Chapter 3

Cryptography

In Chapter 2 we learned that some functions are really hard to compute. This might seem like terrible news, but enables modern cryptography!

3.1 Perfect Encryption

We start with the oldest problem in cryptography: How can we send a secret message?

Definition 3.1 (Perfect Security)**.** *An encryption algorithm has perfect security, if the encrypted message reveals no information about the plaintext message to an attacker, except for the possible maximum length of the message.*

Remarks:

- If an encryption algorithm offers perfect security, any plaintext message of the same length could have generated the given ciphertext.
- Sometimes perfect security is also called information-theoretic security.
- Is there an algorithm that offers perfect security?

```
# m = plaintext message Alice wants to send to Bob
# k = random key known by Alice and Bob, with len(k) = len(m)
# c = ciphertext, the encrypted message m

def encrypt_otp_Alice(m, k)
    Alice sends c = m ⊕ k to Bob   # ⊕ = XOR

def decrypt_otp_Bob(c, k)
    Bob computes m' = c ⊕ k
```

Algorithm 3.2: One Time Pad

Remarks:

- In cryptography, it's always Alice and Bob, with a possible attacker Eve.

Theorem 3.3. *Algorithm 3.2 is correct.*

Proof. $m' = c \oplus k = (m \oplus k) \oplus k = m$. □

Theorem 3.4. *Algorithm 3.2 has perfect security.*

Proof. Given a ciphertext c, for every plaintext message m there exists a unique key k that decrypts c to m, that is $m = c \oplus k$. Therefore, if k is uniformly random, every plaintext is equally likely and thus, ciphertext c reveals no information about plaintext m. □

Remarks:

- Algorithm 3.2 only works if the message m has the same length as the key k. How can we encrypt a message of arbitrary length with a key of fixed length?
- Block ciphers process messages of arbitrary length by breaking them into fixed-size blocks and operating on each block.

```
# m, k, c as defined earlier, now with len(k) << len(m)

def encrypt_ECB(m, k)
    Split m into r len(k)-sized blocks m_1, m_2, ..., m_r
    for i = 1, 2, 3, ..., r:
        c_i = m_i ⊕ k
    c = c_1; c_2; ...; c_r   # ; stands for concatenation
    return c
```

Algorithm 3.5: Electronic Code Book

Remarks:

- In Algorithm 3.5, blocks of the same plaintext result in the same ciphertext, because the same key k is reused to encrypt every block. Furthermore, reusing the same key reveals information about m_1 and m_2: Suppose you have two messages m_1, m_2 encrypted with the same key k, resulting in c_1, c_2. We now have $c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$. So, reusing the same key k in Algorithm 3.2 is insecure. → notebook
- But there are better block based encryptions. CTR-AES (Advanced Encryption Standard with Counter Mode of Operation) is the current state of the art.
- For encryption, Alice and Bob need to agree on a key k first! While this may be feasible for, e.g., secret agents, it is quite impractical for everyday usage.

3.2 Key Exchange

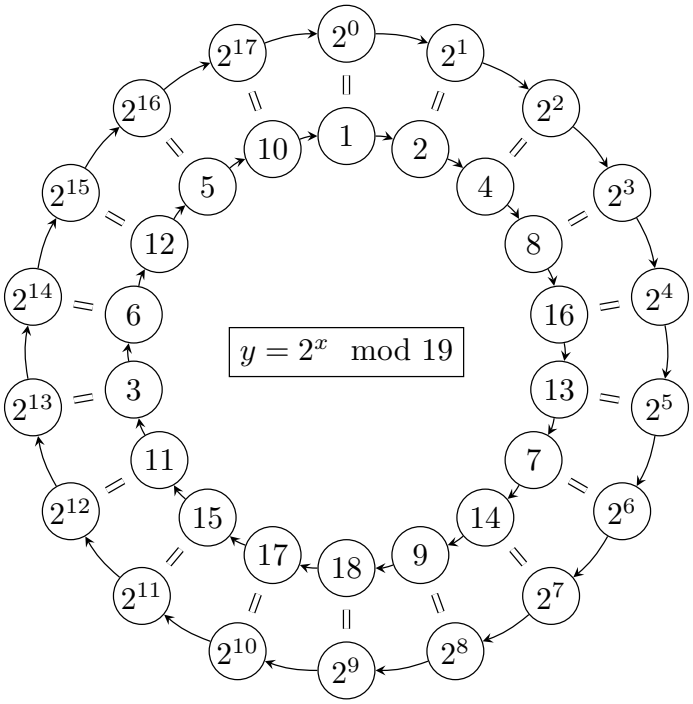
How to agree on a common secret key in public, if you never met before?

Definition 3.6 (Primitive Root)**.** *Let $p \in \mathbb{N}$ be a prime. Then $g \in \mathbb{N}$ is a primitive root of p if the following holds: For every $y \in \mathbb{N}$, with $1 \leq y < p$, there is an $x \in \mathbb{N}$ such that $g^x = y \mod p$.*

→ notebook

Remarks:

- An example for $p = 19$ with $g = 2$:



→ notebook

```
# p = publicly known large prime number
# g = publicly known primitive root of p
def Diffie_Hellman_Alice():
    Pick a random secret key a ∈ {1, 2, ..., p − 1}
    Send k_a = g^a mod p to Bob
    Receive k_b from Bob
    Calculate k = (k_b)^a mod p

def Diffie_Hellman_Bob():
    # same as Alice, swapping all a, b.
```

Algorithm 3.7: Diffie-Hellman Key Exchange

Theorem 3.8. *In Algorithm 3.7, Alice and Bob agree on the same key k.*

Proof. Everything mod p, we have

$$k = (k_b)^a = \left(g^b\right)^a = g^{b \cdot a} = g^{a \cdot b} = (g^a)^b = (k_a)^b = k.$$

Actually, Alice receives $\left(g^b \bmod p\right)$ and computes $\left(g^b \bmod p\right)^a \bmod p$, however under modulo operations $\left(g^b \bmod p\right)^a = \left(g^b\right)^a$. In fact, all the following operations are well-defined in the modulo operation:

$$(a + b) \bmod p = ((a \bmod p) + (b \bmod p)) \bmod p$$

$$(a - b) \bmod p = ((a \bmod p) - (b \bmod p)) \bmod p$$

$$(a \cdot b) \bmod p = ((a \bmod p) \cdot (b \bmod p)) \bmod p$$

$$(g^a \bmod p)^b \bmod p = (g^a)^b \bmod p$$

□

Remarks:

- Algorithm 3.7 does not have perfect security, but instead only computational security.

Definition 3.9 (Computational Security)**.** *An algorithm has computational security, if it is secure against any adversary with polynomial computational resources.*

Remarks:

- The definition of security differs from one cryptographic primitive to another (e.g., encryption, signatures, etc.), but they are typically reduced to the difficulty of a computational problem.
- The computational security of Algorithm 3.7 is based on the difficulty of the discrete logarithm.

Problem 3.10 (Discrete Logarithm or DL)**.** *Given a prime $p \in \mathbb{N}$, a primitive root g of p, and $y \in \mathbb{N}$ with $1 \leq y < p$, find an $x \in \mathbb{N}$ such that $g^x = y \mod p$.*

Remarks:

- In Algorithm 3.7, an adversary overhears g^a and g^b and wants to learn the secret key g^{ab}. But it is unknown whether the adversary actually needs to know how to compute the discrete logarithm to extract the key. Maybe there is another way. Therefore, the following (stronger) assumption captures the security of the protocol better.

Problem 3.11 (Computational Diffie Hellman or CDH)**.** *Given a prime $p \in \mathbb{N}$, a primitive root g of p, and $g^a, g^b \in \mathbb{N}$ with $1 \leq g^a, g^b < p$, compute $g^{ab} \mod p$.*

Lemma 3.12. *CDH $\leq$ DL.*

Proof. We just compute the discrete logarithms of g^a, g^b and then compute g^{ab}. □

Remarks:

- The discrete logarithm (resp. computational Diffie-Hellman) assumption states that it is infeasible to solve DL (CDH) with computationally bounded resources.
- We have no proof that CDH (or DL) is hard, but there is also no known efficient algorithm.
- It is not known whether the opposite direction (DL $\leq$ CDH) holds, though in certain special cases it does.
- Conversely, modular exponentiation can be done in polynomial time using repeated squaring. → notebook
- So far, we have assumed the adversary only listens on the communication channel. This is known as passive security.
- What about stronger adversaries?

Definition 3.13 (Man in the Middle Attack)**.** *A man in the middle attack is defined as an adversary Eve deciphering or changing the messages between Alice and Bob, while Alice and Bob believe they are communicating directly with each other.*

Theorem 3.14. *The Diffie-Hellman Key Exchange from Algorithm 3.7 is vulnerable to a man in the middle attack.*

Proof. Assume that Eve can intercept and relay all messages between Alice and Bob. That alone does not make it a man in the middle attack, Eve needs to be able to decipher or change messages without Alice or Bob noticing. Indeed, Eve can emulate Alice's and Bob's behavior to each other, by picking her own a', b', and then agreeing on common keys $g^{a \cdot b'}$, $g^{b \cdot a'}$ with Alice and Bob, respectively. Thus, Eve can relay all messages between Alice and Bob while deciphering and (possibly) changing them, while Alice and Bob believe they are securely communicating with each other. □

Remarks:

- It is a bit like concurrently playing chess with two grandmasters: If you play white and black respectively, you can essentially let them play against each other by relaying their moves.
- How do we fix this? One idea is to personally meet in private first, exchange a common secret key, and then use this key for secure communication. However, having a key already completely defeats the purpose of a key exchange algorithm.
- Can we do better? Yes, with public key cryptography.

3.3 Public Key Cryptography

Definition 3.15 (Public Key Cryptography)**.** *A public key cryptography system uses two keys per participant: A public key k_p, to be disseminated to everyone, and a secret (private) key k_s, only known to the owner. A message encrypted with the public key of the intended receiver can be decrypted only with the corresponding secret key. Also, messages can be digitally signed; a message verifiable with a public key must have been signed with the corresponding secret key.*

Remarks:

- Popular public key cryptosystems include RSA, Elliptic Curve Cryptography, etc.
- We study a public key cryptosystem based on the DL problem.
- In Diffie-Hellman Key Exchange algorithm (Algorithm 3.7), Alice picked a secret number a and computed a public number $k_a = g^a \mod p$ which Alice sent to Bob. We use the exact same idea in Algorithm 3.16 to generate a pair of public and secret keys (k_p, k_s).

```
# p, g as defined earlier

def generate_key():
    Pick a random secret key k_s ∈ {1, 2, ..., p − 1}
    k_p = g^{k_s} mod p
    return k_p, k_s
```

Algorithm 3.16: Key Generation

3.4 Public Key Encryption

Public key or asymmetric encryption schemes allow users to send encrypted messages directly.

Definition 3.17. *A public key encryption scheme is a triple of algorithms:*

- *A key generation algorithm that outputs a public/secret key pair k_p, k_s.*
- *An encryption algorithm that outputs the encryption c of a message m using the receiver's public key k_p.*
- *A decryption algorithm that outputs the message m using the secret key k_s.*

```
# p, g, m, k_p, k_s as defined earlier

```

```
def encrypt(m, k_p):
    Pick a random nonce x ∈ {1, 2, ..., p − 1}
    c_1 = g^x  mod p
    c_2 = m · k_p^x  mod p
    return c_1, c_2   # encryption c = (c_1, c_2)

def decrypt(c_1, c_2, k_s):
    m' = c_2 · c_1^{k_s·(p−2)}  mod p
    return m'
```

Algorithm 3.18: ElGamal Encryption Algorithm

Theorem 3.19. *ElGamal encryption scheme (Algorithms 3.16, 3.18) is correct.*

Proof. Alice can recover the message: $m' = c_2 \cdot c_1^{k_s \cdot (p-2)} = \left(m \cdot k_p^x\right) \cdot g^{x \cdot k_s \cdot (p-2)} = m \cdot \left(k_p^x\right)^{p-1} = m$. The last step uses the following theorem. □

Theorem 3.20 (Fermat's Little Theorem)**.** *Let p be a prime number. Then, for any $x \in \mathbb{N}$: $x^p = x \mod p$. If x is not divisible by p, then $x^{p-1} = 1 \mod p$.*

Remarks:

- What about the security of ElGamal encryption? In the context of public encryption schemes, we want that an adversary who listens in the communication channel to not be able to extract the message m.

Problem 3.21 (Breaking-ElGamal-Encryption)**.** *Given $(c_1, c_2) = (g^x, m \cdot g^{k_s \cdot x})$ and the public key $k_p = g^{k_s}$, compute the message m.*

Remarks:

- The computational security of Algorithm 3.18 is based on the difficulty of CDH.

Theorem 3.22. *CDH $\leq$ Breaking-ElGamal-Encryption*

Proof. Given (g^a, g^b), we create a problem instance for Breaking-ElGamal-Encryption by setting $c_1 = g^a$, c_2 a random value and $k_p = g^b$. From the definition of the problem we can infer that $c_1 = g^x = g^a$, $k_p = g^{k_s} = g^b$ and thus $c_2 = m \cdot g^{k_s \cdot x} = m \cdot g^{a \cdot b}$. We assume that we can break ElGamal, so we know m. Now we simply compute $m^{p-2} \cdot c_2$ and we get

$$m^{p-2} \cdot c_2 = m^{p-2} \cdot m \cdot g^{a \cdot b} = m^{p-1} \cdot g^{a \cdot b} = g^{a \cdot b}.$$

□

Remarks:

- In other words, we have shown that CDH is "easier" than Breaking-ElGamal-Encryption. As long as the CDH is hard, so is Breaking-ElGamal-Encryption.
- The other direction (Breaking-ElGamal-Encryption $\leq$ CDH) holds as well.
- However, there is a problem with reductions: An encryption scheme that reveals 90% of the plaintext is still considered secure with a reduction approach, as long as the remaining 10% is hard to find. This is clearly unacceptable. We want extracting even a single bit of information to be difficult.
- Both CDH and DL assumptions are not enough to show this. The decisional Diffie Hellman assumption is an even stronger assumption that we rely on.

Problem 3.23 (Decisional Diffie-Hellman or DDH)**.** *Given a prime $p \in \mathbb{N}$, a primitive root g of p, and $g^a, g^b, g^c \in \mathbb{N}$ with $1 \leq g^a, g^b, g^c < p$, decide if $c = a \cdot b$.*

Remarks:

- Note that DDH $\leq$ CDH.
- How can we prove security of ElGamal based on the DDH assumption?
- The idea is to compare ElGamal to a perfectly secure scheme. We have seen an example for perfect security (OTP) which uses XOR. To make the protocols comparable, we now present another version of OTP that uses modulo computation.

```
1  # p, g, m, k_p, k_s as defined earlier
2  # k = random key known by Alice and Bob (shared secret)
3
4  def encrypt_modulo_otp_Alice(m, k)
5      Pick a random nonce x ∈ {1, 2, ..., p − 1}
6      c_1 = g^x  mod p
7      c_2 = m · k  mod p
8      return c_1, c_2   # encryption c = (c_1, c_2)
9
10 def decrypt_modulo_otp_Bob(c_1, c_2, k)
11     Bob computes m' = c_2 · k^(p−2)  mod p
```

Algorithm 3.24: Modulo-OTP

Theorem 3.25. *Algorithm 3.24 is correct.*

Proof. $m' = c_2 \cdot k^{p-2} = (m \cdot k) \cdot k^{p-2} = m \cdot k^{p-1} = m.$ □

Theorem 3.26. *Algorithm 3.24 has perfect security.*

Proof. Given a ciphertext c, for every plaintext message m there exists a unique key k that decrypts c_2 to m, that is $m = c_2 \cdot k^{p-2}$. Therefore, if k is uniformly random, every plaintext is equally likely and thus, ciphertext c_2 reveals no information about plaintext m. □

Remarks:

- Note that the key c_1 is not used at all for encryption and thus is independent of the message. The sole reason why we have that is to make a protocol very similar to ElGamal which we will use in the following proof.
- The idea is to show that in presence of polynomially bounded adversaries ElGamal is just as hard as Modulo-OTP. Since Modulo-OTP is impossible to crack, so is ElGamal (under DDH assumption).

Problem 3.27 (Distinguish(Modulo-OTP, ElGamal))**.** *Given a protocol where Alice sends an encrypted message to Bob, decide whether Alice and Bob are using Modulo-OTP (Algorithm 3.24) or ElGamal (Algorithm 3.18).*

Theorem 3.28. *DDH $\leq$ Distinguish(Modulo-OTP, ElGamal)*

Proof. Given g^a, g^b, g^c, we create an instance for Distinguish(Modulo-OTP, ElGamal), where we set $g^a = g^{k_s}$ and $g^b = g^r$ (both publicly known) and we encrypt any message m with g^c, i.e. $c_2 = g^c \cdot m$. If $g^c = g^{ab}$, then we have the exact situation of ElGamal encryption, since there we encrypt by using $c_2 = g^{k_s \cdot r} \cdot m$. If $g^{ab} \neq g^c$, then we have the exaction situation of Modulo-OTP encryption, where we can set $k = g^c$ since g^c has no relation to public information like g^a and g^b. □

Remarks:

- ElGamal-Encryption has some other features, e.g., it is homomorphic.

Definition 3.29 (Homomorphic Encryption Schemes)**.** *An encryption scheme is said to be homomorphic under an operation $*$ if $E(m_1 * m_2) = E(m_1) * E(m_2)$.*

Remarks:

- In other words, we can directly compute with encrypted data!
- $m_1 * m_2$ and $E(m_1) * E(m_2)$ indicates that ciphertexts and messages can both be operated upon using the same operation. This depends on the representation of ciphertexts, and is not always precisely defined. In the case of ElGamal encryption's homomorphism, we use entry-wise vector multiplication to multiply ciphertexts: $E(m_1) \cdot E(m_2) = (c_{11}, c_{12}) \cdot (c_{21}, c_{22})$.

Lemma 3.30. *The ElGamal encryption scheme (Algorithms 3.16, 3.18) is homomorphic under modular multiplication.*

Proof. We refer the encryption of message m with public key k_p, large prime p, generator g, and a random nonce x as $E(m) = (c_1, c_2) = (g^x, m \cdot k_p^x)$

$$\begin{aligned} E(m_1) \cdot E(m_2) &= (g^{x_1}, m_1 \cdot k_p^{x_1}) \cdot (g^{x_2}, m_2 \cdot k_p^{x_2}) \\ &= (g^{x_1+x_2}, (m_1 \cdot m_2) k_p^{x_1+x_2}) = E(m_1 \cdot m_2) \end{aligned}$$ □

Remarks:

- Not every public encryption scheme is homomorphic under all operations. If an encryption scheme is homomorphic only under some operations, it's called a *partial homomorphic encryption scheme*. For example, we have:
  - Modular multiplication: ElGamal cryptosystem, RSA cryptosystem.
  - Modular addition: Benaloh cryptosystem, Pallier cryptosystem.
  - XOR operations: Goldwasser–Micali cryptosystem.
- There are fully homomorphic encryption schemes that support all possible functions, like Craig Gentry's lattice-based cryptosystem.
- Homomorphic encryption is used in electronic voting schemes to sum up encrypted votes.

3.5 Digital Signatures

Definition 3.31 (Digital Signature Scheme)**.** *A digital signature scheme is a triple of algorithms:*

- *A key generation algorithm that outputs a public/secret key pair k_p, k_s.*
- *A signing algorithm that outputs a digital signature σ on message m using a secret key k_s.*
- *A verification algorithm that outputs* `True` *if the signature σ on the message m is valid using the public key k_p of the signer, and* `False` *otherwise.*

Definition 3.32 (Correctness)**.** *A signature scheme is correct if the verification algorithm on input σ, m, k_p returns* `True` *only if σ is the output of the signing algorithm on input m, k_s.*

Remarks:

- All algorithms (key generation, signing, and verification) should be efficient, i.e., computable in polynomial time.
- Digital signatures offer *authentication* (the receiver can verify the origin of the message), *integrity* (the receiver can verify the message has not been modified since it was signed), and *non-repudiation* (the sender cannot falsely claim that they have not signed the message).
- Widely known signature schemes are ElGamal, Schnorr, and RSA.

```
1  # p, g, m as defined earlier
2  # h = cryptographic hash function like SHA256
3  # k_p, k_s = Alice's public/secret key pair
4  # s, r = the signature sent by Alice
5
6  def sign_Alice(m, k_s):
7      Pick a random x ∈ {1, 2, ..., p − 2}
8      r = g^x  mod p
9      s = x · h(m) − k_s · r  mod p − 1
10     return s, r   # signature σ = (s, r)
11
12 def verify_Bob(m, s, r, k_p):
13     return r^{h(m)} == k_p^r · g^s  mod p
```

Algorithm 3.33: ElGamal Digital Signatures

Remarks:

- The key generation algorithm for ElGamal signatures ElGamal encryption scheme (Algorithm 3.16).

Theorem 3.34. *The ElGamal digital signature scheme (Algorithms 3.16, 3.33) is correct.*

Proof. The algorithm is correct, meaning that a signature generated by (an honest) Alice will always be accepted by Bob. That is because,

$$k_p^r \cdot g^s = g^{k_s \cdot r} \cdot g^{x \cdot h(m) - k_s \cdot r} = g^{k_s \cdot r + x \cdot h(m) - k_s \cdot r} = g^{x \cdot h(m)} = r^{h(m)} \mod p.$$

□

Remarks:

- The random variable x in Line 7 is often called a *nonce* – a number only used once.
- Writing `mod` $p-1$ in Line 9 is not a typo. In the exponent, we always compute modulo $p-1$, since that will make sure that values larger than $p-1$ will be truncated (Theorem 3.20).
- The function $h()$ in Line 9 is a so-called cryptographic hash function. If we did not use $h()$, we had a problem:

Theorem 3.35. *ElGamal signatures without cryptographic hash functions are vulnerable to existential forgery.*

Proof. Let s, r be a valid signature on message m. Then, $(s', r') = (sr, r^2)$ is a valid signature on message $m' = rm/2$ (as long as either m or r is even), because → notebook

$$k_p^{r'} \cdot g^{s'} = \left(g^{k_s}\right)^{r^2} \cdot g^{s \cdot r} = g^{r^2 \cdot k_s} \cdot g^{r \cdot (x \cdot m - r \cdot k_s)} = g^{r^2 \cdot k_s + r \cdot x \cdot m - r^2 \cdot k_s} =$$
$$g^{r \cdot x \cdot m} = (g^x)^{r \cdot m} = r^{2r \cdot m/2} = \left(r^2\right)^{r \cdot m/2} = (r')^{m'} \mod p.$$

□

Remarks:

- Existential forgery is the creation of at least one message-signature pair (m, s), when m was never signed by Alice. Hashing the message in the computation makes the inversion difficult.
- Craig Wright used Satoshi Nakamoto's key in Bitcoin and signed a random message attempting to impersonate the famous creator of Bitcoin. However, when Wright was asked to sign "I am Satoshi" he could not deliver!
- So what is a cryptographic hash function?

3.6 Cryptographic Hashing

Definition 3.36 (Cryptographic Hash Function)**.** *A cryptographic hash function is a function that maps data of arbitrary size to a bit array of a fixed size (the **hash value** or **hash**). A cryptographic hash function is called one-way if it is easy to compute but hard to invert and a cryptographic hash function is called collision-resistant if it is difficult to find two different values that are mapped to the same hash.*

Remarks:

- A hash function is deterministic: the same message always results in the same hash value.
- SHA2, SHA3 (Secure Hash Algorithm 2/3), RIPMED, and BLAKE are some example families of cryptographic hash functions. SHA256 is a specific implementation of the SHA2 construction which outputs a 256 bit output for arbitrary sized inputs. Earlier constructions like MD5 or SHA1 are considered broken/weak now.
- One-way and collision-resistance properties can be phrased as computational problems.

Problem 3.37 (Collision)**.** *Find two different values x and x' such that $h(x) = h(x')$.*

Problem 3.38 (Inversion)**.** *Given a value y, find a value x such that $h(x) = y$.*

Lemma 3.39. *Collision $\leq$ Inversion.*

Proof. Let us choose some random value x and compute $h(x)$. If we invert $h(x)$, it is highly likely we get a value x' such that $x' \neq x$ (collision), since hash functions maps data of arbitrary size to fixed-size values, and hence there many potential inputs. □

Lemma 3.40. *Existence of one-way functions $\Rightarrow \mathsf{P} \neq \mathsf{NP}$.*

Proof. Suppose there exists a function h that is one-way. We define a decision problem (Definition 2.2) as follows: Given an input $(\bar{x}, y)$, decide whether there is a input x such that $h(x) = y$ with $\bar{x}$ being a prefix of x. This decision

problem is in NP: Given $(\bar{x}, y)$ as input and x as possible solution, one can check in polynomial time that $h(x) = y$.

If the decision problem was in P, we could invert h one bit at a time. We start by checking whether $(0, y)$ or $(1, y)$ is true. Whichever is true determines the initial prefix $\bar{x}$. We continue to adding a bit to $\bar{x}$ such that the decision with that added bit continues to be true. This way we construct all bits of x. But since we assumed that h was one-way, the decision problem is not in P. We have now a decision problem that is in NP but not in P and thus $\mathsf{P} \neq \mathsf{NP}$. □

Remarks:

- Since we do not know $\mathsf{P} \neq \mathsf{NP}$, we even less know whether one-way functions exist. But that does not keep us from using them.
- What about the security of digital signatures? In the context of digital signatures, only the owner of the secret key should be able to produce valid signatures.

Problem 3.41 (Forging ElGamal-Signatures)**.** *For a given digital signature scheme, produce a valid message-signature pair without having access to the secret key.*

Remarks:

- The computational security of Algorithm 3.33 is based on the difficulty of inverting one-way functions and computing the discrete logarithm. Intuitively, to forge a signature, a malicious Bob can either find a collision in the hash function, $h(m) = h(m') \mod p - 1$, or extract Alice's secret key k_s. Therefore, Bob must either solve Collision or DL. Both problems are assumed to be hard. However, there is no proof that these are the only ways of forging ElGamal digital signatures.

Conjecture 3.42. *(Collision or DL) ≤ Forging-ElGamal-Signatures*

Remarks:

- Note that Alice must choose a different x for *each* signature, keeping x secret. Otherwise, security can be compromised. In particular, if Alice uses the same nonce x and secret key k_s to sign two different messages, Bob can compute k_s.
- Similarly to digital signatures, message authentication codes are used to ensure a message received by Bob is indeed sent by Alice.

Definition 3.43 (Message Authentication Code or MAC)**.** *A message authentication code is a bitstring that accompanies a message. It can be used to verify the authenticity of the ciphertext in combination with a secret authentication key k_a (different from k) shared by the two parties.*

Remarks:

- Eve should not be able to change the encrypted message and/or the MAC, and get Bob to believe that Alice sent the encrypted message.
- MACs are symmetric, i.e., they are generated and verified using the same secret key.
- Algorithm 3.44 shows a hash based MAC construction.

```
# m, k_p, c as defined earlier
# k_a = key to authenticate c

def encrypt_then_MAC(m, k_p, k_a):
    c = encrypt(m, k_p)
    a = h(k_a; c)
    return c, a
```

Algorithm 3.44: Hash Based Message Authentication Code

Remarks:

- Bob accepts a message c only if he calculates $h(k_a; c) = a$.
- With some hash functions (e.g., SHA2), it is easy to append data to the message and obtain another valid MAC without knowing the key. To avoid these attacks, in practice we use $h(k_a; h(k_a; c))$.
- Now Alice and Bob can securely communicate over the insecure communication channels of the internet, due to the known public keys.
- But how does Bob know that Alice's public key really belongs to Alice? What if it is really Eve's key? Quoting Peter Steiner: "*On the Internet, nobody knows you're a dog.*"

3.7 Public Key Infrastructure

"*Love all, trust a few.*" – William Shakespeare

What can we do, unless we personally meet with everyone to exchange our public keys? The answer is trusting a few, in order to trust many.

Remarks:

- Let's say that you don't know Alice, but both Alice and you know Doris. If you trust Doris, then Doris can verify Alice's public key for you. In the future, you can ask Alice to vouch for her friends as well, etc.

- Trust is not limited to real persons though, especially since Alice and Doris are represented by their keys. How do you know that you give your credit card information to a shopping website, and not an attacker? You probably don't know the owner of the shopping website personally.

Definition 3.45 (Public Key Infrastructure or PKI)**.** *Public Key Infrastructure (PKI) binds public keys with respective identities of entities, like people and organizations. People and companies can register themselves with a certificate authority.*

Definition 3.46 (Certificate Authority or CA)**.** *A certificate authority is an entity whose public key is stored in your hardware device, operating system, or browser by the respective vendor like Apple, Google, Microsoft, Mozilla, Ubuntu, etc.*

Remarks:

- A certificate is an assertion that a known real world person, with a physical postal address, a URL, etc. is represented by a given public key, and has access to the corresponding secret key.
- You can accept a public key if a certificate to that effect is signed by a CA whose public key is stored in your device.
- CA's whose public keys are stored in your device are also called root CA's. Sometimes, there are intermediate CA's whose certificates are signed by root CA's, and who can sign many other end-user certificates. This enables scaling, but also introduces vulnerabilities.
- If a CA's secret key is compromised by a malicious actor, they can sign themselves a certificate saying that they are someone else (say, Google), and then impersonate Google to innocent browsers which trust this CA. A CA's key can be revoked if this happens, or CA's keys can have shorter expiry times.
- Another problem is that your own set of root certificates might be compromised, e.g., if malicious software replaces your browser's root certificates with fakes.

3.8 Transport Layer Security

To communicate securely over the internet, we simply combine the cryptographic primitives we learned so far!

Remarks:

- Alice and Bob don't want Eve to be able to read their messages. Therefore, they encrypt their messages using block based encryption (Section 3.1).
- For the encryption algorithm, they need to agree on a secret key using a key exchange protocol (Section 3.2).

- When Alice receives a message, how can she be sure that the message hasn't been modified on the way from Bob to her? Alice and Bob use message authentication (Section 3.5) to ensure integrity of the communication.
- Let's assume that Alice hasn't met Bob in person before. How can she be sure that she is really communicating with Bob and not with Eve? She would ask Bob to authenticate himself (Sections 3.5, 3.7).

Protocol 3.47 (Transport Layer Security, TLS)**.** *TLS is a network protocol in which a client and a server exchange information in order to communicate in a secure way. Common features include a bulk encryption algorithm, a key exchange protocol, a message authentication algorithm, and lastly, the authentication of the server to the client.*

Remarks:

- TLS is the successor of Secure Sockets Layer (SSL).
- HTTPS (Hypertext Transfer Protocol Secure) is not a protocol on its own, but rather denotes the usage of HTTP via TLS or SSL.
- What other problems can we solve using crypto? The answer is surprisingly many! In the next sections we will discuss some of the most exciting cryptographic primitives beyond TLS.

3.9 Zero-Knowledge Proofs

Problem 3.48 (Waldo)**.** *Peggy and Vic play Where's Waldo. Can Peggy prove she found Waldo without revealing Waldo's location to Vic?*

Remarks:

- In the physical world, Peggy can cover the picture with a large piece of cardboard that has a small, Waldo-shaped hole in its center. She can then place the cardboard such that only Waldo is visible through the hole and therefore prove to Vic she has found Waldo without revealing any information regarding Waldo's location.
- In Zero-Knowledge Proofs (ZKP), the *prover*, Peggy, wants to convince the *verifier*, Vic, of the knowledge of a secret without revealing any information about the secret to Vic.

Definition 3.49 (Zero-Knowledge Proof)**.** *A pair of probabilistic polynomial time interactive programs P, V is a zero-knowledge proof if the the following properties are satisfied:*

- ***Completeness:*** *If the statement is true, then an honest verifier V will be convinced by an honest prover P.*
- ***Soundness:*** *If the statement is false, a cheating prover P cannot convince the honest verifier V that it is true, except with negligible probability.*
- ***Zero-knowledge:*** *If the statement is true, a verifier V learns nothing beyond the statement being true.*

Remarks:

- Soundness concerns the security of the verifier, and zero-knowledge the security of the prover.

Problem 3.50 (Color-Blind)**.** *Vic has two spheres: one red and one blue. Vic is color-blind and thus he cannot differentiate between these two spheres; they look exactly the same to him. Peggy, on the other hand, is not color-blind and wants to prove to Vic that she can differentiate between these two spheres. How can she do this?*

Remarks:

- Peggy wants to convince Vic in zero-knowledge, meaning that she wants to give Vic absolutely no additional information except the fact that she can differentiate these two spheres. For example, she does not want Vic to know which sphere is red and which one is blue.

```
# n = security parameter
def ColorBlind(n):
    repeat n times:
        Vic takes the spheres behind his back
        Vic either switches the spheres or not
        Vic shows the spheres to Peggy
        Peggy answers whether Vic switched the spheres or not
```

Algorithm 3.51: Color-blind

Theorem 3.52. *Algorithm 3.51 is complete.*

Proof. If Peggy knows how to differentiate the spheres (i.e. if she is not color-blind) and Vic behaves according to the protocol, Peggy can always tell whether Vic switched the spheres or not. □

Theorem 3.53. *Algorithm 3.51 is sound.*

Proof. If Peggy cannot differentiate the colors, then she has only 50% of guessing correctly whether Vic switched the spheres or not. In such a case, Peggy is caught with a probability $1/2$, and the probability that she survives n rounds of the protocol undetected is negligible (2^{-n}).

□

Remarks:

- The main intuition is that Peggy's answers do not reveal anything about the spheres. In each round, Peggy simply answers whether the spheres are switched or not, and none of these answers tell which one is red and which one is blue. It seems that this proves that our protocol is zero-knowledge, but we need to be careful, zero-knowledge is a strong attribute. In particular, zero-knowledge demands that Vic cannot convince a third party that Peggy can see the colors. Does our protocol fulfill this requirement?

- Say Vic records the whole interaction with Peggy as a movie and shows the recording to you. Will you be convinced that Peggy can see color?

Theorem 3.54. *Vic cannot convince a third party that Peggy can distinguish colors.*

Proof. Vic and Peggy can create the same movie even if Peggy *is color-blind*. Vic and Peggy can first decide in what order to switch the spheres. After they have agreed to the order, they start recording the movie. A third party seeing the movie can never tell whether it was a recording, where Peggy can see color and was genuinely proving to Vic that she could distinguish the colors, or whether Peggy is color-blind and they agreed on the order beforehand. □

Remarks:

- However, zero knowledge is an even stronger attribute. In particular, zero knowledge requires that no information leaks during the protocol.

Theorem 3.55. *Algorithm 3.51 is not zero-knowledge.*

Proof. Consider a malicious Vic who has not 2 but 4 spheres: One red and one blue (as before), plus a second set of spheres: Red and Blue. These upper-case spheres look exactly the same as the lower-case spheres, but Vic knows their color, because somebody told him. In the first round Vic shows the red and blue spheres, just as in the normal protocol. In the second round however, Vic shows the Red and Blue spheres. Now Peggy's answer (switched or not) will directly reveal the color of the original (lower-case) spheres to Vic. So the protocol was not zero-knowledge. □

Remarks:

- An example of a ZKP is Hamiltonian Cycle (HC), see Problem 2.68. HC is particularly interesting because HC is NP-complete. This means that a ZKP for HC can thus be used as a ZKP for every problem in NP.

```
1  # n = security parameter
2  # G = large graph
3  def ZKP_HamiltonianCycle(G):
4      repeat n times:
5          Peggy creates graph H = permutation of G
6          Peggy hides each entry of H
7          Vic tosses a coin c = [perm, cyc]
8          if c == perm:
9              Peggy opens H and gives the permutation
10             Vic verifies that it is the original graph G
11         elif c == cyc:
12             Peggy opens only the entries of the cycle
13             Vic verifies it is a cycle
```

Algorithm 3.56: Hamiltonian Cycle ZKP

Remarks:

- A permutation H of G is the same graph, but we only permute the names of the nodes of G.

- If $c = \texttt{cyc}$, the prover should be able to open only the cycle. This can be done, for example, by using a matrix representation of graphs as illustrated in Figure 3.59.

Theorem 3.57. *Algorithm 3.56 is complete.*

Proof. If Peggy knows the cycle in G, she can satisfy Vic's demand in both cases. In case $c = \texttt{perm}$, Peggy opens the whole matrix and also returns the renaming of G's nodes in H. In case $c = \texttt{cyc}$, Peggy can easily construct and return a cycle in H by applying the permutation in the original cycle in G. □

Theorem 3.58. *Algorithm 3.56 is sound.*

Proof. If someone knew how to answer both questions, then they can construct the cycle: Take the cycle in H and do the reverse permutation to get the cycle in G.

If Peggy does not know the cycle, the previous argument implies that she can only answer one of these questions. In such a case, Peggy is caught with probability 1/2, and the probability that she survives n rounds of the protocol undetected is negligible (2^{-n}).

Remarks:

- The exact math is a bit tricky and the probability is not precisely 1/2 since there is a (small) chance that Peggy might guess randomly a correct cycle. Formalizing soundness in this example is usually difficult, and out of the scope of this lecture.

□

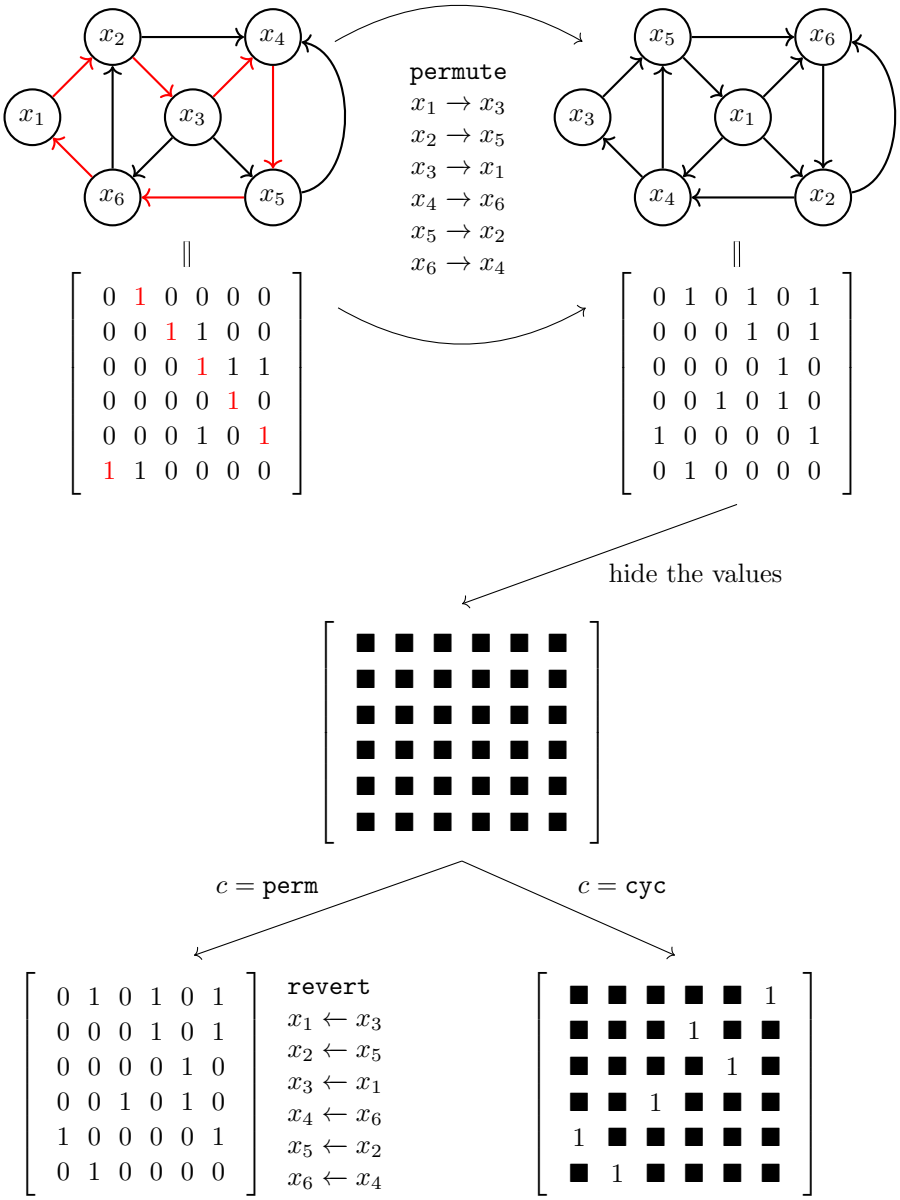


Figure 3.59: An illustration of the ZKP for Hamiltonian Cycle

Remarks:

- What about the zero-knowledge property? The main intuition is that Peggy's answers do not reveal the original cycle in G. In each round, if the challenge is $c =$ `perm` Vic only sees a permutation of G. On the other hand, if the challenge is $c =$ `cyc`, Vic sees that there is a cycle in the hidden graph. However, for all we know, the hidden matrix could be filled all with ones. So in both cases, zero information about the cycle is revealed. As explained in Theorem 3.55, such an argument is not enough.

- Say Vic records the whole interaction with Peggy as a movie and shows the recording to a third party. Will the third party be convinced that Peggy knows the cycle?

Theorem 3.60. *Vic cannot convince a third party that Peggy knows a cycle.*

Proof. Even if Peggy does not know the cycle, Peggy and Vic can create the same movie by agreeing on the order of the challenges beforehand. If the challenge is to open the permutation, Peggy hides a random permutation of the graph. This way she can easily answer the challenge $c =$ `perm`. If the challenge is to open a cycle, Peggy hides a matrix that contains only ones. This way, Peggy can also easily open a random cycle. Similar to the color-blind example, a third party can never tell whether it is a genuine recording or the one where Peggy and Vic decided on the order beforehand. □

Remarks:

- We have seen that our color-blind example also fulfilled this condition and yet it wasn't zero-knowledge. How can we prove that this protocol is zero-knowledge? Remember that zero-knowledge is an even stronger attribute. It requires that absolutely no information leaks.
- One can formally prove that Algorithm 3.56 is zero-knowledge. However, formally proving zero-knowledge is usually difficult and beyond the scope of this lecture.
- But how can we hide the entries of a matrix? We can use commitment schemes.

3.10 Commitment Schemes

Commitment schemes are the digital analogue of a safe.

Definition 3.61 (Commitment Scheme)**.** *A commitment scheme is a two-phase interactive protocol between Alice, the sender, and Bob, the receiver.*

- ***Commit phase:*** *Alice commits to a message m by producing a public commitment c and a secret decommitment d. Alice sends c to Bob.*
- ***Reveal phase:*** *Alice sends m and d to Bob. Bob verifies that the message m corresponds to the commitment c.*

A commitment scheme must be correct, binding and hiding.

- ***Correctness:*** *If both Alice and Bob follow the protocol, then Bob always returns* `True` *in the reveal phase.*
- ***Hiding:*** *A commitment scheme is hiding if Bob cannot extract any information about the committed message before the reveal phase.*
- ***Binding:*** *A commitment scheme is binding if Alice cannot change her commitment after the commit phase.*

Remarks:

- Is there a simple way to create a commitment scheme? How about using hash functions?

```
# m,h as defined earlier
# n = security parameter

def commit_Alice(m):
    Pick a random n-bit string r
    Send c = h(r;m) to Bob

def reveal_Bob(c,m,r): # Bob receives m,r from Alice
    return c == h(r;m)
```

Algorithm 3.62: A Simple Commitment

Theorem 3.63. *Any cryptographic hash function can produce a (computationally binding and computationally hiding) commitment scheme.*

Proof. Assume Alice committed to a value m by sending $h(r, m)$.

- Computationally binding: To change her commitment, Alice needs to find a collision (i.e. an r', m') such that $h(r'; m') = h(r; m)$. This is hard for a computationally bounded Alice.
- Computationally hiding: The main idea is that for Bob to find what is the committed value, he needs to invert the given hash function. This is hard for a computationally bounded Bob. However, this procedure is only heuristically secure. In order to be provably safe, the selected hash function must have special properties.

□

Remarks:

- The protocol is only heuristically computationally hiding, since one might be able to still get partial information about the message. For example, one might learn that the committed value is an odd number. The hash function would still be hard to invert, but the committed value is not hidden anymore.
- One might also think that this scheme is perfectly hiding, since there are infinitely many values that could be committed. We then deduce that even if Bob has unlimited computational power he cannot find the committed value. This is not the case, because we have not specified how the hash function looks like. For example, it might happen that for a particular value there is no collision, and thus Bob with unlimited computational power would be able to find the committed value.

- What if Alice or/and Bob are computationally unbounded?

```
# p, g, m, n as defined earlier
# y = a random value in {1, 2, ..., p − 1}
# x with y = g^x mod p is unknown

def commit_Alice(m):
    Pick a random r ∈ {1, 2, ..., p − 1}
    c = g^m · y^r mod p
    Send c to Bob

def reveal_Bob(m, c, r): # Bob receives m, c, r from Alice
    return c == g^m · y^r mod p
```

Algorithm 3.64: Pedersen Commitment

Theorem 3.65. *Pedersen commitments are correct.*

Proof. Given m, c, r, y, Bob can verify $c = g^m \cdot y^r \mod p$. Thus, the Pedersen commitment scheme is correct. □

Theorem 3.66. *Pedersen commitments are perfectly hiding.*

Proof. Given a commitment c, every message m is equally likely to be the committed message to c. That is because given m, r and any m', there exists (a unique) r' such that $g^m \cdot y^r = g^{m'} \cdot y^{r'} \mod p$. With $y = g^x \mod p$ (such a value x always exists since g is a primitive root), we just have to solve the linear equation $m' + x \cdot r' = m + x \cdot r \mod p - 1$. □

Theorem 3.67. *Pedersen commitments are computationally binding.*

Proof. We show that DL $\leq$ ChangingCommitment, where ChangingCommitment is defined as the problem where Alice needs to find different values m', r' that have the same commitment $c = g^m \cdot y^r$ (i.e. $g^m \cdot y^r = g^{m'} \cdot y^{r'}$).
Given $g, y = g^x$ our goal is to find x. We create an instance for ChangingCommitment by first choosing any m, r and commit to $c = g^m \cdot y^r$. We assume we can solve ChangingCommitment and thus Alice can find m', r' such that $g^m \cdot y^r = g^{m'} \cdot y^{r'}$. The previous equation can also be written as $g^{m-m'} = y^{r'-r}$. We can now compute $x = (m - m') \cdot (r' - r)^{p-3} \mod p - 1$, because

$$\begin{aligned} g^x &= g^{(m-m')\cdot(r'-r)^{p-3}} = \left(g^{(m-m')}\right)^{(r'-r)^{p-3}} = y^{(r'-r)\cdot(r'-r)^{p-3}} \\ &= y^{(r'-r)^{p-2}} = y^{(r'-r)^{p-2} \mod p-1} = y. \end{aligned}$$

The last step assumes that $p - 1$ is prime and thus we can apply Fermat's Theorem. In practice, the group used always has a size that is a prime.

□

Remarks:

- The proof above is oversimplified. Note that if p is prime, then $p-1$ cannot be prime. And we need both to be prime! In practice, people choose a group with a size $q = (p-1)/2$ (example $p = 23$, $q = 11$). Instead of using all numbers from 1 to $p-1$, we choose a subgroup that again has prime order. Then, we have mod p in the base, and mod q in the exponent. This way even in the exponent we can apply Fermat's little theorem.
- If Alice sends both c, r as a commitment to Bob, Pedersen commitments can be perfectly binding and computationally hiding.
- But why compromise at all? Ideally, we want both: perfectly hiding and perfectly binding.

Theorem 3.68. *A commitment scheme can either be perfectly binding or perfectly hiding but not both.*

Proof. Assume a commitment scheme is perfectly binding. Then Alice cannot change her commitment and open another value, even if she is computationally unbounded. This can be the case if and only if there is only a unique value m that can be committed to c. But this means that for a computationally unbounded Bob, he can simply generate all possible commitments and find the value m. Thus the commitment scheme is not perfectly hiding. □

Remarks:

- Commitment schemes have important applications in several cryptographic protocols, such as zero-knowledge proofs, and multiparty computation.

3.11 Threshold Secret Sharing

How does a company share its vault passcode among its board of directors so that at least half of them have to agree to opening the vault?

Definition 3.69 (Threshold Secret Sharing)**.** *Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t,n)-threshold secret sharing scheme.*

→ notebook

```
# s = secret real number to be shared
# t = threshold number of participants to recover the secret
# n = total number of participants

def distribute(s, t, n):
    Generate t − 1 random a_1, ..., a_{t−1} ∈ R
    Obtain a polynomial f(x) = s + a_1 x + ... + a_{t−1} x^{t−1}
```

```
    Generate n distinct x_1, ..., x_n ∈ R \ {0}
    Send (x_i, f(x_i)) to participant P_i

def recover(x = [x_0, x_1, ..., x_t], y = [f(x_0), f(x_1), ..., f(x_t)]):
    f = lagrange(x, y)
    return f(0)
```

Algorithm 3.70: Shamir's (t, n) Secret Sharing Scheme

Theorem 3.71. *Algorithm 3.70 is correct.*

Proof. Any t shares will result in the reconstruction of the same polynomial, hence the secret will be revealed. □

Theorem 3.72. *Algorithm 3.70 has perfect security.*

Proof. A polynomial of degree $t-1$ can be defined only by t or more points. So, any subset $t-1$ of the n shares cannot reconstruct a polynomial of degree $t-1$. Given less than t shares, all polynomials of degree $t-1$ are equally likely; thus any adversary, even with unbounded computational resources, cannot deduce any information about the secret if they have less than t shares. □

Remarks:

- Note that for numerical reasons, in practice modulo p arithmetic is used instead of real numbers. → notebook
- What happens if a participant is malicious? Suppose during recovery, one of the t contributing participants publishes a wrong share $(x_i', f(x_i'))$. The $t-1$ honest participants are blocked from the secret while the malicious participant is able to reconstruct it. To prevent this, we employ *verifiable* secret sharing schemes.

Definition 3.73 (Verifiable Secret Sharing or VSS)**.** *An algorithm that achieves threshold secret sharing and ensures that the secret can be reconstructed even if a participant is malicious is called verifiable secret sharing.*

Remarks:

- Typically, a secret sharing scheme is verifiable if auxiliary information is included that allows participants to verify their shares as consistent.
- VSS protocols guarantee the secret's reconstruction even if the distributor of the secret (the dealer) is malicious.
- So far, we assumed a dealer knows the secret and all the shares. However, we want to avoid trusted third parties and distribute trust. A strong cryptographic notion towards this direction is multiparty computation.

3.12 Multiparty Computation

Alice, Bob, and Carol are interested in computing the sum of their income without revealing to each other their individual income.

```
# a, b, c = Alice's, Bob's and Carol's income

def Sum_MPC():
    Alice picks a large random number r
    Alice sends to Bob m_1 = a + r
    Bob sends to Carol m_2 = b + m_1
    Carol sends to Alice m_3 = c + m_2
    Alice computes s = m_3 − r
    Alice shares s with Bob and Carol
```

Algorithm 3.74: Computation of the Sum of 3 Parties' Income

Theorem 3.75. *Algorithm 3.74 is correct, meaning the output is the desired sum.*

Proof. The output of the algorithm is $m_3 - r = c + m_2 - r = c + b + m_1 - r = c + b + a + r - r = a + b + c$. □

Theorem 3.76. *Algorithm 3.74 keeps the inputs secret.*

Proof. Bob receives $r + a$, hence no information is revealed concerning Alice's income as long as r is large enough. In addition, both Carol and Alice cannot deduce any information about the individual incomes as they are obfuscated. □

Remarks:

- Algorithm 3.74 is an example of secure 3-party computation.
- The generalization of this problem to multiple parties is known as multiparty computation.

Definition 3.77 (Multiparty Computation or MPC)**.** *An algorithm that allows n parties to jointly compute a function $f(x_1, x_2, \ldots, x_n)$ over their inputs $x_1, x_2, \ldots, x_n$ while keeping these inputs secret achieves secure multiparty computation.*

Remarks:

- Formal security proofs in MPC protocols are conducted in the *real/ideal world paradigm.*

Definition 3.78. *The real/ideal world paradigm states two worlds: In the ideal world, there is an incorruptible trusted third party who gathers the participants' inputs, computes the function, and returns the appropriate outputs. In contrast, in the real world, the parties exchange messages with each other. A protocol is secure if one can learn no more about each participant's private inputs in the real world than one could learn in the ideal world.*

Remarks:

- In Algorithm 3.74, we assume all participants are honest. But what if some participants are malicious?
- In MPC, the computation is often based on secret sharing of all the inputs and zero-knowledge proofs for a potentially malicious participant. Then, the majority of honest parties can assure that bad behavior is detected and the computation continues with the dishonest party eliminated or her input revealed.

Chapter Notes

In 1974, Ralph Merkle designed Merkle Puzzles [15], the first key exchange scheme which works over an insecure channel. In Merkle Puzzles, the eavesdropper Eve's computation power can be at most quadric to Alice's and Bob's computational power. This quadratic difference is not enough to guarantee security in practical cryptographic applications. In 1976, Diffie and Hellman introduced a practically secure key exchange scheme over an insecure channel [6].

Diffie Hellman key exchange [6], Schnorr zero-knowledge proofs [19], ElGamal signature and encryption schemes [7] all rely on the hardness of the discrete logarithm problem [3]. So far we have been conveniently vague in our choice of a group, but the discrete logarithm problem is solvable in polynomial-time when we choose an inappropriate group. To avoid this, we can select a group that contains a large subgroup. For example, if $p = 2q + 1$ and q is prime, there is a subgroup of size q, called the quadratic residues of p, which is often used in practice.

Another frequently employed hard problem is integer factorization [12]. The RSA cryptosystem [18], developed in 1977 at MIT by Ron Rivest, Adi Shamir, and Leonard Adleman, depends on integer factorization. RSA was also the first public-key encryption scheme that could both encrypt and sign messages.

A trapdoor one-way function is a function that is easy to compute, difficult to invert without the trapdoor (some extra information), and easy to invert with a trapdoor [6, 25]. The factorization of a product of two large primes, used in RSA, is a trapdoor function. While selecting and verifying two large primes and multiplying them is easy, factoring the resulting product is (as far as known) difficult. However, if one of the prime numbers is given as a trapdoor, then it is easy to compute the other prime number. There are no known trapdoor one-way functions based on the difficulty of discrete logarithms (either modulo a prime or in a group defined over an elliptic curve), because there is no known "trapdoor" information about the group that enables the efficient computation of discrete logarithms. In general, a digital signature scheme can be built by any trapdoor one-way function in the random oracle model [14].

A random oracle [1] is a function that produces a random output for each query it receives. It must be consistent with its replies: if a query is repeated, the random oracle must return the same answer. Hash functions are often modeled in cryptographic proofs as random oracles. If a scheme is secure assuming the adversary views some hash function as a random oracle, it is said to be secure in the random oracle model.

Secure digital signature schemes are unforgeable. There are several versions of unforgeability. For instance, Schnorr signatures, a modification of ElGamal signatures, are existentially unforgeable against adaptively chosen message attacks (EUF-CMA) [20]. In the adaptively chosen message attack, the adversary wants to forge a signature for a particular public key (without access to the corresponding secret key) and has access to a signing oracle, which receives messages and returns valid signatures under the public key in question. The proof that Schnorr digital signatures are EUF-CMA is based on the proof that the Schnorr zero-knowledge proof is sound.

Zero-knowledge proofs are a complex cryptographic primitive; formally defining zero-knowledge proofs was a delicate task that took 15 years of research [2, 10]. One key application for zero-knowledge proofs is in user identification schemes. Another recent one is in cryptocurrencies, such as Monero [23].

The concept of information-theoretically secure communication was introduced in 1949 by American mathematician Claude Shannon, the inventor of information theory, who used it to prove that the one-time pad system was secure [22]. Secret sharing schemes are information theoretically secure. Verifiable secret sharing was first introduced in 1985 by Benny Chor, Shafi Goldwasser, Silvio Micali and Baruch Awerbuch [5]. Thereafter, Feldman introduced a practical verifiable secret sharing protocol [9] which is based on Shamir's secret sharing scheme [21] combined with a homomorphic encryption scheme. Verifiable secret sharing is important for secure multiparty computation to handle active adversaries.

Multiparty computation (MPC) was formally introduced as secure two-party computation (2PC) in 1982 for the so-called Millionaires' Problem, a specific problem which is a Boolean predicate, and in general, for any feasible computation, in 1986 by Andrew Yao [24, 26]. MPC protocols often employ a cryptographic primitive called oblivious transfer.

An oblivious transfer protocol, originally introduced by Rabin in 1981 [17], allows a sender to transfer one of potentially many pieces of information to a receiver, while remaining oblivious as to what piece of information (if any) has been transferred. Oblivious transfer is complete for MPC [11], that is, given an implementation of oblivious transfer it is possible to securely evaluate any polynomial time computable function without any additional primitive! An "1-out-of-n" oblivious transfer protocol [8, 16, 13] is a generalization of oblivious transfer where a receiver gets exactly one database element without the server (sender) getting to know which element was queried, and without the receiver knowing anything about the other elements that were not retrieved. A weaker version of "1-out-of-n" oblivious transfer, where only the sender should not know which element was retrieved, is known as Private Information Retrieval [4].

This chapter was written in collaboration with Zeta Avarikioti, Klaus-Tycho Foerster, Ard Kastrati and Tejaswi Nadahalli.

Bibliography

[1] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73, 1993.

[2] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[3] Dan Boneh. The decision diffie-hellman problem. In *International Algorithmic Number Theory Symposium*, pages 48–63. Springer, 1998.

[4] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[5] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395. IEEE, 1985.

[6] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[7] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[8] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[9] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 285–306. 2019.

[11] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer–efficiently. In *Annual international cryptology conference*, pages 572–591. Springer, 2008.

[12] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, et al. Factorization of a 768-bit rsa modulus. In *Annual Cryptology Conference*, pages 333–350. Springer, 2010.

[13] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829, 2016.

[14] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, 1979.

[15] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[16] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 35(5):1254–1281, 2006.

[17] Michael O Rabin. How to exchange secrets with oblivious transfer.

[18] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[20] Yannick Seurin. On the exact security of schnorr-type signatures in the random oracle model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 554–571. Springer, 2012.

[21] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[22] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[23] Nicolas van Saberhagen. Monero whitepaper. Technical report, 2013.

[24] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[25] Andrew C Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pages 80–91. IEEE, 1982.

[26] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.