

Smart Contracts and Blockchain Technology

Lecture 2. The mining game

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Introduction to the topic

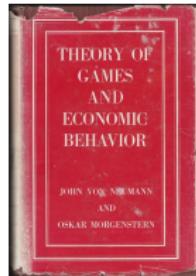
This lecture:

- We plan to have about four lectures on the game-theoretic analysis of
 - mining and
 - consensus formation.
- *Today:* The mining game with homogenous costs

Bitcoin mining as a contest (1)

The game-theoretic approach

Game theory has existed for more than a century by now. Its breakthrough to mainstream economics is often associated with the appearance of the monograph by John von Neumann and Oskar Morgenstern (1945).



Bitcoin mining as a contest (2)

Non-cooperative games

In a **non-cooperative game**, a finite number of players independently and simultaneously choose strategies, which has payoff implications for all of them.

An **equilibrium concept**, such as Nash equilibrium, is applied to make specific predictions.

Examples:

- Cournot model (quantity competition)
- Bertrand model (price competition)

Bitcoin mining as a contest (3)

Non-cooperative games

We will model blockchain mining as a game. As an equilibrium concept, we use the **Cournot-Nash equilibrium**.¹

This means that each player **correctly anticipates** the strategies chosen by her opponents, and chooses an optimal response.

Optimality means here that each player **maximizes her expected payoffs** given equilibrium expectations.

¹Nash equilibrium was initially defined for games with finitely many strategies for each player (Nash, 1950, 1951). The Cournot-Nash equilibrium is a generalization in which players may choose strategies from a continuous strategy set (e.g., from an interval).

Bitcoin mining as a contest (4)

Posing a difficult mathematical problem

Suppose that, in regular time intervals, the crypto protocol formulates a new **mathematical problem**, and organizes a competition between anybody interested to participate.

The **reward** for solving the puzzle is denoted by $R > 0$.

In reality, the reward is denominated in crypto (e.g., bitcoin) and may be composed of several components:

- block reward
- transaction fees
- additional rewards (e.g., so-called uncle rewards in Ethereum)

Bitcoin mining as a contest (5)

Miners

Users participating in the competition are called **miners**. Each miner $i \in \{1, 2\}$ decides about her computational power $h_i \geq 0$ (the “hash rate”).

The computations are assumed to create a **continuous flow of costs** $C(h_i)$, say within a fixed time interval of length $T > 0$.

We assume that miner i 's **cost function** is given by $C(h_i) = c_i \cdot h_i$, where $c_i > 0$ is miner i 's constant marginal cost to produce a unit of computational power.

Bitcoin mining as a contest (6)

Discussion of the assumptions on the cost functions

In reality, miners' investment decisions are more complicated for various reasons.

We impose the following **assumptions**:

- Fixed-cost investments (set-up of hardware and software) are depreciated using the straight-line method.
- Costs of computational power stand for operational variable costs (such as energy consumption and maintenance).
- Costs are expressed in bitcoin (rather than in fiat currency).

Bitcoin mining as a contest (7)

The case of homogeneous costs

For simplicity, we start with the case of two **ex-ante identical** miners.

Thus, we assume that:

- $n = 2$, and
- $c_1 = c_2 \equiv c$ (**homogeneous costs**).

Bitcoin mining as a contest (8)

Excursus: Poisson process

The Poisson process is one of the stochastic processes most commonly used in economics (and many other fields).

Key properties:

- Process in continuous time $t \geq 0$
- Counting process (starting at zero, constant almost everywhere, upward jumps by one at random times)
- Memoryless (delay does not make the “discovery” in the next instant more or less likely)

The Poisson process may be considered as the limit of discrete-time search processes as the time interval $\Delta t > 0$ between two consecutive experiments goes to zero, where the probability of a discovery $p \approx \lambda \cdot \Delta t$ is asymptotically linear in Δt .

Bitcoin mining as a contest (9)

The distribution of the time of discovery

Denote by \tilde{t} the **time of the discovery**. This is a random variable.

Let

$$F(t) = \text{prob}\{\tilde{t} \leq t\} \quad (1)$$

be the **cumulative distribution function** of the probability law that is followed by \tilde{t} .

The **instantaneous probability** of a discovery is

$$f(t) = \lim_{\Delta t \rightarrow 0} \frac{F(t + \Delta t) - F(t)}{\Delta t} = F'(t). \quad (2)$$

Bitcoin mining as a contest (10)

Relationship to the exponential distribution

As the process is memoryless, the **instantaneous probability** of a discovery, conditional on not having discovered the solution before, is **constant**:

$$\frac{f(t)}{1 - F(t)} = \lambda, \quad (3)$$

for some $\lambda > 0$.

Solving the ordinary differential equation (3) leads to

$$F(t) = 1 - \exp(-\lambda t), \quad (4)$$

$$f(t) = \lambda \exp(-\lambda t). \quad (5)$$

Thus, \tilde{t} is **exponentially distributed** with parameter λ .

Bitcoin mining as a contest (11)

Expected waiting time

The expected waiting time is defined as

$$E[\tilde{t}] = \int_0^{\infty} f(t) t dt. \quad (6)$$

Lemma 1. *The **expected waiting time** for the discovery is*

$$E[\tilde{t}] = \frac{1}{\lambda}. \quad (7)$$

Bitcoin mining as a contest (12)

Proof of the lemma

We use the **integration-by-parts** formula

$$\int_a^b u'(t)v(t)dt = u(t)v(t)|_a^b - \int_a^b u(t)v'(t)dt, \quad (8)$$

with functions

$$u(t) = -\exp(-\lambda t) \quad (9)$$

$$v(t) = t. \quad (10)$$

Then,

$$\int_0^\infty \lambda \exp(-\lambda t)tdt = \int_0^\infty \exp(-\lambda t)dt = \frac{1}{\lambda}, \quad (11)$$

as has been claimed. \square

Bitcoin mining as a contest (13)

Waiting time for an individual miner

We assume that solving the puzzle (by try and error) follows a Poisson process with parameter

$$\lambda_i = \frac{h_i}{d}, \quad (12)$$

where d is the **difficulty of the puzzle**.

Let \tilde{t}_i be miner i 's **waiting time** for solving the puzzle. Then, \tilde{t}_i is an exponentially distributed random variable with parameter λ_i .

Moreover, the **expected waiting time for miner i** is

$$E[\tilde{t}_i] = \frac{1}{\lambda_i} = \frac{d}{h_i}. \quad (13)$$

Bitcoin mining as a contest (14)

Waiting time for the market

The time of the first solution in the market is $\tilde{t} = \min(\tilde{t}_1, \tilde{t}_2)$.

Let's assume that individual waiting times \tilde{t}_1 and \tilde{t}_2 are stochastically independent.

Then, \tilde{t} is likewise exponentially distributed because, for any $t \geq 0$,

$$\text{prob}\{\tilde{t} \leq t\} = 1 - (1 - F_1(t))(1 - F_2(t)) \quad (14)$$

$$= 1 - \exp(-\lambda_1 t) \exp(-\lambda_2 t) \quad (15)$$

$$= 1 - \exp(-(\lambda_1 + \lambda_2)t). \quad (16)$$

Bitcoin mining as a contest (15)

Parameters

For the computation above, the parameter of the exponential distribution underlying $E[\tilde{t}]$ has the parameter $\lambda = h/d$, where $h = h_1 + h_2$.

Moreover, the **expected waiting time for the market** is

$$E[\tilde{t}] = \frac{1}{\lambda} = \frac{d}{h}. \tag{17}$$

Note: The parameter d is adjusted by the protocol such that $E[\tilde{t}] = T$ (e.g., in the case of bitcoin, $T = 10$ minutes).

Bitcoin mining as a contest (16)

Two-stage game

Stage 1. Miners simultaneously and independently choose hash rates h_1 and h_2 .

Stage 2. The protocol endogenously adjusts the difficulty level d such that

$$\frac{d}{h} = 10 \text{ minutes.} \quad (18)$$

Therefore, **miner i 's profit** in any time interval of expected length T starting after the discovery of the previous block is

$$\Pi_i(h_1, h_2) = \begin{cases} R - c \cdot h_i & \text{if } i \text{ is first to solve the puzzle} \\ -c \cdot h_i & \text{if } i \text{ is not first to solve the puzzle.} \end{cases} \quad (19)$$

Bitcoin mining as a contest (17)

Property of the Poisson process

The probability for miner 1 to solve the problem is given as

$$p_1 = \int_0^\infty f_1(t)(1 - F_2(t))dt \quad (20)$$

$$= \int_0^\infty \lambda_1 \exp(-\lambda_1 t) \exp(-\lambda_2 t) dt \quad (21)$$

$$= \lambda_1 \int_0^\infty \exp(-(\lambda_1 + \lambda_2)t) dt \quad (22)$$

$$= \frac{\lambda_1}{\lambda_1 + \lambda_2}. \quad (23)$$

Lemma 2. *Miner 1's probability of winning equals*

$$p_1 = \frac{\lambda_1}{\lambda_1 + \lambda_2}. \quad (24)$$

Bitcoin mining as a contest (18)

Contest success function

Note that

$$\frac{\lambda_1}{\lambda_1 + \lambda_2} = \frac{h_1/d}{h_1/d + h_2/d} = \frac{h_1}{h_1 + h_2}. \quad (25)$$

Therefore, regardless of d , the ratios

$$p_1 = \frac{h_1}{h_1 + h_2} \quad (26)$$

$$p_2 = \frac{h_2}{h_1 + h_2} \quad (27)$$

represent the probabilities that miners 1 and 2, respectively, will be first in solving the puzzle.

Bitcoin mining as a contest (19)

The miner's profit

Therefore, **miner i 's expected profit** is

$$E[\Pi_i] = \frac{h_i}{h_1 + h_2} R - c_i h_i \quad (i \in \{1, 2\}), \quad (28)$$

where the ratio is interpreted as zero if $h_1 = h_2 = 0$.

Bitcoin mining as a contest (20)

Exploiting first-order conditions

Maximization of miner 1's expected profit with respect to h_1 leads to the **first-order condition**

$$\frac{Rh_2}{(h_1 + h_2)^2} = c. \quad (29)$$

We focus on symmetric equilibria. Thus,

$$h_1 = h_2. \quad (30)$$

Then

$$\frac{R}{4h_1} = c \quad (31)$$

$$\Rightarrow h_1^* = h_2^* = \frac{R}{4c}. \quad (32)$$

$h_i \cdot c = \text{costs}$

Bitcoin mining as a contest (21)

Nash equilibrium

Proposition 1. *The unique Nash equilibrium of the mining game with homogeneous costs is given by*

$$h_1^* = h_2^* = \frac{R}{4c}. \quad (33)$$

Comparative statics: The hash power (energy consumption, CO₂ footprint)

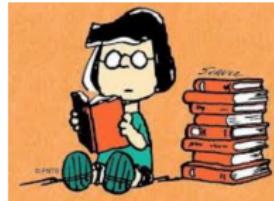
- increases strictly in the reward R ,
- declines with marginal costs of mining c (i.e., Pigouvian taxation would be desirable).

Bitcoin mining as a contest (22)

Bibliographic notes

This lecture is based on the recent **game-theoretic literature on blockchain economics**. This literature studies the **economic incentives and competitive behavior** of blockchain users.

Our model of bitcoin mining follows **Dimitri (2017)**.



Houy (2016) considered a similar model, allowing for endogenous block size.

Bitcoin mining as a contest (23)

References

- Dimitri, N. (2017), Bitcoin mining as a contest, *Ledger* **2**, 31-37.
- Houy, N. (2016), The Bitcoin mining game, *Ledger* **1**, 53-68.
- Nash, J. (1950). Equilibrium points in n -person games. *Proceedings of the National Academy of Sciences* **36**, 48-49.
- Nash, J. (1951). Non-cooperative games, *Annals of Mathematics* **54**, 286-295.
- von Neumann, J., Morgenstern, O. (1945). *Theory of Games and Economic Behavior*.

Smart Contracts and Blockchain Technology

Lecture 3. Extensions of the basic mining model

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

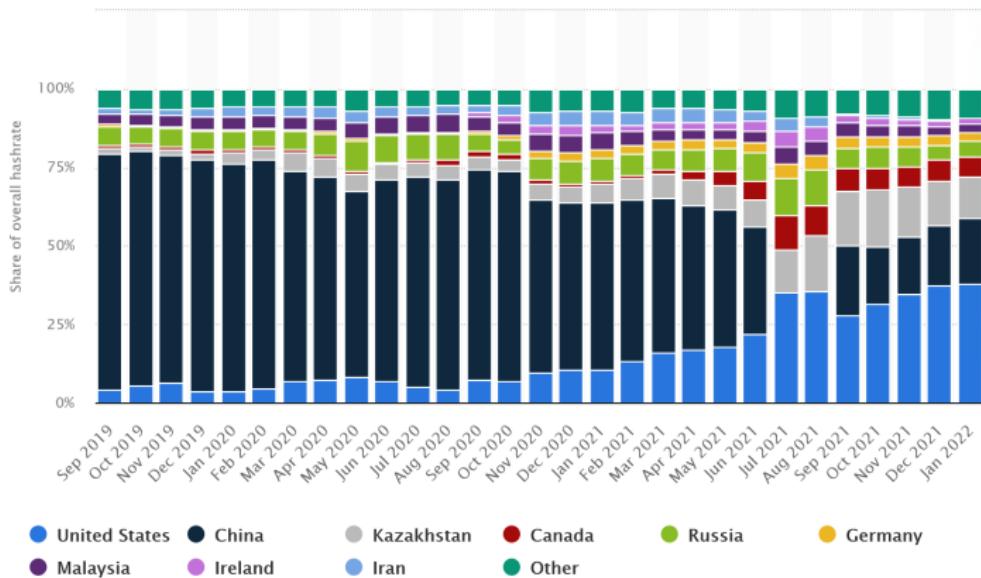
Last lecture: The game played by blockchain miners with homogenous costs.

This lecture: Extensions such as:

- More than two miners and heterogeneous costs
- Intertemporal smoothing (risk aversion)

Extensions of the basic mining model (1)

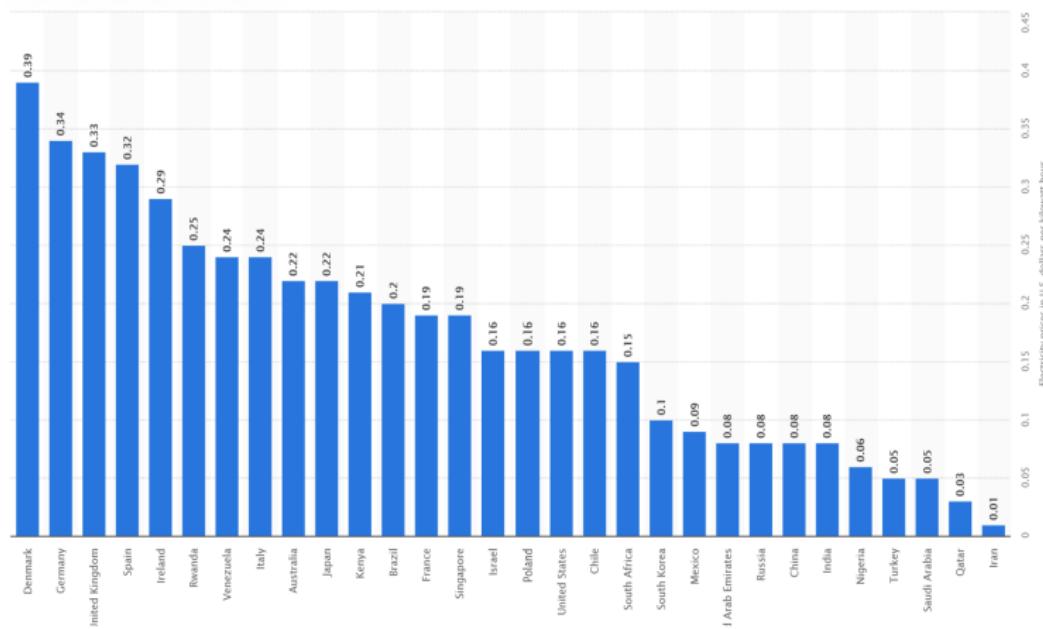
Distribution of Bitcoin mining hashrate 09/2019-01/2022, by country¹



¹Source: www.statista.com. In July 2021, a Chinese court ruled that bitcoin mining is harmful to the climate and incompatible with China's carbon neutrality goal (www.climatechangenews.com).

Extensions of the basic mining model (2)

Household electricity prices in 12/2021, by select country²



²Source: www.statista.com

Extensions of the basic mining model (3)

More than two miners and heterogeneous costs

More than two miners. Rather than just two miners, we allow now for any number $n \geq 2$ of miners. Let $i \in \{1, \dots, n\}$ denote a generic miner.

Heterogeneous costs. Each miner i chooses a hash rate $h_i \geq 0$, and the computations are assumed to create a continuous flow of costs

$$C_i(h_i) = c_i h_i \tag{1}$$

in a given period of time $T > 0$, where $c_i > 0$ is the constant marginal cost for miner i to produce a unit of computational power.³

³We assume for simplicity complete information regarding c_i , i.e., miners know each other's marginal costs.

Extensions of the basic mining model (4)

Waiting times for a miner

Let \tilde{t}_i be miner i 's **waiting time** for solving the puzzle. As has been seen, \tilde{t}_i is an exponentially distributed random variable with parameter

$$\lambda_i = \frac{h_i}{d}. \tag{2}$$

where d is the difficulty of the puzzle.

Then, the **expected waiting time for miner i** is

$$E [\tilde{t}_i] = \frac{d}{h_i}. \tag{3}$$

Extensions of the basic mining model (5)

Waiting time for the market

The **time at which the new block is mined** and added to the blockchain is

$$\tilde{t} = \min\{\tilde{t}_i\}_{i=1,\dots,n}. \quad (4)$$

Let's assume that **individual waiting times** $\{\tilde{t}_i\}_{i=1}^n$ are independent. Then, \tilde{t} is exponentially distributed with parameter

$$\lambda = \frac{h}{d}, \quad (5)$$

where

$$h = \sum_{i=1}^n h_i \quad (6)$$

is the **total hash rate** of the market.

Extensions of the basic mining model (6)

Expected payoffs

Moreover, the **expected waiting time for the market** is

$$E[\tilde{t}] = \frac{d}{h}. \quad (7)$$

As before, d is determined endogenously so that $E[\tilde{t}] = T$.

Miner i 's **expected payoff** is given by

$$E[\Pi_i] = \frac{h_i}{\sum_{j=1}^n h_j} R - c_i h_i, \quad (8)$$

where the ratio is interpreted as zero if the denominator vanishes.

Extensions of the basic mining model (7)

First-order conditions

Derivation of the expected profit with respect to h_i leads to the **first-order condition**

$$\frac{R(h - h_i)}{h^2} = c_i. \quad (9)$$

Summing over all miners $i = 1, \dots, n$, we obtain

$$\sum_{i=1}^n \frac{R(h - h_i)}{h^2} = C \quad (10)$$

$$\Rightarrow \frac{R(n-1)}{h} = C \quad (11)$$

$$\Rightarrow h^* = \frac{R(n-1)}{C}, \quad (12)$$

where $C = \sum_{i=1}^n c_i$.

Extensions of the basic mining model (8)

Nash equilibrium with heterogenous costs

In view of Eq. (9), it follows that **miner i 's optimal has rate is**

$$h_i^* = R(n-1) \frac{C - (n-1)c_i}{C^2}. \quad (13)$$

Thus, if miners are ordered such that $c_1 \leq c_2 \leq \dots \leq c_n$, then $h_1 \geq h_2 \geq \dots \geq h_n$.

Proposition. Suppose that $C > (n-1)c_n$. Then, the unique Nash equilibrium of the bitcoin mining game, with complete information about the contestants' marginal costs, is the profile of hash rates (h_1^*, \dots, h_n^*) , where h_i^* is given by Eq. (13).

Extensions of the basic mining model (9)

Arbitrarily many players with homogeneous costs

Suppose that **all miners are ex-ante identical**, i.e.,

$$c_1 = \dots = c_n \equiv c. \quad (14)$$

Then, $C = nc > (n - 1)c$, and miner i 's hash rate is given by

$$h_i^* = R(n - 1) \frac{C - (n - 1)c}{C^2} = \frac{(n - 1)R}{n^2 c}. \quad (15)$$

The **total hash rate** in the market is

$$h^* = \frac{(n - 1)R}{nc}. \quad (16)$$

In particular, **rent dissipation** h^*c/R goes to 100 percent as $n \rightarrow \infty$.

Extensions of the basic mining model (10)

Numerical example

Suppose that $R = 1$, $n = 3$, and

$$c_1 = 0.08 \quad (\text{e.g., India}) \quad (17)$$

$$c_2 = 0.19 \quad (\text{e.g., Singapore}) \quad (18)$$

$$c_3 = 0.25 \quad (\text{e.g., Switzerland}) \quad (19)$$

Then, $C = 0.08 + 0.19 + 0.25 = 0.52 > (n - 1)0.08$, and

$$h_1 = R(n - 1) \frac{C - (n - 1)c_1}{C^2} \quad (20)$$

$$= 2 \cdot \frac{0.52 - 2 \cdot 0.08}{0.52^2} \quad (21)$$

$$\simeq 2.663, \quad (22)$$

$$h_2 \simeq 1.036, \quad (23)$$

$$h_3 \simeq 0.148. \quad (24)$$

Extensions of the basic mining model (11)

Activity analysis

Definition. We say that miner i is *active* if $h_i > 0$.

Consider a marginally active miner n (with highest cost), and suppose that $n \geq 3$. Then,

$$h_n = R(n-1) \frac{C - (n-1)c_n}{C^2} = 0 \quad (25)$$

$$\Leftrightarrow C - (n-1)c_n = 0 \quad (26)$$

$$\Leftrightarrow c_n = \frac{C}{n-1} \quad \left| -\frac{c_n}{n-1} \right. \quad (27)$$

$$\Leftrightarrow c_n \cdot \frac{n-2}{n-1} = \frac{c_1 + \dots + c_{n-1}}{n-1} \quad (28)$$

$$\Leftrightarrow c_n = \left(1 + \frac{1}{n-2}\right) \frac{c_1 + \dots + c_{n-1}}{n-1}. \quad (29)$$

Extensions of the basic mining model (12)

Activity analysis (continued)

Observations:

- For miner n to be active, it is necessary and sufficient that her marginal cost c_n is low enough compared to cost average of all more efficient miners, i.e., that

$$c_i \leq c_i^* \equiv \left(1 + \frac{1}{n-2}\right) \frac{c_1 + \dots + c_{n-1}}{n-1}. \quad (30)$$

- c_i^* converges to the average cost as $n \rightarrow \infty$ (perfect competition).
- However, the block reward R , provided it is positive, does not matter for the decision regarding activity. Instead, R matters for the decision about the optimal hash rate.

Extensions of the basic mining model (13)

In the maturing market, solo miners have entered pools...⁴

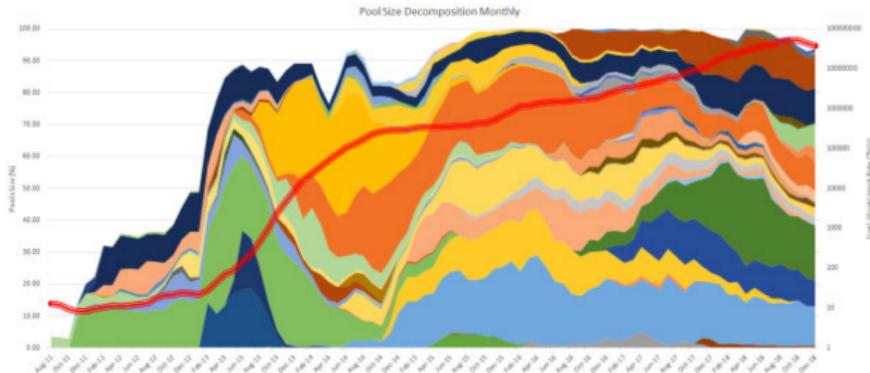


Figure 1

The evolution of Bitcoin mining

This graph plots (1) the growth of aggregate hash rates (right-hand-side vertical axis, in log scale) starting from June 2011 to December 2018 and (2) the size evolution of all Bitcoin mining pools (left-hand-side vertical axis) over this period, with pool size measured by each pool's hash rates as a fraction of global hash rates. Colors represent different pools, and white spaces represent solo mining. Over time, mining pools have increasingly taken over Bitcoin mining, but no pool ever seems to dominate the mining industry for long. The pool hash rates data come from <https://data.bitcoinity.org/bitcoin/hashrate/6m?c=m&g=15&t=a> and <https://btc.com/>, with details given in Section 4.

⁴Source: Cong, He, and Li (2021).

Extensions of the basic mining model (14)

Heuristics of solo mining

Within 30 days, the bitcoin blockchain with $T = 10$ min produces a total number of

$$\frac{60 \text{ min}}{T} \times 24 \times 30 = 4320 \quad (31)$$

new blocks.

Let

$$p_i = \frac{h_i}{\sum_{j=1}^n h_j} \quad (32)$$

denote miner i 's probability of winning a new block within T .

Extensions of the basic mining model (15)

Heuristics of solo mining (continued)

Then, miner i 's probability of winning at least one block in 30 days is

$$\hat{p}_i = 1 - (1 - p_i)^{4320}. \quad (33)$$

E.g., if miner i is one of 100000 ex-ante identical miners, then

$$p_i = 0.001\% \quad (34)$$

$$\Rightarrow \hat{p}_i \simeq 4.2\%. \quad (35)$$

Thus as the market grows, the probability of successfully mining a block within a whole month becomes too small to be the basis of a regular business, i.e., **solo mining** becomes a prohibitively risky strategy for small miners.

Extensions of the basic mining model (16)

The impact of risk aversion

Suppose that miner i exhibits **constant absolute risk aversion (CARA)**, i.e., her utility function is given as

$$u_i(x) = -\exp(-\alpha x),$$

for some $\alpha > 0$. The parameter α is known as the Arrow-Pratt coefficient of absolute risk aversion.

Then, the **certainty equivalent** of the uncertain revenue is given as

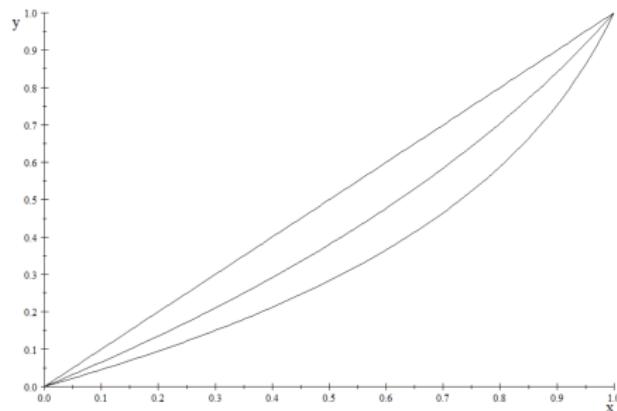
$$\text{CE}(p_i, R, \alpha) = u_i^{-1}(p_i u_i(R) + (1 - p_i) u_i(0)) \quad (36)$$

$$= -\frac{1}{\alpha} \ln \left(\frac{p_i}{\exp(\alpha R)} + 1 - p_i \right) \quad (37)$$

Extensions of the basic mining model (17)

Certainty equivalent

The figure below illustrates the function $CE(p_i, R, \alpha)$ for $\alpha \in \{0, 1, 2\}$ and $R = 1$.⁵



Observation: Risk aversion strictly lowers the incentive for all miners (with less than half of the market's hash power).

⁵The x-axis shows p_i , the y-axis shows $CE(p_i, R, \alpha)$.

Extensions of the basic mining model (18)

Proof of the observation

$$\frac{\partial \text{CE}(p_i, R, \alpha)}{\partial p_i} = \frac{\partial}{\partial p_i} \left(-\frac{1}{\alpha} \ln \left(\frac{p_i}{\exp(\alpha R)} + 1 - p_i \right) \right) \quad (38)$$

$$= \frac{e^{\alpha R} - 1}{\alpha (p_i + e^{\alpha R} - p_i e^{\alpha R})}. \quad (39)$$

Now,

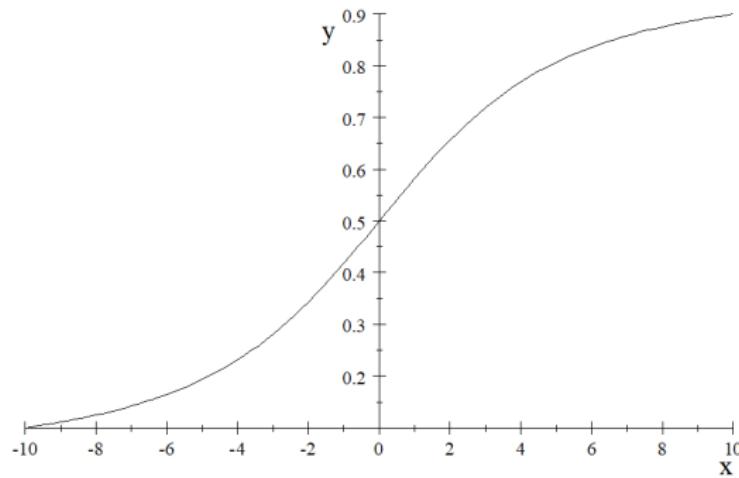
$$\frac{e^{\alpha R} - 1}{\alpha (e^{\alpha R} - p_i (e^{\alpha R} - 1))} \geq R \quad (40)$$

$$\Leftrightarrow p_i \leq p_i^*(\alpha) = \underbrace{\frac{1}{1 - \exp(-\alpha R)} - \frac{1}{\alpha R}}_{> \frac{1}{2}} \quad (41)$$

Extensions of the basic mining model (19)

Illustration

$$x > 0 \Rightarrow \frac{1}{1 - \exp(-x)} - \frac{1}{x} > \frac{1}{2}$$



Extensions of the basic mining model (20)

Equilibrium analysis

With risk aversion, miner i 's **expected payoff** is given by

$$E[u_i(\Pi_i)] = \text{CE}(p_i, R, \alpha) - ch_i, \quad (42)$$

where $p_i = h_i / \sum_{j=1}^n h_j$.

Extensions of the basic mining model (21)

Equilibrium analysis (continued)

The first-order conditions reads

$$\frac{h - h_i}{h^2} \cdot \frac{\partial \text{CE}(p_i, R, \alpha)}{\partial p_i} = c. \quad (43)$$

In a symmetric equilibrium, $h = nh_i$ and $p_i = 1/n$. Therefore, the market hash rate is

$$h^* = \frac{n-1}{nc} \cdot \underbrace{\frac{\partial \text{CE}(1/n, R, \alpha)}{\partial p_i}}_{< R}. \quad (44)$$

Thus, compared to the symmetric reference model with $n \geq 2$ risk-neutral miners, the introduction of risk aversion strictly lowers the equilibrium market hash rate.

Extensions of the basic mining model (22)

A simple model of mining pools

Suppose that there are **two pool managers** that offer (almost) perfect risk diversification for a large population of small miners.

Each pool $m \in \{1, 2\}$ charges a proportional fee of $f_m \in [0, 1]$. Solo mining is assumed to be prohibitively risky.

Denote by H_m the hash power attracted by pool m .⁶ We assume that miners can freely allocate their hash power across mining pools.

⁶Thus, we assume (for simplicity) that pool managers does not do any mining themselves.

Extensions of the basic mining model (23)

Endogenously attracted hash power

Then, the marginal payoff of a miner by allocating a small amount of additional hash power to pool m is

$$\frac{H_2}{(H_1 + H_2)^2} (1 - f_1)R = c \quad (45)$$

$$\frac{H_1}{(H_1 + H_2)^2} (1 - f_2)R = c. \quad (46)$$

Thus

$$H_2(1 - f_1) = H_1(1 - f_2). \quad (47)$$

Extensions of the basic mining model (24)

The pool's problem

Pool 1 has the following problem

$$\begin{aligned} & \max_{f_1 \in [0,1]} && \frac{H_1}{H_1 + H_2} f_1 R \\ & \text{s.t.} && H_2(1 - f_1) = H_1(1 - f_2) \end{aligned} \tag{48}$$

Note that

$$\frac{H_1}{H_1 + H_2} = \frac{1}{1 + H_2/H_1} \tag{49}$$

$$= \frac{1}{1 + \frac{1-f_2}{1-f_1}} \tag{50}$$

$$= \frac{1 - f_1}{2 - f_1 - f_2}. \tag{51}$$

Extensions of the basic mining model (25)

The pool's problem (continued)

Pool 1's problem reduces to has the following problem

$$\max_{f_1 \in [0,1]} \frac{(1-f_1)}{2-f_1-f_2} f_1 R \quad (52)$$

$$= \max_{e_1 \in [0,1]} \frac{e_1}{e_1+e_2} (1 - e_1) R \quad (53)$$

The first-order condition reads

$$\frac{e_2}{(e_1 + e_2)^2} (1 - e_1) - \frac{e_1}{e_1 + e_2} = 0. \quad (54)$$

Extensions of the basic mining model (26)

Equilibrium derivation

Hence,

$$\frac{e_2(1 - e_1)}{e_1 + e_2} = e_1. \quad (55)$$

In a symmetric equilibrium, $e_1 = e_2 \equiv e$, so that

$$e = \frac{1}{3} \quad (56)$$

$$\Rightarrow f_1^* = f_2^* = \frac{2}{3}. \quad (57)$$

Extensions of the basic mining model (27)

Equilibrium has rate

Thus,

$$H^* = H_1^* + H_2^* \quad (58)$$

$$= \frac{R}{6c}. \quad (59)$$

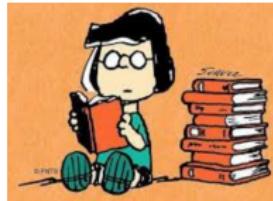
The purely **mining-related rent dissipation** is $\frac{1}{6} = 16.6$ percent, which is surprisingly low (because it is often claimed that pooling leads to excessive rent dissipation).

Extensions of the basic mining model (28)

Bibliographic notes

Cong et al. (2020) argue that pools escalate the arms race between bitcoin miners, leading to even more energy consumption.

Assuming exogenous difficulty, Ma et al. (2018) show that, under free entry, energy consumption is independent of the difficulty of the miner's puzzle.



Extensions of the basic mining model (29)

References

Cong, L.W., He, Z., Li, J. (2021), Decentralized mining in centralized pools, *Review of Financial Studies* **34**, 1191–1235.

Ma, J., Gans, J.S., Tourky, R. (2018), Market structure in bitcoin mining, *NBER Working Paper* 24242.

Smart Contracts and Blockchain Technology

Lecture 4. Consensus Formation

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

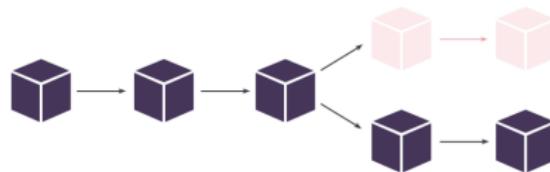
Last lecture: Model of crypto mining (equilibrium, activity analysis, mining pools)

This lecture: Consensus formation (forks, model of the blockchain, conservative mining, longest-chain rule, selfish mining)

Formal model of the blockchain (1)

Forks

A **fork** occurs if a single block has two or more child nodes:



Forks can be quite problematic:

- Loss of liquidity (e.g., in decentralized exchanges)
- Value of a stablecoin after the fork?
- NFTs are no longer unique
- Etc.

Formal model of the blockchain (2)

Examples of forks

Bitcoin (BTC) vs. Bitcoin Cash (BTH):

- The fork was caused on August 1, 2017 due to a disagreement on how to scale the network up to accommodate the strong demand for transactions.¹
- As of October 2022, BTC is valued USD 20'000, BCH is valued USD 110.



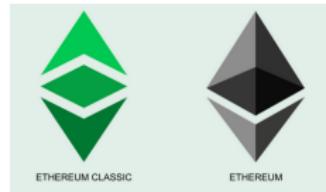
¹Bitcoin has a block size of 1 MB at the cost of dropping the electronic signature of transactions (so-called Segregated Witness technology, SegWit2x), while Bitcoin Cash has a block size of 8 MB, at the cost of larger disc space requirements.

Formal model of the blockchain (3)

Examples of forks (continued)

Ethereum (ETH) vs. Ethereum Classic (ETC):

- Investors in the DAO had lost USD 70m due to an exploit in the smart contract. To revert the hack, Ethereum was hard-forked in 2016.²
- The legacy chain became Ethereum Classic.
- As of October 2022, ETH is valued USD 1'300, ETC is valued USD 24.



²For the full story, see <https://medium.com>.

Formal model of the blockchain (4)

Some popular terminology

Accidental forks. Two or more miners broadcast a new block at nearly the same time. An accidental fork is resolved as subsequent blocks get added and one of the branches becomes longer than the other. Miners will tend to ignore the blocks that are not in the longest chain (**orphaned blocks**).³

Intended forks may result from updates of the software used by nodes of the peer-to-peer network:

- So-called **soft forks** may (but need not) result from backwards-compatible updates.
- **Hard forks** may (but need not) result from backwards-incompatible updates.

³Note the potentially confusing terminology. An orphaned block always has a parent block (unless it is the genesis block). Moreover, orphaned block are defined by the property that they have no child blocks.

Formal model of the blockchain (5)

Definition

Suppose there are $n \geq 2$ **miners**, collected in a set $N = \{1, \dots, n\}$.

A **blockchain** \mathbb{B} consists of:

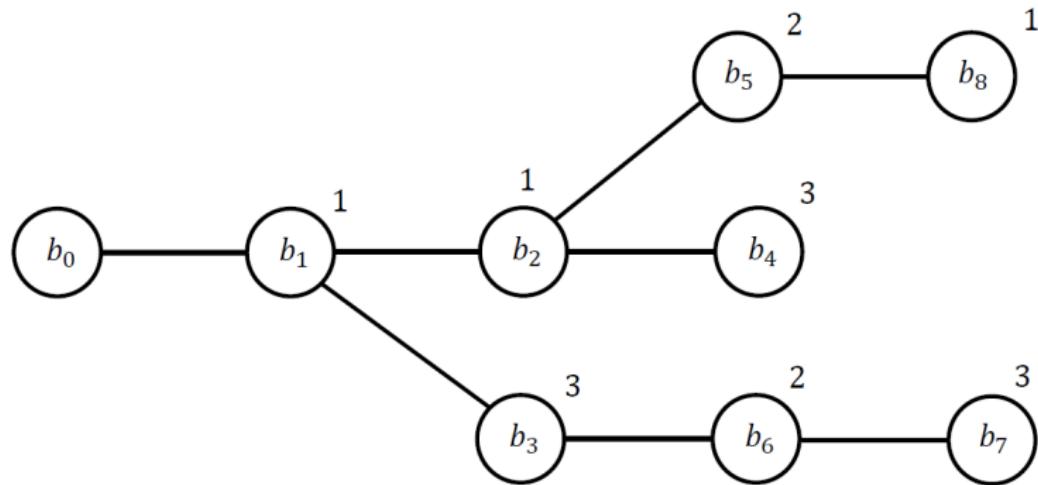
- (i) a **sequence of blocks** $B = \{b_0, b_1, \dots, b_T\}$, where $T \geq 0$;
- (ii) a **parent-child relation** \leqslant on B ;
- (iii) an **assignment map** $\iota : B \setminus \{b_0\} \rightarrow N$.

It is required that:

- (a) each block except the **genesis block** b_0 has precisely one parent, i.e., for any $t' > 0$, there is precisely one t such that $b_t \leqslant b_{t'}$
- (b) the parent has a lower index than the child, i.e., $b_t \leqslant b_{t'}$ implies $t < t'$.

Formal model of the blockchain (6)

Example



An example of a 3-miner blockchain with $T = 8$; the numbers at the circles refer to the values of the assignment map.

Formal model of the blockchain (7)

Chains

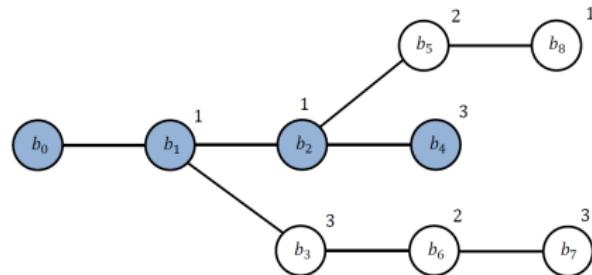
Length: nr. of parent-child relationships

A **chain** of length $K \geq 1$ in the blockchain \mathbb{B} is a set

$C = \{b^{(0)}, \dots, b^{(K)}\}$ such that $b^{(k-1)} \Leftarrow b^{(k)}$ for $k = 1, \dots, K$.

The **original chain** starts at b_0 and, if there is more than one child to a given parent, continues with the child with the lowest index.

E.g., in the example, the original chain is $C^{\text{org}} = \{b_0, b_1, b_2, b_4\}$.



Formal model of the blockchain (8)

Blockchain game, initial stage

Suppose the n miners incrementally construct a blockchain \mathbb{B} by interacting over $T \geq 1$ stages. We denote the intermediate blockchains as $\mathbb{B}_0, \mathbb{B}_1, \dots, \mathbb{B}_T$.

At the start of the game, \mathbb{B}_0 consists only of the genesis block, so that $B_0 = \{b_0\}$, and both \leq_0 and ι_0 are empty.

Formal model of the blockchain (9)

Blockchain game, intermediate stages

Next, at any intermediate stage $t \in \{1, 2, \dots, T\}$, \mathbb{B}_t is constructed from the existing blockchain \mathbb{B}_{t-1} as follows.

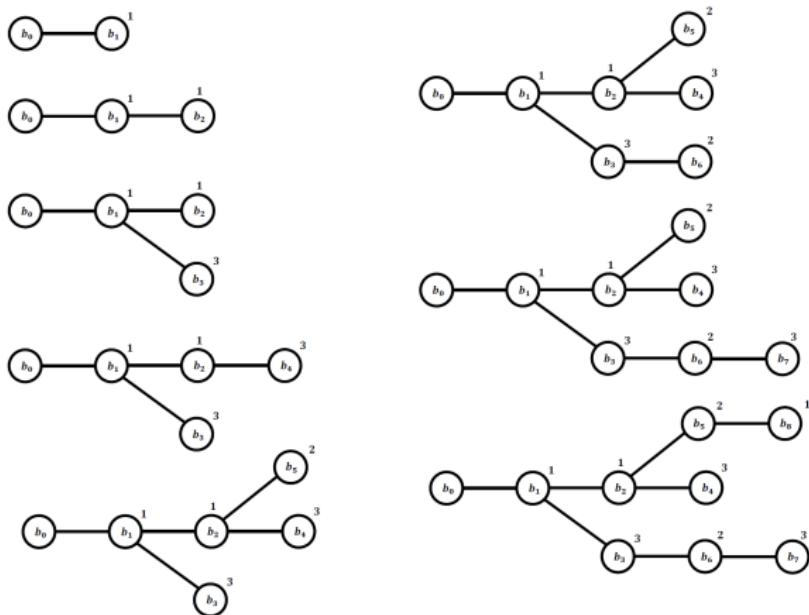
Each miner $i \in N$ selects a block $\hat{b}_{t-1}(i) \in B_{t-1}$ from the existing set of blocks B_{t-1} .

Then, a fair random draw, just as in Dimitri's (2017) mining model, selects the winning miner $i_t^* \in N$ of stage t .

The new block b_t is assigned to i_t^* . Moreover, it is appended as a child to the block $\hat{b}_{t-1}(i_t^*)$ chosen by the winning miner.

Formal model of the blockchain (6)

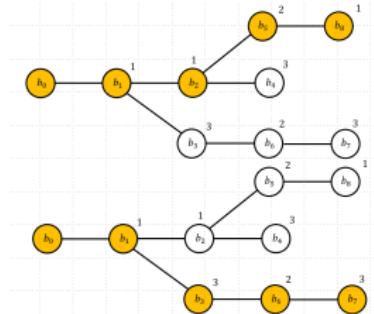
Example



Formal model of the blockchain (10)

Payoffs

After stage T , one of the longest chains C in the blockchain \mathbb{B}_T is drawn with equal probability.



Each miner $i \in N$ receives one *token* for each block $b \in C \setminus \{b_0\}$ assigned to her. Miners are risk-neutral and maximize the number of tokens they receive.

Note: There are efficiency gains from coordination...

Formal model of the blockchain (11)

Nash equilibrium

Definition.

- We say that miner i is *conservative* if she always chooses the last block of the original chain.
- We say that miner i follows the *longest-chain rule* if she always chooses the last block of one of the longest chains.

Note: *Conservative mining is a strategy, whereas the longest-chain rule is a class of strategies.*

Proposition. *Conservative mining constitutes a symmetric Nash equilibrium. Any combination of mining strategies consistent with the longest-chain rule forms a Nash equilibrium.*

Formal model of the blockchain (12)

Proof (for conservative mining)

We assume that all miners $j \in N \setminus \{i\}$ are conservative. We have to show that miner i likewise wishes to be conservative.

Suppose first that i is conservative. Then, the blockchain develops into a single chain consisting of $T + 1$ blocks, and miner i receives one token for each block that she mines.

Suppose, instead, that miner i deviates and works on a block that is not the last block of the original chain. Then, miner i creates a fork with positive probability. As a result, she does not necessarily receive one token for each block that she mines.

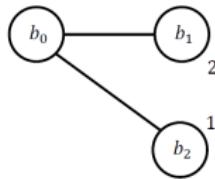
Thus, miner i potentially lowers, but never raises her payoff. Therefore, a deviation from conservative mining can never lead to a strictly higher expected payoff for miner i .

Formal model of the blockchain (13)

Subgame perfection

Conservative mining need not constitute a subgame-perfect equilibrium.

Example 1. Let $n = 2$ and $T = 3$, and consider the blockchain \mathbb{B}_2 :



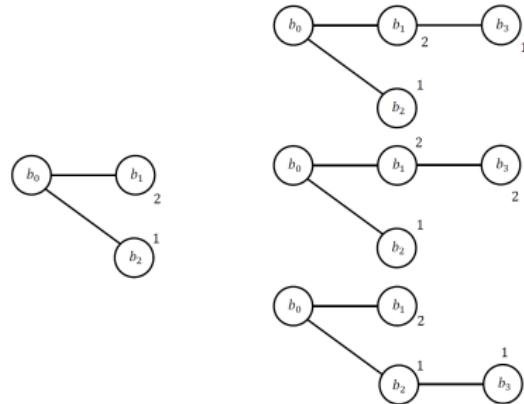
Thus, miner 1 mined b_2 , not following the conservative strategy.

Formal model of the blockchain (14)

Subgame perfection (continued)

At stage $T = 3$, the last block of the original chain is b_1 .

However, it is optimal here for miner 1 to work on b_2 because this allows her, with probability 1/2, to realize a token for the block b_2 .



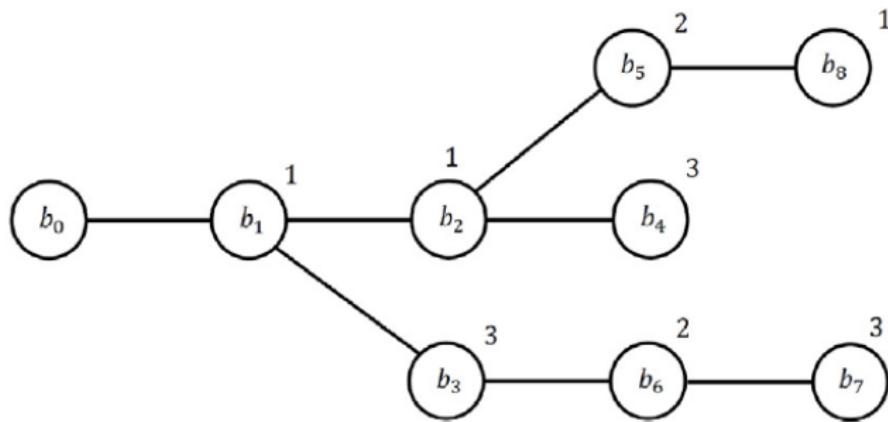
Thus, conservative mining is **not subgame-perfect**.

Formal model of the blockchain (15)

Subgame perfection (continued)

Longest-chain mining need not constitute a subgame-perfect equilibrium either.

Example 2. Let $n = 3$ and $T = 6$, and consider the blockchain \mathbb{B}_5 :



Formal model of the blockchain (16)

Discussion

The model captures the **interplay between two forces**:

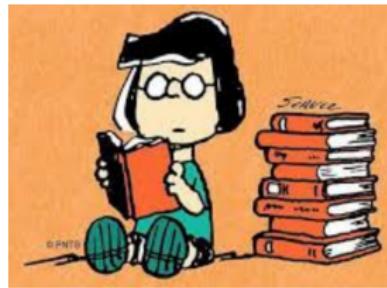
- Coordination problem between players
- Problem of vested interests

Formal model of the blockchain (18)

Bibliographic notes

The framework introduced above is a simplified version of the model used by Biais et al. (2019). See Ewerhart (2020).

Eyal and Sirer (2018) show with the help of a Markov chain model that selfish mining can be profitable.



Formal model of the blockchain (17)

References

- Biais, B., Bisiere, C., Bouvard, M., Casamatta, C. (2019), The blockchain folk theorem, *Review of Financial Studies* **32**, 1662-1715.
- Ewerhart, C. (2020), Finite blockchain games, *Economics Letters* **197**, 109614. ([link](#))
- Eyal, I., Sirer, E.G. (2018), Majority is not enough: Bitcoin mining is vulnerable, *Communications of the ACM* **61**, 95-102.

Smart Contracts and Blockchain Technology

Lecture 5. Equilibrium in the blockchain game

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Blockchain game

This lecture:

- Equilibrium in the blockchain game
- Selfish mining

Equilibrium in the blockchain game (1)

Strategies

Definition. A strategy s_i for miner i selects a block from any given blockchain.

The formal definition goes as follows:

- Suppose that $\mathbb{B} = (B, \leqslant, \iota)$ is a (state of the) blockchain,...
- ...with $B = \{b_0, b_1, \dots, b_T\}$ being the ordered set of blocks,...
- ...then $s_i(\mathbb{B}) \in B$.

Equilibrium in the blockchain game (2)

Some canonical mining strategies

Definition

- Miner i is *conservative* if she always chooses the last block of the original chain.
- Miner i follows the *longest-chain rule* if she always chooses the last block of one of the longest chains.
- Miner i adheres to *naïve mining* if she maximizes the expected number of her tokens under the assumption that the current stage is the last one.

Note: Conservative mining is a well-defined strategy, whereas the longest-chain rule and naïve mining each characterizes a set of strategies.

Equilibrium in the blockchain game (3)

The blockchain game

The **set of players** is $N = \{1, \dots, n\}$.

Denote the **set of strategies** (identical for all miners) by S .

Miner i 's **payoff function** (the expected number of tokens) is denoted by $\Pi_i(s_i; s_{-i}) = \Pi_i(s_1, \dots, s_n)$, where $s_{-i} = (s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n)$.

For any given time horizon $T \geq 0$, this defines a **symmetric noncooperative game**.

Equilibrium in the blockchain game (4)

Nash equilibrium

Definition. An n -tuple of strategies $(s_1^*, \dots, s_n^*) \in S^n$ is a **Nash equilibrium** if $\Pi_i(s_i^*; s_{-i}^*) \geq \Pi_i(s_i; s_{-i}^*)$ for any $s_i \in S$ and $i \in N$.

Thus, each player's strategy maximizes her expected payoff under the assumption that all the other players adhere to their respective equilibrium strategies.

Definition. A Nash equilibrium (s_1^*, \dots, s_n^*) is **symmetric** if $s_1^* = \dots = s_n^*$.

Equilibrium in the blockchain game (5)

Conservative mining

Proposition

Conservative mining constitutes a symmetric Nash equilibrium.

Equilibrium in the blockchain game (6)

Proof of the equilibrium property of conservative mining

We assume that all miners $j \in N \setminus \{i\}$ are conservative.

We have to show that, then, miner i likewise weakly prefers to be conservative.

Equilibrium in the blockchain game (7)

Proof of the equilibrium property of conservative mining (continued)

Suppose first that i is conservative, like all other miners.

Then, the blockchain develops into a single chain consisting of $T + 1$ blocks, and miner i receives one token for each block that she mines.

Equilibrium in the blockchain game (8)

Proof of the equilibrium property of conservative mining (continued)

Suppose, instead, that miner i deviates and works on a block that is not the last block of the original chain.

Then, miner i creates a fork with positive probability.

As a result, she does not necessarily receive one token for each block that she mines.

Equilibrium in the blockchain game (9)

Proof of the equilibrium property of conservative mining (continued)

Thus, by deviating from conservative mining, miner i potentially lowers, but never raises the expected number of tokens.

Therefore, a deviation from conservative mining can never lead to a strictly higher expected payoff for miner i ! \square

Equilibrium in the blockchain game (10)

Miners using different strategies

Proposition. *Any combination of mining strategies consistent with conservative mining, the longest-chain rule, or naïve mining forms a (not necessarily symmetric) Nash equilibrium.*

Proof. Analogous to the previous proof!¹ □

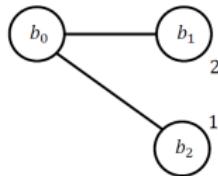
¹See the problem sets.

Equilibrium in the blockchain game (11)

Subgame perfection

Conservative mining need not constitute a subgame perfect equilibrium.

Example 1. Let $n = 2$ and $T = 3$, and consider the blockchain \mathbb{B}_2 :



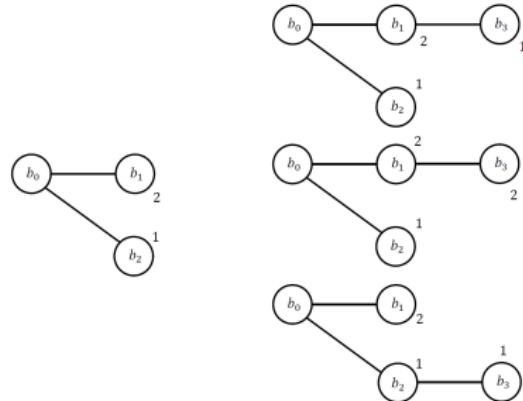
Thus, miner 1 worked on b_2 , not following the conservative strategy.

Equilibrium in the blockchain game (12)

Subgame perfection (continued)

At the beginning of stage $t = 3$, the last block of the original chain is b_1 .

However, it is optimal here for miner 1 to work on b_2 because this allows her, with probability 1/2, to realize a token for the block b_2 .



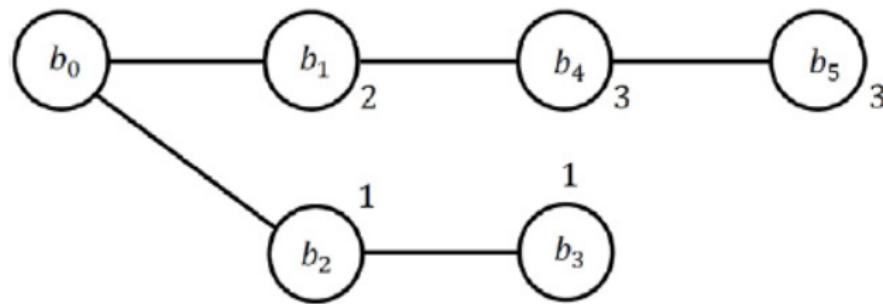
Thus, conservative mining is **not subgame perfect**.

Equilibrium in the blockchain game (13)

Subgame perfection (continued)

Longest-chain mining need not constitute a subgame perfect equilibrium either.

Example 2. Let $n = 3$ and $T = 6$, and consider the blockchain \mathbb{B}_5 :



Equilibrium in the blockchain game (14)

Discussion

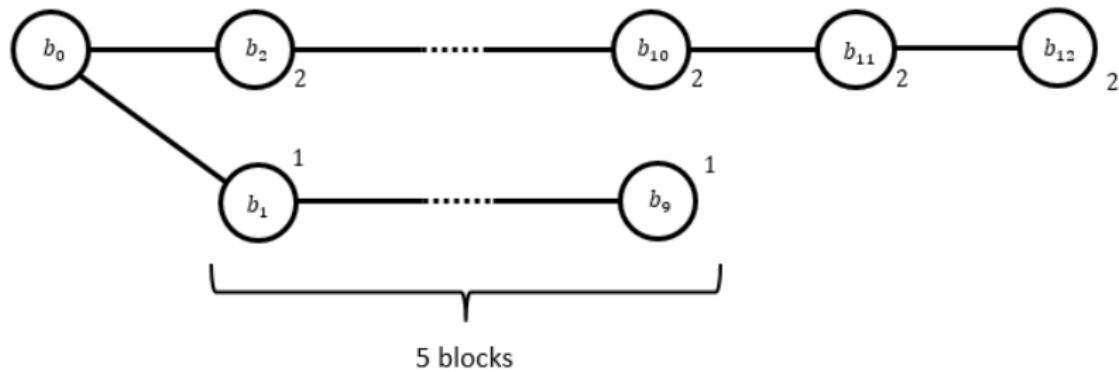
The model captures the **interplay between two forces**:

- Coordination problem between players
- Problem of vested interests

Equilibrium in the blockchain game (15)

Naïve mining is not subgame perfect

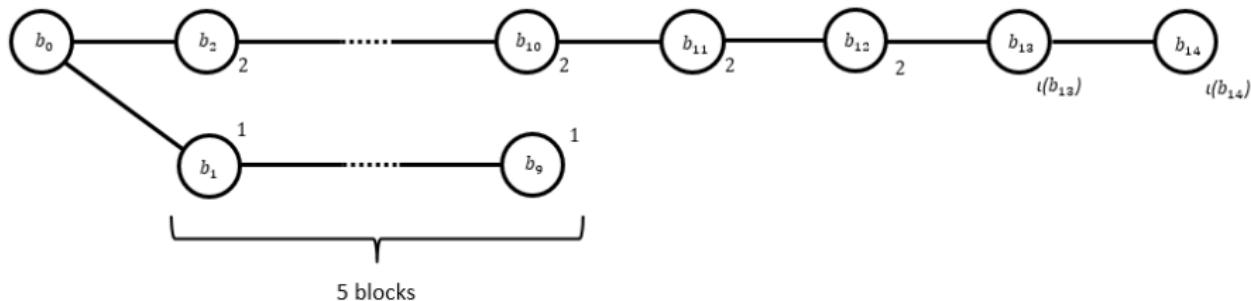
Example 3. Let $n = 2$ and $T = 14$, and consider the following blockchain \mathbb{B}_{12} :



Equilibrium in the blockchain game (16)

Naïve mining is not subgame perfect (continued)

If naïve, both miners work on the longest chain in stages
 $t \in \{13, 14\}$.



The expected payoff for miner 1 is

$$E[\Pi_1] = \frac{1}{4} \cdot 0 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 = 1. \quad (1)$$

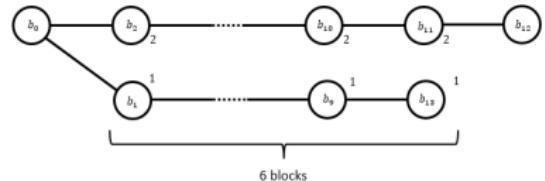
Equilibrium in the blockchain game (17)

Naïve mining is not subgame perfect (continued)

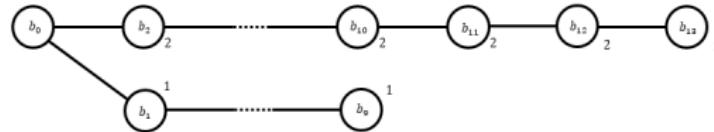
Suppose that miner 1 deviates and decides to work on b_9 in stage $t = 13$, while miner 2 continues to follow the naïve strategy.

Then, there are two scenarios.

Scenario 1 (50%). Miner 1 successfully mines block b_{13} :



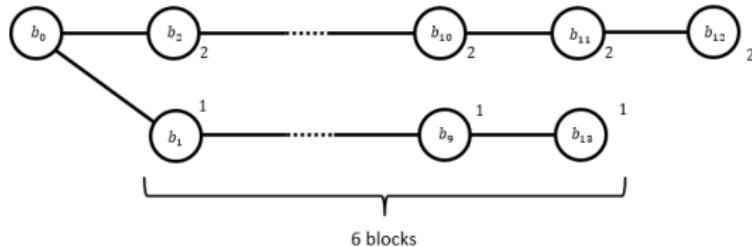
Scenario 2 (50%). Miner 2 successfully mines block b_{13} :



Equilibrium in the blockchain game (18)

Scenario 1

Suppose that miner 1 wins b_{13} .



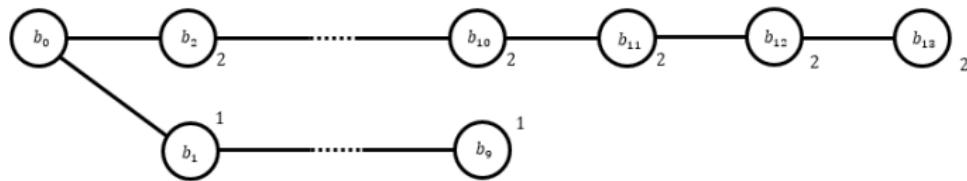
Then, miner 1 works next on b_{13} , while miner 2 works on b_{12} . Miner 1's final payoff is...

- $\Pi_1 = 7$ with probability 25% (if 1 wins and the lower chain is selected),
- $\Pi_1 = 0$ with probability 25% (if 1 wins yet the upper chain is selected), and
- $\Pi_1 = 0$ with probability 50% (if 2 wins).

Equilibrium in the blockchain game (19)

Scenario 2

Suppose that miner 2 wins b_{13} .



Then, there is another stage $t = 14$, and miner 1's payoff is

- $\Pi_1 = 1$ with probability 50% (if 1 wins), and
- $\Pi_1 = 0$ with probability 50% (if 2 wins).

Equilibrium in the blockchain game (20)

Benefit from a deviation

The expected payoff for miner 1 from the deviation is, consequently,

$$E[\Pi_1] = \frac{1}{2} \cdot \frac{1}{4} \cdot 7 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 = 1.125 > 1. \quad (2)$$

Therefore, naïve mining is not subgame perfect! \square

Remark: As of today, a subgame perfect equilibrium is not known for the blockchain game...

Equilibrium in the blockchain game (21)

Selfish mining

It is often argued that the Bitcoin mining protocol is stable provided that **more than half** of the hash power lies with honest miners.

This position ignores the possibility of **selfish mining**:

- A pool may strategically delay the broadcast of a successfully mined block.
- The benefit for the pool is that honest miners waste their hash power on side chains that become orphaned soon after.

Equilibrium in the blockchain game (22)

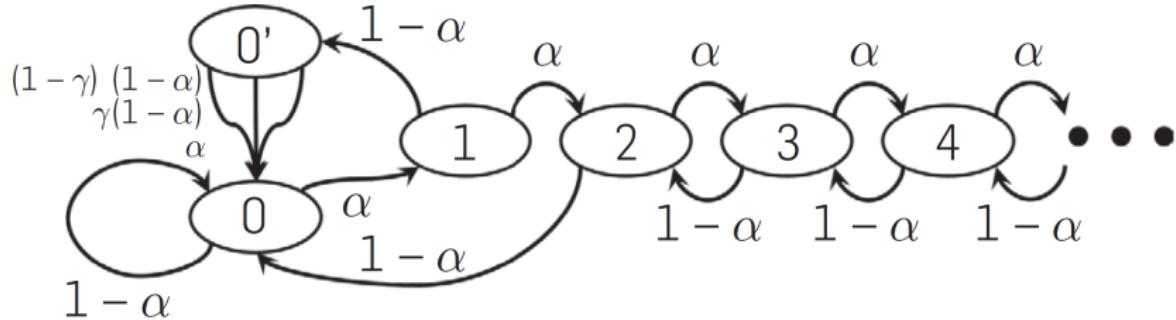
Strategy of selfish mining

Relative hash power of selfish pool: $\alpha \in (0, 1)$

Share of honest miners that work on the pool's block $\gamma \in (0, 1)$

State $0'$: Fork where honest miners and pool each have mined a block

State $a \in \{0, 1, 2, \dots\}$: The pool's inventory of secret blocks

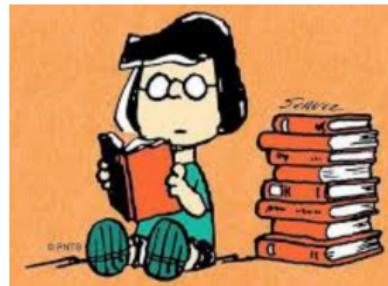


Equilibrium in the blockchain game (23)

Selfish mining

Nash equilibrium was introduced by Nash (1950). Subgame-perfect equilibrium was conceptualized by Selten (1965).

Eyal and Sirer (2018) show with the help of a Markov chain model that selfish mining can be profitable.



Formal model of the blockchain (24)

References

Eyal, I., Sirer, E.G. (2018), Majority is not enough: Bitcoin mining is vulnerable, *Communications of the ACM* **61**, 95-102.

Nash Jr., John F. (1950), Equilibrium points in n -person games, *Proceedings of the National Academy of Sciences* **36**, 48-49.

Selten, R. (1965), Spieltheoretische Behandlung eines Oligopolmodells mit Nachfrageträgheit: Teil I: Bestimmung des dynamischen Preisgleichgewichts, *Zeitschrift für die gesamte Staatswissenschaft/Journal of Institutional and Theoretical Economics*, (Heft 2), 301-324.

Smart Contracts and Blockchain Technology

Lecture 6. Cryptographic underpinnings

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture

- Equilibrium in the blockchain game
- Selfish mining

This lecture: Cryptographic underpinnings

- Binary and hexadecimal numbers
- Hash functions
- Private and public keys
- Finite fields
- Discrete logarithm problem

Cryptographic underpinnings (1)

Basics on binary and hexadecimal numbers

A **binary number** is a technical representation of a number based on powers of 2 (rather than 10 which is used for decimal numbers).

Each **binary digit** is taken from the set $\{0,1\}$.

Example: $0b10100011 = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^1 + 1 \cdot 2^0 = 163$, where the **prefix** 0b indicates a binary number.

Bits are the units of information. For instance, a die with six faces bears information

$$I = \log_2(6) = \frac{\ln 6}{\ln 2} = 2.585 \text{ bit},$$

i.e., somewhat less than three bits.

Cryptographic underpinnings (2)

Basics on binary and hexadecimal numbers (continued)

A **hexadecimal number** is, in complete analogy, a technical representation of a number based on powers of 16.

Each **hexadecimal digit** is taken from the set $\{0, 1, 2, \dots, 9, a, \dots, f\}$.



Example: $0xc03 = 12 \cdot 16^2 + 0 \cdot 16^1 + 3 \cdot 16^0 = 3075$, where the **prefix** $0x$ indicates a hexadecimal number.

Cryptographic underpinnings (3)

Basics on binary and hexadecimal numbers (continued)

Hexadecimal numbers are useful because they are both compact and easily transformed into binary numbers:

- $0x0=0b0000,$
- $0x1=0b0001,$
- $0x2=0b0010,$
- ...
- $0xf=0b1111.$

Two hexadecimal numbers correspond to **one byte** (8 bits), which has been, in particular, the word length of the first commercially successful microprocessors (e.g., Intel 8080, Motorola 6800).

Cryptographic underpinnings (4)

Cryptography is a branch of mathematics used extensively in computer security.

The term means “secret writing” in Greek, and refers originally to the encryption of messages.

However, cryptographic methods are also used:

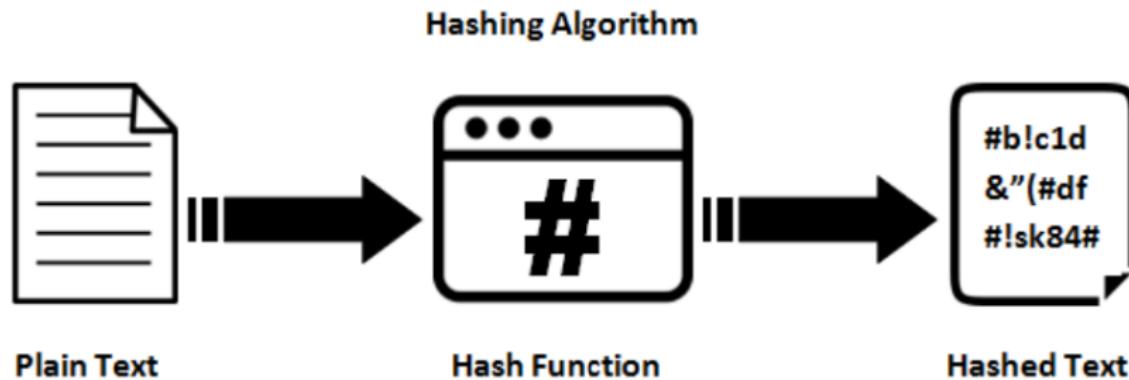
- to identify, and prove the integrity of, data (with a **digital fingerprint**, also known as hash),
- to prove the authenticity of a message (e.g., with a **digital signature**).

These methods are critical, in particular, to the operation of blockchain systems and smart contract applications.

Cryptographic underpinnings (5)

Digital fingerprints

Definition. A hash function maps data of arbitrary size to data of fixed size.



Cryptographic underpinnings (6)

Hash functions on bit strings

Denote by $\mathcal{A} = \{0, 1\}$ the **alphabet** consisting of the set of binary values, and by

$$\mathcal{A}^* = \{ "", "0", "1", "00", "01", "10", "11", "000", "001", \dots \}$$

the set of **words** over \mathcal{A} . Then, a hash function is any mapping

$$\psi : \mathcal{A}^* \rightarrow \mathcal{A}^N = \underbrace{\mathcal{A} \times \dots \times \mathcal{A}}_{N \text{ times}}, \quad (1)$$

where $N \geq 1$ is the length of the hash.

Cryptographic underpinnings (7)

Use cases of digital fingerprints

Data integrity. Comparing hash values can determine whether any changes have been made to a given data set.

Proof of work. Winning the block reward requires finding a hash with given properties.

Password verification. To prevent password theft, only a hash of a user password is communicated to the backend system.

Cryptographic underpinnings (8)

Check bits

For a word $w \in \mathcal{A}^*$, denote by

$$\psi(w) = \begin{cases} 0 & \text{if the number of 1's in } w \text{ is even} \\ 1 & \text{if the number of 1's in } w \text{ is odd} \end{cases} \quad (2)$$

the **parity bit**.

For example, $\psi("1010111") = 1$.

Examples

- American Standard Code for Information Interchange (**ASCII**):
7 bits for the code plus one parity bit, corresponding to one byte
(8 bits)

Cryptographic underpinnings (9)

USASCII code chart

b ₇ b ₆ b ₅				0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1	
B ₄ B ₃ B ₂ B ₁				0	1	2	3	4	5	6	7	
				0 0 0 0 0	NUL	DLE	SP	0	@	P	'	p
				0 0 0 1 1	SOH	DC1	!	1	A	Q	a	q
				0 0 1 0 2	STX	DC2	"	2	B	R	b	r
				0 0 1 1 3	ETX	DC3	#	3	C	S	c	s
				0 1 0 0 4	EOT	DC4	\$	4	D	T	d	t
				0 1 0 1 5	ENQ	NAK	%	5	E	U	e	u
				0 1 1 0 6	ACK	SYN	&	6	F	V	f	v
				0 1 1 1 7	BEL	ETB	'	7	G	W	g	w
				1 0 0 0 8	BS	CAN	(8	H	X	h	x
				1 0 0 1 9	HT	EM)	9	I	Y	i	y
				1 0 1 0 10	LF	SUB	*	:	J	Z	j	z
				1 0 1 1 11	VT	ESC	+	;	K	C	k	{
				1 1 0 0 12	FF	FS	,	<	L	\	l	l
				1 1 0 1 13	CR	GS	-	=	M	J	m	}
				1 1 1 0 14	SO	RS	.	>	N	^	n	~
				1 1 1 1 15	SI	US	/	?	O	—	o	DEL

The **eighth bit** was used to check if the data was correct, e.g., on a punched tape for printer data.

Cryptographic underpinnings (10)

The Luhn Algorithm for credit card numbers

2x	2x	2x	2x	2x	2x	2x	2x	2x	2x	2x	X				
5	4	5	7	6	2	3	8	9	8	2	3	4	1	1	X
2x5=10 (1+0)	2x4=8	2x5=10 (1+0)	2x7=14 (1+2)	2x6=12 (1+2)	2x2=4	2x3=6	2x8=16 (1+6)	2x9=18 (1+8)	2x8=16 (1+6)	2x2=4	2x3=6	2x4=8 (1+8)	2x1=2	2x1=2	=34
4	7	1	2	6	8	8	8	4	8	3	3	8	1	2	=33
														=67	

To calculate the check digit, multiply every even-position digit (when counted from the right) in the number by two. If the result is a two digit number, then add these digits together to make a single digit (this is called the *digital root*).

To this total, we then add every odd-position digit.

This will result in a total (in our example =67). The check-digit is what number needs to be added to this total to make the next multiple of 10. In our case, we'd need to add 3 to make 70. So the check-digit for this fictitious number is 3.

Also, payment operators use simple methods for payment card identification:

$$\psi("4362\ 3245\ 2314\ 0012") = "****\ ****\ ****\ *012"$$

Cryptographic underpinnings (11)

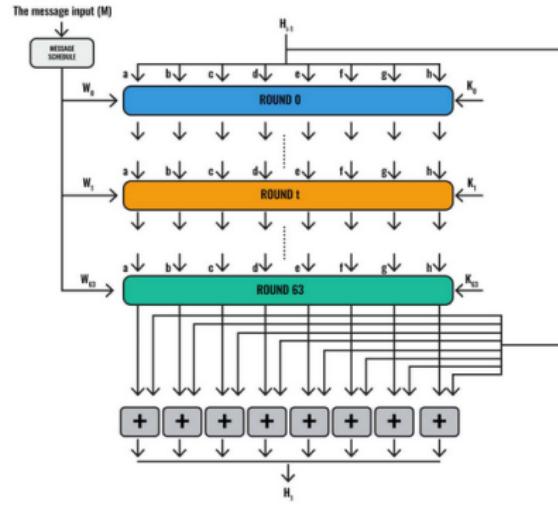
Properties of hash functions used in blockchain systems

- **Determinism.** A given input message always produces the same hash output.
- **Verifiability.** Computing the hash of a message is efficient (linear complexity).
- **Noncorrelation.** A small change in the message (e.g., a 1-bit change) should change the hash output so extensively that it cannot be correlated to the hash of the original message.
- **Irreversability.** Computing the message from its hash is infeasible, equivalent to a brute-force search through all possible messages.
- **Collision protection.** It should be infeasible to calculate two different messages that produce the same hash output.

Cryptographic underpinnings (12)

Secure Hash Algorithm

The **SHA-256** falls into the class SHA-2, which was created by the United States National Security Agency (NSA) as a successor to SHA-1 in 2001.¹



¹Irrespective of the size of input data, the hash will always consist of **256 bits**.

Cryptographic underpinnings (13)

Private keys

Definition. A **private key** is an element of a finite set of feasible keys, $\pi \in \Pi = \{\pi_1, \dots, \pi_N\}$, where $N = \#\Pi$ denotes the cardinality of the set of feasible keys.

Cryptographic underpinnings (14)

Examples of private keys

PIN for payment card (4 digits):

$\Pi = \{“0000”, “0001”, \dots, “9999”\}$, with $N = 10^4 = 10'000$.

Phone PIN (between 4 and 6 digits):

$\Pi = \{“0000”, \dots, “9999”\} \cup \{“00000”, \dots, “99999”\} \cup \{“000000”, \dots, “999999”\}$, with $N = 10^4 + 10^5 + 10^6 = 1'110'000$.

Alphanumeric password (e.g., between 8 and 12 symbols, at least one upper-case and one-lower case letter, at least one symbol):

“#qwerty123”, with $N \approx 10^{24}$.

Cryptographic underpinnings (15)

Risks of private keys

Possession of a private key is the root of user control. Therefore, holders of private keys are exposed to the following risks:

- **Private key loss.** If a private key is lost, it cannot be recovered and control may be lost forever. Therefore, a private key must be backed up and protected from accidental loss.
- **Private key compromise.** A private key must remain secret at all times. Revealing it to a third party is equivalent to sharing control (then, it may not be feasible to disentangle who authenticated a transaction).
 - The key has been revealed to an unauthorized party (stolen key)
 - The key *may* have been revealed to an unauthorized party (uncertainty)
 - An attacker has identified the private key by guesswork and/or trial-and-error techniques.

Cryptographic underpinnings (16)

Most common passwords

Rank	2021
1	123456
2	123456789
3	12345
4	qwerty
5	password
6	12345678
7	111111
8	123123
9	1234567890
10	1234567

Source: nordpass.com.

Cryptographic underpinnings (17)

Generation of a private key

- **Offline** and **not chosen by a human**
- **Secure source of randomness**, e.g.:
 - Mouse-wiggling
 - Cosmic radiation noise from microphone channel
 - Quantum random generator
- Pseudo random number generator must be **cryptographically secure** (CSPRNG)²

²CSPRNG requirements fall into two groups: (1) they pass statistical randomness tests, (2) they hold up well under serious attack, even when part of their initial or running state becomes available to an attacker.

Cryptographic underpinnings (18)

Ethereum private keys

A private key in the Ethereum system is a binary number with 256 digits (corresponding to a hexadecimal number with 64 digits).

Example:

0xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

$N = 16^{64} = 2^{256} \approx 1.1579 \cdot 10^{77}$ (the estimated number of atoms in the visible universe is 10^{80}).

Cryptographic underpinnings (19)

Public keys

The public key is generated from the private key using a function that is easy to compute but practically irreversible. Such functions are called **one-way functions** or **trapdoor functions**.

Examples:

- Exponentiation in a finite field (discrete logarithm problem)
- Scalar multiplication on elliptic curves

Note: Also hash functions are one-way functions. However, in contrast to the functions considered here, they take data of arbitrary size and lack the mathematical properties (homomorphy) needed to run digital signature protocols.

Cryptographic underpinnings (20)

Finite fields

Finite fields are finite mathematical structures that are the basis for cryptographic proofs. An important example are **residue class fields with prime characteristic**.³

Example. The residue class field \mathbb{F}_7 consists of

- the set of residue classes $\{0, 1, 2, 3, 4, 5, 6\}$
- the operation of addition modulo 7, e.g.,

$$2 + 6 \equiv 1 \pmod{7} \tag{3}$$

- the operation of multiplication modulo 7, e.g.,

$$5 \cdot 4 \equiv 6 \pmod{7}. \tag{4}$$

³For any prime power p^m , there exists precisely one finite field. Residue class fields (where $m = 1$) are of an elementary nature.

Cryptographic underpinnings (21)

Primitive roots

Example. In the residue class field \mathbb{F}_7 , consider the powers of 3:

$$3^0 \equiv 1 \pmod{7}$$

$$3^1 \equiv 3 \pmod{7}$$

$$3^2 \equiv 2 \pmod{7}$$

$$3^3 \equiv 6 \pmod{7}$$

$$3^4 \equiv 4 \pmod{7}$$

$$3^5 \equiv 5 \pmod{7}$$

Note that each nonzero residue class corresponds to some power of 3 (this property makes 3 a **primitive root** or **generator** of the multiplicative group $\mathbb{F}_7^* = \mathbb{F}_7 \setminus \{0\}$).

Cryptographic underpinnings (22)

Table of primitive roots

Analogously, one defines \mathbb{F}_p for any prime number $p \in \{2, 3, 5, 7, 11, 13, 17, 19, 23, \dots\}$.

p	Primitivwurzeln modulo p
2	1
3	2
5	2,3
7	3,5
11	2,6,7,8
13	2,6,7,11
17	3,5,6,7,10,11,12,14
19	2,3,10,13,14,15
23	5,7,10,11,14,15,17,19,20,21

Cryptographic underpinnings (23)

The discrete logarithm problem

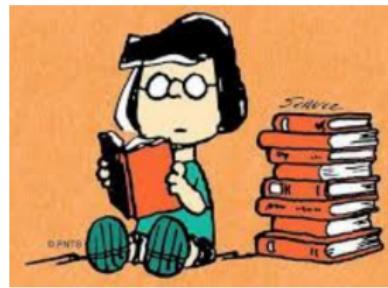
The **discrete logarithm problem** is to find, for a given residue class r , the exponent m such that $g^m \equiv r \pmod{p}$.

For large primes p , this can be a very difficult problem.

Cryptographic underpinnings (24)

Bibliographic notes

This chapter is loosely based on Antonopoulos and Wood (2018, Ch. 4).



Cryptographic underpinnings (25)

References

Antonopoulos, A.M., Wood, G. (2018), *Mastering Ethereum: Building Smart Contracts and Dapps*, O'Reilly Media.

Smart Contracts and Blockchain Technology

Lecture 7. Digital signatures

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Cryptographic underpinnings

- Hash functions
- Residue class fields
- Discrete logarithm problem

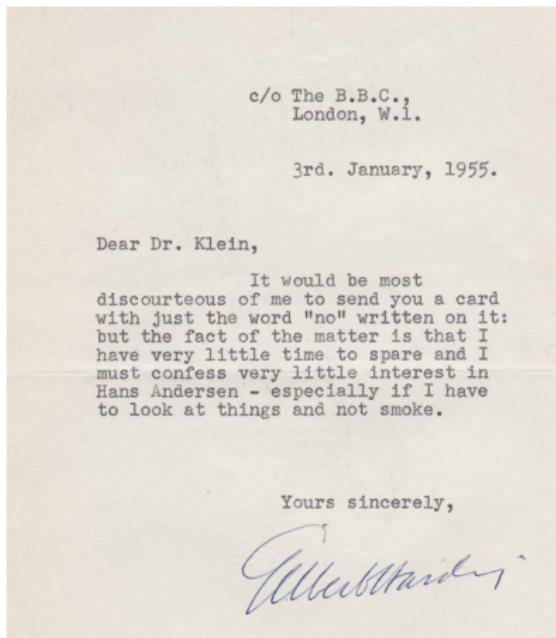
This lecture: Digital signatures

- Private and public keys
- Ethereum wallet addresses
- Elliptic curve cryptography

Digital signatures (1)

A problem

Signing a letter is easy...



...but how do we sign an **electronic message**?

Digital signatures (2)

Public-key cryptography

Public-key cryptography, also known as asymmetric cryptography, is the basis of modern information security.

- Asymmetric means: There is **no shared secret!**

The protocol allows a sender to digitally sign a message. This allows the receiver of the message to verify that:

- the message comes from the sender (**authentication**), and
- the message has not been modified (**integrity**).

Non-repudiation is a closely related legal concept, which captures that there is evidence usable in court that proves that the sender has created a specific message.

Digital signatures (3)

Asymmetric cryptography

Asymmetric cryptography uses a combination of a **private key** and a **public key**. An example for a key pair in Ethereum is:

A 256-bit **private key** in hexadecimal representation:

k=0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315

A corresponding 520-bit **public key**, likewise in hexadecimal representation:

K=0x046e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b
83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0

Public key -> prefix 0x04 (8 bits) and is followed by two elliptic coordinates
-> public key is uncondensed

Digital signatures (4)

Trapdoor functions

The public key is computed as the value of a **trapdoor function** when evaluated at the private key.



A trapdoor function is:

- **difficult to invert** (just as a hash function), but in contrast to a simple hash function, it is also
- **homomorphic**, meaning that arithmetic operations on the input translate into arithmetic operations on the output.

Digital signatures (5)

Examples for trapdoor functions

Exponentiation in finite fields

- Computing the residue class $r \equiv g^m \pmod{p}$ for some generator $g \in \mathbb{F}_p$, with p prime, is simple.¹
- However, computing the **discrete logarithm** m from r (given the prime p and the generator g) is very difficult.

Scalar multiplication on elliptic curves

- Scalar multiplication on elliptic curves over a finite field \mathbb{F}_p is comparably simple (as will be seen).
- However, inverting the operation is nearly impossible if p is sufficiently large.

¹E.g., computing $g^{37} \equiv g^{32} \cdot g^4 \cdot g^1$, with $g^4 \equiv (g^2)^2$ and $g^{32} = ((g^4)^2)^2$ requires only six multiplications. This technique is known as **repeated squaring** or **square & multiply**.

Digital signatures (6)

Determination of the Ethereum address

Private key k (a number)
-> Trapdoor
Public key $k \cdot G = K \rightarrow K = (x_k, y_k)$
-> generator of $(E, +)$ elliptic curve
 $E = (x_E, y_E)$

Start with the public key:

$K = 0x046e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b$
 $83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0$

The prefix 0x04 is dropped before the hash is computed:

$K' = 6e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b$
 $83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0$

Compute the Keccak256 of the byte code:

$\text{Keccak256}(K') =$

$0x2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9$

To obtain the **Ethereum wallet address**, keep the last 20 bytes
(and add the prefix 0x, as usual):

$0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9$

Digital signatures (7)

Elliptic curves

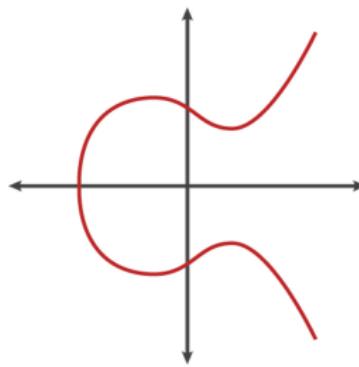
An **elliptic curve E** over the real numbers consists of:

- the set of solutions $(x, y) \in \mathbb{R}^2$ of a cubic equation of the form

$$y^2 = x^3 + ax + b, \quad (1)$$

where $a, b \in \mathbb{R}$ are constants such that $\Delta \equiv 4a^3 - 27b^2 \neq 0$, and

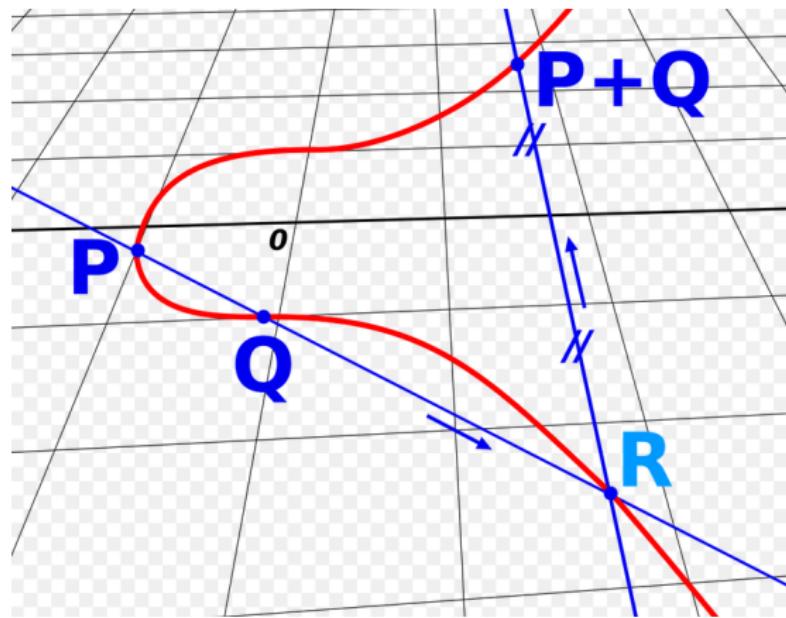
- a **point at infinity**, denoted by \mathcal{O} .



Digital signatures (8)

Addition on elliptic curves

Surprising fact: Two points $P, Q \in \mathbf{E}$ can be geometrically **added** to obtain a new point $P + Q \in \mathbf{E}$.



Digital signatures (9)

Addition on elliptic curves (cont.)

This also works in “nongeneric cases”:

$$P + \mathcal{O} = P, \tag{2}$$

$$P + (-P) = \mathcal{O}, \tag{3}$$

where $-P = (x, -y)$ is the **inverse** of P .

Lemma:

- $P + Q = Q + P$ (**commutativity**)
- $(P + Q) + S = Q + (P + S)$ (**associativity**) → brackets are not needed...

Digital signatures (10)

Algebraic counterpart of addition

The **slope** of the line connecting two points P and Q (in generic position) is given as

$$\sigma = \frac{y_P - y_Q}{x_P - x_Q}. \quad (4)$$

Then the **sum** $P + Q$ has the coordinates

$$x_{P+Q} = \sigma^2 - x_P - x_Q, \quad (5)$$

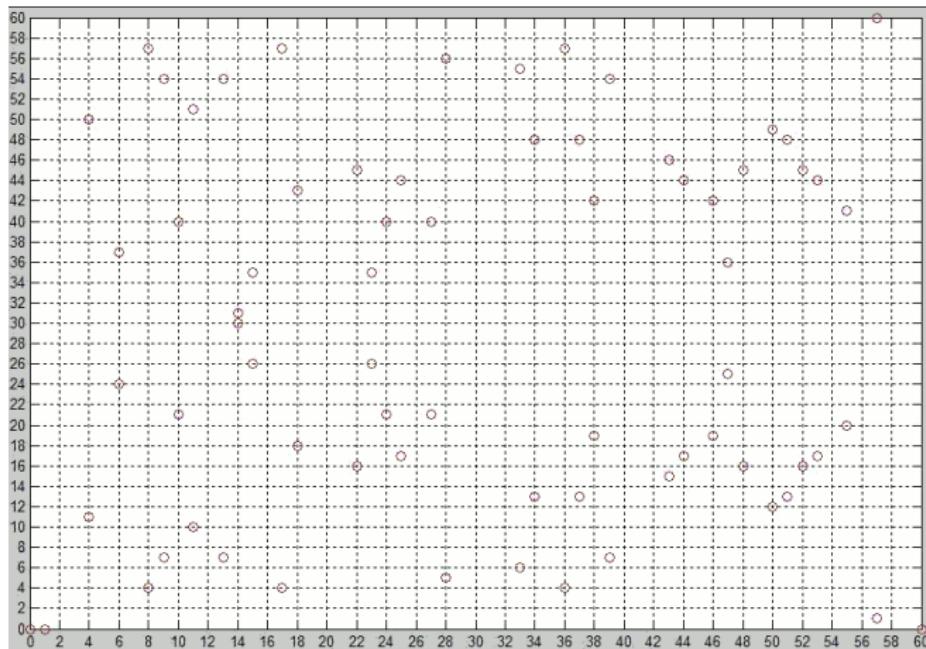
$$y_{P+Q} = -y_P + \sigma(x_P - x_{P+Q}). \quad (6)$$

Another surprising fact: These geometrically motivated formulas work in the same way for elliptic curves over finite fields (i.e., where $(x, y) \in \mathbb{F}_p$ for some prime p).

Digital signatures (11)

Elliptic curve over a finite field

Shown is the set of solutions to $y^2 = x^3 - x$ over the finite field \mathbb{F}_{61} .



Digital signatures (12)

Scalar multiplication on elliptic curves

Scalar multiplication by n corresponds to an iterated addition:

$$n \cdot P = \underbrace{P + \dots + P}_{n \text{ times}} \quad (7)$$

This is the **trapdoor function**:

- Scalar multiplication is easy.²
- However, determining n from $Q = n \cdot P$ and P can be very hard.

This is called the **discrete logarithm problem for elliptic curves**.

²Use a variant of the square & multiply technique: “double & add”

Digital signatures (13)

The elliptic curve used by Ethereum and Bitcoin

Ethereum and Bitcoin use the same elliptic curve, called **secp256k1**.

That elliptic curve is defined over the finite field \mathbb{F}_p through

$$y^2 \equiv x^3 + 7 \pmod{p}, \quad (8)$$

where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, which is a very large prime.³

The generator used is (prefix 04 followed by x and y coordinates):

G=0479BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798

483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8

³ In fact, $p = 115\,792\,089\,237\,316\,195\,423\,570\,985\,008\,687\,907\,853\,269\,984\,665\,640\,564\,039\,457\,584\,007\,908\,834\,671\,663$.

Digital signatures (14)

Signing a message

Suppose you plan to send an **electronic message** m , and you wish to add a digital signature so that the receiver can verify that the received message is authentic and unchanged.

You create a cryptographically secure random number q , the **ephemeral private key**.

Then, you compute the **ephemeral public key** $Q = q \cdot G$ on the elliptic curve $\mathbf{E} = \text{secp256k1}$.

Then, the **digital signature** $(r, s) \in \mathbb{F}_p \times \mathbb{F}_p$ is given as

$$r = x_Q, \tag{9}$$

$$s \equiv q^{-1}(\text{Keccak256}(m) + rk) \bmod p. \tag{10}$$

Digital signatures (15)

Verifying a signature

Suppose you receive a message m , augmented by a digital signature (s, r) .

Calculate in \mathbb{F}_p :

$$w \equiv s^{-1} \pmod{p}, \quad (11)$$

$$u_1 \equiv \text{Keccak256}(m)w \pmod{p}, \quad (12)$$

$$u_2 \equiv rw \pmod{p}, \quad (13)$$

Next, calculate on the elliptic curve $\mathbf{E} = \text{secp256k1}$. the point

$$\hat{Q} = u_1 \cdot G + u_2 \cdot K. \quad (14)$$

If $x_{\hat{Q}} = r$, then **the signature is valid!**

Digital signatures (16)

Crucial point

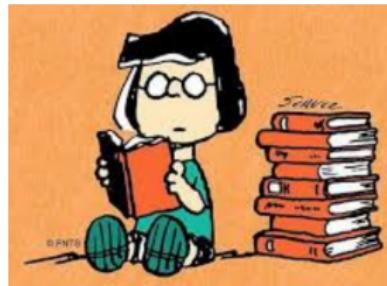
Each signature requires a new ephemeral private key.

Digital signatures (17)

Bibliographic notes

This chapter is based on Antonopoulos and Wood (2018, Chapters 4 and 6).

The basic idea underlying asymmetric cryptography is due to Diffie and Hellman (1976) and Merkle (1978).



Digital signatures (18)

References

Antonopoulos, Andreas M., and Gavin Wood. Mastering ethereum: building smart contracts and dapps. O'Reilly Media, 2018.

Diffie, W., & Hellman, M. E. (1976). New directions in cryptography (1976) <https://doi.org/10.1109>.

R. C. Merkle, "Secure communication over an insecure channel." Common. Ass. Comput. Mach., vol. 21. pp. 294-299, Apr. 1978.

Smart Contracts and Blockchain Technology

Lecture 8. Smart contracts

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Digital signatures

This lecture: Smart contracts

- What is a smart contract?
- Smart contract platforms
- Ethereum
- Solidity

Smart contracts (1)

What is a smart contract?

A **smart contract** is a piece of code that can be deployed and run on a blockchain system.



Smart contracts (2)

Characteristics of smart contracts

Immutable: Once deployed, the code of a smart contract cannot be changed.¹

Decentralized: Execution of the code, and validation of the results, happens identically on numerous computers in parallel.

Deterministic: Starting from a given state, the code always delivers the same results (i.e., there is no randomness).

¹In practice, the immutability constraint can be relaxed in a number of ways. First, smart contracts may **self-destruct**. Second, smart contracts can be embedded into **decentralized autonomous organizations** that admit changes to their protocols in response to the outcome of votes. Similarly, contracts may be set up in a modular way so as to allow changes to be implemented by linking to **new modules**.

Smart contracts (3)

Some use cases

- Multi-signature wallets
- Voting and decentralized autonomous organizations (DAOs)
- Tokens, cryptoassets, and stablecoins
- Crowdfunding platforms
- Decentralized exchanges
- Smart insurance
- NFTs
- Central Bank Digital Currency (CBDC)

Smart contracts (4)

Contracts vs. smart contracts

Note: A smart contract is not a contract in the legal sense!

A **contract** in the legal meaning is an agreement between parties, creating mutual obligations that are enforceable by law.



Important characteristics of a legally enforceable contract are:

- mutual assent, expressed by a valid **offer and acceptance**
- adequate consideration and capacity
- legality

The popular phrase "**The code is the law**" reflects the anarchic mindset of early blockchain enthusiasts.

Smart contracts (5)

Contract addresses

Any Ethereum address in use is either an externally owned address or a contract account:

- An **externally owned account (EOA)** is an account created by or for human users of the Ethereum network.
- A **contract account** is an account containing code that executes whenever it receives a transaction from another account (EOA or contract).

Deployment: A transaction is sent to the contract creation address 0x0.

Smart contracts (6)

Dormant nature

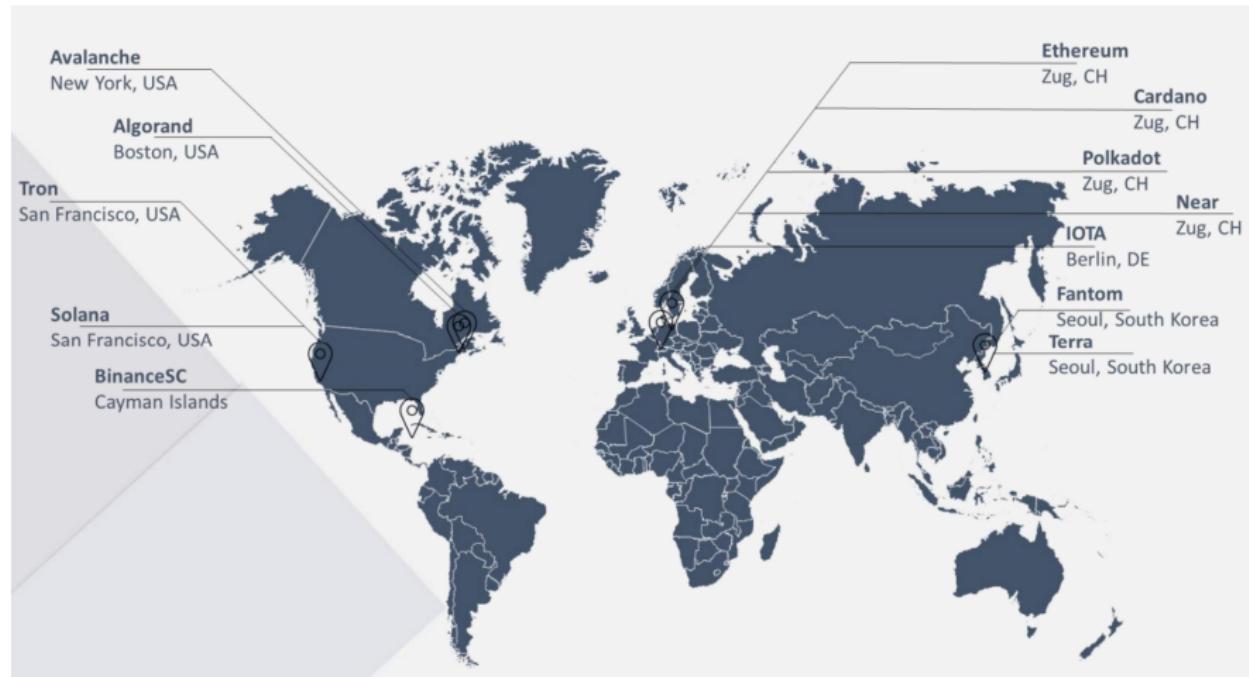
Smart contracts do not run “on their own” or “in the background.”
Rather they stay dormant until called by a transaction.²



²The picture represents the vow scene in the movie *Corpse Bride*, a 2005 animated movie.

Smart contracts (7)

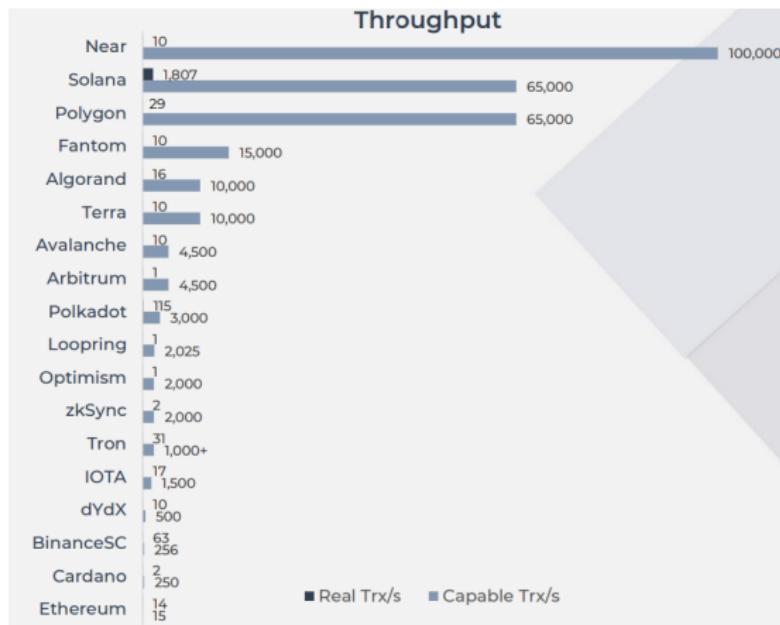
Layer-1 smart contract platforms³



³Source: Blockchain Presence

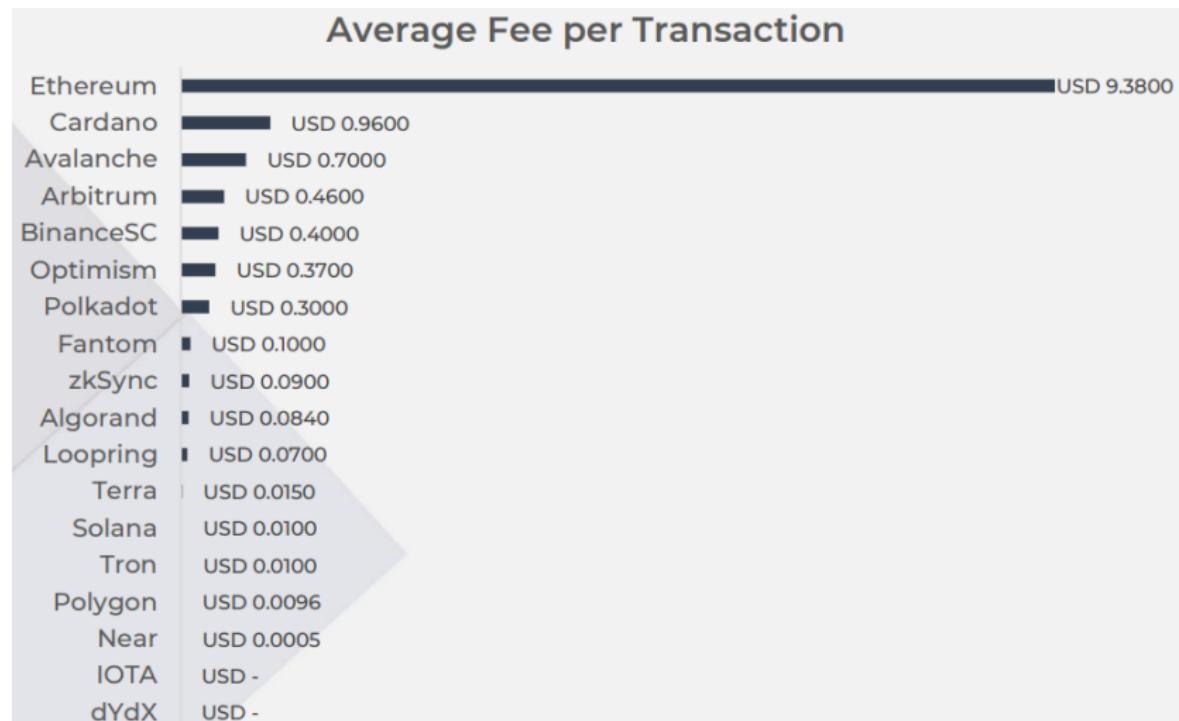
Smart contracts (8)

Transactions per second



Smart contracts (9)

Transaction costs



Smart contracts (10)

Ethereum

- the most decentralized smart contract platform
- by far the biggest developer base
- ability to be upgraded over time through EIPs
- introduced the most used token standards, e.g., ERC-20, ERC-721, ERC-1155
- merged with the Beacon chain to shift from PoW to PoS
- serves as the settlement layer for the several layer-2 solutions



Smart contracts (11)

Smart contract programming languages for Ethereum

Solidity⁴

- inspired by Javascript (the programming language that allowed the transition to an interactive web experience)

Vyper

- inspired by Python (the programming language that made coding feasible for everybody)

EVM bytecode

- the Ethereum virtual machine (EVM) is a stack-based virtual machine that executes bytecode ("machine code").⁵

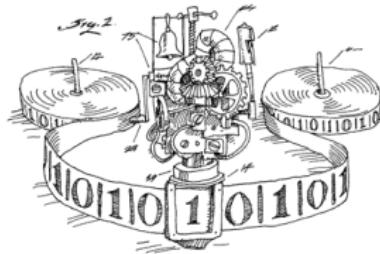
⁴See the [Solidity Documentation](#).

⁵Virtual means that EVM instructions are not directly executed at the hardware level, but are first converted into machine code before being executed by a central processing unit (CPU). See the [Ethereum yellow paper](#).

Smart contracts (12)

Turing completeness

A blockchain system with smart contract capability is called **Turing complete** if it can be used to simulate the operations of any Turing machine.⁶



Ethereum is Turing complete.⁷

⁶ A **Turing machine**, named after the British scientist Alan Turing (1912-1954), is a theoretical model of a computer with unbounded memory executing a finite code sequence.

⁷ In practice, however, gas costs render this concept rather theoretical.

Smart contracts (13)

The general structure of a smart contract written in Solidity

```
1 // SPDX-License-Identifier: GPL-3.0-only
2 pragma solidity >=0.7.0 <0.9.0;
3
4 contract Faucet {
5     // Accept any incoming amount
6     receive () external payable {}
7
8     // Give out ether to anyone who asks
9     function withdraw(uint withdraw_amount) public {
10         // Limit withdrawal amount
11         require(withdraw_amount <= 0.1 ether);
12         payable(msg.sender).transfer(withdraw_amount);
13     }
14 }
```

Smart contracts (14)

Some explanations

Comments. The double slash “//” introduces a comment line.⁸

License specification. It is good practice to include a license statement in your code. Under the [SPDX-License](#) GPL-3.0-only, others may freely copy, modify, and use your code.

Compiler instructions.⁹ The line starting with “pragma solidity” is not part of the smart contract but indicates that the code is written in Solidity. The constraints “>=0.7.0 <0.9.0” specify the compiler versions with which the code is compatible.

⁸ Comments are meant to inform readers of the code and have no implications for the workings of the smart contract.

⁹ A **compiler** is a software tool that converts code written in a high-level programming language such as Solidity into a lower-level language such as EVM bytecode.

Smart contracts (15)

`msg.sender`

`msg.sender` represents the address that initiated the contract call, not necessarily the originating EOA that sent the transaction.

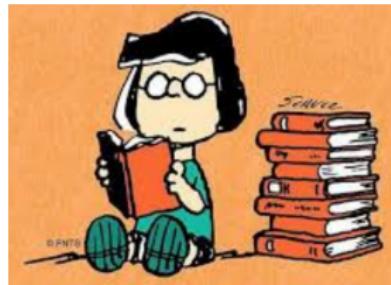
If your contract was called directly by an EOA transaction, then `msg.sender` is the address that signed the transaction, but otherwise it will be a contract address.

Smart contracts (16)

Bibliographic notes

The term “smart contracts” has been coined in the 1990s by Nick Szabo.

A useful introduction to smart contracts and Solidity can be found in [Antonopoulos and Wood \(2018, Chapter 7\)](#).



Smart contracts (17)

References

Antonopoulos, Andreas M., and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, 2018.

Smart Contracts and Blockchain Technology

Lecture 9. Solidity variables and types

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Smart contracts

This lecture: Solidity variables and types

- Value types
- Reference types
- Visibility

Solidity variables and types (1)

Type declarations

Solidity is a **statically typed programming language**, which means that the type of each variable needs to be specified in the form of an **explicit type declaration** at compile time.¹

Moreover, newly declared variables always have a **default value**:

```
uint x; // declares x and sets it to 0x0
```

¹In contrast, *Python* is a dynamically typed language because it does not require the declaration of variables at compile time, and the checking of types takes place at runtime.

Solidity variables and types (2)

Value types vs. reference types

Solidity distinguishes between value types and reference types:

- Variables of **value types** are always passed on by value, i.e., they are copied when they are used as function arguments or in assignments.
- Variables of **reference type** can be addressed through multiple, different names.²

²It is useful to think of variables of the reference types as *pointers*, so that several pointers may identify the same data.

Solidity variables and types (3)

Value types

- Boolean
- Integer
- Fixed-point number³
- Address (payable)
- Contract
- Fixed-size byte array
- Literal
- Enum
- User defined
- Function

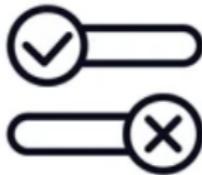
³Fixed-point numbers are partially supported only. Floating-point numbers (i.e., scientific notation) are not supported at all.

Solidity variables and types (4)

Booleans

Booleans represent precisely one bit of information.
Correspondingly, they take one of two values, true or false.

```
bool flag; // declares a boolean
```



Note: The value of any boolean is initially `false`.

```
flag = true; // flag is set
```

Solidity variables and types (5)

Use of booleans in a control structure

Booleans may be used in control structures, for instance:⁴

```
if flag == true {  
    // do something  
}
```

Equivalent is the following:

```
if flag {  
    // do something  
}
```

⁴Control structures are discussed in the next lecture.

Solidity variables and types (6)

Some operators

! “not” (logical negation)

&& “and” (logical conjunction)

|| “or” (logical disjunction)

!= “not equal to” (inequality)

Solidity variables and types (7)

Integers

Unsigned integers

`uint`

`uint8`, `uint16`, `uint24`, ..., `uint256`⁵

Signed integers

`int`

`int8`, `int16`, `int24`, ..., `int256`⁶

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

⁵The number indicates the number of bits in the machine-level representation.
The type `uint256` is equivalent to `uint`.

⁶The type `int256` is equivalent to `int`.

Solidity variables and types (8)

Type conversions

An **implicit conversion** is applied by the compiler if it makes sense semantically and no information is lost:

- during assignments,
- when applying operators,
- when passing arguments to functions.

```
uint8 x;  
uint16 y;  
uint32 z = x + y; // this works
```

An **explicit type conversion** adds clarity at the cost of formalism:

```
uint16 a = 0x1234;  
uint32 b = uint32(a); // b will be 0x00001234 now
```

Solidity variables and types (9)

Integer operations

- Arithmetic operators: $+$, $-$, unary $-$ (only for signed integers), $*$, $/$, $\%$ (modulo), $**$ (exponentiation)

Division or modulo by zero causes a so-called panic error.⁷

⁷As will be discussed, an error will undo all changes made to the state during a message call, and it will “bubble up” unless caught by a try instruction.

Solidity variables and types (10)

Address

An Ethereum address is a 20 byte hexadecimal value.

```
address constant customer1 =  
0xc0fee254729296a45a3885639AC7E10F9d54979;
```

It is possible to query the ether balance of an address using the property `.balance`.

```
if customer1.balance == 0 revert();
```

Solidity variables and types (11)

Members of address

.balance returns the **balance** of the address in Wei as uint256.

.code returns the **code** at the address as bytes memory (can be empty).⁸

.codehash returns the Keccak-256 hash of the code at address as bytes32.⁹

⁸This allows to check if an address is a contract address or not.

⁹This can be used to check if the code on a given address is as expected.

Solidity variables and types (12)

Low-level contract calls

Any call method:

- operates on an address (rather than a contract instance),
- takes a given payload as bytes memory argument,
- returns success condition and return data as (bool, bytes memory), and
- forwards all available gas, adjustable.

`.call(payload)`

`.delegatecall(payload)`

`.staticcall(payload)`¹⁰

¹⁰This is basically the same as `call`, but will revert if the called function modifies the state in any way.

Solidity variables and types (13)

Address payable

A contract can make a payment only to an address that has been declared payable.¹¹

```
address payable borrower;
```

```
borrower = payable(customer1); // conversion needed
```

address payable has two additional members, both sending an amount in Wei as uint256 and forwarding a 2300 gas stipend (not adjustable):

.transfer(amount) reverts on failure (either payment not accepted or insufficient funds).

.send(amount) returns a bool (equal to false on failure).

¹¹Implicit conversion from payable to address is possible, but not vice versa.

Solidity variables and types (14)

Enum types

New data types may be defined in the form of **enums**:

```
contract softdrinks {  
    enum Size{ SMALL, MEDIUM, LARGE }  
    Size choice; // variable of type Size  
    function setChoice() public {  
        choice = Size.LARGE;  
    }  
    function getChoice() public returns (Size) {  
        return choice;  
    }  
}
```

Solidity variables and types (15)

Fixed-size byte arrays

Raw byte code of given size may be represented by **fixed-size byte arrays**:

`bytes1, bytes2, ..., bytes32`

The number indicates the number of bytes.

Solidity variables and types (16)

Reference types

Reference types:

- Structs
- Arrays
- Mappings

If you use a reference type, you always have to explicitly provide the data area where the type is stored:

- **storage** (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract), or
- **memory** (whose lifetime is limited to an external function call),
- **calldata** (special data location that contains the function arguments).

Solidity variables and types (17)

Structs

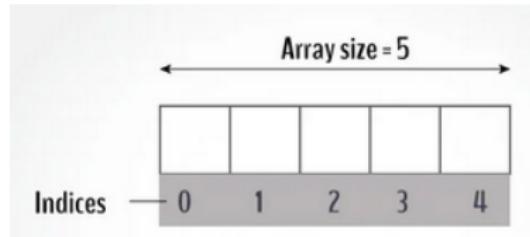
New data types may be defined in the form of **structs**:

```
struct Book {  
    string title;  
    string author;  
    uint book_id;  
}  
  
Book storage book; // variable of type Book  
  
book.title = "The Tell-Tale heart";  
book.author = "Edgar Allen Poe";  
book.book_id = 1;
```

Solidity variables and types (18)

Arrays

Storage arrays either have a **fixed size** or a **dynamic size**:¹²



```
uint[6] a; // declares a static array
uint[] d; // declares a dynamic array
bytes b; // used for arbitrary-length raw byte data
string s; // used for arbitrary-length string
           (UTF-8) data
a[4] = 8; // assigns a value to array element
```

¹²Memory arrays are static once created (but the size can be set at runtime).

Solidity variables and types (19)

Strings

```
pragma solidity ^0.5.0;
contract SolidityTest {
    string data = "test";
}
```

Solidity variables and types (20)

Methods for dynamic storage arrays

On dynamic storage arrays:

- `.length` returns the number of elements of the array.
- `.push(value)` appends value at the end of the array.
- `.push()` appends a zero-initialised element.
- `.pop()` removes the element at the end of the array.

Note: `.push(value)`, `.push()`, and `.pop()` do not work for strings.

Solidity variables and types (21)

Mappings

Mappings allow to create and manage lists in a flexible way.

Example:

```
contract LedgerBalance {  
    mapping(address => uint) public balances;  
    function updateBalance(uint newBalance) public {  
        balances[msg.sender] = newBalance;  
    }  
}
```

Solidity variables and types (22)

Mappings (continued)

Variables of **mapping type** are declared using the syntax

```
mapping(KeyType => ValueType) VariableName
```

The KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed.

ValueType can be any type, including mappings, arrays and structs.

Note: Mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information.

Solidity variables and types (23)

State variable visibility

`private`

...can only be accessed from within the contract they are declared in.

`internal`

...can be accessed both from within the contract they are declared in and in derived contracts.¹³ This is the default visibility level for state variables.

`public`

...differ from internal ones in that the compiler automatically generates **getter functions** for them, which allows other contracts to read their values.¹⁴

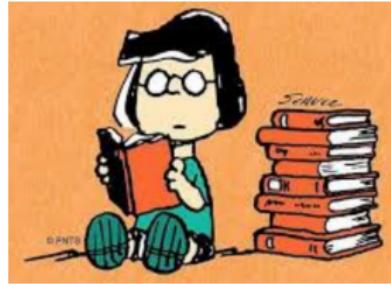
¹³ Derived contracts will be discussed in the next lecture.

¹⁴ For instance, to read the value of variable `x` in a contract `myContract`, one uses `myContract.x`.

Solidity variables and types (24)

Bibliographic notes

This slide deck is based on Section 3.6 of the official documentation of the programming language Solidity ([link](#)).



Solidity variables, types, and expressions (25)

References

Solidity Documentation, Release 0.8.17, Ethereum Foundation,
September 8, 2022.

Smart Contracts and Blockchain Technology

Lecture 10. Solidity functions and contracts

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Basic language concepts of Solidity

- Booleans, integers, and addresses
- Type conversion (implicit and explicit)
- Elementary operations and members

This lecture: Core language concepts

- Functions
- Contracts

Solidity functions and contracts (1)

Functions

Functions are executable units of code, typically defined inside a contract:¹

```
contract myFirstContract {  
    function myFirstFunction() {  
        \\ do something  
    }  
}
```

The declaration of a function may specify:

- **parameters** as input, and
- **return variables** as output.²

¹However, they can also be defined outside of contracts.

²Both parameters and return variables are optional.

Solidity functions and contracts (2)

Example

```
function Double(uint x) public pure returns (uint y) {  
    y = 2*x;  
}
```

Equivalent:

```
function Double(uint x) public pure returns (uint) {  
    return(2*x);  
}
```

The function is **pure** because it does not read or modify the state.³

³Functions may also be of type **view** if they read the state but do not modify it. These properties are referred to as the **mutability** of the function.

Solidity functions and contracts (3)

Function calls

Functions are called either internally or externally:

- **Internal function calls** target a function inside the current code unit, which includes the current contract and inherited functions, in particular.
- **External function calls** target a function outside of the current code unit. They require an address and a so-called function signature.

Solidity functions and contracts (4)

Function visibility

Function calls may be prohibited using visibility specifiers:

External functions can be called via message calls only.⁴

Public functions can be called both via message calls and internally.

Private functions can be accessed from the current contract only.

Internal functions are like private functions but can also be accessed from contracts deriving from it.

```
inheritance:  
contract Parent {  
    //...  
}  
  
contract Child is Parent {  
    //...  
}
```

⁴I.e., `f()` does not work, but `this.f()` works.

By using it, a child contract can use all elements of parent contract

Solidity functions and contracts (5)

Function modifiers

Modifiers amend the semantics of functions in a declarative way:

```
contract Purchase {  
    address public seller;  
    modifier onlySeller() {  
        require(  
            msg.sender == seller,  
            "Not authorized");  
        _;  
    }  
    function abort() public view onlySeller {  
        // ...  
    }  
}
```

Solidity functions and contracts (6)

Events

Events are used for logging. Once emitted, they can be read by any server that has access to a node of the blockchain network.

```
contract Auction {  
    event BidRaised(address bidder, uint amount);  
  
    function bid() public payable {  
        // ...  
        emit BidRaised(msg.sender, msg.value);  
    }  
}
```

Solidity functions and contracts (7)

`msg.sender` and `msg.value`

During the execution of a function call, we have access to `msg.sender` and `msg.value`, in particular:

- `msg.data (bytes calldata)`: complete calldata
- `msg.sender (address)`: sender of the message (current call)
- `msg.sig (bytes4)`: first four bytes of the calldata (i.e. function identifier)
- `msg.value (uint)`: number of wei sent with the message

Solidity functions and contracts (8)

tx.origin and tx.gasprice

Further, we have access to tx.origin and tx.gasprice:

- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Solidity functions and contracts (9)

Selfdestruct

A contract may be programmed to self-destruct:

```
selfdestruct(0xAd8...866);
```

This operation will:

- erase the current contract code as well as the contract storage,⁵
- transfer the balance at the contract address to the address provided.

Note. `selfdestruct` is not recommended (ether sent to the contract is lost). Instead, change some internal state which causes all functions to revert.

⁵However, the history of the blockchain may be retained by the nodes. Under `delegatecall`, the calling contract gets erased.

Solidity functions and contracts (10)

Contracts

Contracts in Solidity are similar to classes in object-oriented languages.

They contain persistent data in state variables, and functions that can modify these variables.

Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible.

A contract can refer to itself using this:

`address(this).balance`

Solidity functions and contracts (11)

Creating a contract

Contracts can be created “from outside” via Ethereum transactions or from within Solidity contracts.

IDEs, such as Remix, make the creation process seamless using UI elements.⁶

One way to create smart contracts programmatically is via the web3.js library.⁷

⁶IDE = Interactive Development Environment. UI = User Interface.

⁷The ending .js stands for JavaScript, the programming language most prevalently used in internet applications.

Solidity functions and contracts (12)

Constructor

The **constructor** of a contract is an (optional) function that is executed once at the time of contract creation:⁸

```
constructor() {  
    // do something  
}
```

After the constructor has executed, the code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls.⁹

⁸The constructor should be placed on top of all functions of a contract.

⁹The deployed code does not include the constructor code nor internal functions only called from the constructor.

Solidity functions and contracts (13)

Receive

The **receive** function is an (optional) function that accepts incoming payments.

```
receive() external payable {  
    // do something  
}
```

Warning. A contract without a receive function can receive ether as a recipient of a coinbase transaction or as a destination of a selfdestruct. A contract cannot react to such transfers and thus also cannot reject them.

Solidity functions and contracts (14)

Fallback function

A contract may have a **fallback function**. This function is executed if either the other functions do not match, or if no data was supplied at all and there is no receive function.

```
fallback() external payable {  
    // do something  
}
```

This function must have external visibility. In order to receive ether, it must be marked as payable (as in the example above). With parameters, the fallback function looks as follows:

```
fallback (bytes calldata input) external payable  
returns (bytes memory output) {}
```

Solidity functions and contracts (15)

Example

```
contract AcceptingPayments {  
    address payable owner;  
    constructor() {  
        owner = payable(msg.sender);  
    }  
    receive() external payable {}  
    fallback() external payable {}  
    function collect() {  
        require(  
            owner == payable(msg.sender),  
            "not authorized"  
        );  
        selfdestruct(owner);  
    }  
}
```

Solidity functions and contracts (16)

Error handling

There are three ways to (conditionally) trigger an error:

- `assert`
- `require`
- `revert`

In addition, errors may be caught by calling contracts:

- `try and catch`

Solidity functions and contracts (17)

Assert

To check a condition that should be true for any input, you may use the **assert function**:

```
function GetItRight() {  
    assert(1+1==2);  
}
```

If the condition is not met, a panic error is triggered.

Solidity functions and contracts (18)

Require

To check a condition, you may use the **require function**:

```
require(msg.sender == owner; "not authorized");
```

Solidity functions and contracts (19)

Revert function

To trigger an unconditional error, you may use the **revert function**:¹⁰

```
revert("This should not have happened!");
```

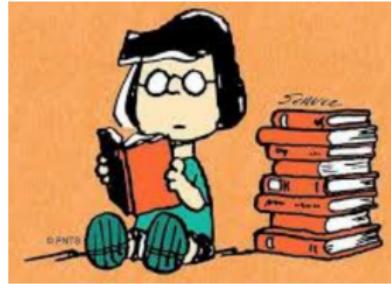
The error data will be passed back to the caller and can be caught there using a try/catch statement.

¹⁰ To save gas, one may alternatively define an error message and call the revert statement.

Solidity functions and contracts (20)

Bibliographic notes

This slide deck is based on Section 3.9 of the official documentation of the programming language Solidity ([link](#)).



Solidity functions and contracts (21)

References

Solidity Documentation, Release 0.8.17, Ethereum Foundation,
September 8, 2022.

Smart Contracts and Blockchain Technology

Lecture 11. Control structures in Solidity

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture: Core language concepts

- Functions
- Contracts

This lecture: Advanced elements of Solidity

- Error handling
- Data types (cont.)

Control structures in Solidity (1)

Error handling

There are three ways to (conditionally) trigger an error:

- `assert`
- `require`
- `revert`

In addition, errors may be caught by calling contracts:

- `try and catch`

Control structures in Solidity (2)

Assert

To check a condition that should be true for any input, you may use the **assert** function:

```
1 // SPDX-License-Identifier: GPL-3.0-only
2 pragma solidity >=0.8.0 <0.9.0;
3 contract c {
4     function k(string memory s) public pure returns (bytes32) {
5         return keccak256(abi.encodePacked(s));
6     }
7
8     // check if keccak256 works
9     bytes32 constant emptyString = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
10    function GetItRight() public pure {
11        assert(emptyString == keccak256(abi.encodePacked("")));
12    }
13 }
```

If the condition is not met, a panic error is triggered.

Control structures in Solidity (3)

Require

To check a condition, you may use the **require function**:

```
require(msg.sender == owner; "not authorized");
```

Control structures in Solidity (4)

Revert function

To trigger an unconditional error, you may use the **revert function**:¹

```
revert("This should not have happened!");
```

The error data will be passed back to the caller and can be caught there using a try/catch statement.

¹To save gas, one may alternatively define an error message and call the **revert statement**.

Control structures in Solidity (5)

Try/catch

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.8.1;
3 interface DataFeed { function getData(address token) external returns (uint value); }
4 contract FeedConsumer {
5     DataFeed feed;
6     uint errorCount;
7     function rate(address token) public returns (uint value, bool success) {
8         // Permanently disable the mechanism if there are more than 10 errors.
9         require(errorCount < 10);
10        try feed.getData(token) returns (uint v) {
11            return (v, true);
12        } catch Error(string memory /*reason*/) {
13            // This is executed in case revert was called inside getData and a reason string was provided.
14            errorCount++;
15            return (0, false);
16        } catch Panic(uint /*errorCode*/) {
17            // This is executed in case of a panic, i.e. a serious error like division by zero or overflow.
18            errorCount++;
19            return (0, false);
20        } catch (bytes memory /*lowLevelData*/) {
21            // This is executed in case revert() was used.
22            errorCount++;
23            return (0, false);
24        }
25    }
26 }
```

Control structures in Solidity (6)

Data types (cont.)

Reference types:

- Structs
- Arrays and strings
- Mappings

If you use a reference type, you always have to explicitly provide the data area where the type is stored:

- **storage** (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract), or
- **memory** (whose lifetime is limited to an external function call),
- **calldata** (= stack, special data location that contains the function arguments).

Control structures in Solidity (7)

Storage and memory

Storage is like a computer hard drive. State variables are storage data. These state variables reside in the smart contract data section on the blockchain. Writing variables into storage is expensive (in terms of gas consumption).

Memory is a temporary place to store data, like RAM. Function arguments and local variables in functions are memory data. If the function is external, args will be stored in the stack (calldata). EVM has limited space for memory so values stored here are erased between function calls.

Control structures in Solidity (8)

Structs

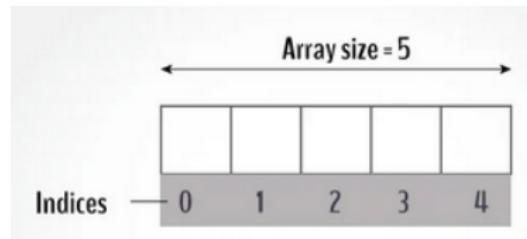
New data types may be defined in the form of **structs**:

```
struct Book {  
    string title;  
    string author;  
    uint book_id;  
}  
  
Book storage book; // variable of type Book  
  
book.title = "The Tell-Tale heart";  
book.author = "Edgar Allen Poe";  
book.book_id = 1;
```

Control structures in Solidity (9)

Arrays

Storage arrays either have a **fixed size** or a **dynamic size**:²



```
uint[6] a; // declares a static array
uint[] d; // declares a dynamic array
bytes b; // arbitrary-length byte data
string s; // arbitrary-length UTF-8 data
a[4] = 8; // assigns a value to array element
```

²Memory arrays are static once created (but the size can be set at runtime).

Control structures in Solidity (10)

Unicode Transformation Format 8 bit (UTF-8)

UTF-8 is the predominant character encoding.

UTF-8 is backward compatible with ASCII but variable-length:

- The first 128 characters of Unicode correspond one-to-one with ASCII. E.g., “\$” = 0x24.
- The checksum bit in ASCII is reinterpreted to signal that more bytes are used to encode the character. E.g., “€” = 0xe282ac.

Examples of UTF-8 encoding

Character	Binary code point	Binary UTF-8		Hex UTF-8
\$ U+0024	010 0100	00100100		24
£ U+00A3	000 1010 0011	11000010 10100011		C2 A3
€ U+0939	0000 1001 0011 1001	11100000 10100100 10111001		E0 A4 B9
€ U+20AC	0010 0000 1010 1100	11100010 10000010 10101100		E2 82 AC
한 U+D55C	1101 0101 0101 1100	11011011 10010101 10011100		ED 95 9C
ଓ U+10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000		F0 90 8D 88

Control structures in Solidity (11)

Fixed-size byte arrays

Raw byte code of given size may be represented by **fixed-size byte arrays**:

`bytes1, bytes2, ..., bytes32`

The number indicates the number of bytes.

Control structures in Solidity (12)

Strings

```
pragma solidity ^0.5.0;
contract SolidityTest {
    string data = "test";
}
```

Control structures in Solidity (13)

Methods for dynamic storage arrays

On dynamic storage arrays:

- `.length` returns the number of elements of the array.
- `.push(value)` appends value at the end of the array.
- `.push()` appends a zero-initialised element.
- `.pop()` removes the element at the end of the array.

Note: `.push(value)`, `.push()`, and `.pop()` do not work for strings.

Control structures in Solidity (14)

Mappings

Mappings allow to create and manage lists in a flexible way.

Example:

```
contract LedgerBalance {  
    mapping(address => uint) public balances;  
    function updateBalance(uint newBalance) public {  
        balances[msg.sender] = newBalance;  
    }  
}
```

Control structures in Solidity (15)

Mappings (continued)

Variables of **mapping type** are declared using the syntax

```
mapping(KeyType => ValueType) VariableName
```

The KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed.

ValueType can be any type, including mappings, arrays and structs.

Note: Mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information.

Control structures in Solidity (16)

Enum types

New data types may be defined in the form of **enums**:

```
contract softdrinks {  
    enum Size{ SMALL, MEDIUM, LARGE }  
    Size choice; // variable of type Size  
    function setChoice() public {  
        choice = Size.LARGE;  
    }  
    function getChoice() public returns (Size) {  
        return choice;  
    }  
}
```

Control structures in Solidity (17)

State variable visibility

`private`

...can only be accessed from within the contract they are declared in.

`internal`

...can be accessed both from within the contract they are declared in and in derived contracts.³ This is the default visibility level for state variables.

`public`

...differ from internal ones in that the compiler automatically generates **getter functions** for them.⁴

³ Derived contracts inherit from existing **base contracts**. They are defined using the keyword `is` in the contract declaration.

⁴ For instance, to read the value of variable `x` in a contract `myContract`, one uses `myContract.x`.

Outlook on the remaining lectures

Lecture 12: Token programming

- ERC20 tokens
- Alternative token standards

Lecture 13: Decentralized finance

- Decentralized exchanges
- Mixers

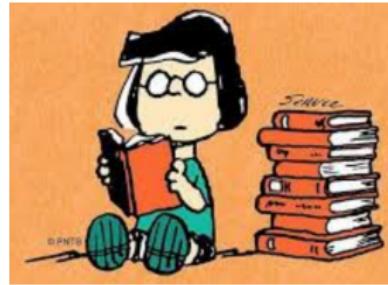
Lecture 14: Beyond crypto

- Use cases
- Future of blockchain technology

Control structures in Solidity (18)

Bibliographic notes

This slide deck is based on Section 3.9 of the official documentation of the programming language Solidity ([link](#)).



Control structures in Solidity (19)

References

Solidity Documentation, Release 0.8.17, Ethereum Foundation,
September 8, 2022.

Smart Contracts and Blockchain Technology

Lecture 12. Token Programming

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Introduction and overview

Last lecture

- Advanced Solidity language concepts

This lecture

- Tokens
- ERC20 standard
- NFTs

Token programming (1)

What is a token

A **token** is a privately issued special-purpose coin-like items of insignificant intrinsic value, such as a laundry token:



Blockchain tokens are conceptually identical, but differ from laundry tokens in several ways:

- flexibility
- global scale
- transaction costs
- transparency

Token programming (2)

Token classes

Payment tokens:

- Cryptocurrencies (i.e., **native tokens** or **coins** embodied in blockchain systems)¹
- Tokens representing cryptocurrencies on other chains

Asset tokens:

- Stablecoins, equity tokens, and DOA shares
- Digital collectibles (e.g., NFTs)

Utility tokens:

- Resource tokens (e.g., for CPU time)
- Certificates (e.g., access rights, identity tokens,...)

¹These “tokens” do not rely on smart contract functionality and are intrinsic to the underlying blockchain system (e.g., bitcoin, ether).

Token programming (3)

Non-fungible tokens

Securities and tokens are called **fungible** when users consider individual units as perfect substitutes.²



Non-fungible tokens (NFTs) correspond (in a somewhat vague way) to a unique tangible or intangible item (e.g., a painting). Each non-fungible token is associated with a unique identifier, such as a serial number.

²Banknotes, e.g., are fungible, even though there are limitations (serial numbers and age, for instance).

Token programming (4)

Legal nature of tokens

Blockchain enthusiasts often refer to the ability of blockchain systems to convert extrinsic assets into intrinsic assets. However, developments are rather slow.³

- A token or NFT typically does not embody any legal claim.⁴
- Tokens may be rather illiquid, i.e., it may be hard to sell them on short notice at a good price.

Note: Tokens are not risky because of counterparty risk (i.e., the risk that a legal obligation is not fulfilled by a counterparty), it is the “triple-L” risk (i.e., legal nature, issuer liability, and liquidity are all questionable).

³An exception are equity tokens under Swiss law.

⁴E.g., virtual items in a computer game may not be considered as assets ([here](#)). This need not stop tax authorities to impose wealth tax on cryptoassets though.

Token programming (5)

Duck test

During the initial ICO hype around 2017, scammers have often tried to hide the fact that they were actually offering (worthless) securities.⁵ To explain why this will not stop prosecution, regulators have referred to the so-called **duck test**:

“If it looks like a duck, swims like a duck, and quacks like a duck,
then it probably is a duck.”



⁵ An **Initial Coin Offering (ICO)** is a crowdfunding mechanism used to raise money by selling tokens. Unlike public offerings in the highly regulated securities markets, ICOs used to be essentially unregulated.

Token programming (6)

Who needs utility tokens?

Opportunities: Token issuers inherit the market enthusiasm, early adopters, technology, innovation, and liquidity of the entire token economy.

Challenges: Utility tokens introduce unnecessary costs/risks and adoption barriers for startups.

Token programming (7)

Tokens on Ethereum

Sending ether is an intrinsic action of the Ethereum platform, but sending or even owning tokens is not:

- The ether balance of Ethereum accounts is handled **at the protocol level**, whereas the token balance of Ethereum accounts is handled **at the smart contract level**.

In order to create a new token on Ethereum, you must create a new smart contract that handles everything, including:

- ownership
- transfers, and
- access rights.

Token programming (8)

Standards

Standards help interoperability...



In a smart contract, token standards are the **minimum specifications** for an implementation (usually functions and events).⁶

Token contract that follow the standards may easily interact with wallets, exchanges, and existing user interfaces.

⁶How the minimum specifications are implemented does not matter. Moreover, additional functionality may be added ad libitum.

Token programming (9)

ERC20 required functions

The vast majority of tokens are currently based on the ERC20 standard.

Single-step work-flow:

- **totalSupply:** Returns the total units of this token that currently exist.⁷
- **balanceOf:** Given an address, returns the token balance of that address.
- **transfer:** Given an address and amount, transfers that amount of tokens to that address, from the balance of the address that executed the transfer.

⁷ERC20 tokens can have a fixed or a variable supply.

Token programming (10)

ERC20 required functions (continued)

Two-step work-flow:

- **transferFrom:** Given a sender, recipient, and amount, transfers tokens from one account to another.
- **approve:** Given a recipient address and amount, authorizes that address to execute several transfers up to that amount, from the account that issued the approval.
- **allowance:** Given an owner address and a spender address, returns the remaining amount that the spender is approved to withdraw from the owner.

Token programming (11)

ERC20 required events

Transfer. Event triggered upon a successful transfer (call to either transfer or transferFrom).⁸

Approval. Event logged upon a successful call to approve.

⁸Even for zero-value transfers.

Token programming (12)

ERC20 optional functions

In addition to the required functions listed in the previous section, the following optional functions are also defined by the standard:

- **name:** Returns the human-readable name (e.g., “US Dollars”) of the token.
- **symbol:** Returns a human-readable symbol (e.g., “USD”) for the token.
- **decimals:** Returns the number of decimals used to divide token amounts.⁹

⁹For example, if decimals is 2, then the token amount is divided by 100 to get its user representation.

Token programming (13)

ERC20 interface

```
contract ERC20 {  
    function totalSupply() constant returns (uint theTotalSupply);  
    function balanceOf(address _owner) constant returns (uint balance);  
    function transfer(address _to, uint _value) returns (bool success);  
    function transferFrom(address _from, address _to, uint _value) returns  
        (bool success);  
    function approve(address _spender, uint _value) returns (bool success);  
    function allowance(address _owner, address _spender) constant returns  
        (uint remaining);  
    event Transfer(address indexed _from, address indexed _to, uint _value);  
    event Approval(address indexed _owner, address indexed _spender, uint _value);  
}
```

Token programming (14)

ERC20 data structures

```
mapping(address => uint256) balances;
```

This implements an internal table of token balances, by owner, and allows the token contract to keep track of who owns the tokens. Each transfer is a deduction from one balance and an addition to another balance.¹⁰

¹⁰If Alice wants to send 10 tokens to Bob, her wallet sends a transaction to the token contract's address, calling the transfer function with Bob's address and 10 as the arguments. The token contract adjusts Alice's balance (-10) and Bob's balance (+10) and issues a Transfer event.

Token programming (15)

ERC20 data structures (continued)

```
mapping (address => mapping (address => uint256)) public allowed;
```

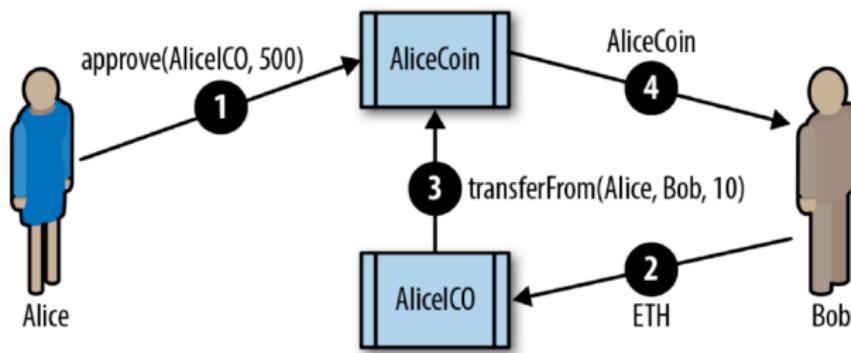
The owner of a token can delegate authority to a spender, allowing them to spend a specific amount (allowance) from the owner's balance.

The ERC20 contract keeps track of the allowances with a two-dimensional mapping, with the primary key being the address of the token owner, mapping to a spender address and an allowance amount.

Token programming (16)

The two-step workflow

Illustration. To sell tokens for an ICO, Alice approves the AliceICO crowdsale contract address to distribute a certain amount of tokens. The crowdsale contract can then transferFrom the token contract owner's balance to Bob, the buyer of the token:



Token programming (17)

ERC20 implementations

Consensys EIP20. A simple and easy-to-read implementation of an ERC20-compatible token.

OpenZeppelin StandardToken. This implementation is ERC20-compatible, with additional security precautions. It forms the basis of OpenZeppelin libraries implementing more complex ERC20-compatible tokens with fundraising caps, auctions, vesting schedules, and other features.

Token programming (18)

Truffle and Ganache

Truffle. Comprehensive suite of tools for smart contract development.

Ganache. Fires up a personal Ethereum blockchain which you can use to run tests, execute commands, and inspect state while controlling how the chain operates.

Token programming (19)

Warnings

You cannot pay for a transaction's gas with a token and the token contract can't pay the gas for you.

Example 1. When trying to send a token, your wallet informs you that you need ether to pay the **gas**.

Example 2. Polygon MATIC exists both as a native token and an ERC20 token, and the transfer is accomplished by a **plasma bridge**.

Token programming (20)

ERC721: Non-fungible Token Standard

The ERC721 proposal is for a standard for non-fungible tokens (a.k.a. **deeds**).

The use of the word “deed” is intended to reflect the “ownership of property” part, even though these are not recognized as legal documents in any jurisdiction (yet):¹¹

- **virtual item**, such as an in-game item or digital collectible
- **physical item** whose ownership is tracked by a token (a house, a car, or an artwork).
- **items of negative value**, such as loans (debt), liens, easements, etc.

¹¹A **deed** is a legal document that is signed and delivered, especially one regarding the ownership of property or legal rights.

Token programming (21)

ERC20 vs. ERC721

Look at the internal data structure used in ERC721:

```
// Mapping from deed ID to owner  
mapping (uint256 => address) private deedOwner;
```

Whereas ERC20 tracks the balances that belong to each owner, with the owner being the primary key of the mapping, ERC721 tracks each deed ID and who owns it, with the deed ID being the primary key of the mapping.

Token programming (22)

ERC721 interface

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
    event Approval(address indexed _owner, address indexed _approved,
                  uint256 _deedId);
    event ApprovalForAll(address indexed _owner, address indexed _operator,
                          bool _approved);

    function balanceOf(address _owner) external view returns (uint256 _balance);
    function ownerOf(uint256 _deedId) external view returns (address _owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId)
        external payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operator, boolean _approved) payable;
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

Token programming (23)

Common extensions to token standards

- **Owner control.** The ability to give specific addresses, or sets of addresses (i.e., multisignature schemes), special capabilities, such as minting, recovery, etc.
- **“Monetary policy.”** The ability to add to or reduce the total supply of tokens, at a predictable rate or by “fiat” of the creator of the token. This also includes caps on supply.
- **Crowdfunding.** The ability to offer tokens for sale, for example through an auction or market sale.
- **Recovery backdoors.** Functions to recover funds, reverse transfers, or dismantle the token that can be activated by a designated address or set of addresses.
- **White-/blacklisting.** The ability to restrict actions (such as token transfers) to specific addresses. There is usually a mechanism for updating the lists.

Token programming (24)

Final words on tokens and ICOs

Tokens have been an explosive development in the Ethereum ecosystem.

Nevertheless, the importance and future impact of these standards should not be confused with an endorsement of past and current token offerings. Many of the tokens on offer in Ethereum today are barely disguised **scams, pyramid schemes, and money grabs**.

Separate this from the long-term vision and impact of smart contract technology.

Token programming (25)

Bibliographic notes

This slide deck is based on Chapter 10 of Antonopoulos and Wood (2022). ([link](#))

An insightful discussion of money, tokens, and other means of payment can be found in the classic monograph by Fisher (1911). ([link](#))

The classification of tokens used above has been proposed by the Swiss financial supervisory authority ([Finma](#)). The ERC20 standard was introduced in November 2015 by Fabian Vogelsteller as an Ethereum Request for Comments (ERC).



Control structures in Solidity (29)

References

Antonopoulos and Wood (2022), Mastering Ethereum.

Fisher, I. (1911). The Purchasing Power of Money.

Smart Contracts and Blockchain Technology

Lecture 1. Introduction and overview

Christian Ewerhart

University of Zurich

Fall 2022

Copyright © 2022, Christian Ewerhart.

All rights reserved.

Without permission of the author, it is not allowed to distribute this script or parts of it.

Welcome!!!

Smart Contracts & Blockchain Technology

This lecture is embedded into a broader study program

Smart Contracts and Blockchain Technology

- Lecture with integrated exercises (6 ECTS)¹
- Seminar (3 ECTS)

Written thesis in collaboration with Blockchain Presence

- BA thesis (18 ECTS)
- MA thesis (30 ETCS)
- Individual learning unit (6 ECTS)

¹The course is offered the first time this fall. If successful, it will be offered regularly in the fall term.

Schedule

Lecture (Wednesday 8:00 a.m. - 9:45 a.m.)

- Prof. Dr. Christian Ewerhart
- 80 percent of the final grade²

Tutorial (Friday 12:15 p.m. - 13:45 p.m.)³

- Haoyuan Zeng (ZGSE)
- 20 percent of the final grade⁴

²The final exam takes place at 8 a.m. on January 11, 2023. Since access to the internet needs to be prohibited, the sustainability principle implies that the exam will be held in a closed-book format.

³The tutorial starts this week!

⁴Solutions of selected problems should be handed in via OLAT and Mumbai blockchain, as detailed in the problem sets.

Economics of blockchain (1)

Trust and consensus

In many realms of economic reality, **trust** or a **reliable third party** is needed to reach a certain goal.



Examples:

- Payments (banks and central banks)
- Real estate transactions (notaries)
- Elections (public authorities)

Blockchain technology provides a way to replace such institutions by a decentralized **consensus protocol**.

Economics of blockchain (2)

Public blockchains

The **blockchain** itself...

- ...consists of a heap of **blocks** (essentially linear and starting from a “genesis block”),...



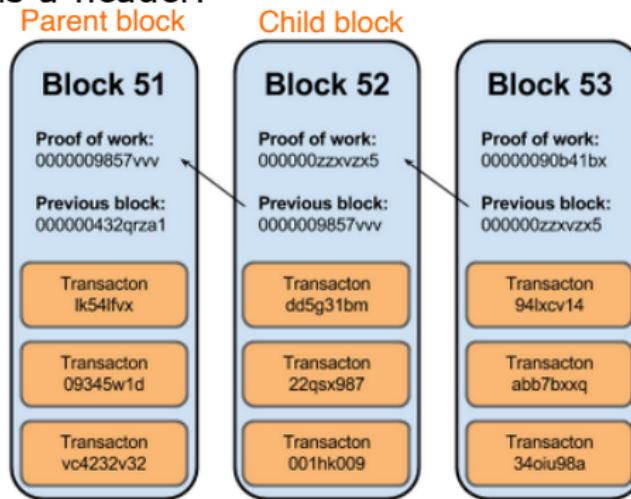
...that is maintained as a **public and distributed database** (or “ledger”) in a peer-to-peer network.



Economics of blockchain (3)

Blocks and transactions

In the bitcoin blockchain, each **block** consists of a list of **transactions** plus a header.



New blocks are added to the blockchain using the **proof-of-work** validation protocol.

Economics of blockchain (4)

Incentives for miners

Every new block creates a **reward**, consisting of

- a block subsidy, and
- transaction fees.

Each **miner** tries to add a block to the blockchain that distributes the reward in his own interest (“coinbase transaction”)!

However, to get consensus for his block, the miner has to present the solution to a **difficult computational problem** (or “crypto puzzle”).



Economics of blockchain (5)

Understanding the miner's problem

The **SHA256** algorithm takes any text message and transforms it into a 64-digit hexadecimal string.

Hexadecimal numbers:
- digits -> 0, 1, 2, 3, 4, ..., a, b, c, d, ...

0x = prefix for a hexadecimal number
> 0xb = 11



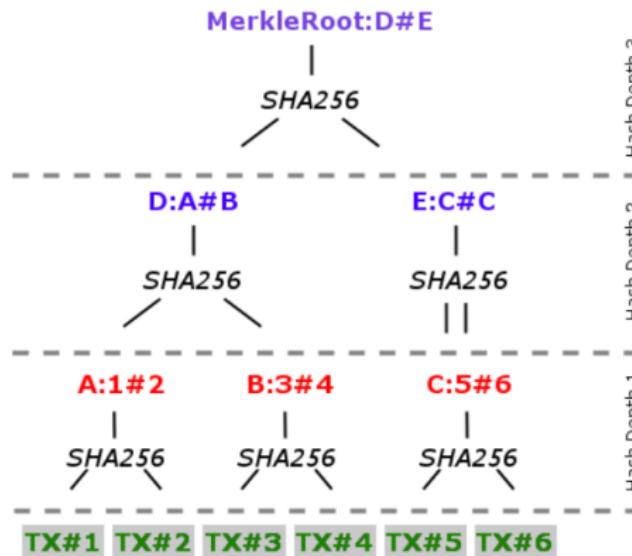
The algorithm is **deterministic** and computationally **non-invertible**.

Economics of blockchain (6)

Merkle root

SHA256 -> hexadecimal code with 64 digits

The transactions of a block are condensed, using iterated SHA256 hashes, into a single **Merkle root**.



Economics of blockchain (7)

Block header

The Merkle root is a component of the **block header**.

Size	Field	Description
4 bytes	Version	The Bitcoin Version Number
32 bytes	Previous Block Hash	The previous block header hash
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The timestamp of the block in UNIX.
4 bytes	Difficulty Target	The difficulty target for the block.
4 bytes	Nonce	The counter used by miners to generate a correct hash.

The **block hash** is determined by hashing the block header through SHA256 (twice).

Economics of blockchain (8)

Nonce

To solve the cryptopuzzle, the miner changes the **nonce** many times, until the block hash is below a certain threshold.

version	02000000
previous block hash (reversed)	17975b97c18ed1f7e255adf297599b55 330edab87803c817010000000000000000
Merkle root (reversed)	8a97295a2747b4f1a0b3948df3990344 c0e19fa6b2b92b3a19c8e6badc141787
timestamp	358b0553
bits	535f0119
nonce	48750833
transaction count	63
coinbase transaction	
transaction	
...	

Block hash

0000000000000000
e067a478024addfe
cdc93628978aa52d
91fabd4292982a50



That threshold (corresponding to the difficulty level) is dynamically adjusted so as to have a new block in regular time intervals.

Economics of blockchain (9)

Block time

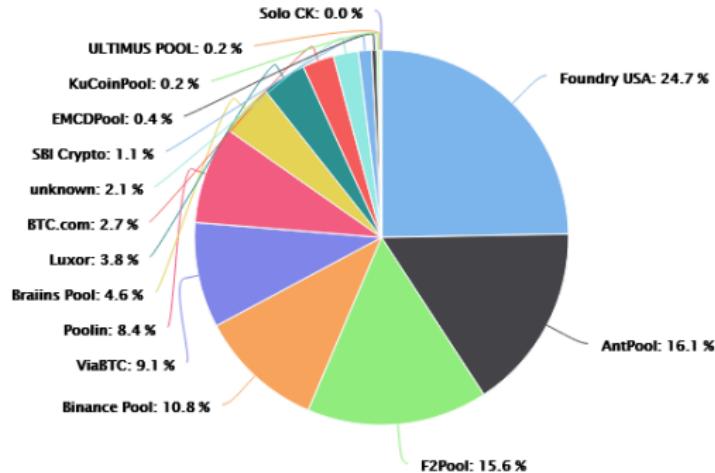
Height	Age	Transactions	Mined by	Size
541912	16 minutes ago	2331	SlushPool	959888
541911	20 minutes ago	2895		917302
541910	31 minutes ago	1754	AntMiner	922638

The **block time** is the average time it takes for the network to generate one extra block in the blockchain. The block time for Bitcoin is about 10 minutes. The blocktime for Ethereum is approximately 15 seconds.

Economics of blockchain (10)

Bitcoin mining pools

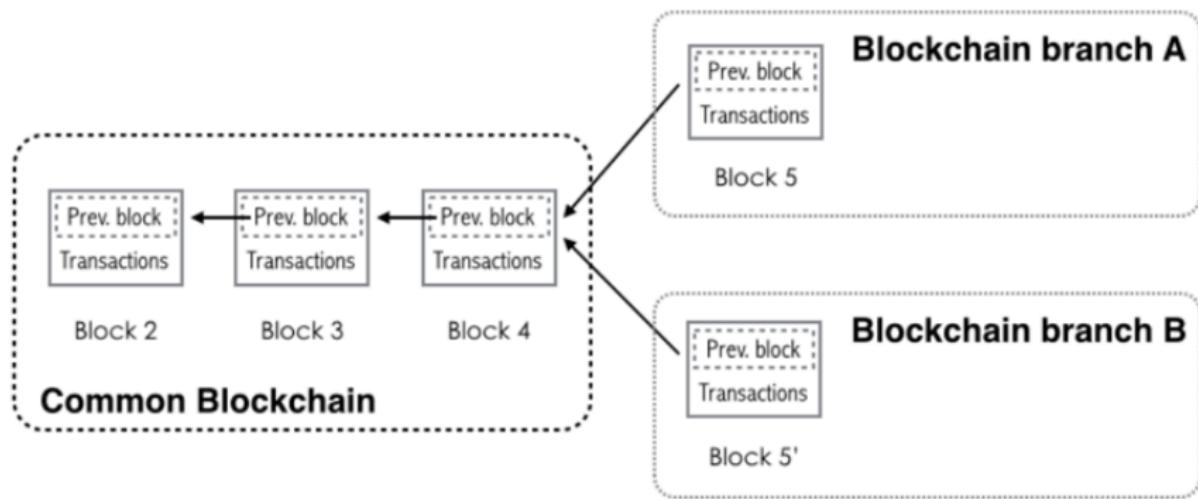
Mining power distribution as of August-September 2022:⁵



⁵Source: [BTC.com](#)

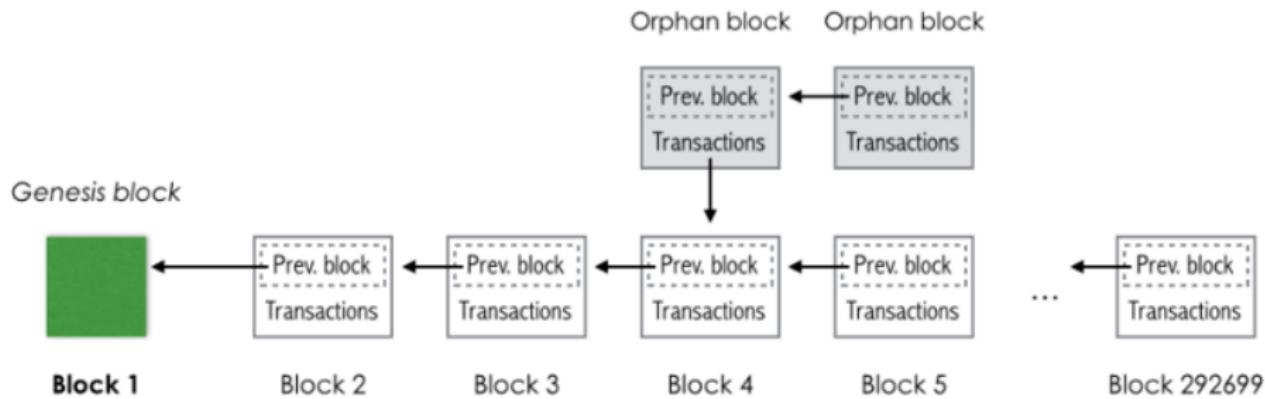
Economics of blockchain (11)

Forking



Economics of blockchain (12)

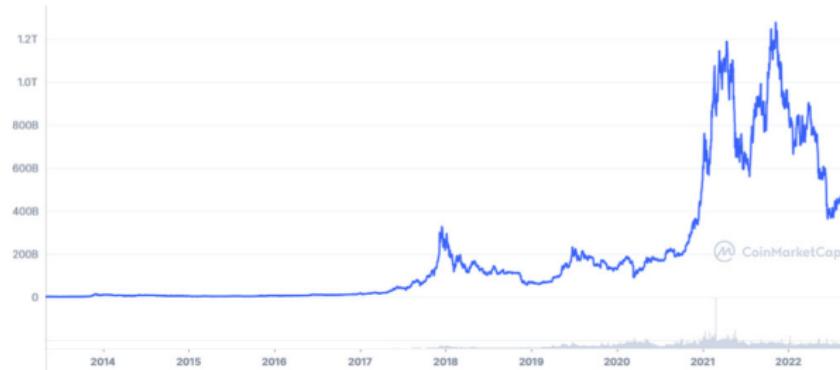
Orphan blocks



Economics of blockchain (13)

Some figures

Bitcoin market cap in USD since February 2013:⁶



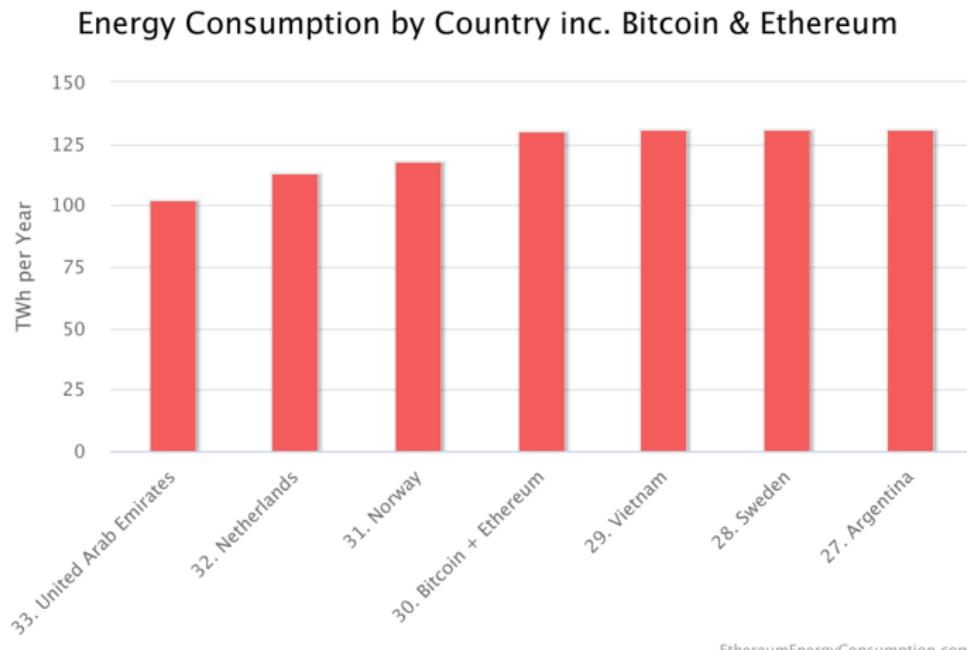
Hypes in December 2017 and December 2021...

Compare: UBS Group \$58B, CS Group \$13B

⁶Source: coinmarketcap.com

Economics of blockchain (14)

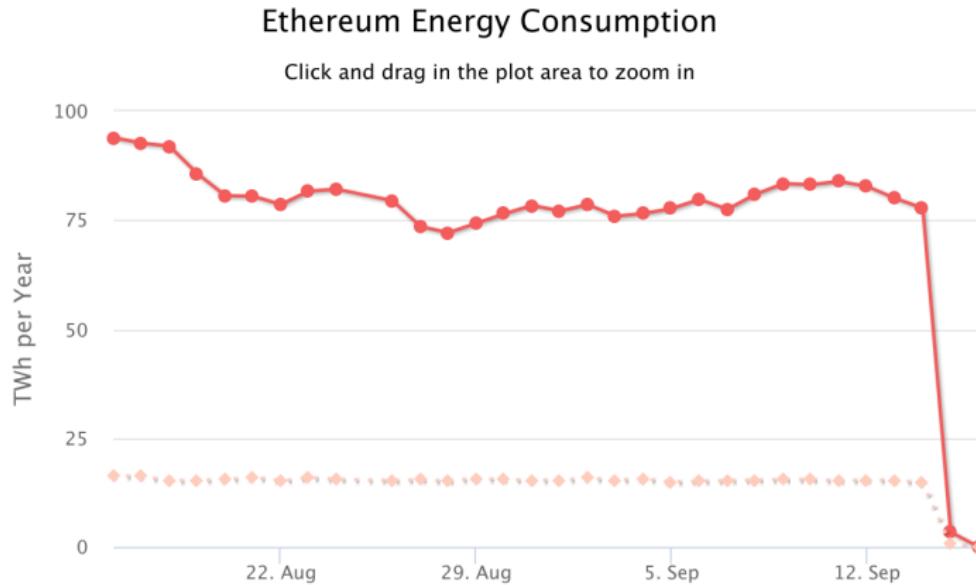
Energy consumption



Economics of blockchain (15)

Ethereum's move from proof-of-work to proof-of-stake

Ethereum's merge might have lowered the world's energy consumption by 0.2 percent⁷



⁷Source: Vitalik Buterin, EthereumEnergyConsumption.com

Economics of blockchain (16)

Smart contracts and blockchain oracles

Smart contracts is software that is stored and executed on a blockchain; usually, they govern transfers in cryptocurrencies.

Almost all smart contract applications require input from the real world. This input is received from so-called **blockchain oracles**, such as Chainlink, Band Protocol, API3, etc.

Economics of blockchain (17)

History of blockchain technology

- 2008 Bitcoin, the first decentralized blockchain, was launched under the acronym Satoshi Nakamoto.⁸
- 2015 Ethereum, the first decentralized smart contract platform, was conceptualized and launched by Vitalik Buterin and Gavin Wood, in particular.⁹

⁸See the [Bitcoin Whitepaper](#). It is not known who hides their identity behind the acronym.

⁹See the excellent monograph [*Mastering Ethereum*](#), by Andreas M. Antonopoulos and Gavin Wood. To access this document, you need to sign up at [Github](#) first.

Economics of blockchain (18)

The double-spending problem

Imagine you wish to create a protocol that allows to transfer value electronically (by email, say).

Suppose you do not want to make use of a bank (maybe because society went through a financial crisis that led to a lot of inequality).

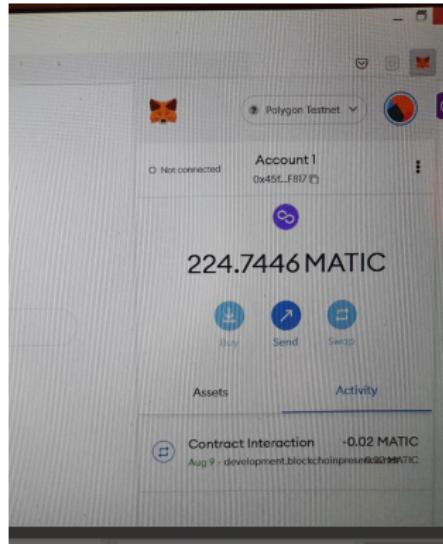
Then, your major problem would be the so-called **double-spending problem**: How do you make sure that the **same amount** is not used **twice**?

Blockchain technology solves this problem.

Economics of blockchain (19)

Wallets

The simplest way to open a wallet is using the browser plug-in Metamask.



Economics of blockchain (20)

The blockchain trilemma

Public blockchains suffer from the so-called **blockchain trilemma**:

- Decentralization
- Security
- Scalability

Overview

Main topics of the lecture:

- I. Mining and consensus formation (4 lectures)
- II. Cryptographic underpinnings (2 lectures)
- III. Smart contract programming (4-5 lectures)
- IV. Decentralized finance (2-3 lectures)

Next week, we will start with part I (Mining and consensus formation).

Some final points

If possible, bring a laptop or tablet to the next tutorial.

Thanks for the attention and see you next week!