Demystifying Blockchain
Mechanics and how CRIX helps us

Bharathi Srinivasan
Wolfgang Karl Härdle

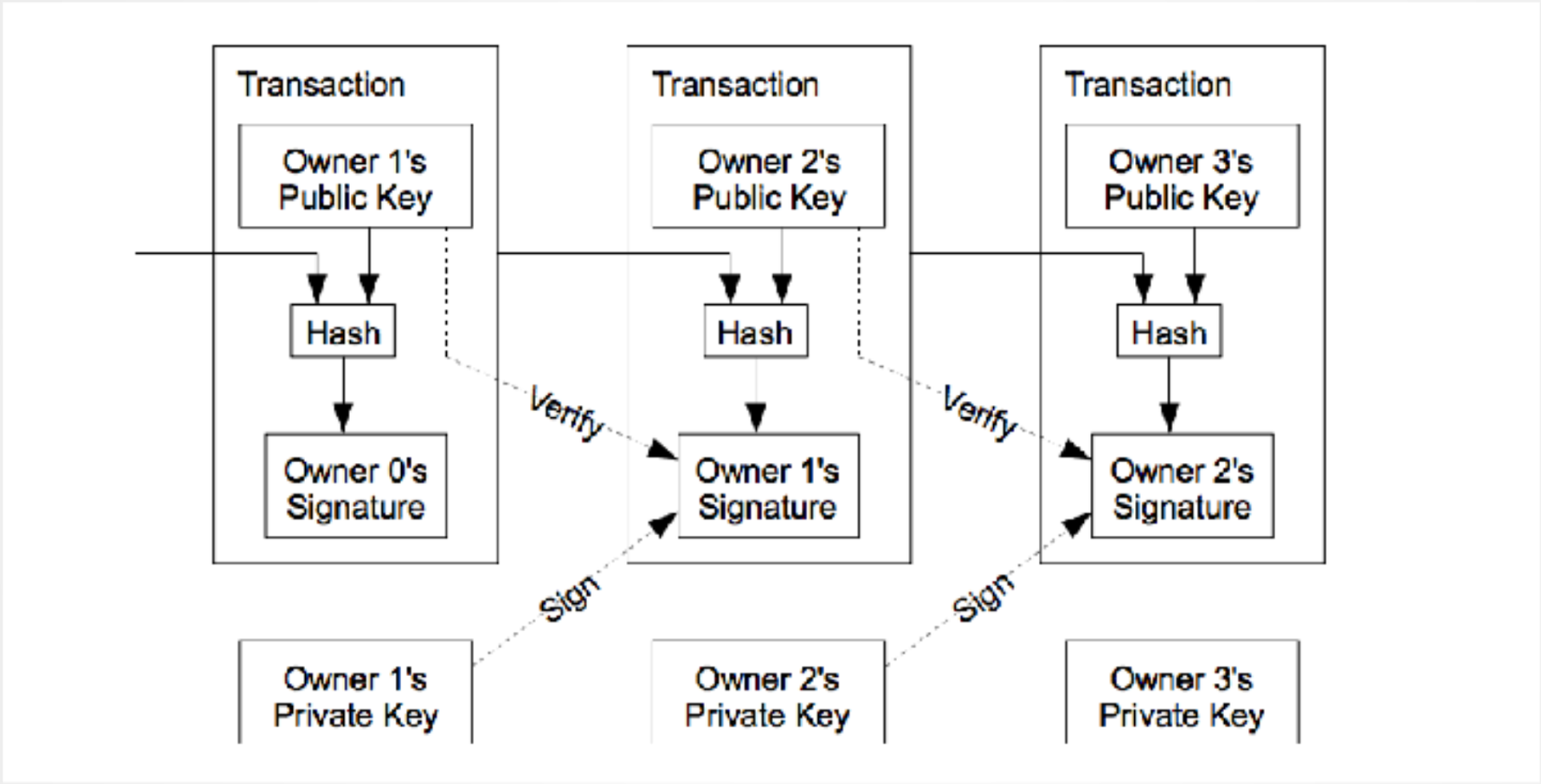
Ladislaus von Bortkiewicz Chair of Statistics
Humboldt-Universität zu Berlin
lvb.wiwi.hu-berlin.de

Outline

1. Laws governing cryptocurrencies
   - Hashing function
     - Proof-of-Work
     - Proof-of-Stake
     - Casper
   - Digital Signatures
     - ECDSA
   - Security against attacks
2. Econometric Analysis of CRIX
   - Stochastic properties
   - Distributional Properties
   - Transformations
   - Observing Heteroskedasticity
3. Implementing Blockchain on Python

Laws of Crypto-Land

Electronic coin : chain of digital signatures

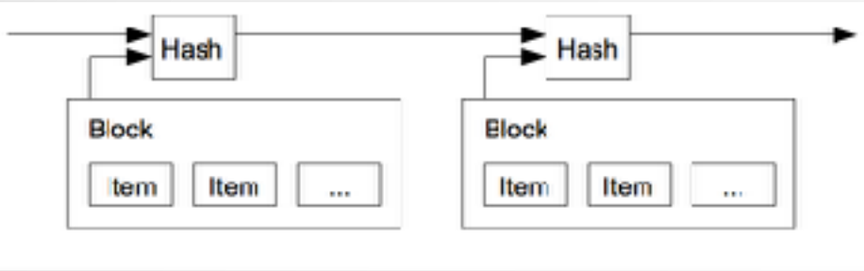


Each owner transfers the coin to the next by digitally signing an identifier (hash) of the previous transaction and the public address of the next owner.

Steps to run the Network

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult proof-of-work for its block.
4. When a node finds a proof-of-work, it broadcasts the block to all nodes.
5. Nodes accept the block only if all transactions in it are valid and not already spent.
6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Ingredient #1 : Hashes




```
> digest("The quick brown fox did some crypto","sha256")
[1] "c5a96abd038aae61b84f19d2b940aeb95b8b1a15e9d2e7a981816214e5d136a3"
> digest("The quick brown Fox did some crypto","sha256")
[1] "73e95c9c8f2e07566db6341c09422a8b552742d6faaa9fb0a517ee0f6926ea85"
```

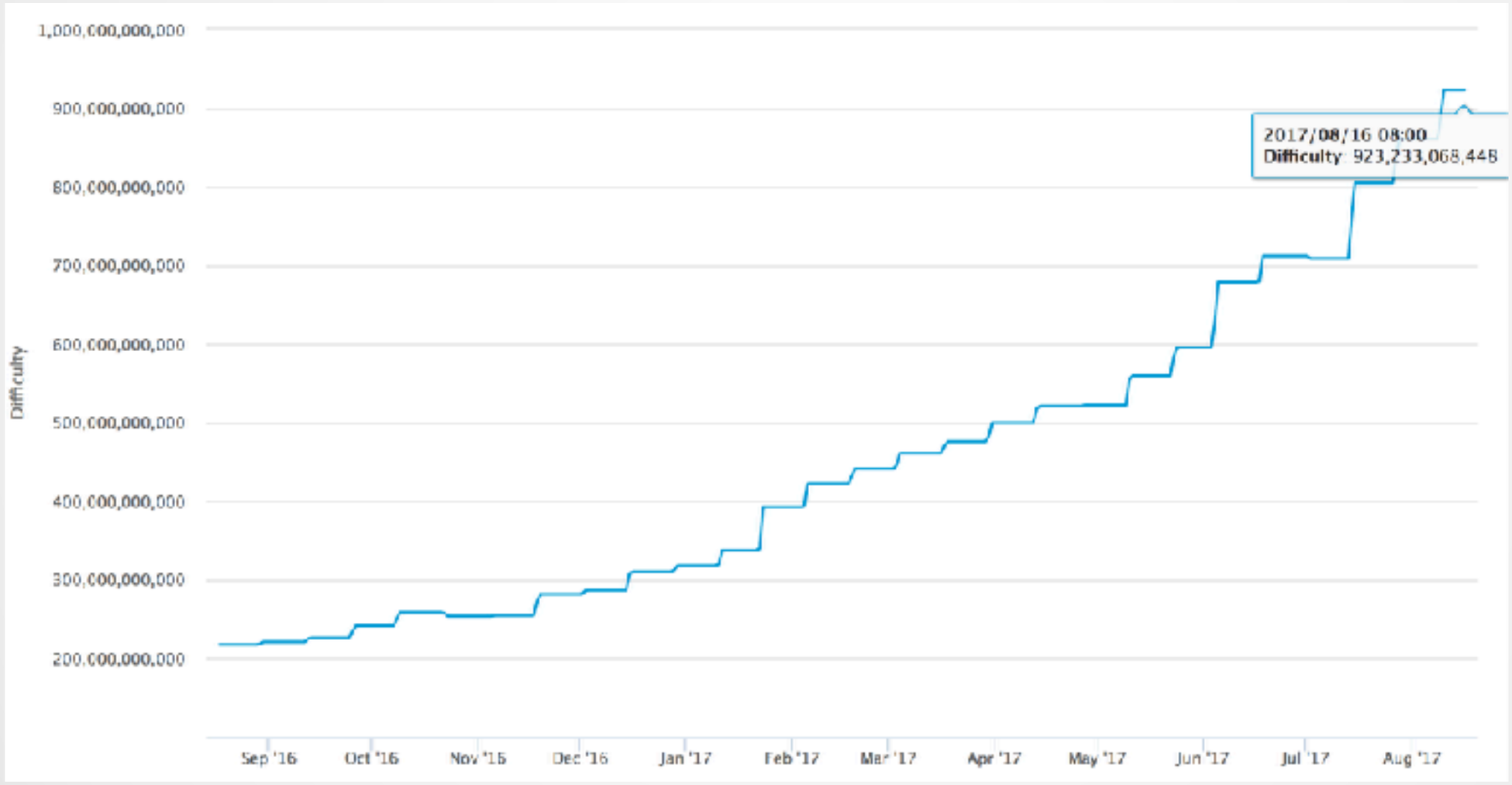
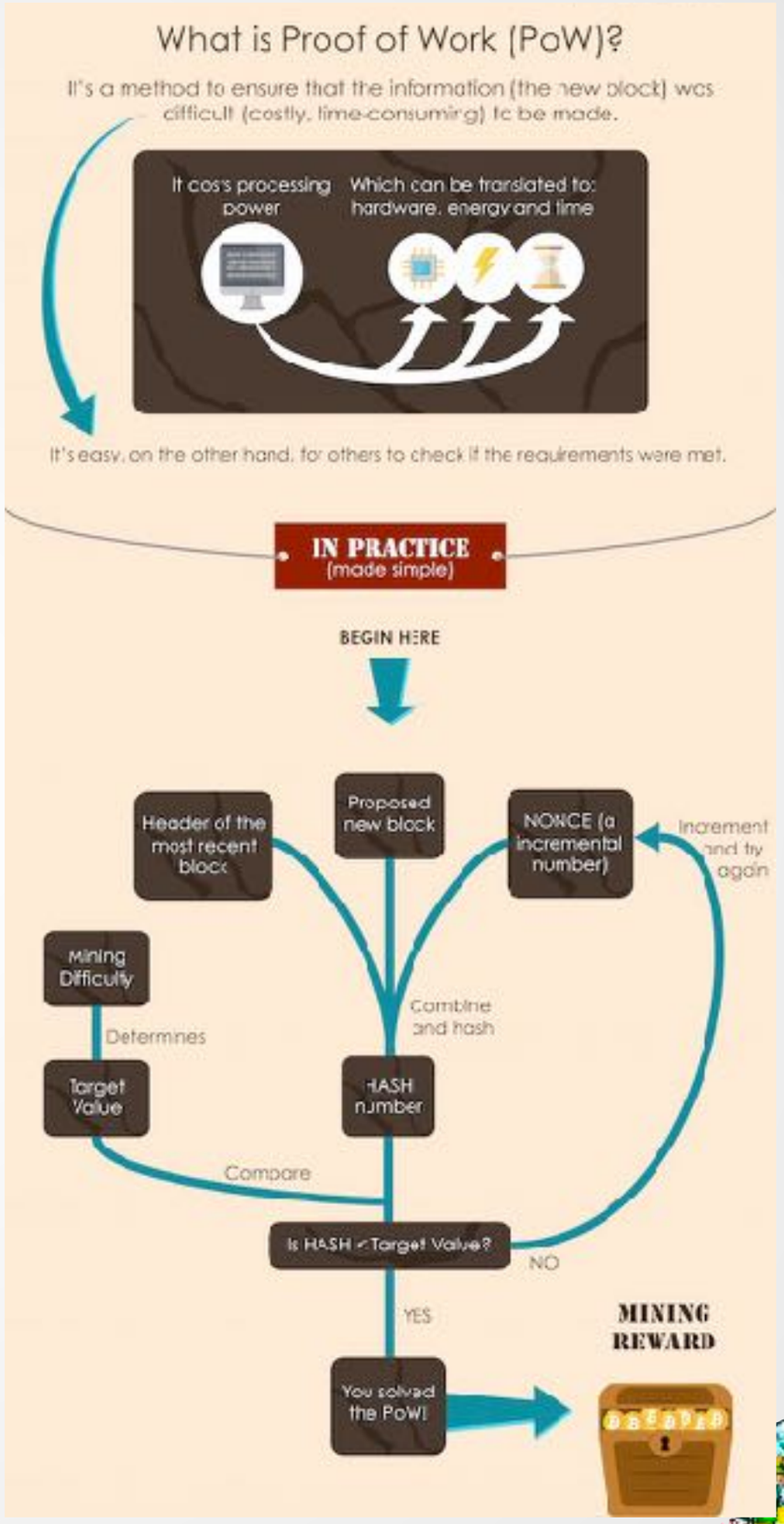
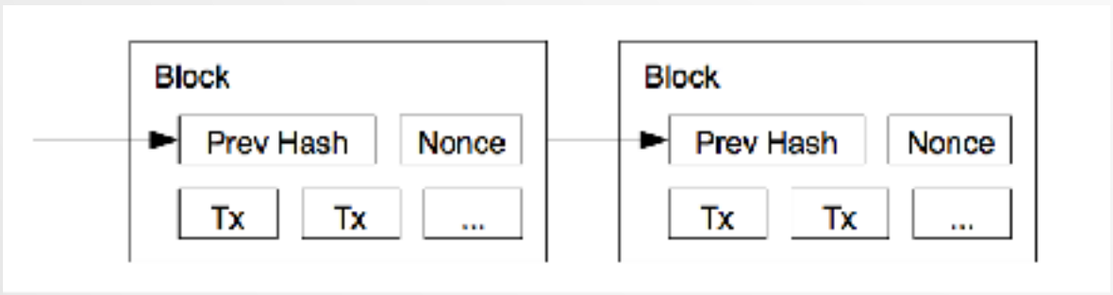


Fig: The difficulty of finding a new block is adjusted periodically a function of how much hashing power is deployed by the network of miners

Proof-of-Work

- Similar to Adam Back's Hashcash
- Incrementing a nonce until a value is found that gives the block's hash the required zero bits
- Solves the problem of determining representation in majority decision making
- "one-CPU-one-vote"
- Difficulty is determined by a moving average of blocks per hour





Why is Ethereum moving from Proof-of-Work to Proof-of-Stake?

Bitcoin and Ethereum burn over $1 million worth of electricity and hardware costs per day as part of their consensus algorithm

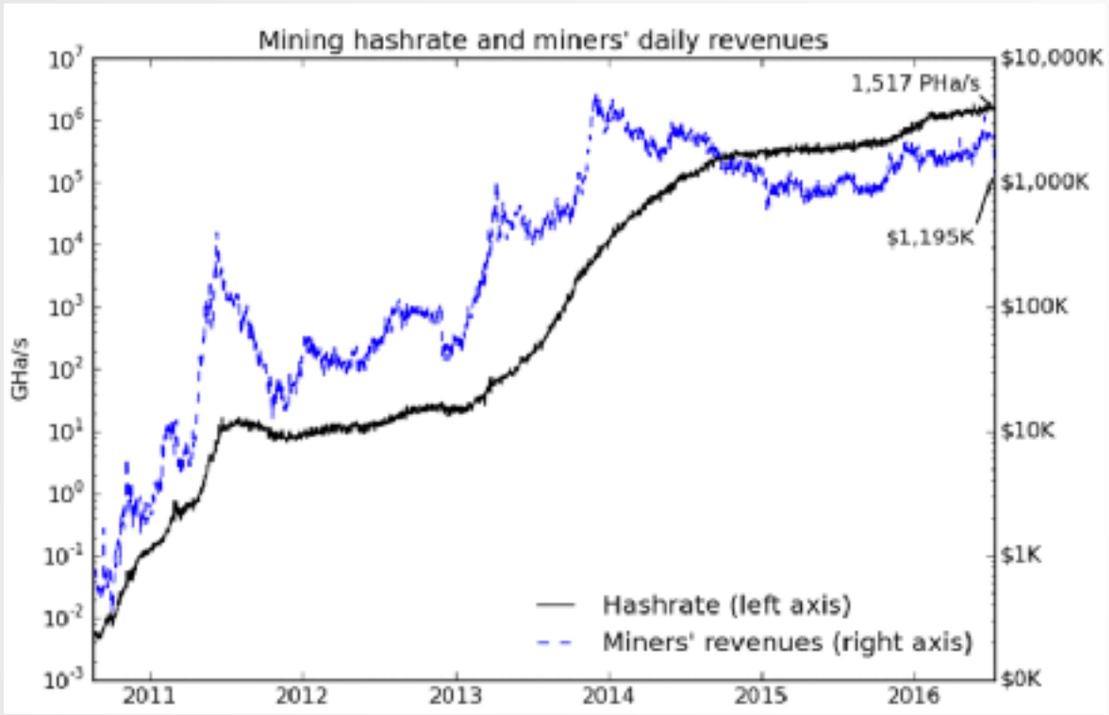
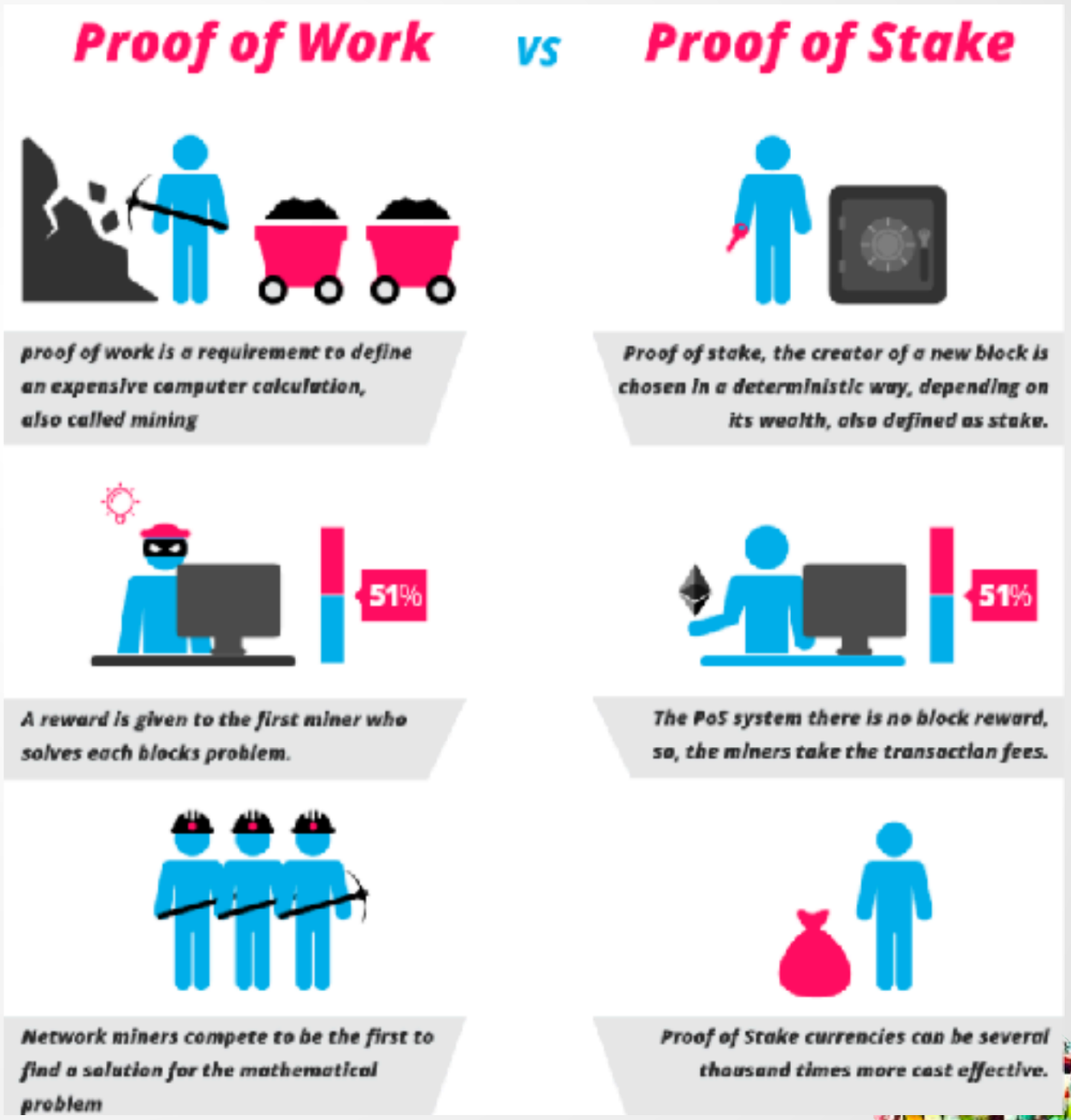


Fig: The total hashrate of Bitcoin mining (in gigahash per second) has increased with mining revenues(in thousands of dollars). The current hash rate is 1517 petahash per second.

PoW: “mining”
PoS: “minting”

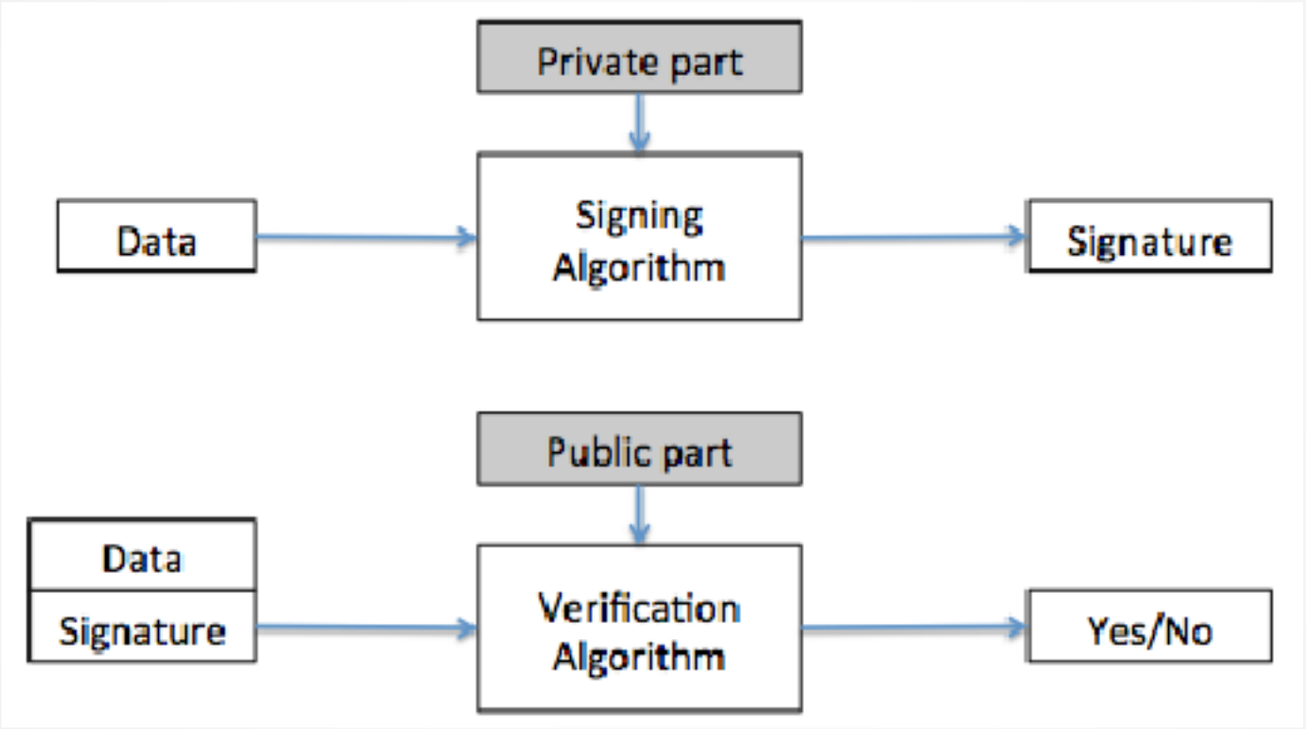


Introducing Casper “the Friendly Ghost”

- deposit + penalty based Proof of Stake
- Validators : any user who signs up by locking their ether into a deposit
- Creating new blocks done through a consensus algorithm in which all validators participate
- Deposit returns provides incentives for honest actions.
- Minimal Slashing Conditions: If a validator makes a malicious move, deposit is burned

Ingredient #2: Signatures

Signing key	
Public part	454F4D3E1..
Private part	56F23F2D..



Math behind Blockchain: Elliptic Curve Digital Signature Algorithm

Can an attacker generate a chain faster than the honest chain?

- Can be characterised as a Binomial Random Walk
- Probability of an attacker catching up with a deficit - Gambler's Ruin problem:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \leq q \\ (q/p)^z & \text{if } p > q \end{cases}$$

- Given our assumption that p>q, probability drops exponentially as the number of blocks the attacker has to catch up with increases

How does Bitcoin circumvent the double spending problem?

- In a double spend attack, the dishonest sender starts creating an alternate version of his initial transaction
- Assuming the honest blocks took the average expected time per block, the attacker's potential progress with a Poisson distribution with expected value:

$$\lambda = z\frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

```
q=0.1
z=0     P=1.0000000
z=1     P=0.2045873
z=2     P=0.0509779
z=3     P=0.0131722
z=4     P=0.0034552
z=5     P=0.0009137
z=6     P=0.0002428
z=7     P=0.0000647
z=8     P=0.0000173
z=9     P=0.0000046
z=10    P=0.0000012

q=0.3
z=0     P=1.0000000
z=5     P=0.1773523
z=10    P=0.0416605
z=15    P=0.0101008
z=20    P=0.0024804
z=25    P=0.0006132
z=30    P=0.0001522
z=35    P=0.0000379
z=40    P=0.0000095
z=45    P=0.0000024
z=50    P=0.0000006
```

- If the recipient waits until z blocks have been linked after the block with his transaction, probability of an attack diminishes with increasing z

Stochastic Properties of CRIX

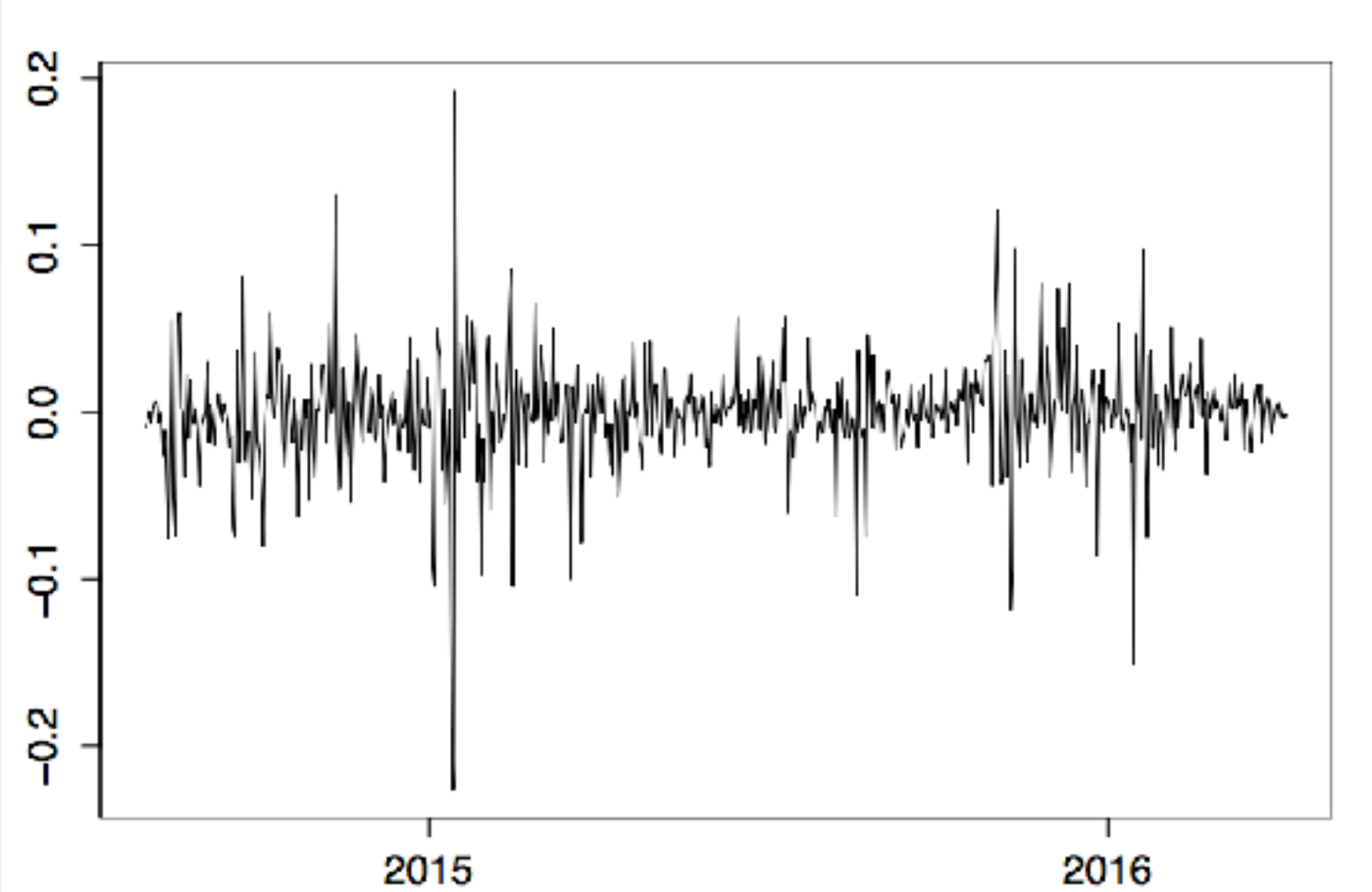


Fig: Log returns of CRIX index from Aug 2nd, 2014 to April 6th, 2016

- Variance adjusted with the GARCH model

Distributional Properties of CRIX

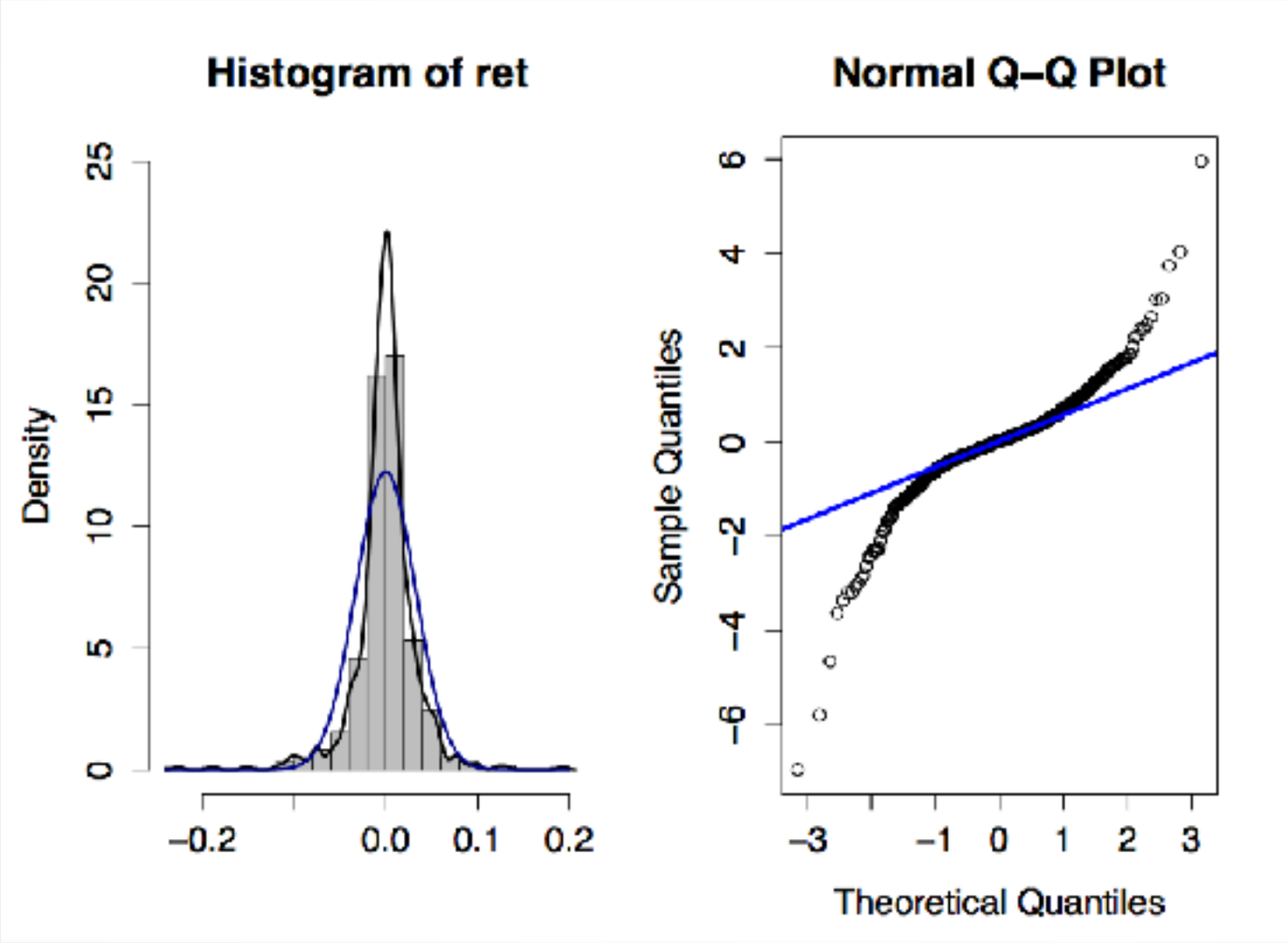


Fig: Histogram and QQ Plot of CRIX returns

- CRIX returns is not normally distributed

Box-Jenkins Procedure

ARIMA Model Selection with AIC and BIC

ARIMA model selected	AIC	BIC
ARIMA(2,0,0)	-2468.83	-2451.15
ARIMA(2,0,2)	-2474.25	-2447.73
ARIMA(2,0,3)	-2472.72	-2441.78
ARIMA(4,0,2)	-2476.35	-2440.99
ARIMA(2,1,1)	-2459.15	-2441.47
ARIMA(2,1,3)	-2464.14	-2437.62

- The ACF pattern suggests that the existence of strong autocorrelations in lag 2 and 8, partial autocorrelation in lag 2, 6 and 8.
- These results suggest that the CRIX return series can be modeled by some ARIMA process, for example ARIMA(2, 0, 2).

Observing Conditional Heteroskedasticity with GARCH Model

GARCH models	Log likelihood	AIC	BIC
GARCH(1,1)	1305.355	-4.239	-4.210
GARCH(1,2)	1309.363	-4.249	-4.213
GARCH(2,1)	1305.142	-4.235	-4.199
GARCH(2,2)	1309.363	-4.245	-4.202

Table: Comparison of GARCH Models

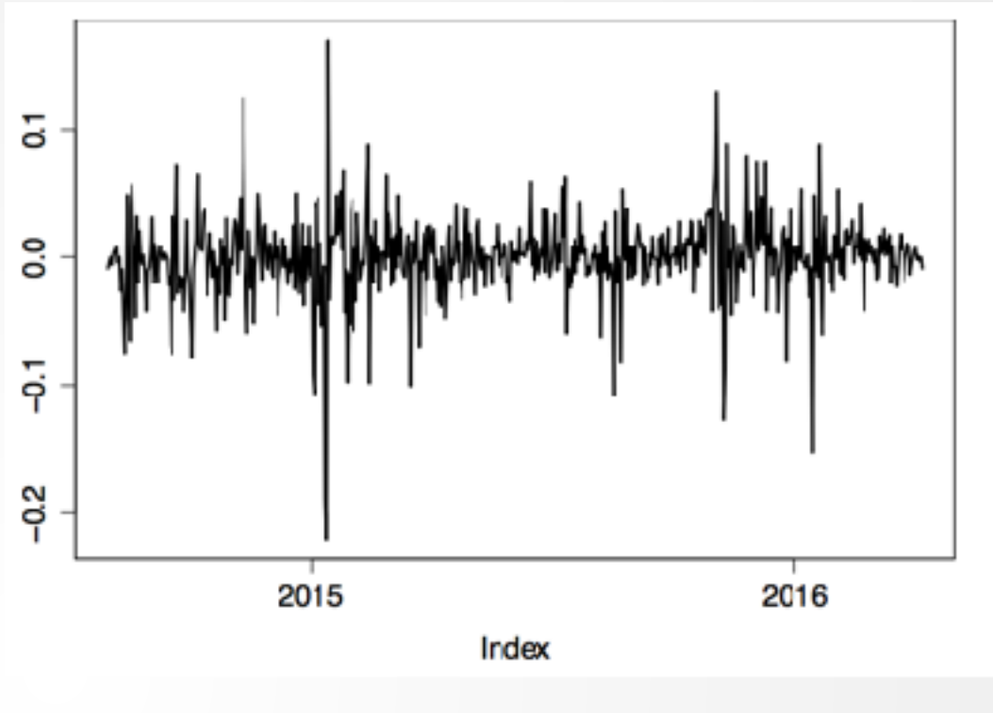


Fig: The ARIMA(2,0,2) - GARCH(1,1) residuals

- Although the model performance of GARCH(1,2) is better than GARCH(1,1), all parameters of GARCH(1,1) are significant.
- All the values are within the confidence bands, so the model residuals have no dependence structure left over different lags.
- Does not capture the leverage effect

Variants of GARCH Model

- The Kolmogorov distance between residuals of the selected model and normal distribution is 0.495 with a p-value of 2.861e-10
- We reject the the null hypothesis that the model residuals re drawn from the normal distribution
- EGARCH(1,1) model with student t distributed innovation term is fitted

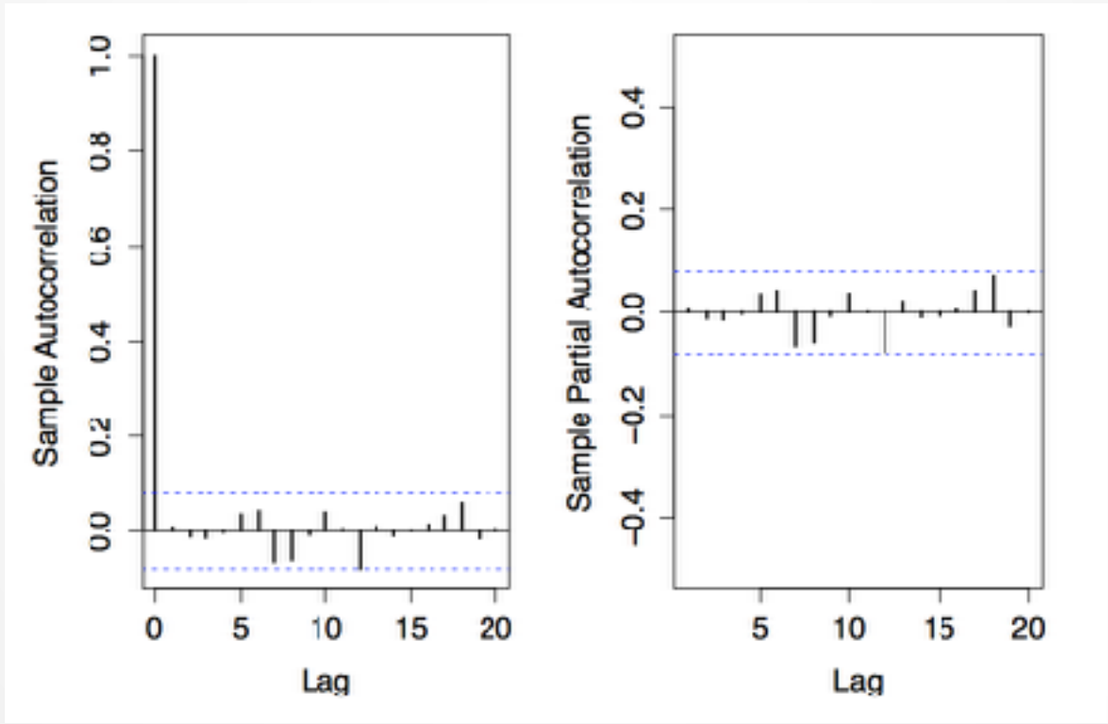


Fig: The ACF and PACF residuals of ARIMA - t- EGARCH process

- The small values indicate independent structure of model residuals

Creating a block

```
import hashlib
import block_params


class Block():

    def __init__(self, params):
        self.index = params.index
        self.previous_hash = params.previous_hash
        self.timestamp = params.timestamp
        self.data = params.data
        self.hash = self.calc_hash()

    def params(self):
        return block_params.BlockParams(
            self.index,
            self.previous_hash,
            self.timestamp,
            self.data
        )

    @classmethod
    def genesis_block(cls):
        params = block_params.BlockParams.genesis_params()
        return cls(params)

    def calc_hash(self):
        return hashlib.sha256(str(self.params()).encode()).hexdigest()

    def has_valid_index(self, previous_block):
        return self.index == previous_block.index + 1

    def has_valid_previous_hash(self, previous_block):
        return self.previous_hash == previous_block.hash

    def has_valid_hash(self):
        return self.calc_hash() == self.hash
```

Voila! The Blockchain

```
import time
import block
import block_params


class BlockChain():

    def __init__(self):
        self.blockchain_store = self.fetch_blockchain()

    def latest_block(self):
        return self.blockchain_store[-1]

    def generate_next_block(self, data):
        index = len(self.blockchain_store)
        previous_hash = self.latest_block().hash
        timestamp = int(time.time())

        params = block_params.BlockParams(index, previous_hash, timestamp, data)
        new_block = block.Block(params)
        self.blockchain_store.append(new_block)

    def fetch_blockchain(self):
        return [block.Block.genesis_block()]

    def receive_new_block(self, new_block):
        previous_block = self.latest_block()

        if not new_block.has_valid_index(previous_block):
            print('invalid index')
            return
        if not new_block.has_valid_previous_hash(previous_block):
            print('invalid previous hash')
            return
        if not new_block.has_valid_hash():
            print('invalid hash')
            return

        self.blockchain_store.append(new_block)
```

Thank you!

Ladislaus von Bortkiewicz Chair of Statistics
Humboldt-Universität zu Berlin
lvb.wiwi.hu-berlin.de