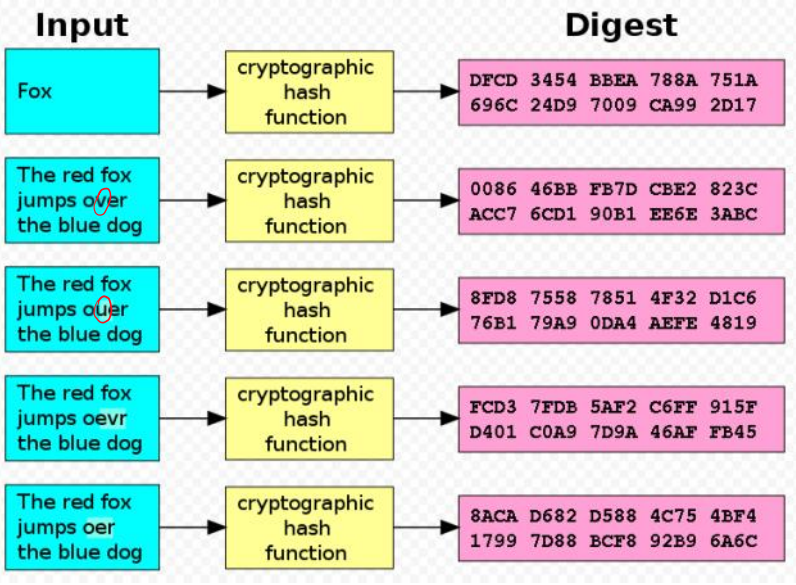
1. Rates of cryptocurrences: Bitcoin, Ethereum, Monero.
2. Biometric methods of identification together with cryptographic means of identification.
3. DRM - digital rights management system.
4. Traitor tracing in DRM.
5. Marlin DRM system.
6. Blockchain technology and smart contracts.
7. Smart contracts, application in: different course works
   - 1. Topic: in social activity!
   - 2. Topic: in Energy produc., supply system.
   - 3. Topic: in Logistic
   - 4. ---- etc.

4-th Industrial Revolution

IoT → Blockchain → Smart contract

H-Functions



One-wayness of H-function

1. It is easy to compute $H(m)$ when m is any finite message: $H(m) = h$.
2. It is infeasible to find any m' when h is given such that $H(m') = h$.

sha-256 (in Octave) → 28 least significant bits of sha-256

Is standardized H-function providing v. good randomness

```
>> sha256('Aliceaddr,Bobaddr,1BTC,nonce=1600027')
ans =
192D34BBF64C7D57291750207C8929E6295116600E653A57CE623EAA709DE657

>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1600027')
h = 09DE657
```

Merkle_Tree

Handbook of Applied Cryptography by A. Menezes, P. van Oorschot and S. Vanstone

13.4.1 Authentication trees

Authentication trees provide a method for making public data available with verifiable authenticity, by using a tree structure in conjunction with a suitable hash function, and authenticating the root value. Applications include:

1. *authentication of public keys* (as an alternative to public-key certificates). An authentication tree created by a trusted third party, containing users' public keys, allows authentication of a large number of such keys.
2. *trusted timestamping service*. Creation of an authentication tree by a trusted third party, in a similar way, facilitates a trusted timestamping service (see §13.8.1).
3. *authentication of user validation parameters*. Creation of a tree by a single user allows that user to publish, with verifiable authenticity, a large number of its own public validation parameters, such as required in one-time signature schemes (see §11.6.3).

To facilitate discussion of authentication trees, binary trees are first introduced.

Binary trees

A *binary tree* is a structure consisting of vertices and directed edges. The vertices are divided into three types:

1. a *root vertex*. The root has two edges directed towards it, a left and a right edge.
2. *internal vertices*. Each internal vertex has three edges incident to it – an upper edge directed away from it, and left and right edges directed towards it.
3. *leaves*. Each leaf vertex has one edge incident to it, and directed away from it.

The vertices incident with the left and right edges of an internal vertex (or the root) are called the *children* of the internal vertex. The internal (or root) vertex is called the *parent* of the associated children. Figure 13.5 illustrates a binary tree with 7 vertices and 6 edges.

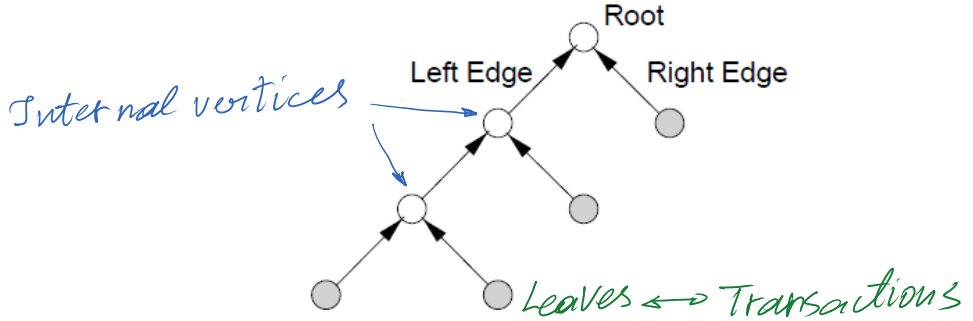
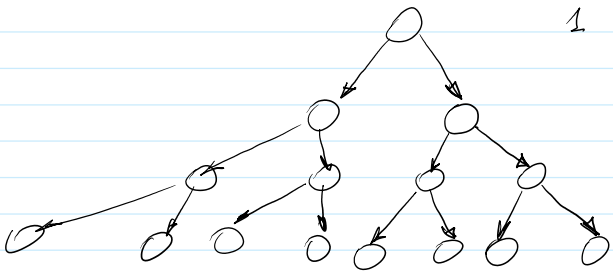


Figure 13.5: *A binary tree (with 4 shaded leaves and 3 internal vertices).*

Malthus



step 1 : $2 = 2^1$
step 2 : $4 = 2^2$
step 3 : $8 = 2^3$
step 4 : $16 = 2^4$
5 : $32 = 2^5$
6 : $64 = 2^6$
7 : $128 = 2^7$
8 : $256 = 2^8$

step n-th : 2^n

Constructing and using authentication trees

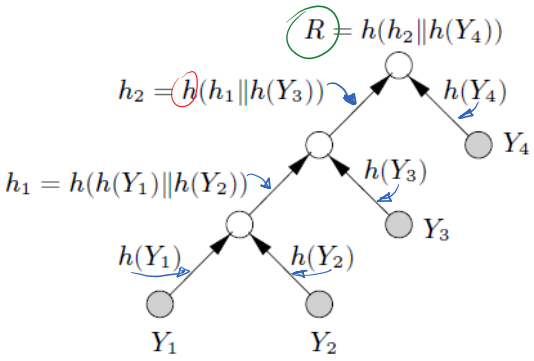
Consider a binary tree T which has t leaves. Let h be a collision-resistant hash function. T can be used to authenticate t public values, $Y_1, Y_2, \ldots, Y_t$, by constructing an *authentication tree* T^* as follows.

1. Label each of the t leaves by a unique public value Y_i. → Transaction
2. On the edge directed away from the leaf labeled Y_i, put the label $h(Y_i) = h_i$.
3. If the left and right edge of an internal vertex are labeled h_1 and h_2, respectively, label the upper edge of the vertex $h(h_1 \| h_2)$.
4. If the edges directed toward the root vertex are labeled u_1 and u_2, label the root vertex $h(u_1 \| u_2)$.

Once the public values are assigned to leaves of the binary tree, such a labeling is well-defined. Figure 13.6 illustrates an authentication tree with 4 leaves. Assuming some means to authenticate the label on the root vertex, an authentication tree provides a means to authenticate any of the t public leaf values Y_i, as follows. For each public value Y_i, there is a unique path (the *authentication path*) from Y_i to the root. Each edge on the path is a left or right edge of an internal vertex or the root. If e is such an edge directed towards vertex x, record the label on the other edge (not e) directed toward x. This sequence of labels (the *authentication path values*) used in the correct order provides the authentication of Y_i, as illustrated by Example 13.17. Note that if a single leaf value (e.g., Y_1) is altered, maliciously or otherwise, then authentication of that value will fail.

|| - concatenat. operation

$h_1 = 12AF$
$h_2 = BC7$
$h_1 \| h_2 =$
$= 12AFBC7$



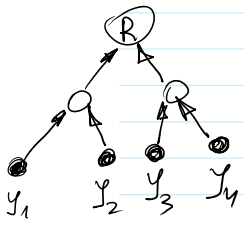


Figure 13.6: *An authentication tree.*

13.17 Example (*key verification using authentication trees*) Refer to Figure 13.6. The public value Y_1 can be authenticated by providing the sequence of labels $h(Y_2)$, $h(Y_3)$, $h(Y_4)$. The authentication proceeds as follows: compute $h(Y_1)$; next compute $h_1 = h(h(Y_1)) \| h(Y_2))$; then compute $h_2 = h(h_1 \| h(Y_3))$; finally, accept Y_1 as authentic if $h(h_2 \| h(Y_4)) = R$, where the root value R is known to be authentic. □

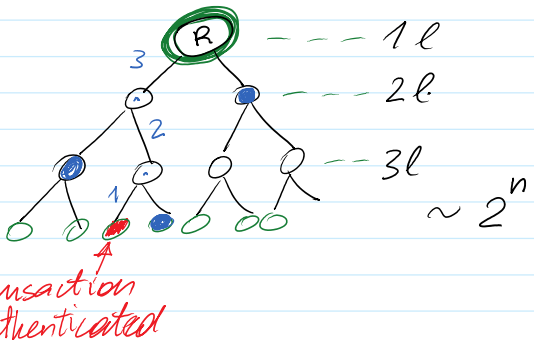
The advantage of authentication trees is evident by considering the storage required to allow authentication of t public values using the following (very simple) alternate approach: an entity A authenticates t public values $Y_1, Y_2, \ldots, Y_t$ by registering each with a trusted third party. This approach requires registration of t public values, which may raise storage issues at the third party when t is large. In contrast, an authentication tree requires only a single value be registered with the third party.

If a public key Y_i of an entity A is the value corresponding to a leaf in an authentication tree, and A wishes to provide B with information allowing B to verify the authenticity of Y_i, then A must (store and) provide to B both Y_i and all hash values associated with the authentication path from Y_i to the root; in addition, B must have prior knowledge and trust in the authenticity of the root value R. These values collectively guarantee authenticity, analogous to the signature on a public-key certificate. The number of values each party must store (and provide to others to allow verification of its public key) is $\lg(t)$, as per Fact 13.19.

13.18 Fact (*depth of a binary tree*) Consider the length of (or number of edges in) the path from each leaf to the root in a binary tree. The length of the longest such path is minimized when the tree is *balanced*, i.e., when the tree is constructed such that all such paths differ in length by at most one. The length of the path from a leaf to the root in a balanced binary tree containing t leaves is about $\lg(t)$.

$t = 7$ level $1 = 2^1$
$2 = 4$
$3 = 8$
$n \quad 2^n$

$\log_2 2^3 = 3$



13.19 Fact (*length of authentication paths*) Using a balanced binary tree (Fact 13.18) as an authentication tree with t public values as leaves, authenticating a public value therein may be achieved by hashing $\lg(t)$ values along the path to the root.

13.20 Remark (*time-space tradeoff*) Authentication trees require only a single value (the root value) in a tree be registered as authentic, but verification of the authenticity of any particular leaf value requires access to and hashing of all values along the authentication path from leaf to root.

13.21 Remark (*changing leaf values*) To change a public (leaf) value or add more values to an authentication tree requires recomputation of the label on the root vertex. For large balanced trees, this may involve a substantial computation. In all cases, re-establishing trust of all users in this new root value (i.e., its authenticity) is necessary.

The computational cost involved in adding more values to a tree (Remark 13.21) may motivate constructing the new tree as an unbalanced tree with the new leaf value (or a subtree of such values) being the right child of the root, and the old tree, the left. Another motivation for allowing unbalanced trees arises when some leaf values are referenced far more frequently than others.

Elliptic-curve cryptography (**ECC**) is an approach to public-key cryptography based on the algebraic structureof elliptic curves over finite fields.
ECC requires smaller keys compared to non-ECC cryptography (based on plain Galois fields) to provide equivalent security.[1]
Elliptic curves are applicable for key agreement, digital signatures, pseudo-random generators and other tasks. Indirectly, they can be used for encryption by combining the key agreement with a symmetric encryption scheme.
They are also used in several integer factorization algorithms based on elliptic curves that have applications in cryptography, such as Lenstra elliptic-curve factorization.

Public-key cryptography is based on the intractability of certain mathematical problems.
Early public-key systems are secure assuming that it is difficult to factor a large integer composed of two or more large prime factors.
For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible: this is the "elliptic curve discrete logarithm problem" (ECDLP).
The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points.
The size of the elliptic curve determines the difficulty of the problem.

The primary benefit promised by elliptic curve cryptography is a smaller key size, reducing storage and transmission requirements, i.e. that an elliptic curve group could provide the same level of security afforded by an RSA-based system with a large modulus and correspondingly larger key: for example, a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key.

The U.S. National Institute of Standards and Technology (NIST) has endorsed elliptic curve cryptography in its Suite B set of recommended algorithms, specifically elliptic

curve Diffie–Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signature.

The U.S. National Security Agency (NSA) allows their use for protecting information classified up to top secret with 384-bit keys.[2]
However, in August 2015, the NSA announced that it plans to replace Suite B with a new cipher suite due to concerns about quantum computing attacks on ECC.[3]

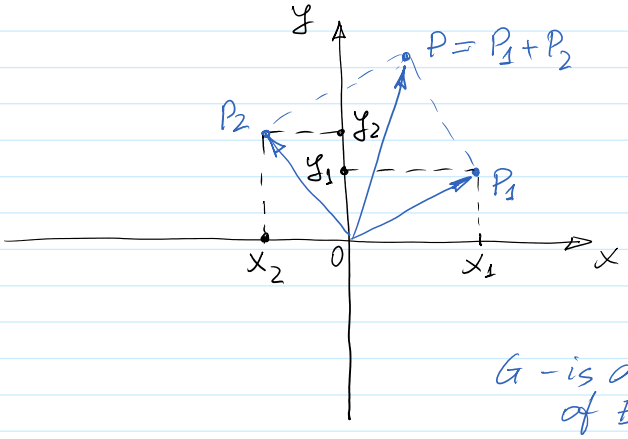
m - message to be signed of arbitrary finite length.

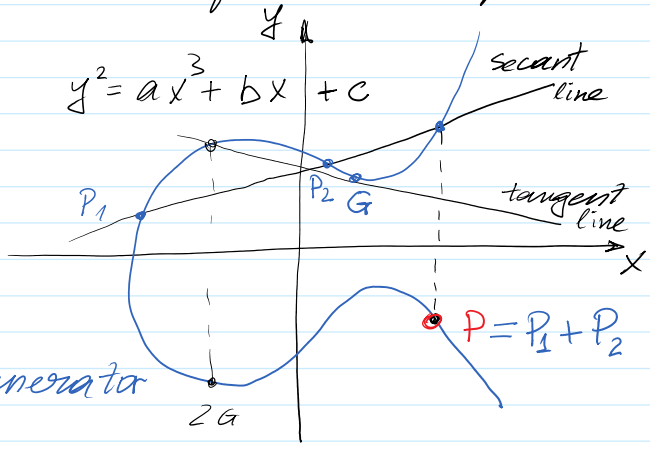
H( ) - cryptographic secure H-function

Hash and sign paradigm: signature is formed on the h-value: $h = H(m)$

Tradition signature schemes used multiplication operation modulo p, where p is prime number in numerical sets of integers, e.g. $\mathcal{L}_p^* = \{1, 2, 3, \dots, p-1\}$

ECDSA is using a very special addition operation between the points in elliptic curves defined in the plane.





G - is a generator of EC

Public parameters of ECCS : (Type of EC, G)

A: $PrK_A = (x)$, $PuK_A = (P_A)$; H(); $P_A = x \cdot G$ ($2 \cdot G$, $3 \cdot G$)

B: $PrK_B = (y)$, $PuK_B = (P_B)$; H(); $P_B = y \cdot G$

$PuK_A \rightarrow$

$\leftarrow PuK_B$

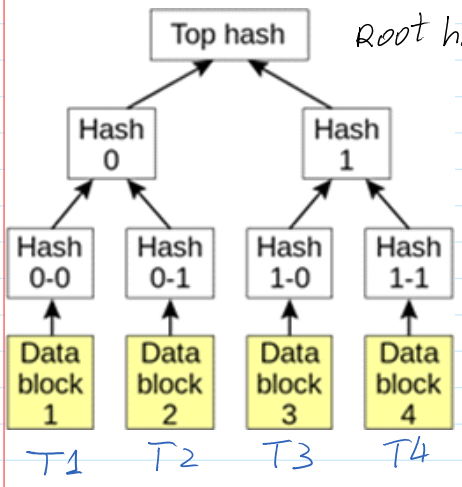
$Sig(PrK, h) = s$ $\quad (m, s) \rightarrow$

Verification:

1. $H(m) = h$
2. $Ver(PuK_A, h) = \begin{cases} True \\ False \end{cases}$

Bitcoin transactions are permanently recorded in the network through files called blocks. Maximum size

of the block is currently limited to 1 MB but it may be increased in the future. Each block contains a UNIX time timestamp, which is used in block validity checks to make it more difficult for adversary to manipulate the block chain. New blocks are added to the end of the record (block chain) by referencing the hash of the previous block and once added are never changed. A variable number of transactions is included into a block through the merkle tree (fig 3.). Transactions in the Merkle tree are hashed using double SHA256 (hash of the hash of the transaction message).



Root hash = Rh : signed by (PrK , PuK)

sig(PrK, Rh) = S

Transactions are included into the block's hash indirectly through the merkle root (top hash of a merkle tree). This allows removing old transactions (fig. 4) without modifying the hash of the block. Once the latest transaction is buried under enough blocks, previous transactions serve only as a history of the ownership and can be discarded to save space.

```
>> h=h28('Aliceaddr, Bobaddr, 1BTC, nonce=1000000')
h = 08625A0
```

```
>> h28('123456')
ans = ADC6C92
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000000')
h = F5F7583
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000001')
h = 6A152F0
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000002')
h = 65F8AA1
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000003')
h = 5F938C5
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000004')
h = 31E2102
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000005')
h = 8215835
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000006')
h = 2BF8D14
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000007')
h = 5DCDEF8
>> h=h28('Aliceaddr,Bobaddr,1BTC,nonce=1000008')
h = 4E7F384
```

```
>> h=h28('Aliceaddr, Bobaddr, 1BTC,nonce=1000027')
h = 044785C
```

$\dot{x} = Ax + f$ → T.A. ~800 € ↓ ~450 €

ocam → .flv