Cryptographic Basics

Gallersdörfer, U., Holl, P., & Matthes, F. (2019). "Blockchain-based Systems Engineering“. Lecture Slides. TU Munich.

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

Outline

1. Cryptographic hash functions
   - Properties of cryptographic hash functions
   - Additional properties of hash functions for usage in cryptocurrencies and Blockchain
   - Applications
   - SHA-family

2. Hash pointers & data structures
   - Hash pointers
   - Blockchains
   - Merkle Trees

3. Bloom filters

4. Digital signatures

5. Digression: Quantum resistance of signature schemes and hash functions

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..

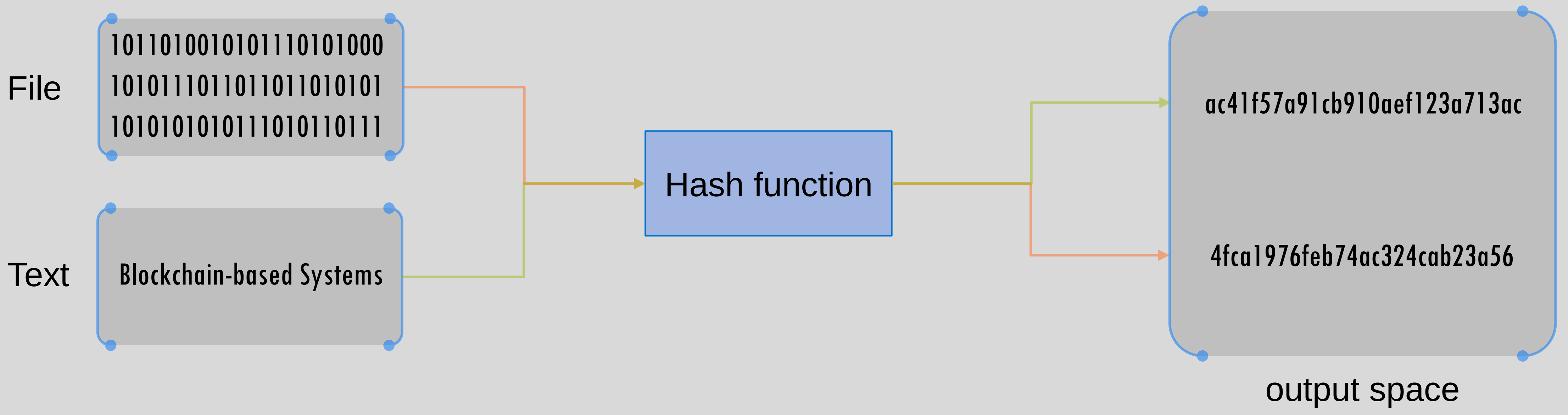
CC BY-SA 4.0

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

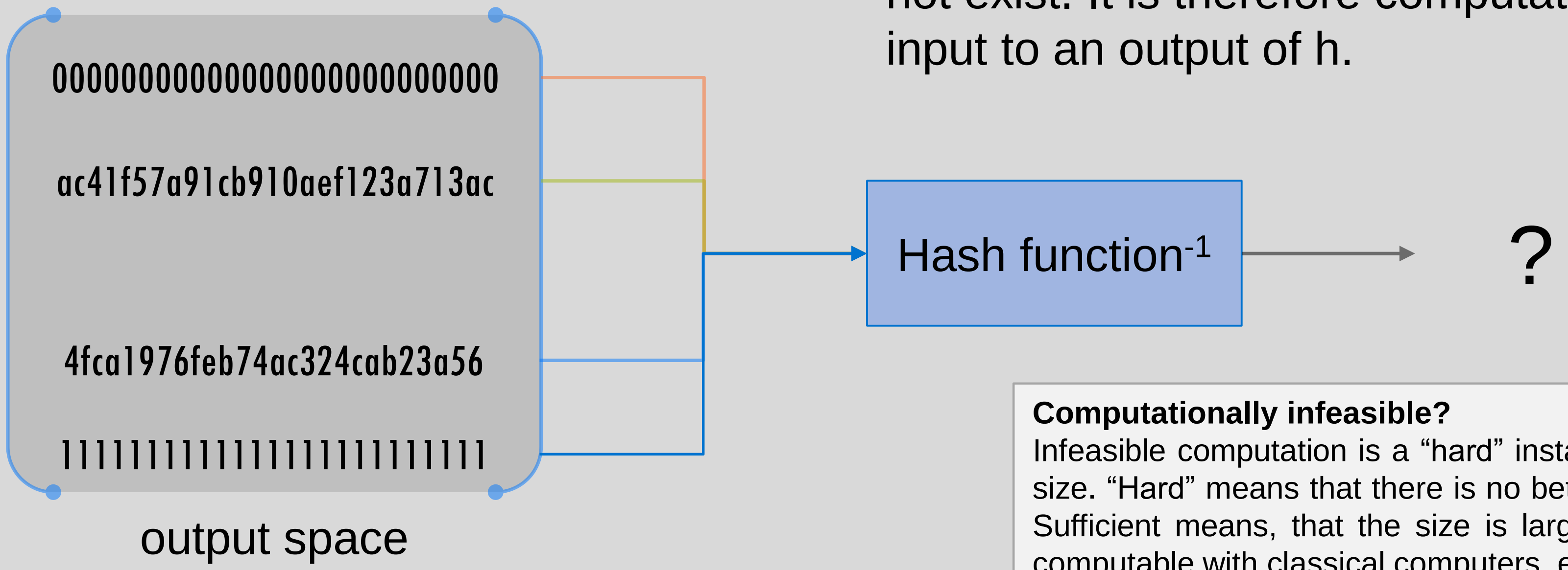
CC BY-SA 4.0

Additional properties for cryptographic hash functions

Definition: Pre-image resistance

- *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.

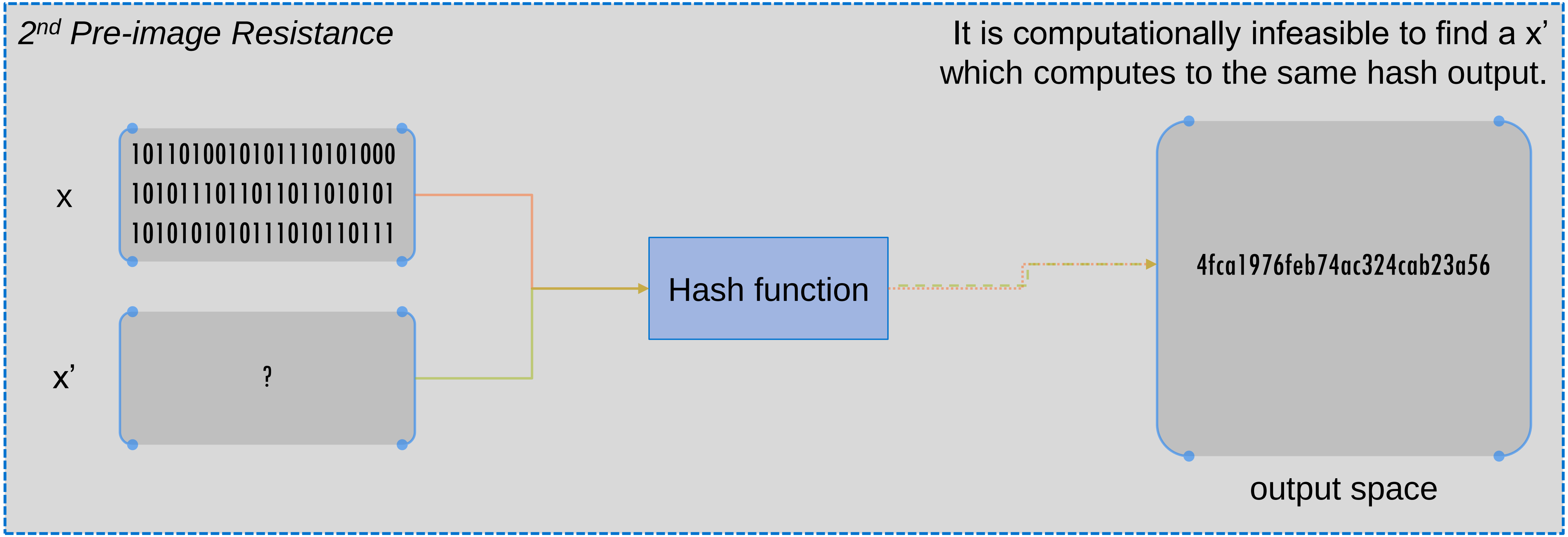
Computationally infeasible?
Infeasible computation is a “hard” instance of an NP-Hard problem of sufficient size. “Hard” means that there is no better way than trying all possible solutions. Sufficient means, that the size is large enough that it can be considered not computable with classical computers,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

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 Unpublished.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

CC BY-SA 4.0

Additional properties for **cryptographic** hash functions (cont.)

Definition *2nd Pre-image Resistance*
Given *x* it is computationally infeasible to find any second input *x'* with *x != x'* such that *h(x) = h(x')*.

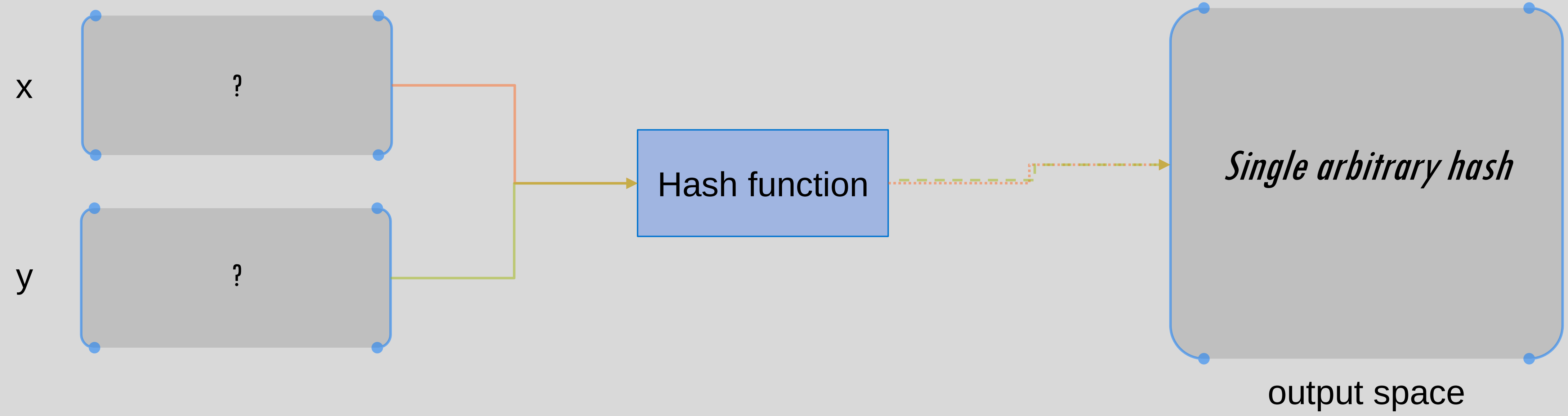


CC BY-SA 4.0

Additional properties for **cryptographic** hash functions (cont.)

Definition *Collision Resistance*
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).

Collision Resistance It is infeasible to find two values that hash to the same output.

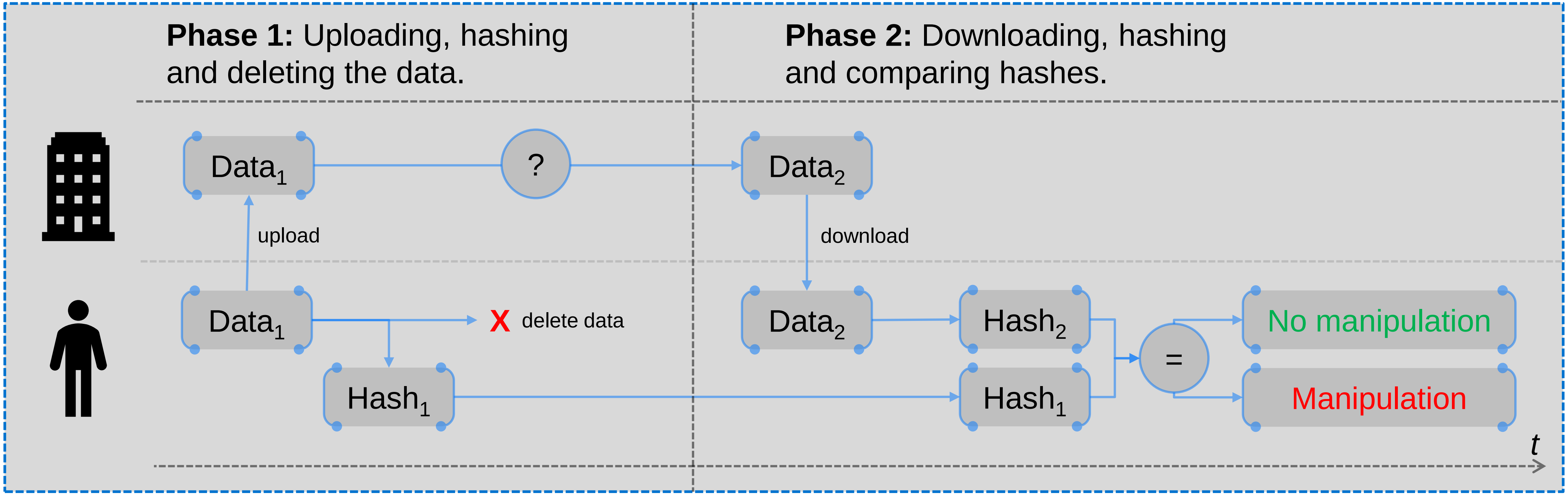


CC BY-SA 4.0

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 preimage 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.



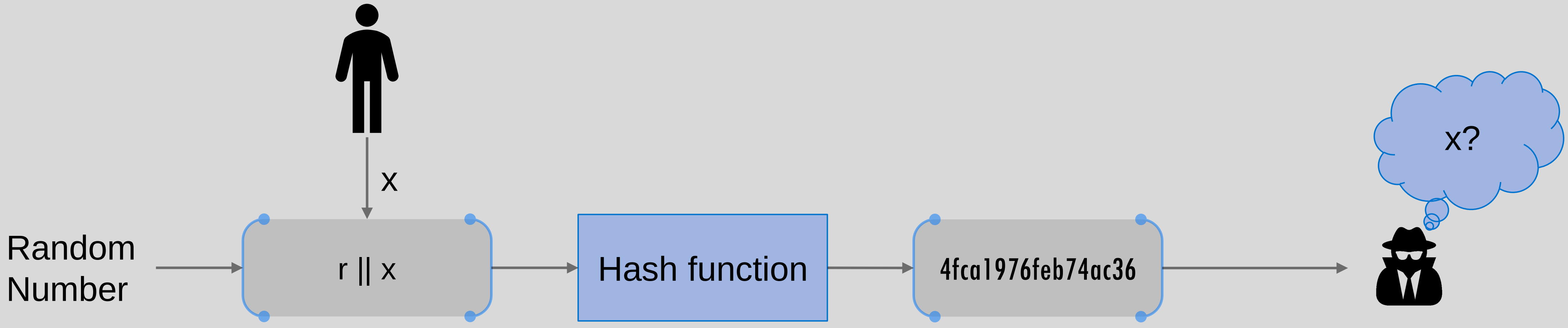
 CC BY-SA 4.0

Hiding is an additional desirable property for cryptocurrencies or Blockchain

Definition *Hiding*
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

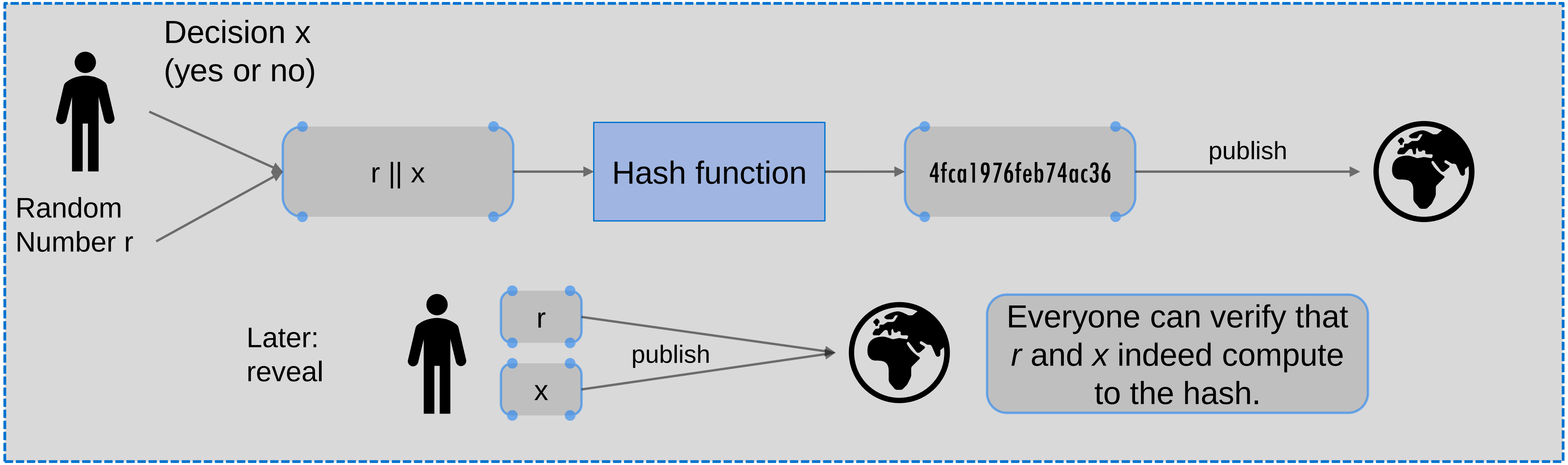
CC BY-SA 4.0

Application: Commitments

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

Commitment Scheme
Two algorithms:

- com := **commit**(*msg, nonce*)
  - *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*.



CC BY-SA 4.0

Application: 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

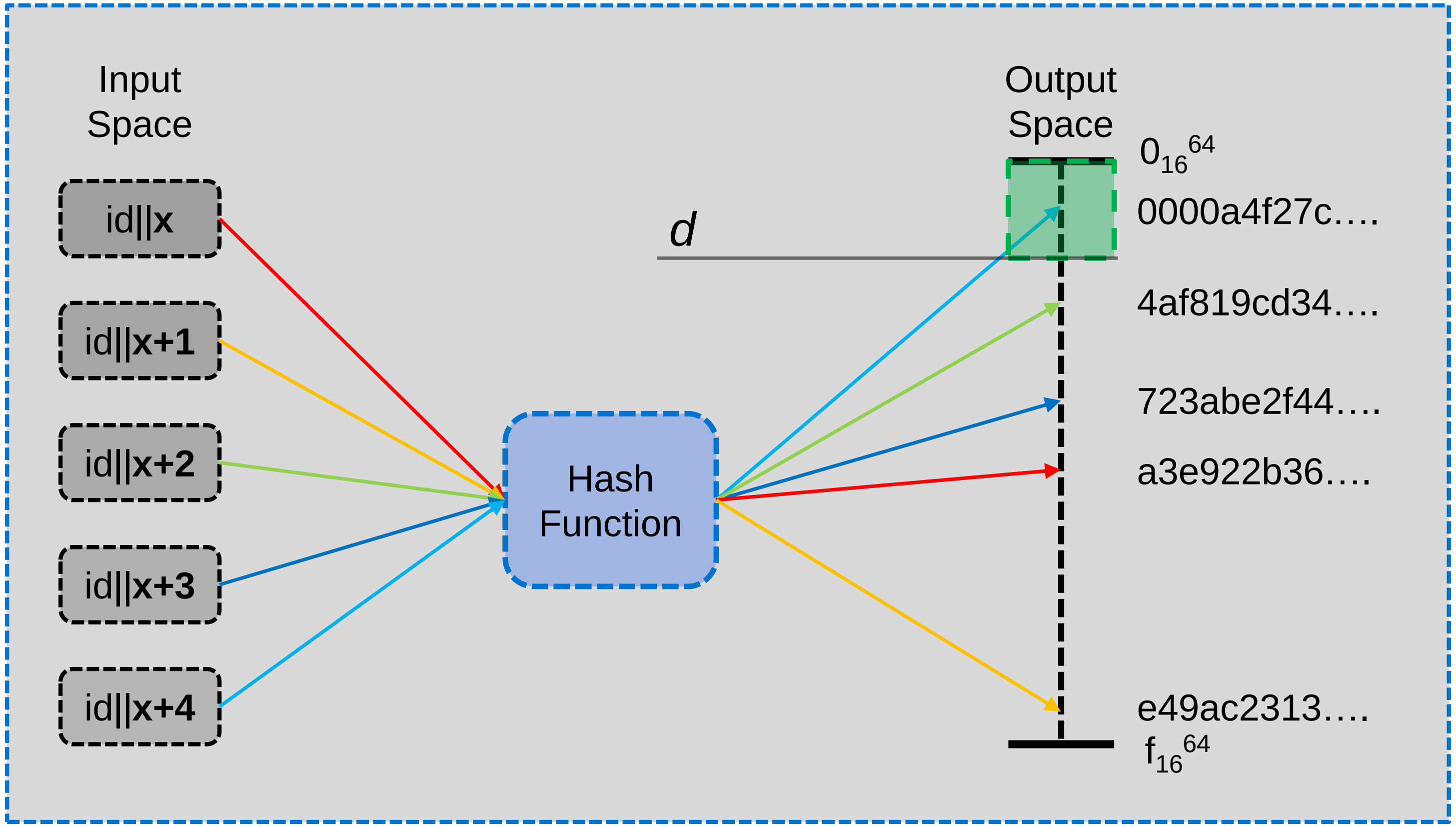
A search puzzle is a mathematical problem which requires searching a very large space in order to find a solution. In particular, there are no shortcuts in finding the solution. h has an n-bit output, therefore it can take any of 2^n values. Solving the puzzle requires finding an input so that the output 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.

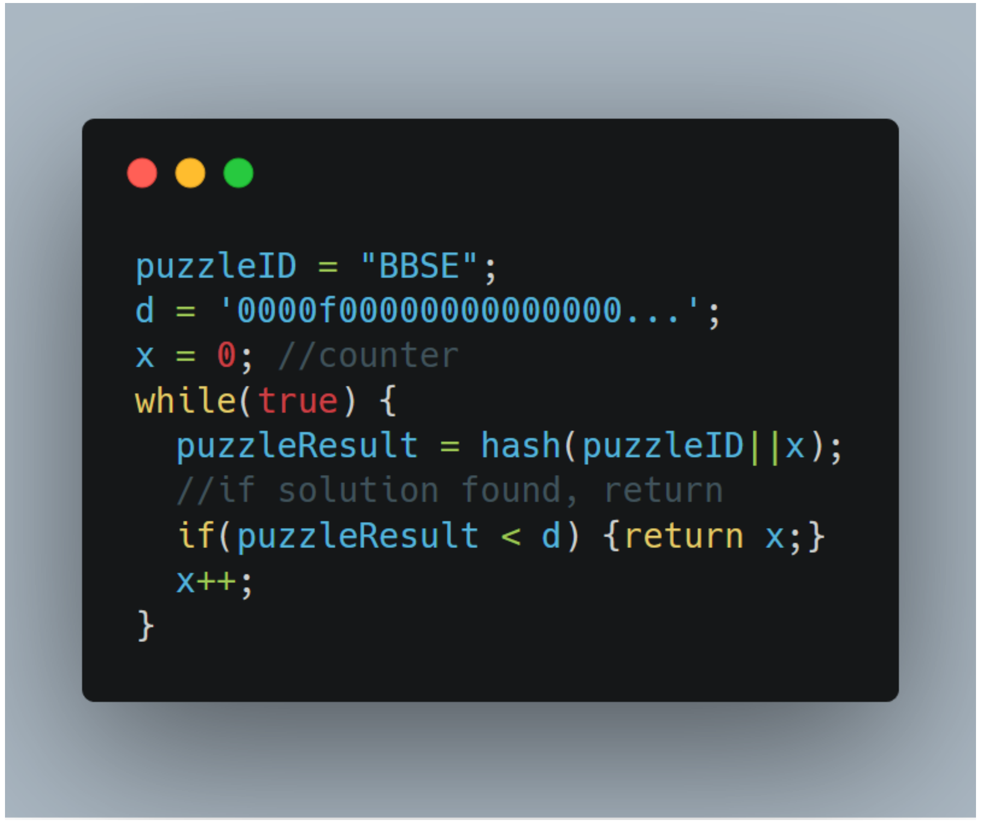
A demo can be found here: https://anders.com/blockchain/block.html

CC BY-SA 4.0

Application: 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.





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.

CC BY-SA 4.0

Cryptographic hash functions – 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 NIST[1]. 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.

[1] National Institute for Standards and Technology

CC BY-SA 4.0

Outline

1. Cryptographic hash functions
   - Properties of cryptographic hash functions
   - Additional properties of hash functions for usage in cryptocurrencies and Blockchain
   - Applications
   - SHA-family

2. Hash pointers & data structures
   - Hash pointers
   - Blockchains
   - Merkle Trees

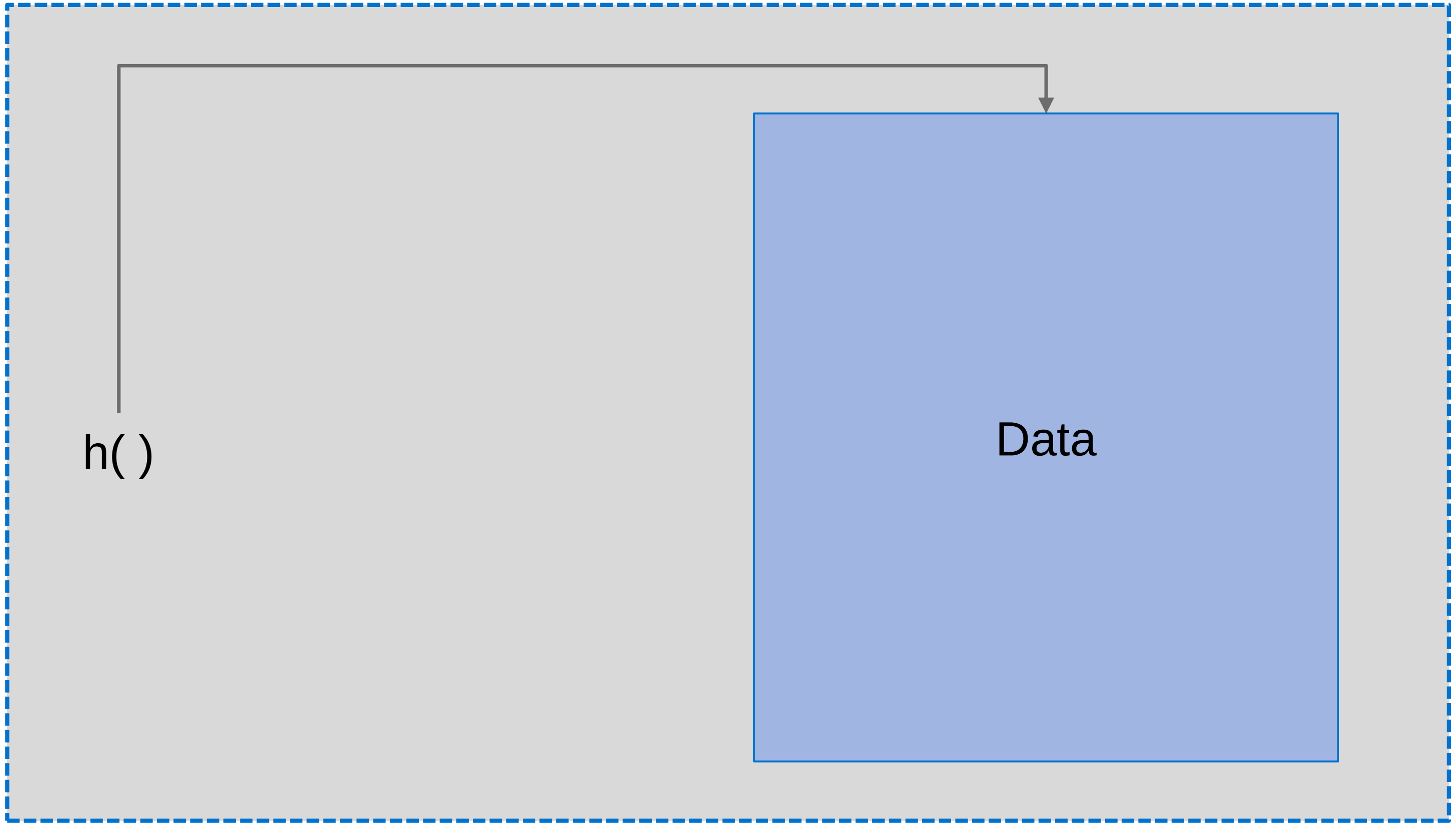
3. Bloom filters

4. Digital signatures

5. Digression: Quantum resistance of signature schemes and hash functions

CC BY-SA 4.0

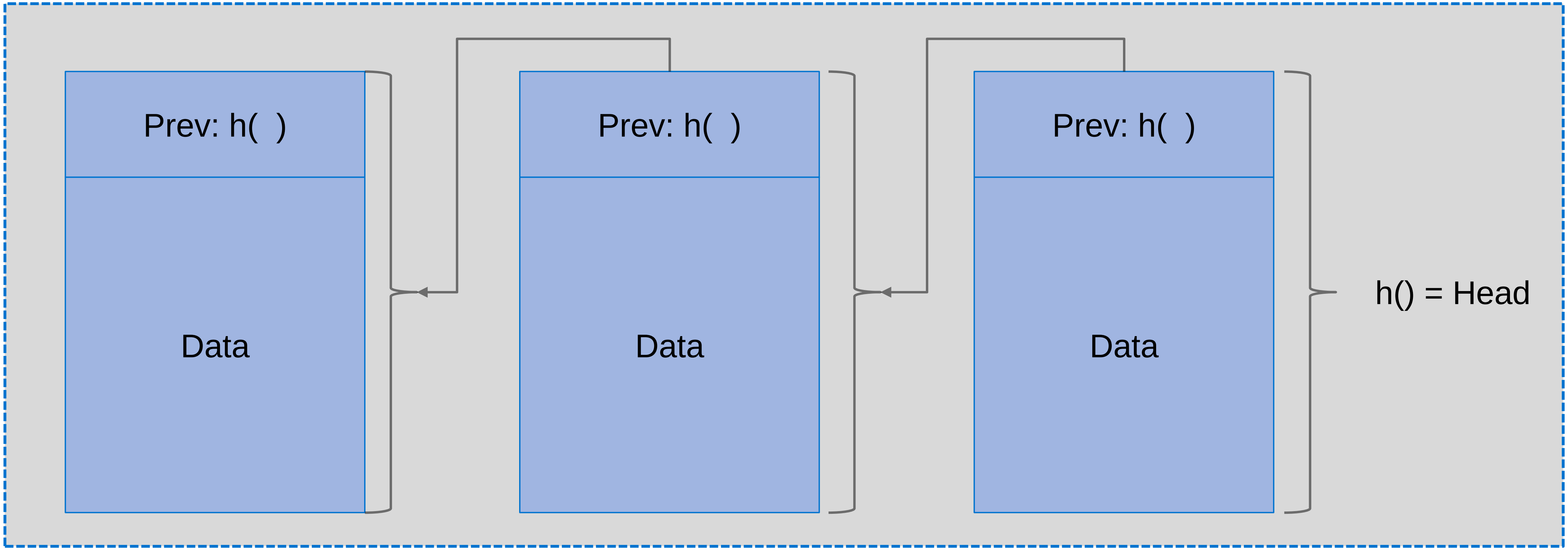
Hash pointer



A hash pointer contains the information to the location of the data enriched with a cryptographic hash of it. The difference between a regular pointer and a hash pointer is, that a hash pointer allows you to verify that the information has not changed.

CC BY-SA 4.0

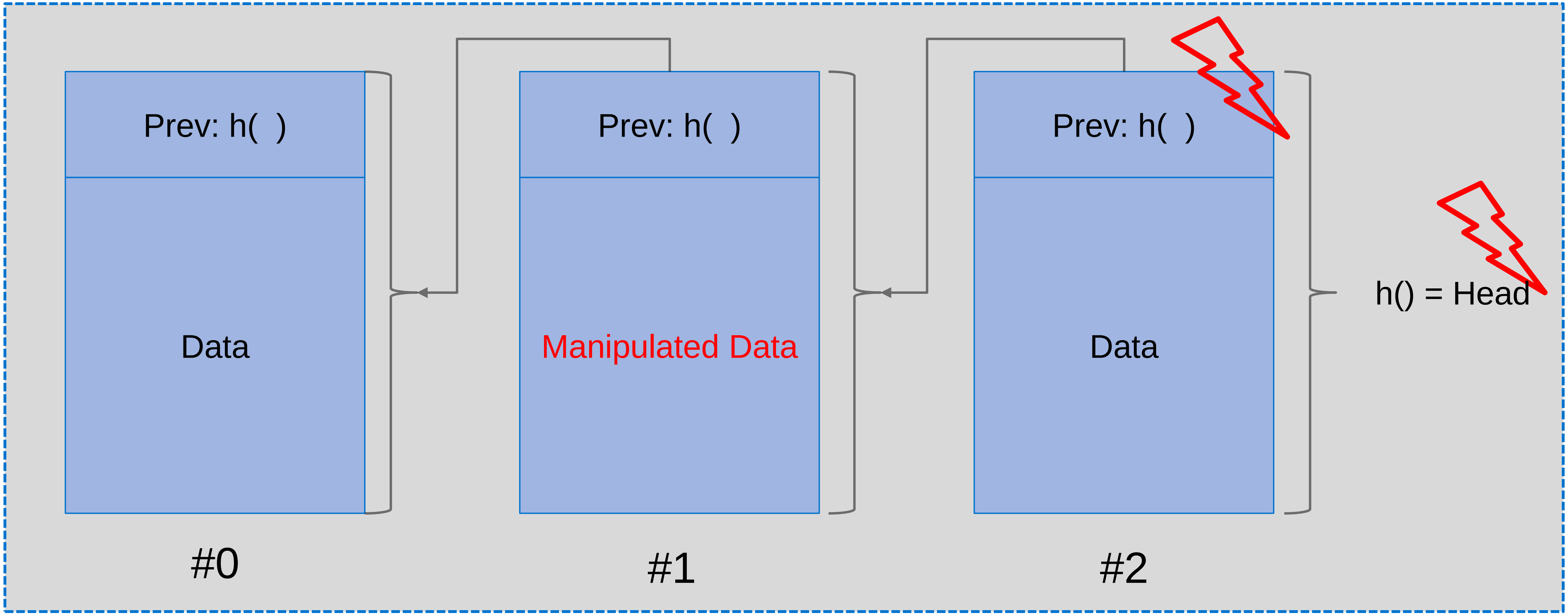
Blockchain



A linked list of hash-pointer 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, as with a recalculation of all hashes, the hashes in blocks after the manipulation changes. By only storing the head of the Blockchain, the integrity is ensured completely.

CC BY-SA 4.0

Blockchain (cont.)

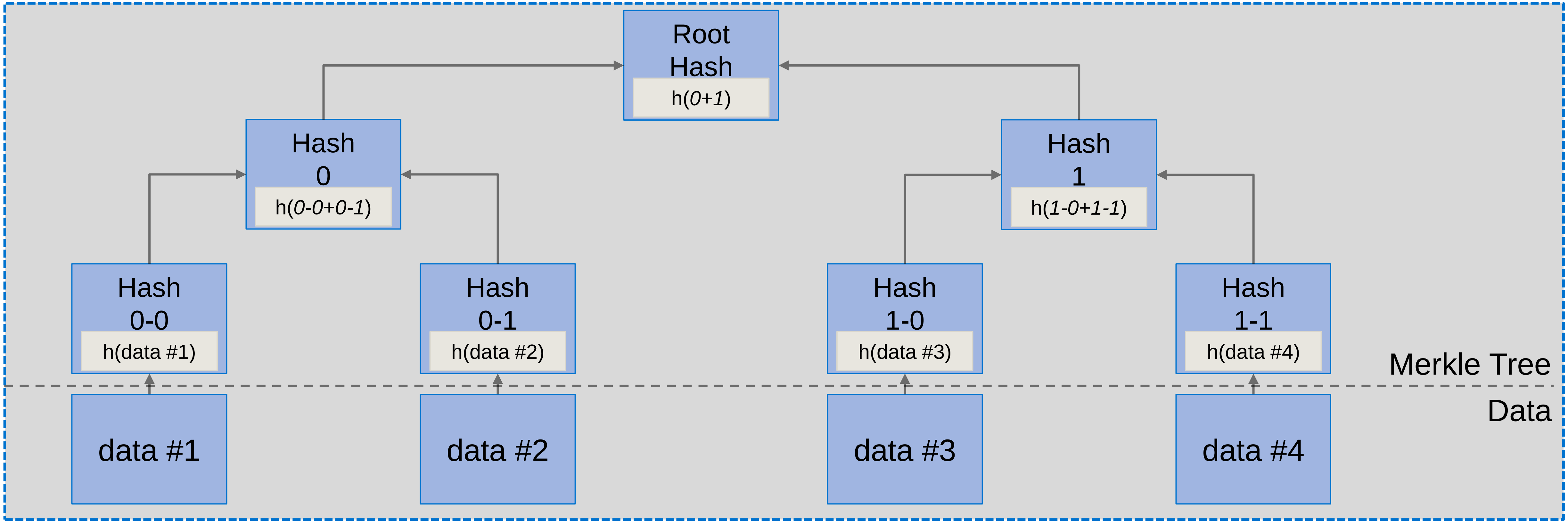


Example of manipulated data in Block #1.

CC BY-SA 4.0

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*)

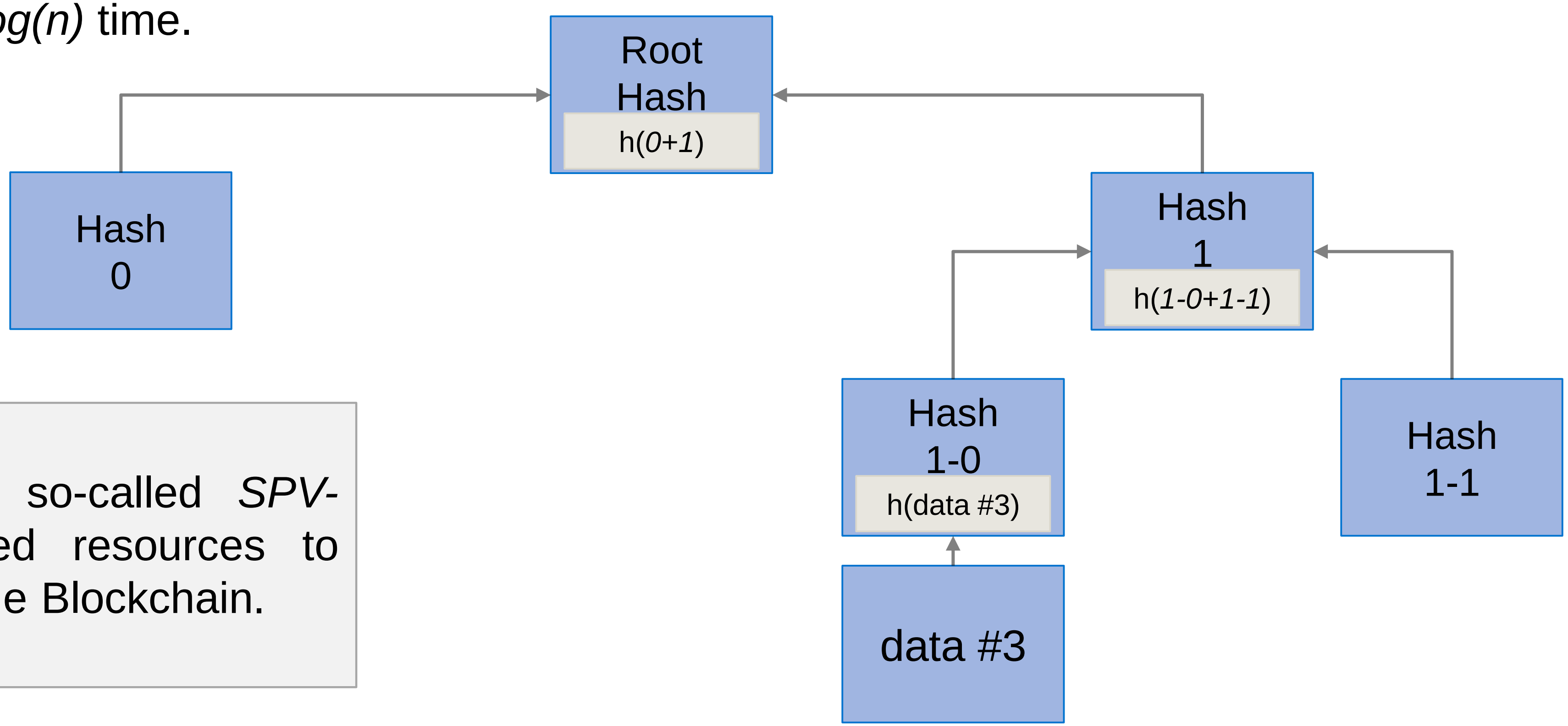


[1] Sometimes also referred to as hash tree.

CC BY-SA 4.0

Merkle Trees (cont.) – 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].

CC BY-SA 4.0

Outline

1. Cryptographic hash functions
   - Properties of cryptographic hash functions
   - Additional properties of hash functions for usage in cryptocurrencies and Blockchain
   - Applications
   - SHA-family

2. Hash pointers & data structures
   - Hash pointers
   - Blockchains
   - Merkle Trees

3. Bloom filters

4. Digital signatures

5. Digression: Quantum resistance of signature schemes and hash functions

CC BY-SA 4.0

Bloom filters

A Bloom filter is a probabilistic data structure which allows to test if an element is a member of a set.

How would you check if an element is a member of a set?

<u>Traditional approach: e ∈ S?</u>

Set S: { Apple;Apricot;Avocado;Banana;Blackberry;Blueberry;Coconut;Cranberry;Cucumber;Damson;Dragonfruit;Grape;Raisin;Grapefruit;Lemon;Lime;Lychee;Mango;Melon;Cantaloupe;Honeydew;Watermelon;Miracle fruit }

Element e: { Lime }

```
function isElem (_elem) {
  foreach(elem in set) {
    if(elem == _elem) {
      return true;
    }
  } return false;
}
```

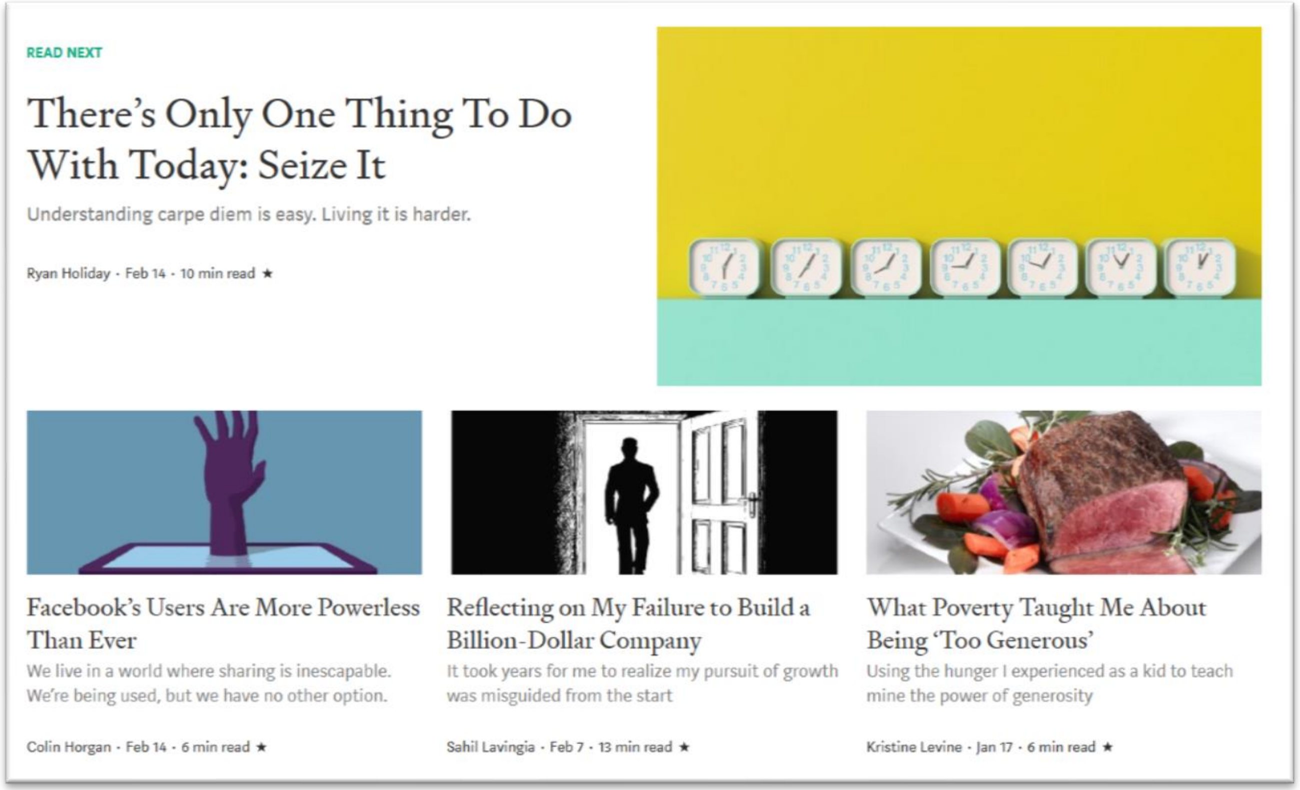
➔ If the set grows larger, membership checks for multiple items can be time-consuming.

CC BY-SA 4.0

Example: Recommend articles in a news page the user has not seen.

You are the web developer for the website of a news paper. At the bottom of each article, the website displays recommended articles to read for the user. The algorithm for good recommendations works fine already, however does not take into account if an article has been read yet. If it has, it should not be displayed.

With thousands individual users each month which read many stories over time, storing all reads and testing if an article is in this set is too slow for a good surfing experience on the website of the newspaper.



What to do?

Inspired by Jamie Talbot's article "What are Bloom filters?", 2015. https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff

CC BY-SA 4.0

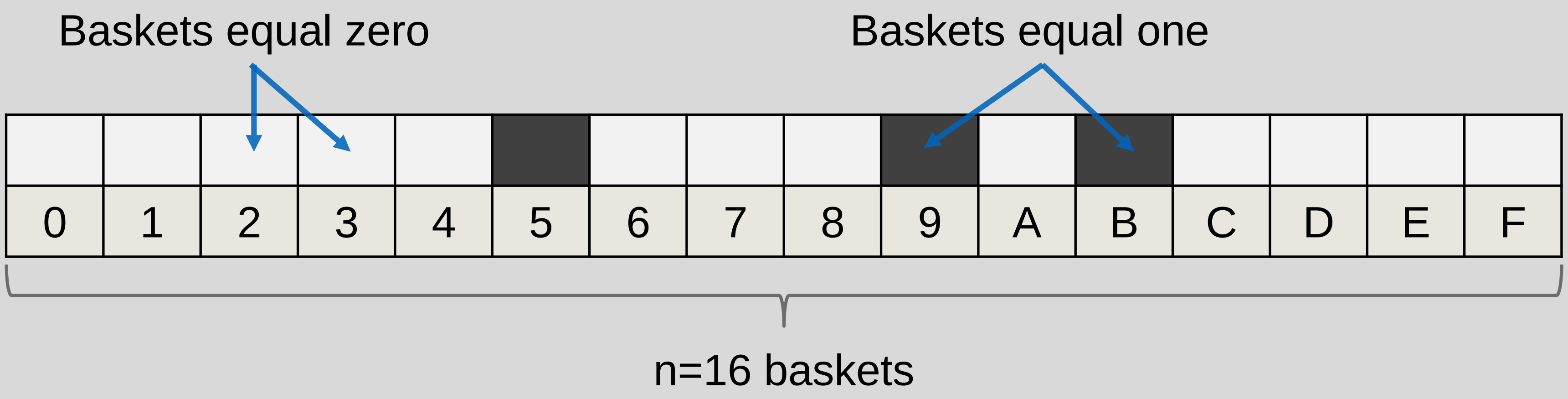
Bloom filters (cont.)

A Bloom filter is a probabilistic data structure which allows to test if an element e is a member of a set S. It is set up as a bit array.

A query to the filter either returns

- True (“e possibly in set S”)
- False (“e definitely not in set S”)

Structure of a Bloom Filter



h=3 hash functions[1]

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

[1]Out of simplicity, we only consider the first character of the resulting hash.

Bloom Filters were introduced in Bitcoin in 2012 with the BIP 37.We will learn about BIPs later.
Further information about Bloom Filters: Gervais, Arthur, et al. "On the privacy provisions of bloom filters in lightweight bitcoin clients." *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014.

CC BY-SA 4.0

Bloom filter internals: Phase 0

Set Information

Set S: { *empty* }

Element e: *empty*

Hashing results

h_1=
h_2=
h_3=

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

Explicit State

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Phase 0: Filter Setup

The filter is initialized with n buckets (0 → n-1). Each bucket is filled with zero. The hash-functions are defined.

This box explains every step.

CC BY-SA 4.0

Bloom filter internals: Phase 1

Set Information

Set S: { *apple* }

Element e: *apple*

Hashing results

h_1= 1F3870BE274F6…
h_2= D0BE2DC421BE…
h_3= 3A7BD3E2360A…

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

Explicit State

0	**1**	2	**3**	4	5	6	7	8	9	A	B	C	**D**	E	F

Phase 1: Element addition (e="apple")
We add our first element to the filter. We hash it with three different hash functions (h1, h2, h3) and set their corresponding buckets to one. Following buckets are set to one: 1, 3, and D.

CC BY-SA 4.0

Bloom filter internals: Phase 1

Set Information

Set S: { *apple; lime* }

Element e: *lime*

Hashing results

h_1= 6
h_2= C
h_3= E

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

Explicit State

0	1	2	3	4	5	**6**	7	8	9	A	B	**C**	D	**E**	F

Phase 1: Element addition (e="lime")
We add our second element to the filter. We hash it with three different hash functions (h1, h2, h3) and set their corresponding buckets to one. Following buckets are set to one: 6, C, and E.

CC BY-SA 4.0

Bloom filter internals: Phase 1

Set Information

Set S: { *apple; lime; lemon* }

Element e: *lemon*

Hashing results

$h_1 = 3$
$h_2 = D$
$h_3 = F$

$h_1 = \text{MD5}\{0\}$
$h_2 = \text{SHA-1}\{0\}$
$h_3 = \text{SHA-2}\{0\}$

Explicit State

0	1	2	**3**	4	5	6	7	8	9	A	B	C	**D**	**E**	F

Phase 1: Element addition (e="lemon")
We add our third element to the filter. Again, we hash it. This time, two out of three buckets are already set to one. In these buckets one remains.

CC BY-SA 4.0

Bloom filter internals: Phase 2

Set Information

Set S: { *apple; lime; lemon* }

Element search

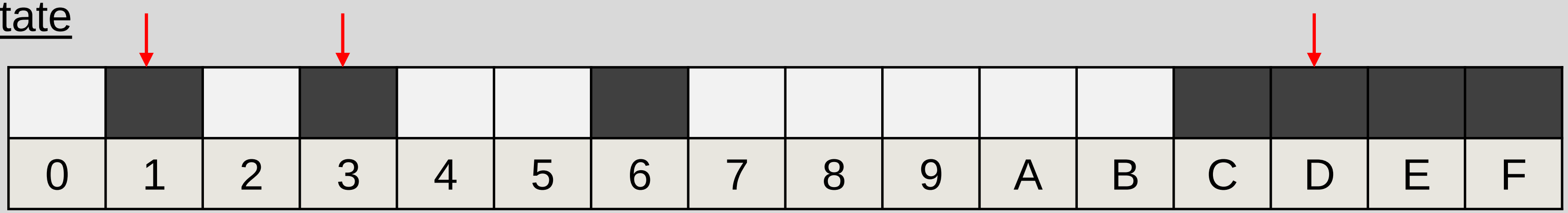
apple

Hashing results

h_1= 1
h_2= D
h_3= 3

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

Explicit State



Phase 2: Element Validation (e="apple")
Now we are able to search if an element is contained in the bloom filter. We added the apple before, therefore we receive *true* from the filter.

CC BY-SA 4.0

Bloom filter internals: Phase 2

Set Information

Set S: { *apple; lime; lemon* }

Element search

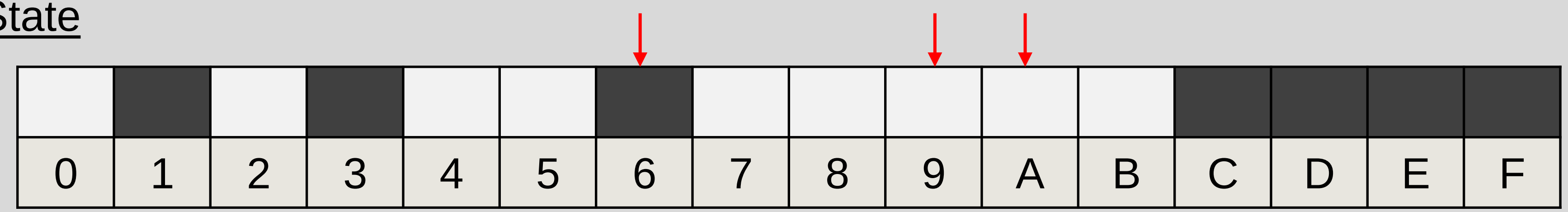
mango

Hashing results

h_1= A
h_2= 9
h_3= 6

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

Explicit State



Phase 2: Element Validation (e="mango")

Is "mango" contained in the set? Hashing mango results in A, 9, and 6, of which only one bucket is set to one.

All h buckets have to be set to one for a match.

CC BY-SA 4.0

Bloom filter internals: Phase 2

Set Information

Set S: { *apple; lime; lemon* }

Element search

grapefruit

Hashing results

h_1= 6
h_2= F
h_3= E

h_1= MD5{0}
h_2= SHA-1{0}
h_3= SHA-2{0}

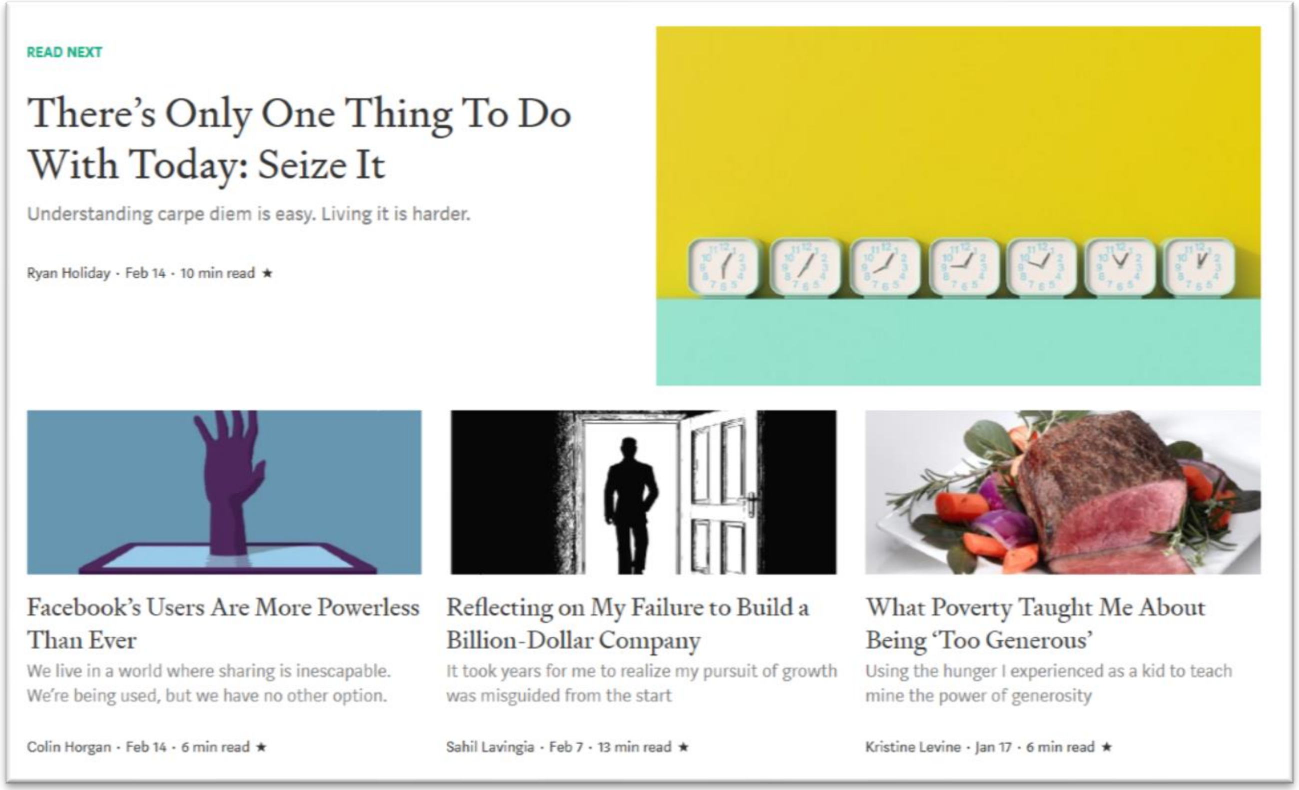
Explicit State

Phase 2: Element Validation (e="grapefruit")
Other elements might generate a false positive, as their buckets might be filled by chance. This is the case with grapefruit, as the functions generate the hashes 6, F, and E.

False positive generations occur in Bloom filters. The probability of their occurrence depends on the number of buckets n and the number of hash functions h.

CC BY-SA 4.0

Solved: Recommend articles in a news page the user has not seen.



What to do?

- Store a bloom filter for each user.
- Add each read story to the filter.
- Check the filter to find out if a story has been read:
  - True: Do not display story
  - False: Display story and store in filter

Why are false positives not a problem in this case?

CC BY-SA 4.0

Why are false positives not a problem in this case?

There is no downsides to false positives, as it eliminates articles, which have not been read by the user. If the user misses an article he did not see yet, he will not notice. In this case, it is important that the user does not see a story twice. This is ensured by the bloom filter.

CC BY-SA 4.0

Outline

1. Cryptographic hash functions
   - Properties of cryptographic hash functions
   - Additional properties of hash functions for usage in cryptocurrencies and Blockchain
   - Applications
   - SHA-family

2. Hash pointers & data structures
   - Hash pointers
   - Blockchains
   - Merkle Trees

3. Bloom filters

4. Digital signatures

5. Digression: Quantum resistance of signature schemes and hash functions

CC BY-SA 4.0

Digital signatures – Overview

- Digital signatures are based on **asymmetric cryptography** algorithms like RSA or ECC.
- 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.

Definition *Digital Signature Scheme*
Three algorithms:

- (*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**

Note, that we do not cover the basics of asymmetric encryption in this course. If you are not familiar with this topic, we strongly encourage you to look at [IN2209] IT-Security by C. Eckert and T. Kittel .

CC BY-SA 4.0

What does unforgeable mean?

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.

CC BY-SA 4.0

Digital signatures – 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

- Due to smaller key sizes in ECC, Bitcoin uses ECDSA.

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

CC BY-SA 4.0

Digital signatures – Practical concerns

- 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.

- 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. (See recommendations on previous slide)

CC BY-SA 4.0

Digital signatures for identity creation

- 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.

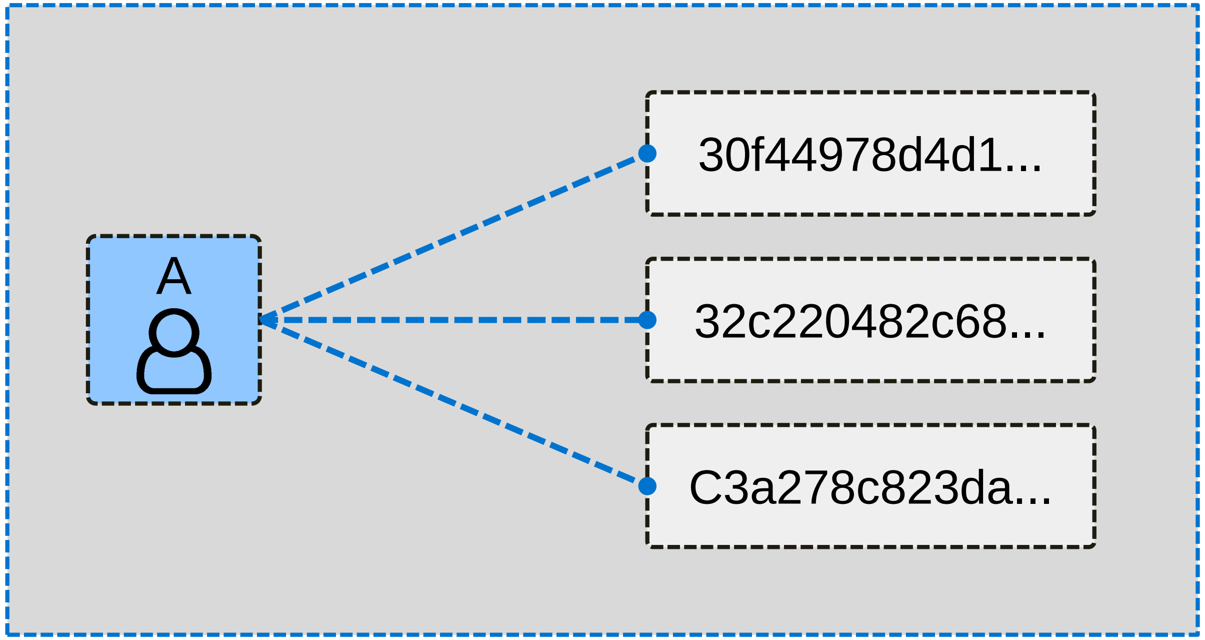
[1] Your statements may leak information, allowing to connect your real world identity to pk. *You are pseudonymous.*

[2] This is covered in the next section.

CC BY-SA 4.0

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**.

CC BY-SA 4.0

Outline

1. Cryptographic hash functions
   - Properties of cryptographic hash functions
   - Additional properties of hash functions for usage in cryptocurrencies and Blockchain
   - Applications
   - SHA-family

2. Hash pointers & data structures
   - Hash pointers
   - Blockchains
   - Merkle Trees

3. Bloom filters

4. Digital signatures

5. Digression: Quantum resistance of signature schemes and hash functions

CC BY-SA 4.0

Digression: 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 an enough powerful quantum computer [SHOR1999]

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

What are the implications?

- Can decentralized identity management work in a post-quantum world?
- Can Bitcoins be stolen? How do we prevent them from being 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.

CC BY-SA 4.0

Can decentralized identity management work in a post-quantum world?

- In the case of Bitcoin, it could work under certain assumptions:
  - Hashing itself is not broken by quantum computers
  - Therefore: 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
  - Therefore: You can transfer me Bitcoin to an address where the public key is unknown

Can Bitcoins be stolen? How do we prevent them from being stolen?

- As before: If public key unknown, then Bitcoins cannot be stolen. (If public, Bitcoins are stolen)
- How do you prevent them being stolen when you issue a transaction?
- If the quantum computer takes longer than 1-2 minutes to compute your private key, then you can transfer Bitcoin 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!

 CC BY-SA 4.0