Program Analysis for System Security and Reliability (Spring 2020)

Summary

Author:
Yannick Merkli

4 Pages

Contents

1 Introduction to blockchains and smart contracts

1.1 Background

1.1.1 Hash functions and crypto puzzles

Hash functions are functions that take an arbitrary-sized input and map it to a fixed-size output.
Hash functions are further:

- Deterministic (equal inputs hash to the same value)
- Uniform (inputs are mapped evenly to the space of possible outputs)
- Efficient (should be fast to compute)

Since we have $|input\_space| >> |output\_space|$ (the input space is inherently unbound), collisions exist. However, cryptographic hash functions ensure collisions are hard to find.

Cryptographic hash functions
Consider a hash function $h : X \to Y$:

- Pre-image resistant: given $y \in Y$, it is infeasible to find $x \in X$ such that $h(x) = y$
- Second pre-image resistant: given $x \in X$, it is infeasible to find $x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$
- Collision resistance: It is infeasible to find a pair $(x, x') \in X \times X$ such that $x \neq x'$ and $h(x) \neq h(x')$. Collision resistance implies second pre-image resistance.

Applications of cryptographic hash functions:

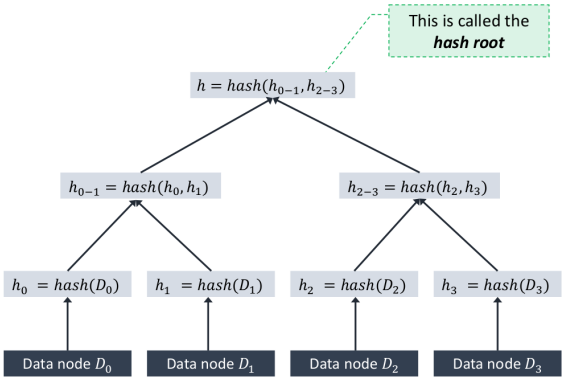
- Data equality: If we know that $h(x) = h(y)$ and h is a cryptographic hash function, then it is safe to assume that $x = y$ (due to collision resistance).
- Data integrity:
  - To verify the integrity a piece of data d, we can remember its hash, $hash = h(d)$
  - Later, when we obtain d' from untrusted source, we can verify whether $hash = h(d')$, to check whether $d = d'$
  - Useful because $hash$ is small (e.g. 256 bits)

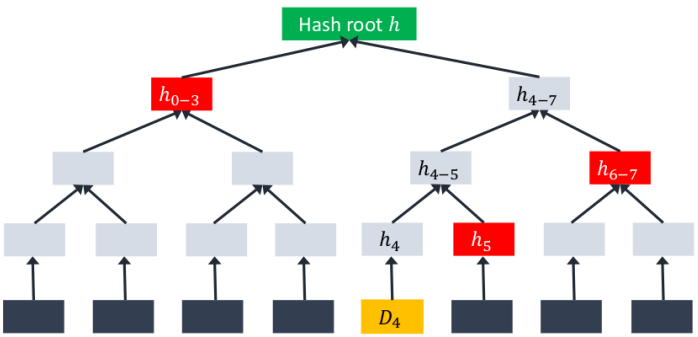
  Example: blockchain, don't need to remember the whole chain, just a hash of it
- Crypto puzzles:
  - Puzzle-friendly: For any output y, if r is chosen from a probability distribution with high min-entropy (true random), then it is infeasible to find x such that $h(r|x) = y$
  - Search puzzle: Given a puzzle ID id, chosen from a probability distribution with high min-entropy, and an output target $T \subset Y$, find a solution x such that $h(id|x) \in T$. Puzzle-friendliness implies that no solving strategy is much better than trying random values of x (i.e. brute-force).

  Example: Bitcoin proof-of-work.

1.1.2 Merkle trees

A Merkle tree is built bottom up. Start by hashing all the data and then recursively build the tree by hashing the hashes of the two children which becomes the parent hash.





Data integrity verification: The **hash root** h is typically obtained from a trusted source. One can verify the integrity of the data elements D_0, D_1, D_2, D_3 by reconstructing the hash root and comparing it to the trusted root.

Verifying whether D_4 is a member (i.e., a leaf): Obtain the hash root h from a trusted source. Then request the nodes h_5, h_{6-7}, h_{0-3} (from possibly untrusted source). Finally, compute $h_4, h_{4-5}, h_{0-3}, h'$ and check

whether $h = h'$.
Membership verification requires $\log(n)$ elements, only *one* of which needs to be from a trusted resource (the hash root h). This is especially useful when the set of data elements is large.
Possible application: check if blockchain transaction is valid (i.e. in the tree).

1.1.3 Digital signatures

The goal of a digital signature is to allow only one user to sign but anyone to verify the signature. We have the following API for signatures:

- $(sk, pk) = generateKeys(keySize)$
  - sk is the secret key, which the owner needs to keep private
  - pk is the public key, which is distributed to all users
- $sig = sign(sk, msg)$
- $verify(pk, msg, sig)$

Establish an identity:

- User generates (sk, pk) key pair
- $hash(pk)$ is the public name of the user
- sk allows the user to endorse a statement $stmt$ using a digital signature $sig = sign(sk, smt)$
- anyone can verify statements endorsed by the user using $verify(pk, stmt, sig)$

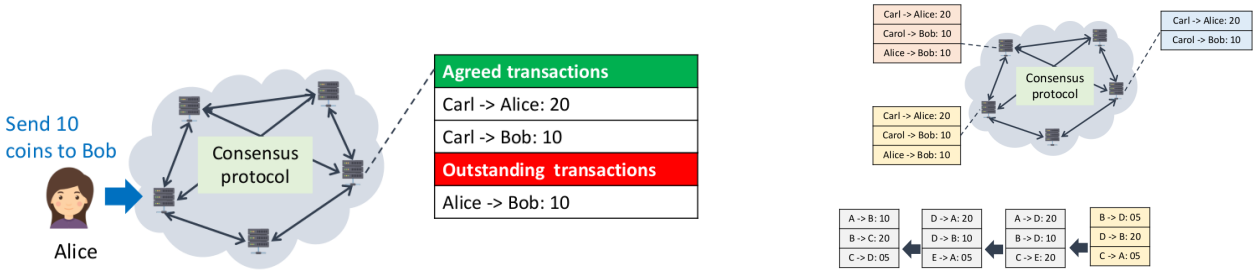
1.2 Bitcoin

In a traditional financial system, we have a centralized ledger that keeps track of everyone's balance (i.e. the amount of money). This would essentially be a very big table. In cryptocurrencies such as Bitcoin, we have a *distributed ledger*. In a distributed ledger, we don't have a central table with all balances. There are multiple balances in the network. However, all nodes need to agree on the state of the ledger. For this, the ledgers run a distributed consensus protocol. Distributed consensus needs to fulfill:

- Termination: Every correct process decides on some value
- Agreement: All correct processes decide on the same value
- Validity: If all correct processes propose the same value, then any correct process must decide on that value

Traditional motivation: reliability in distributed systems (node replication).

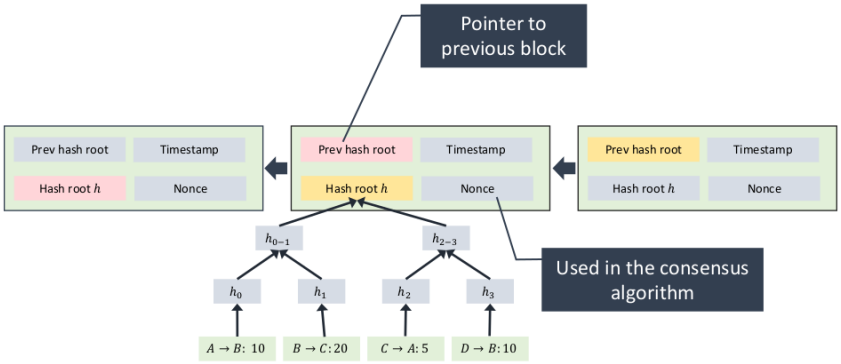
To make a transaction, users broadcast transactions to the network nodes. All nodes have a sequence of all blocks of agreed transactions they have reached consensus on. Each node has a set of outstanding transactions (to be added to a block in the blockchain). Distributed consensus in necessary since all nodes need to agree on the set of accepted transactions. Otherwise, problems such as double-spending arise.



The blockchain itself as a datastrucure is essentially a linked list of blocks. A block contains a set of accepted transactions. More precisely, a block contains:

- Prev hash root: Merkle hash root of the last block
- Hash root h: Merkle hash root of the current block. The Merkle hash tree contains all transactions included in the current block. The Merkle tree enables record integrity of all transaction and efficient membership verification.
- Timestamp
- Nonce

Distributed consensus is hard: nodes may crash, nodes may be byzantine, network is imperfect (not all pairs of nodes are connected, messages may have arbitrary delays, no global time).



1.2.1 Consensus algorithm (simplified)

1. New transactions are broadcast to all nodes
2. Each node collects new transactions into a block
3. In each round, a **random node** gets to broadcast its block
4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures)
5. Nodes express their acceptance of the block by including its hash in the next block they create

The question is how to select a random node. This selection needs to be careful since if we were to select nodes without randomness, it could be that nodes belonging to the same entity pick the transactions repeatedly which would for example allow the entity to censor transactions. Several schemes exist on how to pick a node: proof-of-work (select in proportion to computing power), proof-of-stake (select in proportion to ownership),...

1.2.2 Proof of work in Bitcoin

Nodes need to solve a hard puzzle. Nodes have the previous block with its hash root h_{prev} and the Merkle tree of the current block (with root h), consisting of all new transactions. The nodes then need to find a nonce X such that:

$$hash(h_{prev}|h|X) \leq difficulty$$

Changing the parameter $difficulty$ adjusts the expected time it takes until a node finds a nonce X (i.e. lower $difficulty$ means it takes longer). The current difficulty of Bitcoin is 15,546,745,765,529, which results in a average rate of 10.2min per block. The current hash power is 103,559,611,798 GH/s.
Proof-of-work makes it easy to verify: other nodes verify $hash(h_{prev}|h_{tx}) \leq difficulty$.
Key security assumption: Attacks are infeasible if majority of miners weighted by hash power follow the protocol.

1.2.3 Double spending

If the same output is spent in two different transactions, we have a double-spending. This results in a fork of the blockchain. Honest nodes always extend the longest valid branch. Over time, shorter branches will be abandoned. There are incentives for nodes to act honest and extend the longest chain:

- Block reward: The creator of a block can include a special coin-creation transaction in the block which will send a fixed amount of bitcoin (halfed every 210'000 blocks, currently 12.5BTC) to an chosen recipient address of this transaction. Block creator "receives" the reward only if the block ends up on the long-term consensus branch.
- Transaction fees: Creator of transaction may choose to make output value less than input value: $\sum inputs \geq \sum outputs$. Remainder $(\sum inputs - \sum outputs)$ is transaction fee that goes to the block creator.
  By increasing the transaction fees, you can control how fast your transaction is processed (a higher transaction fee gives a bigger incentive for miners to include the transaction in the block).

Each transaction output is in one of two possible states: spent or unspent. The sum of all unspent transaction outputs (UTXOs) builds part of the shared state of the blockchain.

1.2.4 Bitcoin scripting

A transaction can contain a special script. On the most basic level, we have an unlocking script: this controls who can spend an unspent transaction output, which gives ownership. The sender uses the recipient's public key as parameter for `CHECKSIG`. The recipient then provides a signature generated with its private key (which

can only be done by the holder of the private key), which is then checked by `CHECKSIG`.
Common Bitcoin scripts include: P2PK (Payment to public key), P2PKH (Payment to public key hash), P2MS (Pay to multi-signatures) (need multiple keys to unlock).

There also exists a stronger version of Bitcoin scripts: BitML - a language for Bitcoin smart contracts. A possible application is a timed commitment:

- Alice wants to commit to a secret s, but reveal it sometime later.
- Bob wants to be assured that he will either: learn Alice's secret s within time t or be compensated after time t.
- To achieve this, Alice can chooes and broadcasts $h = H(S)$ and releases two transactions:
  - (reveal h. withdraw A): Alice can reveal s and spend the coin herself (i.e. send back to herself)
  - (after t: withdraw B): Bob can spend the coin after time t

Bitcoin scripts are limited though: no loops, no signature verification on arbitrary messages, no multiplication and shifting, no arithmetic on long numbers, no concatenation on bitstrings. Bitcoin scripting is not Turing complete.

1.3 Smart contracts

A smart contract formalizes the terms of a contract in code.

A smart contract is a **computerized transaction protocol** that executes the **terms of a contract**. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and **minimize the need for trusted intermediaries**. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.

	Traditional	Smart
Specification	Natural language	Code
Identity & consent	Signatures	Digital signatures
Dispute resolution	Judges, arbitrators	Decentralized platform
Nullification	By judges	???
Payment	As specified	built-in
Escrow	Trusted third party	built-in

1.3.1 Ethereum

A decentralized platform designed to run smart contracts:

- Similar to a world computer that executes code and maintains the state of all smart contracts
- The latest block stores the latest local states of all smart contracts
- Transactions result in executing code (calling a function) in target smart contracts
- Transaction change the state of one more more contracts
- Ethereum smart contracts are Turing-complete

References