10: STOP and output “SETUP not used”
11: end if
12: end if
Output: private key \( x_2 \)
## Appendix D

### Safe Primes

<table>
<thead>
<tr>
<th>$p$</th>
<th>$P$</th>
<th>$q$</th>
<th>$p$</th>
<th>$P$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.25</td>
<td>2</td>
<td>587</td>
<td>0.248296</td>
<td>293</td>
</tr>
<tr>
<td>7</td>
<td>0.111111</td>
<td>3</td>
<td>719</td>
<td>0.248609</td>
<td>359</td>
</tr>
<tr>
<td>11</td>
<td>0.206612</td>
<td>5</td>
<td>839</td>
<td>0.248808</td>
<td>419</td>
</tr>
<tr>
<td>23</td>
<td>0.228733</td>
<td>11</td>
<td>863</td>
<td>0.248841</td>
<td>431</td>
</tr>
<tr>
<td>47</td>
<td>0.233056</td>
<td>23</td>
<td>887</td>
<td>0.248873</td>
<td>443</td>
</tr>
<tr>
<td>59</td>
<td>0.237954</td>
<td>29</td>
<td>983</td>
<td>0.248983</td>
<td>491</td>
</tr>
<tr>
<td>83</td>
<td>0.237954</td>
<td>41</td>
<td>1019</td>
<td>0.249019</td>
<td>509</td>
</tr>
<tr>
<td>107</td>
<td>0.240655</td>
<td>53</td>
<td>1187</td>
<td>0.249158</td>
<td>593</td>
</tr>
<tr>
<td>167</td>
<td>0.244012</td>
<td>83</td>
<td>1283</td>
<td>0.249221</td>
<td>641</td>
</tr>
<tr>
<td>179</td>
<td>0.244414</td>
<td>89</td>
<td>1307</td>
<td>0.249235</td>
<td>653</td>
</tr>
<tr>
<td>227</td>
<td>0.245595</td>
<td>113</td>
<td>1319</td>
<td>0.249242</td>
<td>659</td>
</tr>
<tr>
<td>263</td>
<td>0.246198</td>
<td>131</td>
<td>1367</td>
<td>0.249268</td>
<td>683</td>
</tr>
<tr>
<td>347</td>
<td>0.247118</td>
<td>173</td>
<td>1439</td>
<td>0.249305</td>
<td>719</td>
</tr>
<tr>
<td>359</td>
<td>0.247215</td>
<td>179</td>
<td>1487</td>
<td>0.249328</td>
<td>743</td>
</tr>
<tr>
<td>383</td>
<td>0.247389</td>
<td>191</td>
<td>1523</td>
<td>0.249434</td>
<td>761</td>
</tr>
<tr>
<td>467</td>
<td>0.247859</td>
<td>233</td>
<td>1619</td>
<td>0.249382</td>
<td>809</td>
</tr>
<tr>
<td>479</td>
<td>0.247912</td>
<td>239</td>
<td>1823</td>
<td>0.249451</td>
<td>911</td>
</tr>
<tr>
<td>503</td>
<td>0.248012</td>
<td>251</td>
<td>1907</td>
<td>0.249476</td>
<td>953</td>
</tr>
<tr>
<td>563</td>
<td>0.248224</td>
<td>281</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D.1: Safe primes $p$ (such that $p = 2q + 1$ with $q$ prime) with corresponding probabilities $P$, rounded to six decimal digits, and prime $q$. 
Bibliography


