Chapter 3

Cryptography

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

3.1 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. AES (Advanced Encryption Standard) 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

→ 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

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

□

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.).
- 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 $g^x = y \mod p$.*

Problem 3.11 (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 $c = a \cdot b$.*

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

Proof. We just compute the discrete logarithms of g^a, g^b, g^c to decide if $c = a \cdot b$ □

Remarks:

- The discrete logarithm assumption states it is infeasible to solve DL given computationally bounded resources.
- We have no proof that DL is hard, but there is no known efficient algorithm.
- Conversely, modular exponentiation can be done in polynomial time using repeated squaring. → notebook
- The decisional Diffie-Hellman assumption states it is difficult to solve DDH.

Lemma 3.13. *Algorithm 3.7 is secure against passive adversaries under the DDH assumption.*

Proof. There is a polynomial-time reduction from breaking passive security of Diffie-Hellman key exchange to violating the DDH assumption. Intuitively, the additional information obtained for the key $k = g^{a \cdot b}$ by an eavesdropper in Algorithm 3.7, can be used to distinguish whether $c = a \cdot b$. We omit the proof as it is advanced, but similar security proofs will be introduced in Section 3.8. □

Remarks:

- Passive security means that any eavesdropping adversary cannot extract from the ciphertext significantly more information for the encrypted message than someone with access only to the message length.
- What about stronger adversaries?

Definition 3.14 (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 3.15. *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 3.16 (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 discrete logarithm 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 3.17 to generate a pair of public and secret keys (k_p, k_s).

```
1  # p, g as defined earlier
2
3  def generate_key():
4      Pick a random secret key k_s ∈ {1, 2, ..., p − 1}
5      k_p = g^{k_s} mod p
6      return k_p, k_s
```

Algorithm 3.17: Key Generation

3.4 Digital Signatures

Definition 3.18 (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.*

A digital signature scheme should be correct, and unforgeable.

Definition 3.19 (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.*

Definition 3.20 (Unforgeability)**.** *A signature scheme is unforgeable if no adversary can produce a valid message-signature pair without receiving it from external sources.*

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.

```
# p, g, m as defined earlier
# h = cryptographic hash function like SHA256
# k_p, k_s = Alice's public/secret key pair
# s, r = the signature sent by Alice

def sign_Alice(m, k_s):
    Pick a random x ∈ {1, 2, ..., p − 1}
    r = g^x mod p
    s = x · h(m) − k_s · r mod p − 1
    return s, r   # signature σ = (s, r)

def verify_Bob(m, s, r, k_p):
    return r^h(m) == k_p^r · g^s mod p
```

Algorithm 3.21: ElGamal Digital Signatures

Theorem 3.22. *The ElGamal digital signature scheme (Algorithms 3.17, 3.21) 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,

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

. □

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 because of Fermat's Little Theorem (see below, Theorem 3.23).
- The function $h()$ in Line 9 is a so-called cryptographic hash function.

Theorem 3.23 (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 $x^{p-1} = 1 \mod p$.*

Definition 3.24 (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 easy to compute but hard to invert.*

Remarks:

- A hash function is deterministic: the same message always results in the same hash value.
- Being hard to invert can be formalized:

Definition 3.25 (Collision Resistance)**.** *Weak collision resistance: Given an input x, it is difficult to find a different input x' such that $h(x) = h(x')$. Strong collision resistance: It is difficult to find two different values x and x' such that $h(x) = h(x')$.*

Definition 3.26 (One-Way)**.** *A function that is practically infeasible to invert is called one-way.*

Remarks:

- 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.

Lemma 3.27. *One-way $\leq$ strong collision resistance.*

Proof. Suppose a hash function h that is not one-way. Now, let us choose some random value x and compute $h(x)$. Since we can invert $h(x)$, it is highly likely we get another value x', such that $x' \neq x$ (collision). That is because a hash function maps data of arbitrary size to fixed-size values, hence there are many collisions (but they are typically hard to find). Thus, h is not strong collision resistant. □

Lemma 3.28. *Existence of one-way functions* $\Rightarrow \mathsf{P} \neq \mathsf{NP}$.

Proof. Suppose there is a function f that is not one-way. We define a decision problem (defined in Chapter 2) as follows. Given an input $(\bar{x}, y)$, decide whether there is an x with $\bar{x}$ a prefix of x, such that $f(x) = y$. This decision problem is not in P, otherwise for a given y one can compute in polynomial time an x such that $f(x) = y$ and thus f would not be one-way. This would be done by extending the prefix (starting from the empty string) character by character until we find x. This decision problem is in NP, since given $(\bar{x}, y)$ as input and x as a solution, one can check in polynomial time that $f(x) = y$ and $\bar{x}$ is a prefix of x. This means, that we have a decision problem that is in NP but not in P and thus $\mathsf{P} \neq \mathsf{NP}$. □

Remarks:

- The existence of one-way functions is still an open problem.

Theorem 3.29. *ElGamal signatures are unforgeable under the discrete logarithm assumption.*

Proof. To forge a signature, a malicious Bob must 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 break the collision resistance property of the cryptographic hash function or solve DL. Both problems are assumed to be hard.

Note that Alice must chose an x uniformly at random for *each* signature, and make sure no information on x is leaked. 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. □

Remarks:

- Why do we need the cryptographic hash function in Algorithm 3.21?

Theorem 3.30. *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 a s either m or r is even), because → notebook

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

□

Remarks:

- Existential forgery is the creation of at least one message-signature pair (m, s), when m was never signed by Alice.
- 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!
- Similarly to digital signatures, message authentication codes are used to ensure a message received by Bob is indeed sent by Alice. However, MACs are symmetric, i.e., they are generated and verified using the same secret key.

Definition 3.31 (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.
- Algorithm 3.32 shows a hash based MAC construction.

```
1  # m, k_p, c as defined earlier
2  # k_a = key to authenticate c
3
4  def encrypt_then_MAC(m, k_p, k_a):
5      c = encrypt(m, k_p)
6      a = h(k_a; c)
7      return c, a
```

Algorithm 3.32: 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 without knowing the key and obtain another valid MAC. 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.*"

3.5 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 some infamous Nigerian princess Eve? You probably don't know the owner of the shopping website personally.

Definition 3.33 (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 3.34 (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.

3.6 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 (see Section 3.1).
- For the encryption algorithm, they need to agree on a secret key using a key exchange protocol (see 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 (see Section 3.4) 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 (see Sections 3.4, 3.5).

Protocol 3.35 (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.

3.7 Public Key Encryption

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

Definition 3.36. *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.*

Remarks:

- The key generation algorithm for ElGamal encryption scheme is the same as in ElGamal signatures (Algorithm 3.17).

```
# 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 3.37: ElGamal Encryption Algorithm

Theorem 3.38. *ElGamal encryption scheme (Algorithms 3.17, 3.37) 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 Theorem 3.23. □

3.8 Security of PK Encryption

For each scheme studied so far, we have seen that they are correct. But are they secure? How do we know that there is no simple attack on a public-key encryption scheme?

Remarks:

- In cryptography, there are two popular ways to prove a scheme is secure: simulation-based security and game-based security. In this section we will focus on game-based security. In Section 3.10 we will discuss simulation-based security.
- There are various game-based security models for asymmetric public-key encryption schemes, most prominently the IND-CPA model.

Definition 3.39 (Indistinguishability under Chosen Plaintext Attack or IND-CPA)**.** *Consider the following game between an adversary (trying to break the security) and a challenger (challenging the adversary), where the adversary is a probabilistic polynomially-bounded algorithm.*

1. *The challenger generates a key pair (k_s, k_p) based on a security parameter n (e.g., a key size in bits), and publishes k_p to the adversary, while k_s remains secret.*

2. *The adversary performs $O(\mathsf{poly}(n))$ computations (encryptions or other operations).*
3. *The adversary sends two distinct plaintexts of his choice m_0, m_1 ($m_0 \neq m_1$) to the challenger.*
4. *The challenger selects a bit $b \in \{0,1\}$ uniformly at random and sends the challenge ciphertext $c = \texttt{encrypt}(k_p, m_b)$ to the adversary.*
5. *The adversary performs any number of additional computations.*
6. *Finally, the adversary outputs a guess for the value of b.*

An encryption scheme is IND-CPA secure if any probabilistic polynomial time adversary has only a negligible "advantage" over random guessing.

Remarks:

- A negligible "advantage" wins above game with probability $\frac{1}{2} + \varepsilon$.
- Intuitively, an encryption scheme is IND-CPA secure if the adversary does not learn any additional information on how to decrypt a message.
- IND-CPA security is equivalent to semantic security, where an adversary that sees the ciphertext has no advantage against an adversary that does not see the ciphertext.
- Does IND-CPA security hold for deterministic encryption schemes?

Theorem 3.40. *All deterministic public-key encryption schemes fail IND-CPA security.*

Proof. Consider any deterministic encryption scheme. The adversary can always win the IND-CPA game: First, the adversary picks two messages m_0, m_1 and encrypts them to c_0 and c_1 respectively using the public key k_p. Then, when the adversary receives the ciphertext c from the challenger, she simply compares c to c_0 and c_1 and always outputs the correct bit! □

Remarks:

- For instance, the traditional RSA encryption scheme fails IND-CPA security as it is deterministic. To prove RSA security, RSA-OAEP is used.
- Therefore, only randomized encryption schemes are of interest, like ElGamal.

Theorem 3.41. *ElGamal is secure in the IND-CPA model under the DDH assumption.*

Proof. We will show that if ElGamal is not IND-CPA secure, then the DDH assumption does not hold (DDH $\leq$ IND-CPA ElGamal). In particular, given an adversary A that gains a non-negligible advantage ε over random guessing in the IND-CPA game, we can construct an adversary (efficient algorithm) B

that given $g^a, g^b, g^c \in \mathbb{N}$ with $1 \leq g^a, g^b, g^c < p$, can decide if $c = a \cdot b$ (DDH) with probability $1/2 + \varepsilon$.

To that end, algorithm B feeds (g, p, g^a) as input to algorithm A (g^a is the public key for A). When the adversary sends the two messages m_0, m_1 to the challenger, algorithm B (posing as the challenger) chooses b randomly from $\{0, 1\}$ and returns to A the ciphertext $c = (g^b, g^c \cdot m_b)$. Algorithm A then returns a value b'. If $b = b'$, algorithm B returns `True`, else `False`.

In case g^a, g^b, g^c is a DDH tuple, i.e., $a \cdot b = c$, we expect algorithm A to produce the correct output with a good probability, i.e., at least with advantage ε. Therefore, adversary B will guess correctly with probability at least $1/2 + \varepsilon$. On the other hand, if g^a, g^b, g^c is a random tuple, then the ciphertext will have an unpredictable structure and thus adversary A will return the correct value with probability $1/2$ (random guess). Therefore, adversary B can distinguish between the DDH tuple and the random tuple with at least a non-negligible advantage ε, which violates the DDH assumption. □

Remarks:

- What about stronger notions of game-based security? There are two widely used models, Indistinguishability under Chosen Ciphertext Attack (IND-CCA) and Indistinguishability under Adaptive Chosen Ciphertext Attack (IND-CCA2).
- Both IND-CCA and IND-CCA2 are defined similarly to IND-CPA, but in addition the adversary is given access to a decryption oracle which decrypts arbitrary ciphertexts at the adversary's request.
- In IND-CCA the adversary can query the decryption oracle only until he receives the challenge. On the contrary, in IND-CCA2, the adversary can also query the decryption oracle after he has received the challenge – but he cannot query the challenge for decryption.

Definition 3.42 (IND-CCA)**.** *Same as IND-CPA (Definintion 3.39) except step 2 which is replaced by "The adversary performs $O(\mathsf{poly}(n))$ computations or queries to the decryption oracle".*

Definition 3.43 (IND-CCA2)**.** *Same as IND-CPA (Definintion 3.42) except step 5 which is replaced by "The adversary performs any number of additional computations or queries to the decryption oracle".*

Theorem 3.44. *ElGamal is not secure in the IND-CCA2 model.*

Proof. Consider the following scenario: The adversary sends the challenger the two messages m_0, m_1 and receives the ciphertext c which corresponds to the encryption of one of the messages m_b. Then, the adversary picks a random message m and queries the decryption oracle with the ciphertext $c' = c \cdot \texttt{encrypt}(k_p, m)$. The decryption oracle cannot refuse to respond to the query since the input is different to the challenge, and returns $m_b \cdot m$. Next, the adversary simply removes m and recovers the challenge message. □

Remarks:

- IND-CCA security of ElGamal is an open problem.
- The reason ElGamal is not IND-CCA2 secure is that it is multiplicatively homomorphic.
- Any homomorphic encryption scheme is not IND-CCA2 secure.

Definition 3.45 (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 dicrectly 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 pairwise vector multiplication to multiply ciphertexts: $E(m_1) \cdot E(m_2) = (c_{11}, c_{12}) \cdot (c_{21}, c_{22})^T$.

Lemma 3.46. *ElGamal encryption scheme (Algorithms 3.17, 3.37) 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 $E(m) = (c_1, c_2) = (g^x, m \cdot k_p^x)$

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

□

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.
- A close cryptographic notion to homomorphism is malleability.

Definition 3.47 (Malleability)**.** *An encryption scheme is malleable if, given the encryption c of a message m, one can generate another ciphertext c' which decrypts to a related message $m' = f(m)$, where f is a known function, without knowing or learning m.*

Lemma 3.48. *ElGamal encryption scheme is malleable.*

Proof. An attacker can change (c_1, c_2) to (c_1, c_2') where $c_2' = z \cdot c_2 \mod p$. With everything mod p

$$m' = c_2' \cdot s'^{p-2} = (z \cdot m \cdot s) \cdot s^{p-2} = z \cdot m \cdot s^{p-1} = z \cdot m$$

Resulting in a valid encryption of $z \cdot m$. □

Remarks:

- Malleability and indistinguishability are closely related. For instance IND-CCA2 is equivalent to non-malleability under the same attack.
- 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 Commitment Schemes

Suppose Alice and Bob want to play XOR over the telephone. Both choose a random bit. If the chosen bits are the same, Alice wins; if they are different, Bob wins. They do not trust each other, therefore they need a scheme that ensures an unbiased result that is verifiable by both parties. For this purpose they can use so-called commitment schemes.

Remarks:

- In a commitment scheme, Alice locks (commits) her bit in a box and gives the box to Bob. Bob reveals his bit, and only then Alice provides Bob with the key to open the box to reveal her bit.
- Commitment schemes are the digital analog of a safe.

Definition 3.49 (Commitment Scheme)**.** *A commitment scheme is a two-phase interactive protocol between Alice, the sender, and Bob, the receiver. A commitment scheme must be correct, binding and hiding.*

- ***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.*

Definition 3.50 (Hiding)**.** *A commitment scheme is hiding if Bob cannot extract any information about the committed message before the reveal phase.*

Definition 3.51 (Binding)**.** *A commitment scheme is binding if Alice cannot change her commitment.*

Definition 3.52 (Correctness)**.** *If both Alice and Bob follow the protocol, then Bob always returns* `True` *in the reveal 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 3.53: A Simple Commitment

Theorem 3.54. *Any cryptographic hash function can produce a (computationally binding and computationally hiding) commitment scheme.*

Proof. Algorithm 3.53 demonstrates how to extract a commitment scheme from a cryptographic hash function. The probability that some random r', m' exist such that $h(r'; m') = h(r; m)$ is 2^{-n}. To find such an r', m' is infeasible for a computationally bounded Alice, hence the commitment scheme is computationally binding. The scheme is also computationally hiding, but we omit the proof because it is complex. □

Remarks:

- 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 3.55: Pedersen Commitment

Theorem 3.56. *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 3.57. *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 r' such that $g^m \cdot y^r = g^{m'} \cdot y^{r'} \mod p$, specifically $m' + x \cdot r' = m + x \cdot r \mod p-1$, where $y = g^x \mod p$. □

Theorem 3.58. *Pedersen commitments are computationally binding.*

Proof. Towards contradiction, suppose Pedersen commitments are not computationally binding. This means that a "polynomial" Alice can successfully compute two pairs m, r and m', r' such that (everything modulo p)

$$\begin{aligned} g^m \cdot y^r &= g^{m'} \cdot y^{r'} \\ \Leftrightarrow g^{m'(p-2)} \cdot y^{r(p-2)} \cdot g^m \cdot g^r &= g^{m'(p-2)} \cdot y^{r(p-2)} \cdot g^{m'} \cdot y^{r'} \\ \Leftrightarrow g^{m+m'(p-2)} \cdot y^{r+r(p-2)} &= g^{m'+m'(p-2)} \cdot y^{r'+r(p-2)} \\ \Leftrightarrow g^{m+m'(p-2)} \cdot (y^r)^{(p-1)} &= (g^{m'})^{(p-1)} \cdot y^{r'+r(p-2)} \\ \Leftrightarrow g^{m+m'(p-2)} &= y^{r'+r(p-2)} \end{aligned}$$

Then, Alice can compute (using the extended Euclidean Algorithm) the multiplicative inverse of $r' + r(p-2)$, i.e., Alice can find a z such that $z(r' + r(p-2)) = 1 \mod p-1$. Consequently, Alice can compute the discrete logarithm $\log_g y = x = z(m + m'(p-2))$ since $g^{z(m+m'(p-2))} = y \mod p$. But this contradicts the discrete logarithm assumption, therefore such an adversary does not exist. □

Remarks:

- 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 a perfectly hiding and perfectly binding commitment scheme.

Theorem 3.59. *A commitment scheme can either be perfectly binding or perfectly hiding but not both.*

Proof. Given a commitment c, a computationally unbounded adversary can simply generate the commitments for every m until finding one that outputs c. In a perfectly binding scheme c uniquely identifies m. Hence, the scheme is not perfectly hiding. □

Remarks:

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

3.10 Zero-Knowledge Proofs

Peggy and Victor play Where's Waldo. Can Peggy prove she found Waldo without revealing Waldo's location to Victor?

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 Victor 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*, Victor, of the knowledge of a secret without revealing any information about the secret to Victor.

Definition 3.60 (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, no verifier V learns anything beyond the statement being true.*

Remarks:

- Soundness concerns the security of the verifier, and zero-knowledge the security of the prover.
- Examples of ZKPs are the Hamiltonian cycle (Problem 2.60) for a large graph, or the Schnorr protocol (also known as $\Sigma-$protocol).

```
# n = security parameter
# G = large graph

def ZKP_HamiltonianCycle(G):
    repeat n times:
        Peggy creates graph H, isomorphic to G
        Peggy commits to H  # using a commitment scheme
        Victor tosses a coin c = [heads, tails]
        if c == heads:
            Victor asks Peggy for the G → H mapping
            Peggy returns the mapping and the decommitment of H
            Victor verifies the mapping and the commitment
```

```
13          else:
14              Victor asks Peggy for the Hamiltonian cycle
15              Peggy returns the decommitment of the cycle only
16              Victor verifies the cycle and the commitment
```

Algorithm 3.61: Hamiltonian Cycle ZKP

Remarks:

- A graph H is isomorphic to G if H is just like G except that all the nodes (and edges) have random names. Think of G as a "symmetric-looking" graph, e.g., every node in G has the same number of neighbors.
- If c is tails, the commitment should allow Victor to verify the cycle. This can be done by, for example, committing to every edge (or lack thereof) separately. Then, Victor only decommits the edges that form the cycle.

Theorem 3.62. *Algorithm 3.61 is complete.*

Proof. If Peggy knows a Hamiltonian cycle in G, she can easily satisfy Victor's demand in both cases. In case c is heads, Peggy returns the isomorphism (the renaming of G's nodes in H) and reveals the H she committed to. In case c is tails, Peggy can easily construct and return a Hamiltonian cycle in H by applying the isomorphism to the cycle in G. □

Theorem 3.63. *Algorithm 3.61 is sound.*

Proof. If Peggy does not know the information, she can guess which question Victor will ask and generate either a graph H isomorphic to G (in which she does not know a Hamiltonian cycle) or a Hamiltonian cycle for an unrelated graph H'. If she does not know a Hamiltonian cycle for G she cannot do both. Therefore, Peggy's chance of fooling Victor is 2^{-n}, where n is the number of rounds the protocol is repeated. □

Theorem 3.64. *Algorithm 3.61 is zero-knowledge, if the commitment is perfectly-hiding.*

Proof. The main idea is that Peggy's answers do not reveal the original Hamiltonian cycle in G. In each round, Victor only learns H's isomorphism to G or that there exists a Hamiltonian cycle in the commited graph. Thus, the information remains unknown since Victor would need both answers for a single round to discover the cycle in G. But how can we be sure that that Victor learns absolutely nothing?

To prove zero-knowledge, we can show that the whole transcript that Victor sees can be generated by Victor himself without the help of Peggy. For each round, if c is heads, Victor can generate a commitment of a isomorphism of G and then simply reveal it. If c is tails, Victor can generate a commitment of a fully connected graph and reveal a Hamiltonian cycle. The transcript generated by Victor is indistinguishable from the transcript that is produced by interacting

with Peggy (provided that commitments are perfectly-hiding). In other words, the transcript that was produced by interacting with Peggy helps Victor just as much as the transcript generated by himself. Since the same transcript can be generated without even knowing the Hamiltonian cycle of G, the real transcript seen by Victor cannot contain any information on the Hamiltonian cycle in G. □

Remarks:

- For all realistic purposes, it is infeasible to break the soundness of a zero-knowledge proof with a reasonable number of rounds.
- One classic three-round protocol that exhibits the properties of a zero-knowledge proof is the Schnorr protocol.
- In Schnorr's protocol, Peggy wants to prove to Victor she knows x, the discrete logarithm of a publicly known value $y = g^x \mod p$ without revealing any information about x.

```
# g, p, x, y as defined earlier

def ZKP_Schnorr():
    Peggy picks a random r ∈ {1, 2, ..., p − 1}
    Peggy sends t = g^r mod p to Victor
    Victor picks a random challenge c ∈ {1, 2, ..., p − 1}
    Victor sends c to Peggy
    Peggy sends to Victor u = r + x · c mod p − 1
    Victor verifies g^u == t · y^c mod p
```

Algorithm 3.65: Schnorr ZKP

Theorem 3.66. *Algorithm 3.65 is complete.*

Proof. If both Peggy and Victor are honest, we have $g^u = g^{r+x\cdot c} = g^r \cdot (g^x)^c = t \cdot y^c \mod p$. □

Theorem 3.67. *Algorithm 3.65 is zero-knowledge.*

Proof. Victor can generate an indistinguishable communication transcript from the actual communication transcript, without Peggy's help, as follows: Given g, p, y, Victor randomly picks $c', u' \in \{1, 2, \ldots, p-1\}$, and outputs $(t', c', u') = (g^{u'} \cdot y^{c'\cdot(p-2)}, c, u)$. Note that $t' \cdot y^{c'} = g^{u'} \cdot y^{c'\cdot(p-2)} \cdot y^{c'} = g^{u'} \cdot (y^{c'})^{p-1} = g^{u'}$, so the verification holds. The communication transcript between Peggy and Victor is $(t, c, u) = (g^r, c, r + x \cdot c)$, where $r, c \in \{1, 2, \ldots, p-1\}$ are randomly picked. Thus, the two transcripts are indistinguishable. By the same reason as in the previous example, the transcript seen by Victor cannot contain any information about the discrete logarithm of g^x. □

Remarks:

- Proving soundness for Algorithm 3.65 is complicated and omitted.
- The analysis of Algorithm 3.65 is very subtle, when the challenge space is very big (not polynomial). In particular, the protocol is zero-knowledge only when the verifier is honest. This and many other subtleties pose the main challenge for the mathematically precise definition of zero-knowledge.
- Can we design non-interactive zero-knowledge proofs?
- The Fiat-Shamir heuristic is a technique for converting an interactive proof of knowledge into a non-interactive proof of knowledge by replacing the challenge with the outcome of a known cryptographic hash function. For instance, the non-interactive version of Schnorr ZKP protocol is the following: Peggy generates $t = g^x \mod p$ and uses a cryptographic hash function h to calculate the challenge $h(t)$. Peggy, then, calculates the proof of knowledge $u = r + x \cdot h(t)$ and makes t, u public. As a result Peggy can calculate the challenge on her own while anyone can verify the proof.
- Fiat-Shamir enables the creation of digital signature schemes from an interactive zero-knowledge proof.

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 3.68 (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 3.69: Shamir's (t, n) Secret Sharing Scheme

Theorem 3.70. *Algorithm 3.69 is correct.*

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

Theorem 3.71. *Algorithm 3.69 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 3.72 (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 3.73: Computation of the Sum of 3 Parties' Income

Theorem 3.74. *Algorithm 3.73 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 3.75. *Algorithm 3.73 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 3.73 is an example of secure 3-party computation.
- The generalization of this problem to multiple parties is known as multiparty computation.

Definition 3.76 (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 3.77. *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 3.73, 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, Tejaswi Nadahalli, Ard Kastrati, and Klaus-Tycho Foerster.

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 Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 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.