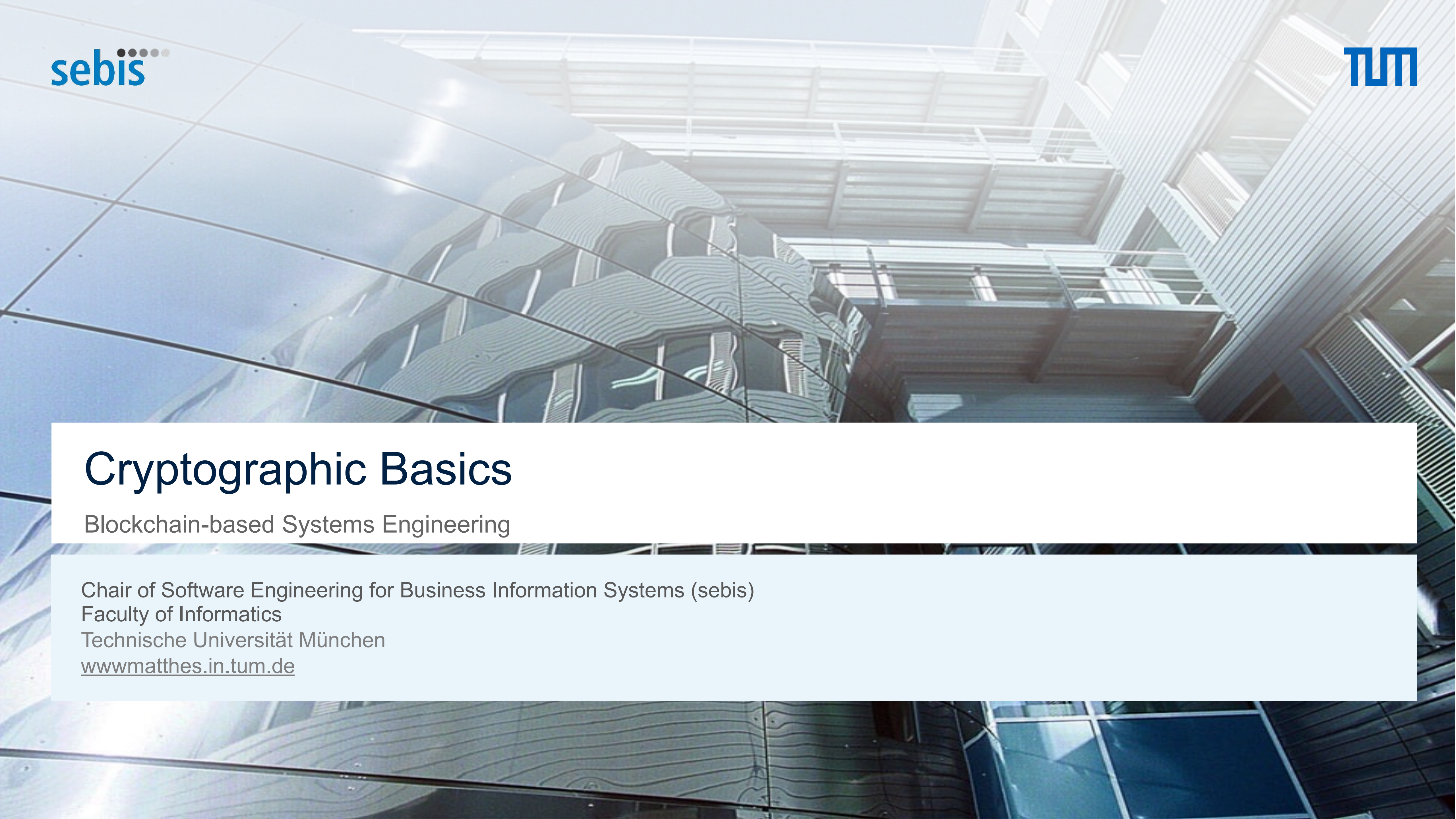


Cryptographic Basics

Blockchain-based Systems Engineering

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

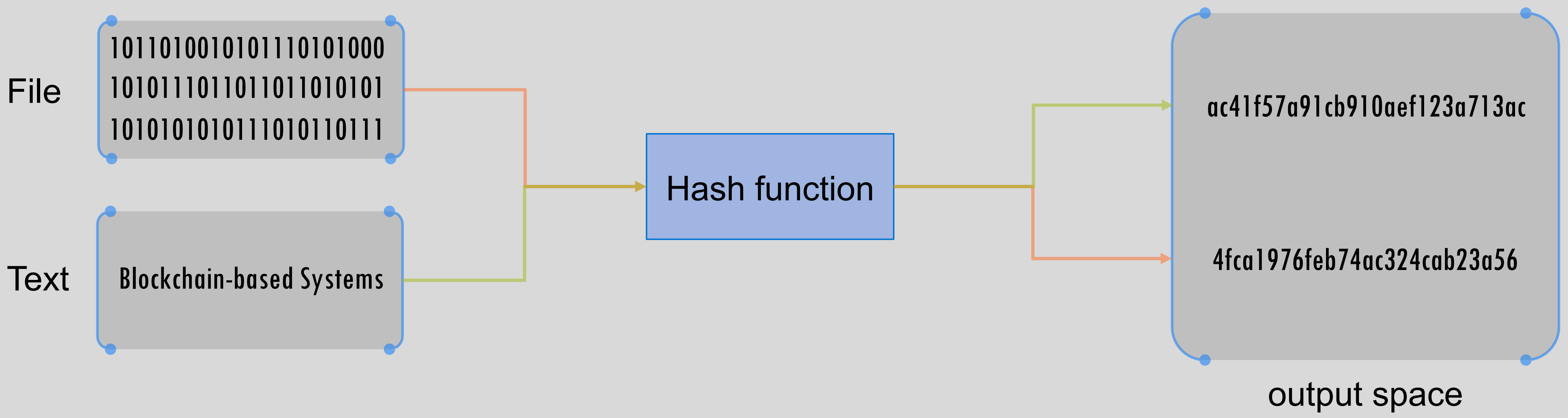
This chapter is heavily inspired by two high quality lectures @ TUM: : [IN2209] IT-Security by C. Eckert and T. Kittel & [IN2101] Network Security by G. Carle and H.Niedermayer
Please also take a look into "IT-Sicherheit: Konzepte, Verfahren, Protokolle" by C. Eckert and „Bitcoin and Cryptocurrency Technologies" by Arvind Narayanan which cover this topic, too.

Hash Functions

Definition: A function *h* is called a ***hash function*** if

- *Compression: h* maps an input *x* of arbitrary finite bit length to an output *h(x)* of fixed bit length *n*: h: $\{0,1\}^* \rightarrow \{0,1\}^n$
- *Ease of computation:* Given *h* and *x* it is *easy* to compute *h(x)*

Compression Independently from the size of the input, the output of h is within a fixed space.



What are the further desirable properties of **cryptographic** hash functions?

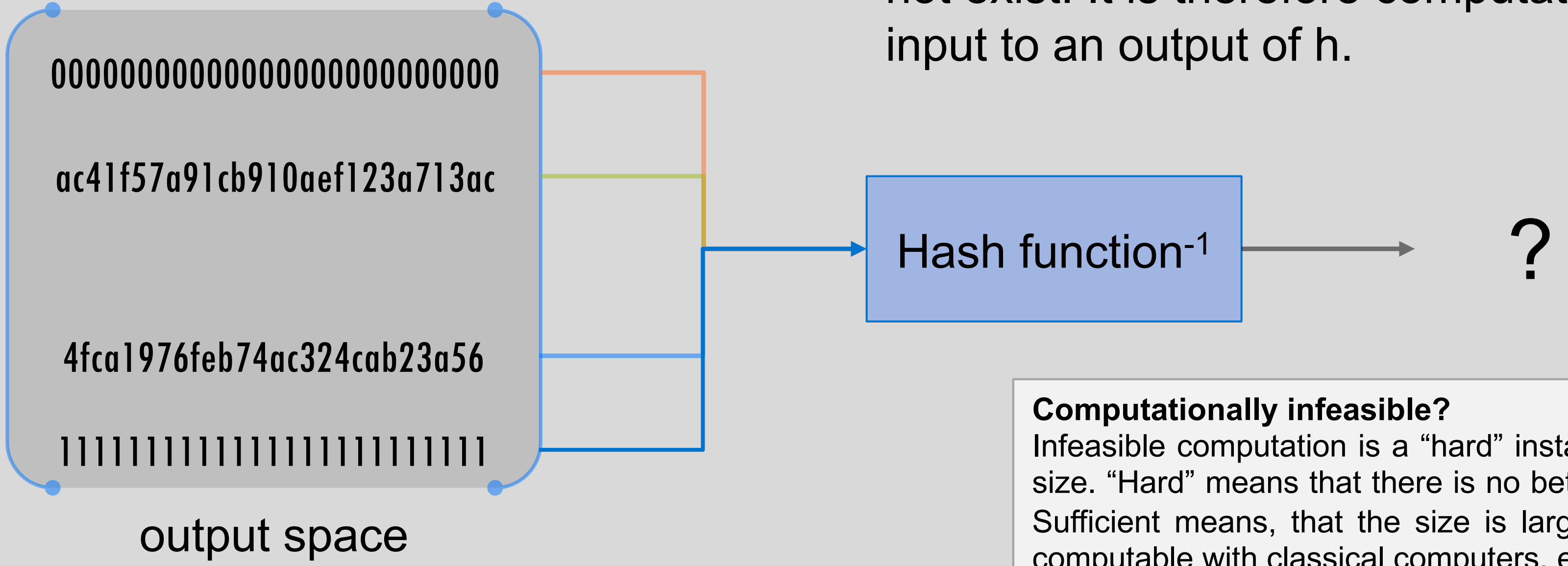
Properties of Cryptographic Hash Functions - Preimage Resistance

Definition:

- *h* is a hash function
- for essentially all pre-specified outputs *y*, it is computationally infeasible to find an *x* such that *h(x)* = *y*
- *h* is also called a **one-way function**.

One-Way Function

If a function h is a one-way-function, then a function h^{-1} does not exist. It is therefore computationally infeasible to find the input to an output of h.



Computationally infeasible?
Infeasible computation is a "hard" instance of an NP-Hard problem of sufficient size. "Hard" means that there is no better way than trying all possible solutions. Sufficient means, that the size is large enough that it can be considered not computable with classical computers, e.g. size = 256 ➔ 2^{256} .

[HEND2012] Henderson, Tim A. D. Cryptography and Complexity. Unpublished. Case Western Reserve University. MATH 408. Spring 2012.

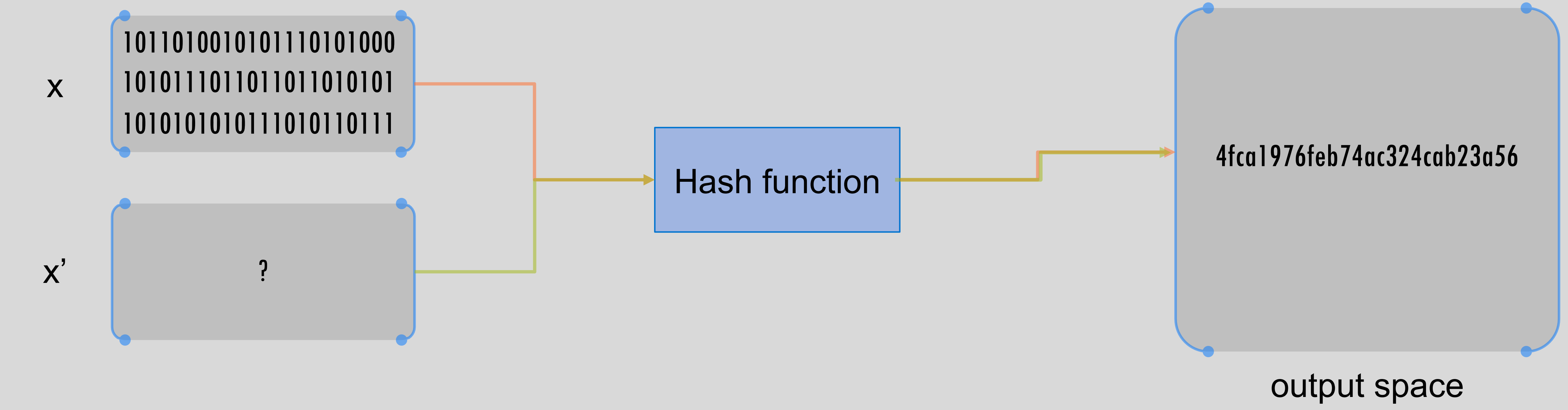
Properties of Cryptographic Hash Functions - 2nd Preimage Resistance

Definition:

- Given *x* it is computationally infeasible to find any second input *x'* with *x != x'* such that *h(x) = h(x')*.

2nd Pre-image Resistance

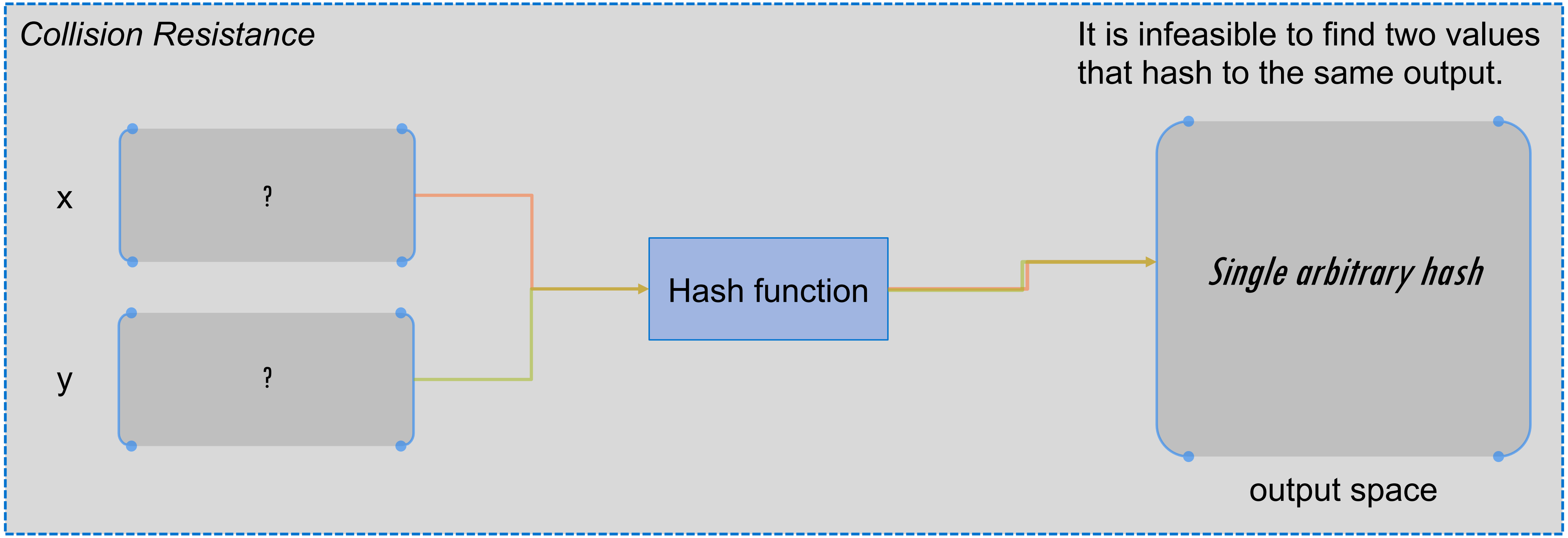
It is computationally infeasible to find a x' which computes to the same hash output.



Properties of Cryptographic Hash Functions - Collision Resistance

Definition:

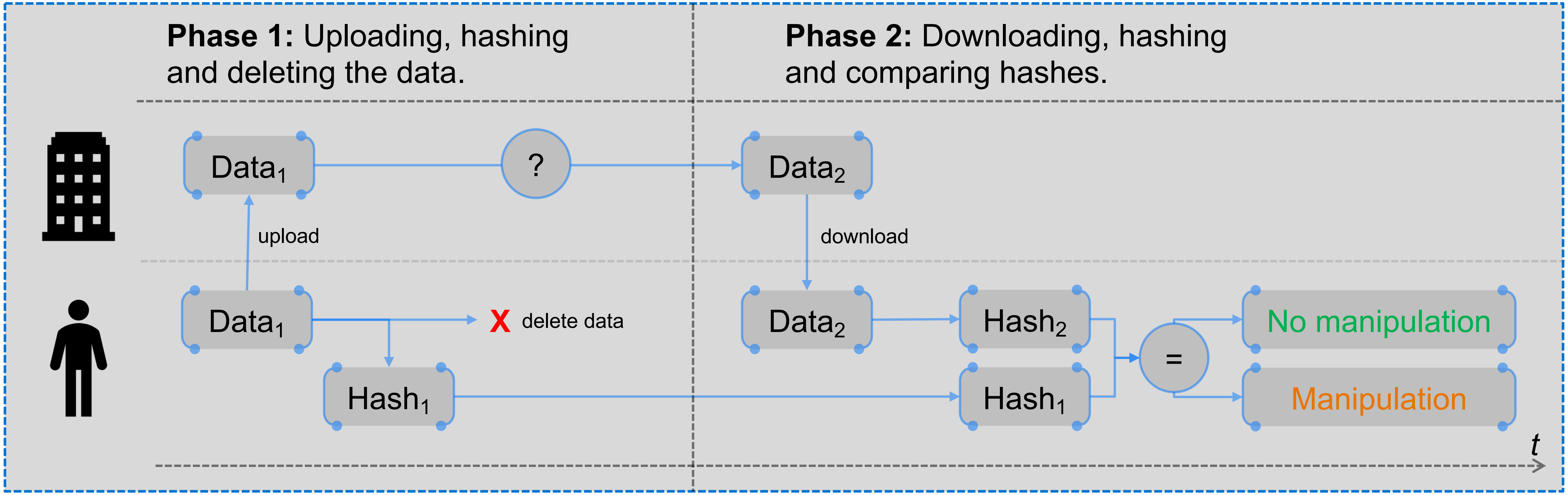
- A hash function h is said to be collision resistant if it is infeasible to find two values, x and y, such that x != y, yet h(x) = h(y).



Application - Message Digests

Suppose you want to store information on an external hosting service. After a successful upload on the external service you want to free up space by deleting that information from your hard drive. You plan to download the data later. However, you want to make sure that the external party cannot modify your content in the mean time without you knowing about the manipulation. How do you proceed?

As of the property 2nd pre-image resistance of the hash function it is not possible to generate the same hash with different contents. Therefore, if the external service manipulates your data, the hash changes. With that, manipulation can be detected.



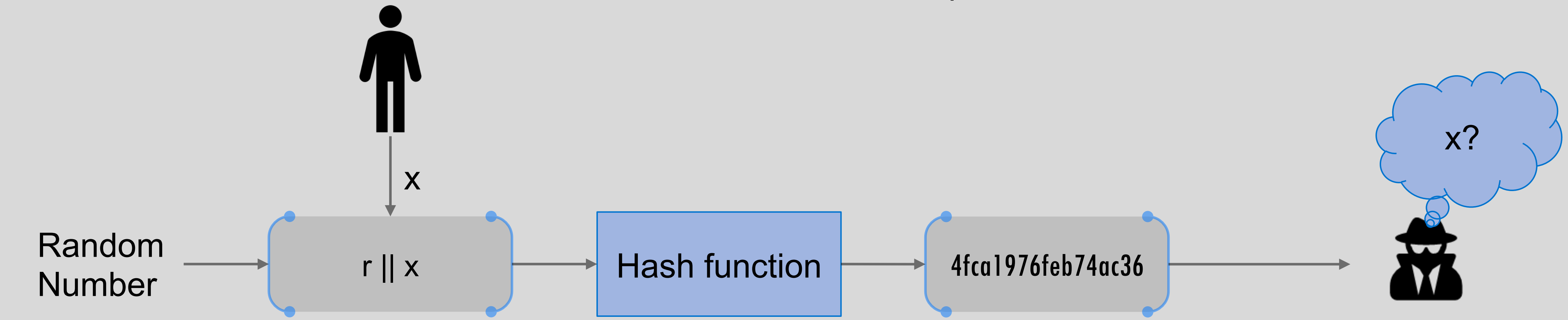
Hiding as an Additional Desirable Property

Definition:

- A hash function *h* is said to be hiding if a secret value r is chosen randomly[1], then, given h(r||x), it is infeasible to find x.

Hiding

The hash function in combination with the random number protects the value x contained in the hash.



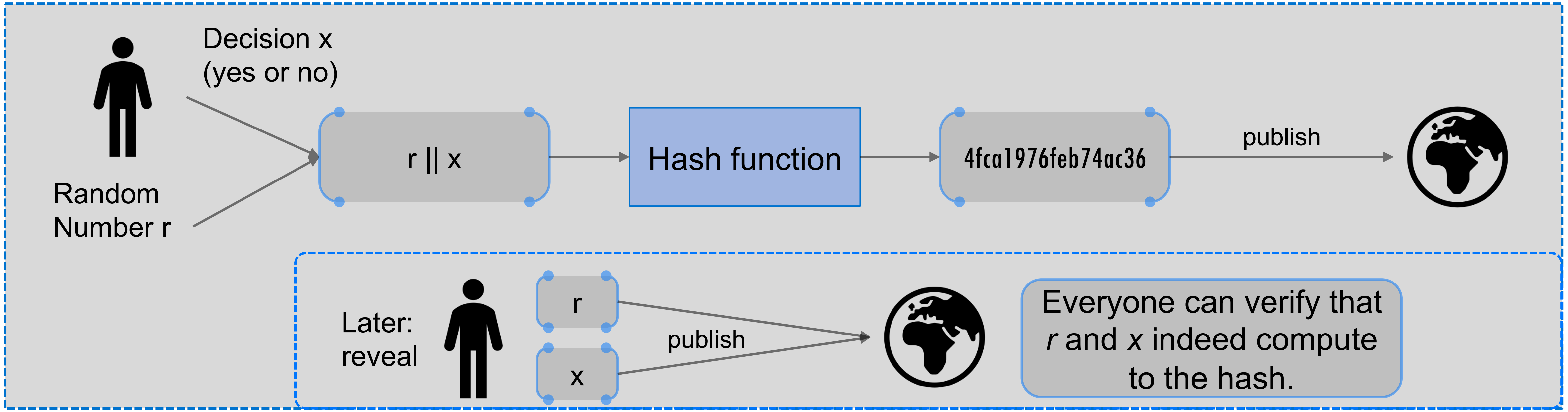
[1]chosen from a probability distribution that has high min-entropy

Application - Commitments

A person can commit him/herself to a value without revealing it immediately.

Two algorithms of *Commitment Scheme*:

- com := **commit**(*msg, nonce*[1])
  - *msg* is the message and *nonce* is the random number. The hash of the concatenation is returned.
- verification := **verify**(*com*, *msg, nonce*)
  - Checks and returns whether *msg* and *nonce* produce the same result as *com*.



[1]Nonce, *number only used once,* is a number used to prevent brute force attacks. It is an arbitrarily chosen number appended to an input to increase the size of the input space.

Search Puzzle

Search puzzle consists out of:

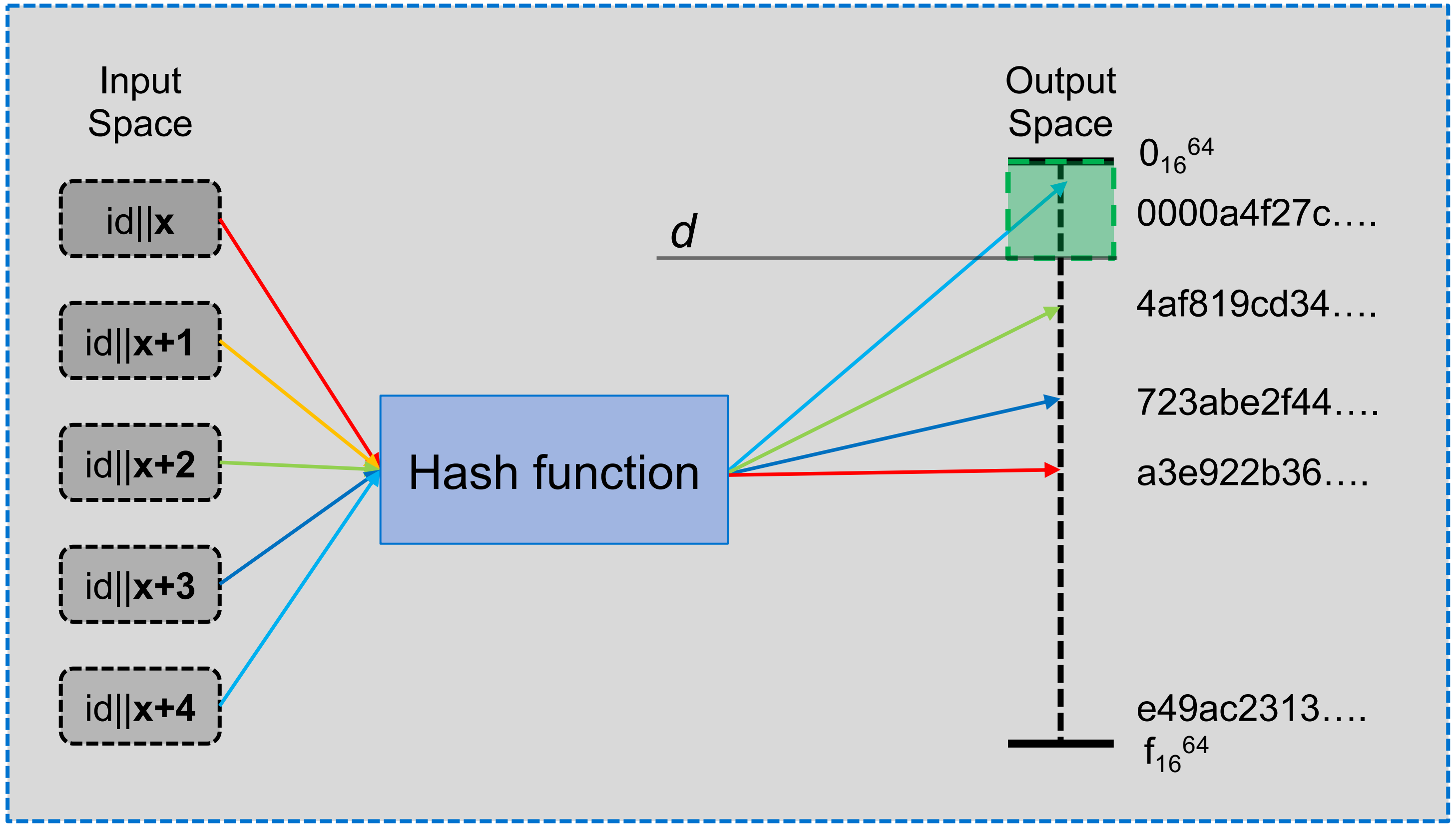
- A hash function h
  - Computes the *puzzle results*
- A value id
  - Is the *puzzle-ID* (makes solutions to the puzzle unique)
- A target set Y
  - For a valid solution, the *puzzle result* must lie within the target set Y
- Computation
  - The puzzle-ID is concatenated with a value x and hashed. x changes until the puzzle result lies within Y

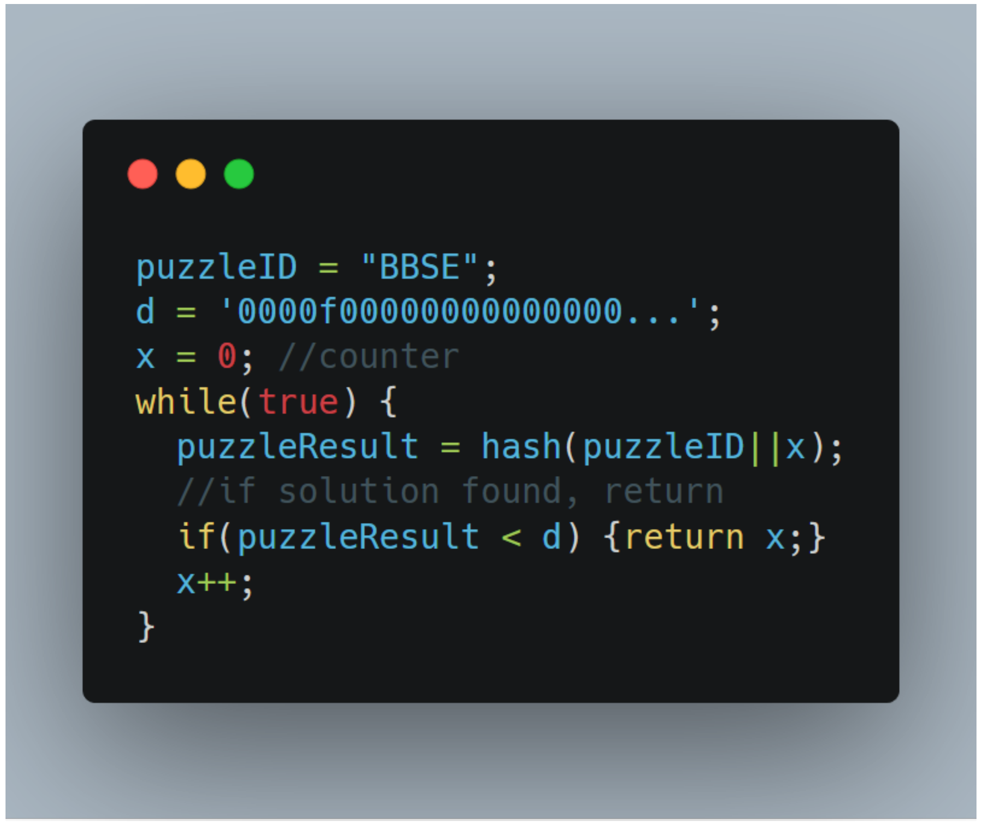
A search puzzle is a mathematical problem which requires searching a very large space in order to find a solution. In particular, there are no shortcuts in finding the solution. h has an n-bit output, therefore it can take any of 2^n values. Solving the puzzle requires finding an input so that the output falls within the set Y. Depending on the size of the set Y, the puzzle can be more or less difficult, e.g., if the set contains all n-bit strings it is trivial, if it contains only one string, it is maximally hard.

A demo can be found here: https://anders.com/blockchain/block.html

Search Puzzle Visualized

For simplicity, we define the target set Y as {0, 1, …, d}, therefore we only have to check if the result of the hash function is smaller than the difficulty d. We define the *puzzleID* as "BBSE"[1]. A pseudocode that implements this search puzzle would look like the following.





```
puzzleID = "BBSE";
d = '0000f000000000000000...';
x = 0; //counter
while(true) {
  puzzleResult = hash(puzzleID||x);
  //if solution found, return
  if(puzzleResult < d) {return x;}
  x++;
}
```

🟩 Target Set Y

[1] Note, that the *puzzleID* should not be known in advance, as solutions could be precomputed. To prevent attacks, puzzleIDs should be chosen randomly.

SHA-Family

There are many **different hash algorithms**:

- Message Digest 4 / 5 (MD4 / MD5) Considered broken!
- Secure Hash Algorithm 1 (SHA-1) Considered broken!
- Secure Hash Algorithm 2 / 3 (SHA-2 / SHA-3) At the moment safe to use, favor SHA-3 over SHA-2.

Most important: Never do your own crypto! Please use reference implementations!

The SHA-family

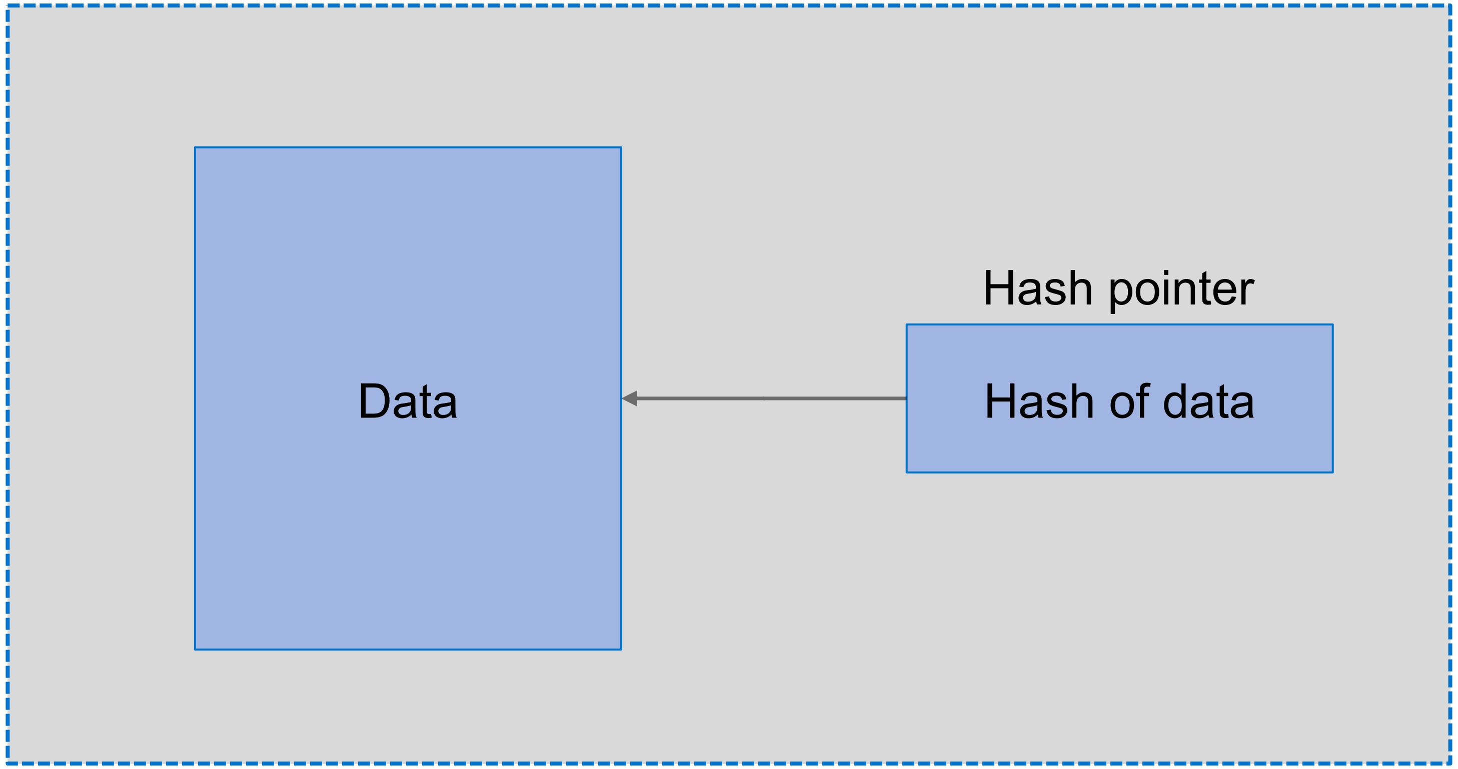
The SHA-family describes a group of standardized hash functions by the National Institute for Standards and Technology (NIST). The SHA-1 & SHA-2 algorithm were developed by NIST and NSA. As of first attacks on SHA-1 in 2004, NIST started a tender process to find a new, more secure SHA-3 algorithm. In 2012, Keccak was announced as the SHA-3 standard.

Keccak itself is not a single hash algorithm, but a family of hash algorithms with different parameters.

Outline

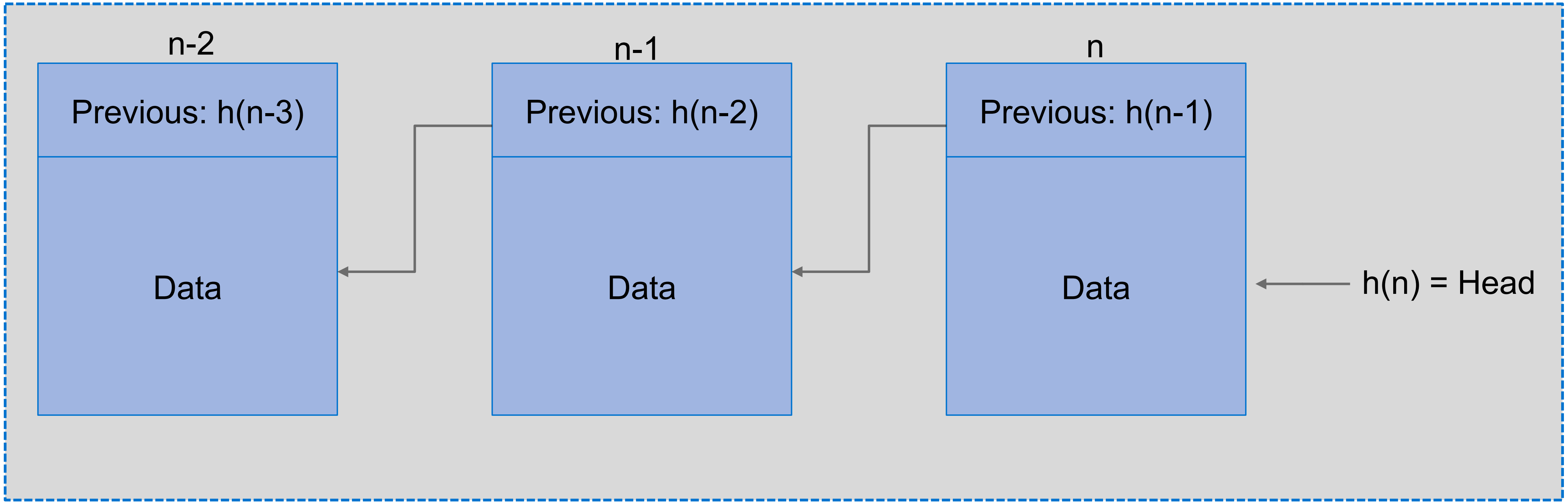
This chapter is heavily inspired by two high quality lectures @ TUM: : [IN2209] IT-Security by C. Eckert and T. Kittel & [IN2101] Network Security by G. Carle and H.Niedermayer
Please also take a look into "IT-Sicherheit: Konzepte, Verfahren, Protokolle" by C. Eckert and „Bitcoin and Cryptocurrency Technologies" by Arvind Narayanan which cover this topic, too.

Hash Pointer



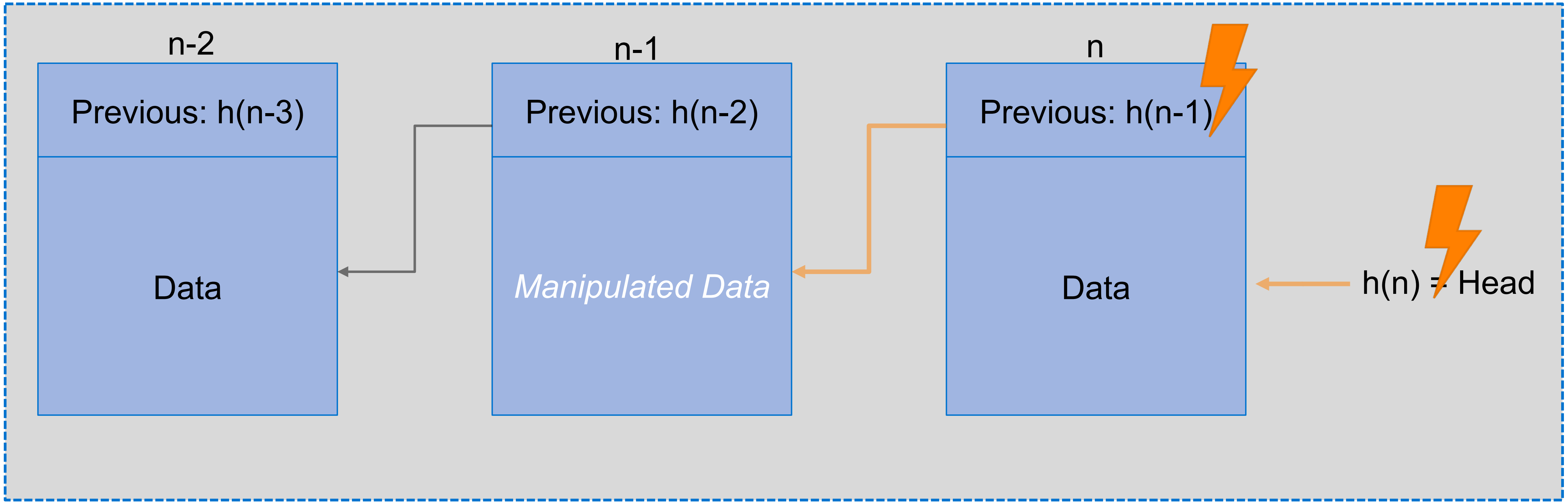
A hash pointer allows us to verify that the information has not changed. Unlike a regular pointer, a hash pointer **does not** contain the information to the location of the data enriched with a cryptographic hash of it.

Blockchain



A linked list of hash pointers is shown. That is typically referred to as a Blockchain. Instead of normal pointers, hash pointers are used. With this, the integrity of the complete blockchain is ensured, because even if all hashes up to the head are recalculated, detecting the change in head hash would only require storing the correct head hash instead of the whole chain.

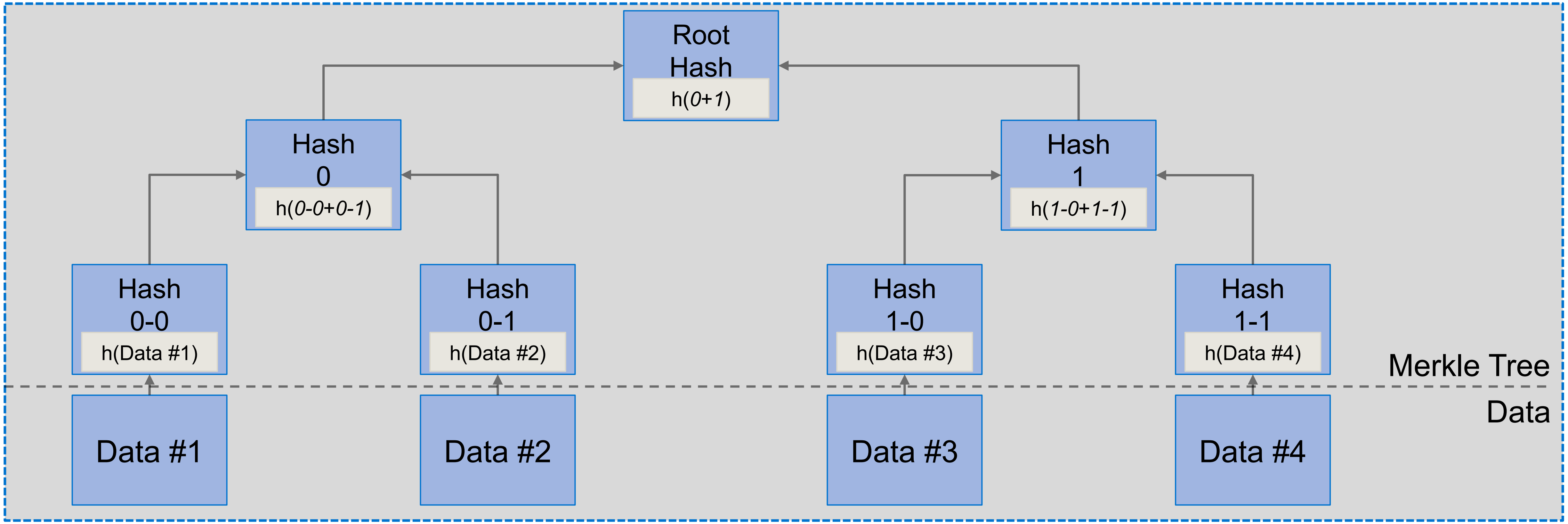
Manipulated Data in Blockchain



Example of manipulated data in Block n-1.

Merkle Trees

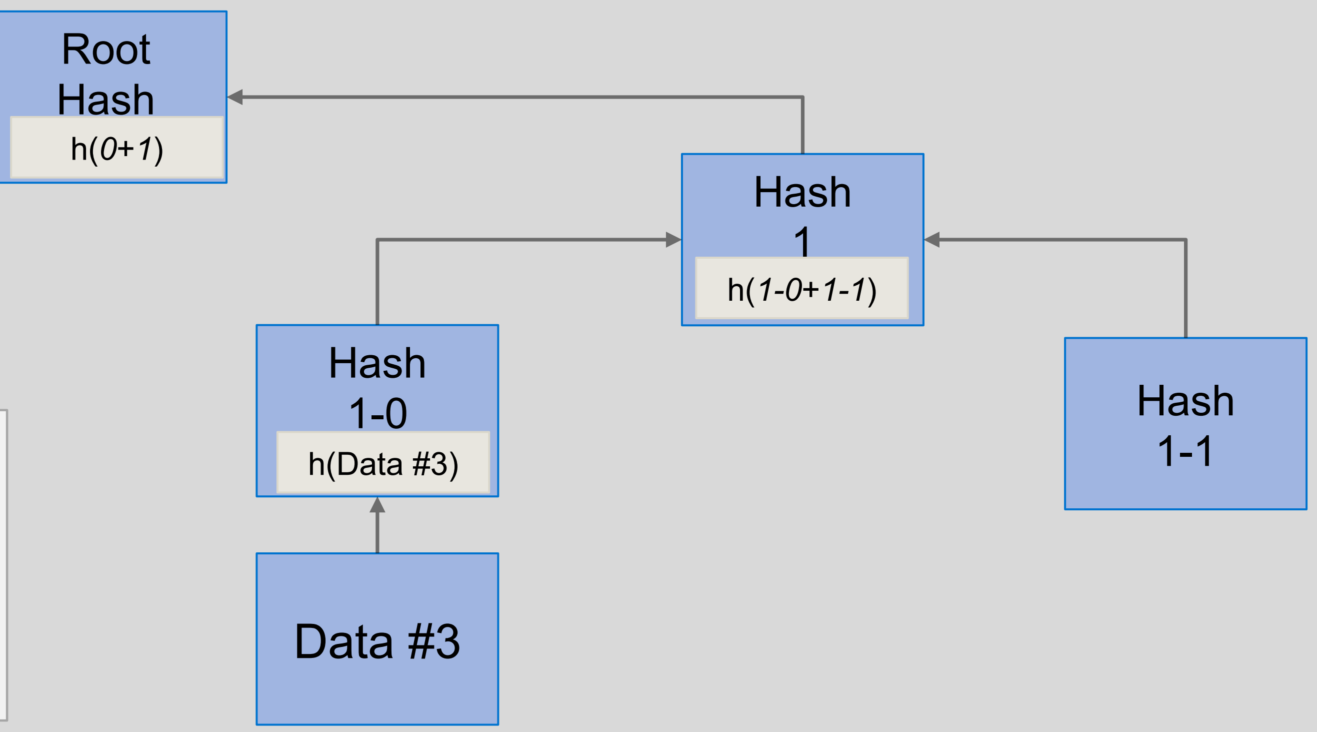
- A Merkle Tree[1] is a **data structure** using cryptographic hashes, basically a **binary tree** with **hash pointers**. It is used as an **efficient and secure way** to **verify** large data structures.
- It especially provides an efficient way to
  - proof that a certain data block is contained in a Merkle Tree (*Proof of Membership*)
  - proof that a certain data block is **not** contained in a *sorted* Merkle Tree (*Proof of Non Membership*)



[1] Sometimes also referred to as hash tree.

Proof of Membership

- We want to ensure that a certain data block is contained in the Merkle Tree without hashing the complete tree.
- Only the hashes of corresponding nodes and leaves have to be checked / validated (without disclosing other content).
- E.g., we want to evaluate whether “Data #3” is contained in the Merkle Tree or not.
- This enables verification in *log(n)* time.



Why?
This is especially useful for so-called *SPV-Clients*[1], as they have limited resources to validate the contents of the whole Blockchain.
We will see this later.

[1] Simple Payment Verification

Details about the implementation of Merkle Trees in Bitcoin can be found at the Bitcoin developer reference documentation.

Outline

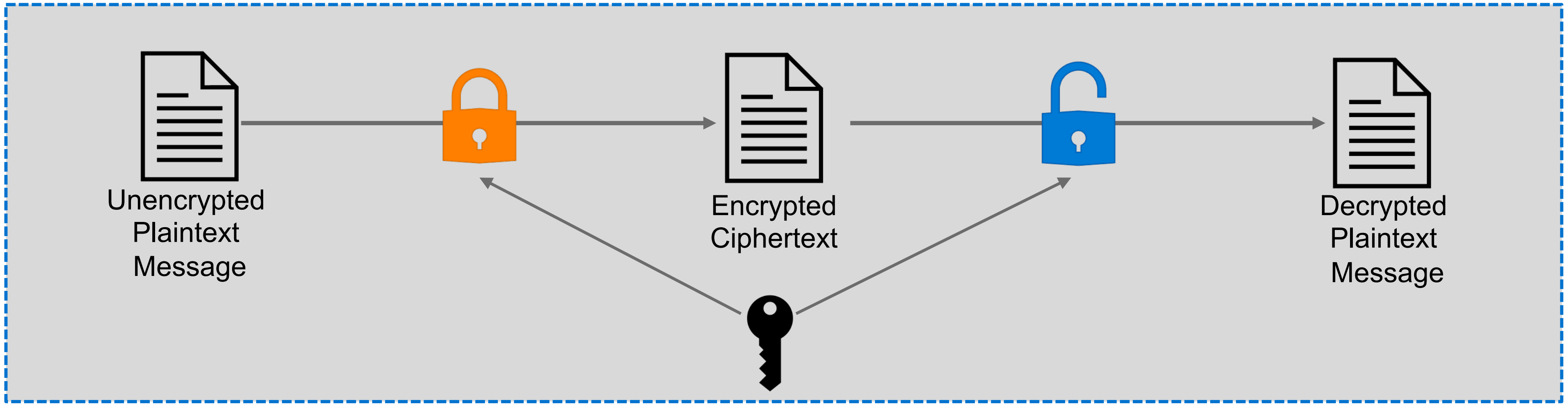
This chapter is heavily inspired by two high quality lectures @ TUM: : [IN2209] IT-Security by C. Eckert and T. Kittel & [IN2101] Network Security by G. Carle and H.Niedermayer
Please also take a look into "IT-Sicherheit: Konzepte, Verfahren, Protokolle" by C. Eckert and „Bitcoin and Cryptocurrency Technologies" by Arvind Narayanan which cover this topic, too.

Symmetric Cryptography

The system is called **symmetric**, in terms of two qualities:

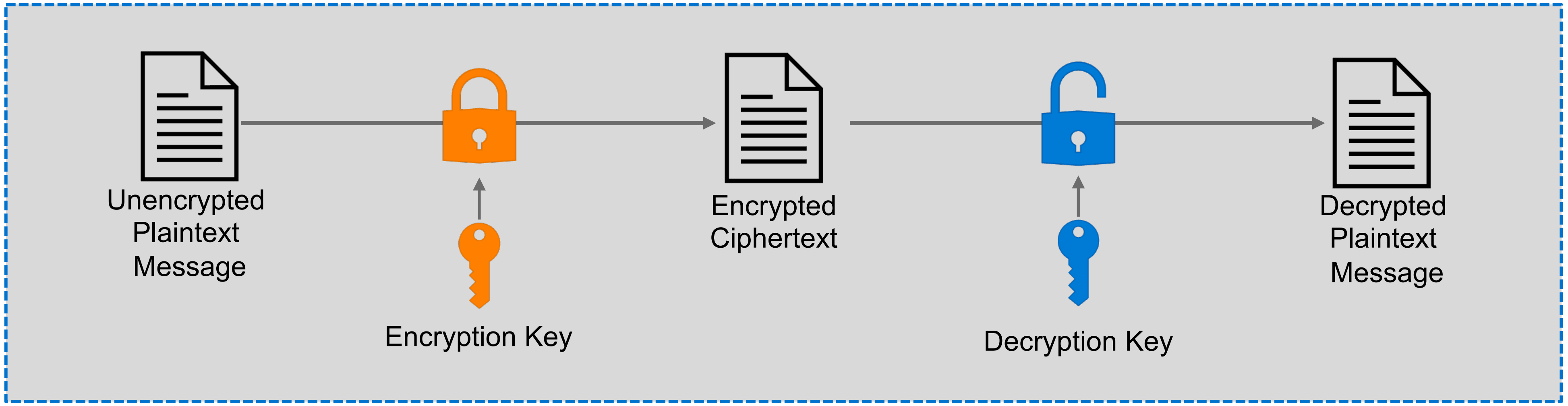
- Encryption and decryption are done using the same secret key.
- The encryption and decryption functions are similar.

- The key must be **exchanged** between the organizations using symmetric encryption so that it may be utilized in the decryption process.
- With symmetric encryption, data is changed to a form that cannot be understood by anybody who does not have the secret key to decrypt it. When the receiver with the key receives the message, the algorithm reverses its action so that the message is returned to its original.



Asymmetric Cryptography

- In **asymmetric** cryptography, **pairs of related keys** are used (one **public** and one **private key**). To generate these key pairs, one-way functions are utilized. One sender cannot read the messages of another sender, even if both have the receiver's public key, due to the one-way nature of the encryption algorithm.
- It is not essential for the person who encrypts the message to have a secret key. The important part is that the receiver can only decrypt the message using a secret key. The receiver publishes a public encryption key that is known to everyone and also has a matching private key for decryption.
- Asymmetric cryptography is often used to authenticate data using digital signatures.



Digital Signature Algorithms

Two major digital signature schemes are available:

- **RSA**-based signature schemes, such as RSA-PSS
  - Invented 1997 by Rivest, Shamir and Adleman
  - Based on the assumption that the factorization of large prime number multiplicated is very hard, but easy with additional information (so called trapdoor one-way-functions)
- **ECC**-based signature schemes, such as ECDSA
  - Suggested independently by Neal Koblitz and Victor S. Miller in 1985
  - Based on discrete logarithms
- The BSI recommends following key sizes for asymmetric cryptography
  - RSA: min. 2048 Bit
  - ECDSA: min. 256 Bit

Note:

- Many signature algorithms are **based on entropy**
  - We need a good source of entropy, otherwise private keys can be leaked.
- Digital signatures can only **sign a small amount** of **data**
  - Signing the hash of the message is sufficient, as the hash function is collision resistant.

An informative video about elliptic curves: *https://www.youtube.com/watch?v=NF1pwjL9-DE&ab_channel=Computerphile*

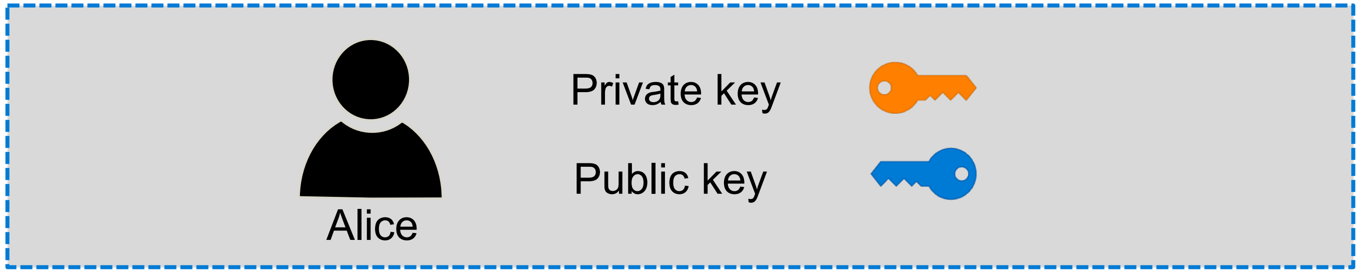
Bundesamt für Sicherheit in der Informationstechnik: Kryptographische Verfahren: Empfehlungen und Schlüssellängen, *see* *https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf*

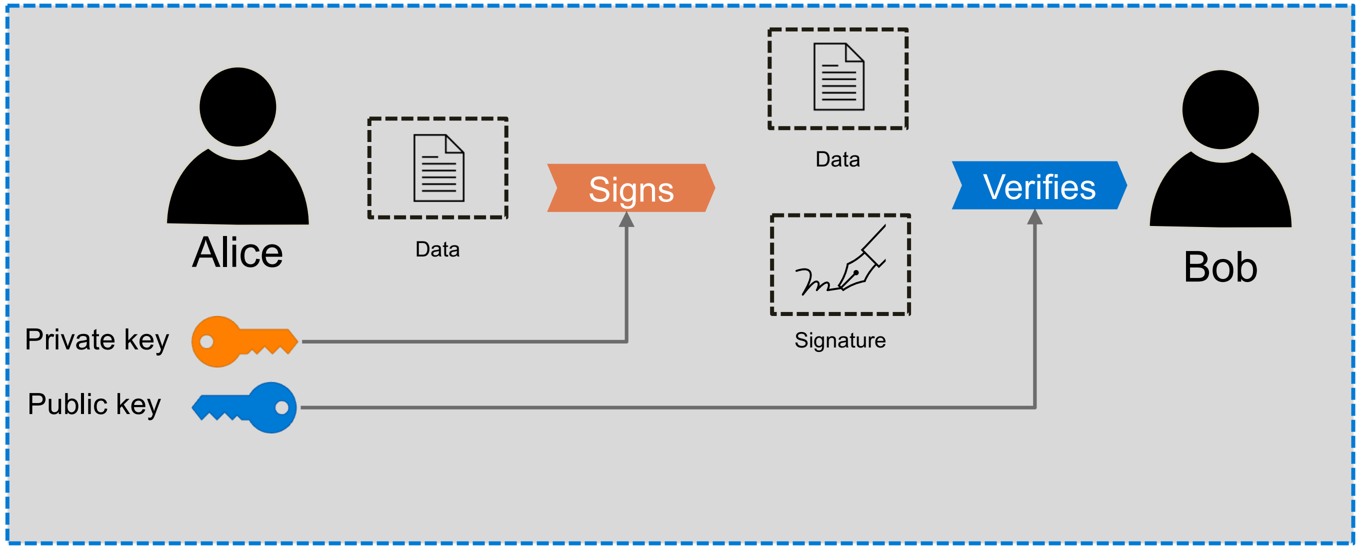
Outline

This chapter is heavily inspired by two high quality lectures @ TUM: : [IN2209] IT-Security by C. Eckert and T. Kittel & [IN2101] Network Security by G. Carle and H.Niedermayer
Please also take a look into "IT-Sicherheit: Konzepte, Verfahren, Protokolle" by C. Eckert and „Bitcoin and Cryptocurrency Technologies" by Arvind Narayanan which cover this topic, too.

Digital Signatures

- A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document.
- Digital signatures are based on **asymmetric cryptography** algorithms like RSA or ECC (Due to smaller key sizes in ECC, Bitcoin uses ECDSA).
- We need two properties of (analogue) signatures to hold in the digital world:
  - **Only** an **entity** is able to **create** a **signature** of its own, but **everyone** can **verify** it.
  - This **signature** is **tied to data** that gets signed. A signature cannot be used for different data.





Digital Signatures (cont.)

Three algorithms of *digital signature scheme*:

- (*sk*, *pk*) := **generateKeys**(*keysize*)
  - *sk* is the secret key and is used to sign messages. *pk* is the public key and is given to everyone. With the pk, they can verify the signature.
- sig := **sign**(*sk*, *message*)
  - The sign method takes the *message* and the secret key, *sk*, as input and returns a signature for *message* under *sk*.
- isValid := **verify**(*pk*, *message*, *sig*)
  - The verify method takes a *message*, a *signature*, and a *public key* as input. It will return *true* if the signature was generated out of the message and the secret key, otherwise *false.*
- Such that verify(pk, message, sign(sk, message)) == true and **signatures are unforgeable**

Unforgeability:

- The attacker knows your public key *pk*.
- The attacker sees your signature *sig* on an arbitrary amount of *messages*
- Unforgeable means, that the attacker **is not able** to create a signature on a message that he has not seen.

Identity Creation with Digital Signatures

- Digital Signature Schemes can be used as **identity systems**
  - The public key ***pk*** acts as an **identity**
  - The private key ***sk*** is the **password** to this identity to act on behalf of this identity

- This has some **advantages**:
  - **New identities** can be generated at will with **generateKeys** from our digital signature scheme
  - At first, these new identities cannot be used to uncover your real-world identity[1]

- Additionally:
  - You want to **hash** your public key pk in order to receive an "identity", as
    - Public keys are very large
    - Public keys may be vulnerable to quantum computing attacks[2]

- To validate a statement, one has to check
  1. if the *pk* hashes to the identity and
  2. if the message verifies under the public key pk.

Practical Concerns:

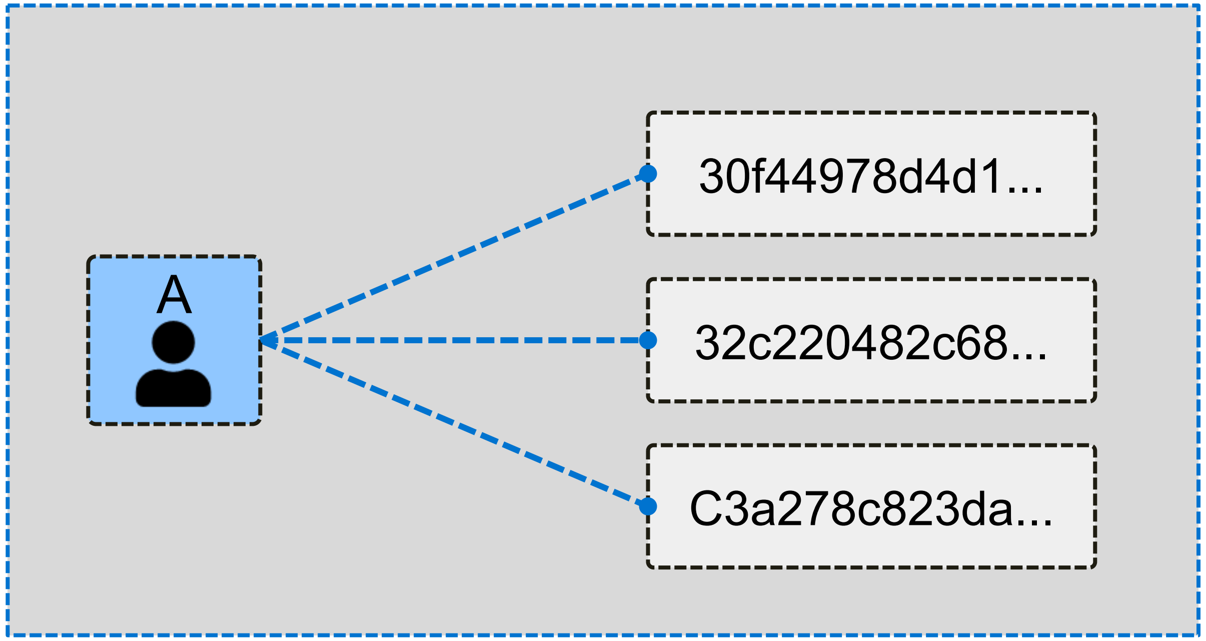
- The **private keys** are **not recoverable**. Once the file is lost, there is no way to act under this entity, can result in lost money, assets, or more.

- An **appropriate key length** should be considered. If the key length is too short, it could be computed in the future.

[1] Your statements may leak information, allowing to connect your real world identity to pk. *You are pseudonymous.*
[2] This is covered in the next section

Decentralized Identity Management

- This approach enables a decentralized identity management
  - No need for registering at a central authority
  - **Arbitrary amount** of identities
  - **Simple verification**



- All cryptocurrencies / blockchain-based systems handle it this way.

- The **address** is (in Ethereum) the **hash of a public key**.

Outline

This chapter is heavily inspired by two high quality lectures @ TUM: : [IN2209] IT-Security by C. Eckert and T. Kittel & [IN2101] Network Security by G. Carle and H.Niedermayer
Please also take a look into "IT-Sicherheit: Konzepte, Verfahren, Protokolle" by C. Eckert and „Bitcoin and Cryptocurrency Technologies" by Arvind Narayanan which cover this topic, too.

Quantum Resistance of Signature Schemes and Hash Functions

- **Signature Schemes** based on the integer factorization problem, the discrete logarithm problem, or the elliptic curve discrete logarithm problem **can be solved** with Shor’s algorithm with enough powerful quantum computer. [SHOR1999]

- **Hash functions** like SHA3 are considered to be **relatively secure** against quantum computers. [Bern2009]

- Can decentralized identity management work in a post-quantum world?
  - **Assuming hashing is not broken**, as long as a public key is not known to a hash of a public key, it is computationally infeasible to calculate the private key. Thus, users can securely receive coins as long as their **public key is unknown**.
- Can Bitcoins be stolen? How can we prevent them from being stolen?
  - If an address only receives coins and **never signs a transaction**, then **it won't expose its public key**. Thus, the public key will remain unknown.
  - Once you sign a transaction and publish it, you release all the information needed (public key and signature) to the public. Then, a malicious entity with quantum-computing capabilities can **recover your private key from your public key** and **your signature can be forged**. Thus, **your Bitcoins can be stolen**!
  - If the quantum computer takes longer than 1-2 minutes to compute your private key, then you can transfer your Bitcoins if you always use a new address (to transfer, but also as a return address).
- **In Bitcoin, it is considered bad hygiene to reuse addresses. In a post-quantum world, it will get your funds stolen!**

[SHOR1999] Shor, Peter W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." SIAM review 41.2 (1999): 303-332.

[BERN2009] Bernstein, Daniel J. "Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete." SHARCS 9 (2009): 105.