Report for Distributed Systems Seminar on storing parts of Dublin Bus GPS sample data in a Blockchain Data Structure*

Anders Martoja[1]

Abstract— **Blockchain is new disruptive technology that allows to store data in a decentralized manner without the need of a central authority figure. The data, that is stored in a blockchain, is hashed to provide security and is publicly accessible. Hashing data into a secure format is crucial for blockchain. A widely used hashing function in blockchain technologies is SHA-256.**

The author of this report provides an overview of SHA-256 hashing function and an introduction into the inner workings of the blockchain technology. Additionally, the author of this report has implemented the most basic version of a blockchain, simply hashing GPS related data into blocks and then appending them to the chain. The results from the implementation show that creating a blocks that contains GPS related data and then appending them to the chain consumes very little time.

I. INTRODUCTION

Blockchain is a buzzword in academia and industry. Kodak[1], which is a imaging company that is invested in several fields, *e.g.* Print Systems, Enterprise Inkjet Systems, Consumer and Film, Advanced Materials and 3D Printing Technology *etc.*, has released a KODAKOne platform and KODAKCoin cryptocurrency[2]. Which caused their stock prices to rise[3]. While profits are a strong driving factor for blockchains, the technology allows for distribution of databases, so that what ever is stored in the blockchain can be taken as truth. Blockchains enable data sharing between hosts so that the hosts do not need to necessarily trust each other - without a central authority figure[5].

Author's main goal for this report was to implement a basic version of a blockchain, so that GPS related data, could be stored in an immutable data structure. No verifiers, miners, distribution of the blockchains or adequate solutions for the Byzantine fault tolerance[4] are taken into consideration in the work at hand.

This report is organized as follows. Section 2 covers the necessary background for this report, SHA-256 hash function and overview of blockchain technology. Section 3 describes the implementation of the solution. Section 4 shows the results with respect to time. Section 5 then draws conclusions from Section 3 and 4 to highlight the current challenges and future work. Finally, Section 6 presents the conclusion for the report.

*This work was not supported by any organization

[1]Anders Martoja is a 1st year Master's student at the Institute of Computer Science, University of Tartu, Juhan Liivi 2, Tartu, Estonia `anders.martoja at ut.ee`

II. BACKGROUND

This section gives an overview of technologies that are necessary to accomplish the main goal of this report. In the beginning of subsection A, the author gives briefly describes hashing in general and its' properties. The second half of subsection A is dedicated to SHA-256 hash function, which ties into subsection B, that focuses on blockchain, the latter uses SHA-256.

A. SHA-256

Hash function can be any function that takes data of erratic size and maps it to data that is of fixed size. Hash function outputs are called hash values, hash codes, digests or hashes. [6]

In cryptography a hash function allows for easy verification of the digest, when the input data is known. When the input for the hash function is unknown, the hash function is purposefully built to hinder the reconstruction of the input, simply from the hash value. [6]

A n-bit cryptographic hash function takes an input of arbitrary length and produces n-bit digests, with two important properties:

- one-way - when one is in possession of a hash, it should require roughly 2^n computations to reproduce the message that hashes to the given hash value.
- collision-resistant - finging any two messages, that produce exactly the same hash values after going through a hash function, should roughly require $2^{\frac{n}{2}}$ computations.

Before hashing the data the data undergoes a 2-step preprocessing. Firstly, the data is converted into binary and afterwards gets padded, this results in a 512-bit padded message. [7] The pad consists of l, which is the length of a message in bits, followed by a ”1”, which represents the end of the message. Then *k*-zero bits are added. *k* is the solution to the following equation:

$$l + 1 + k \equiv 448 \ mod \ 512$$

k has to be non-negative and to satisfy the requirement of being the smallest solution. The pad is completed by a 64-bit block which is the binary representation of the decimal number *l*.

After completing the pad for the message, it is parsed into *N* 512-bit blocks, in the form of $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Each of these blocks are then divided into 32-bit words $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{(i)}$. [7]

The main SHA-256 compression cycle has two main components:

- SHA-256 compression function, Fig. 1.
- SHA-256 message schedule, Fig. 2.

The compression function utilizes four logic functions and the message schedule uses two. In the following list, the first four functions are from the compression function and the last ones are from scheduling function. Each of these function take a 32-bit words as input and also output 32-bit words. [7]

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x)$$

$$\Sigma_1(x) = S^6(x) \oplus S^{11}(x) \oplus S^{25}(x)$$

$$\sigma_0(x) = S^7(x) \oplus S^{18}(x) \oplus R^3(x)$$

$$\sigma_1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)$$

S^n stands for right rotation by n bits and R^n stands for right shift by n bits.

Before compression ensues the SHA-256 algorithm initializes registers a, b, ..., h with 32-bit word hash values. [7]

The small squares with crosses in the center denote $mod \ 2^{32}$ addition. The main loop Fig. 1 starts off by performing all of the above mentioned logic functions, and then starts reassigning the values according to the Fig. 1. SHA-256 message schedule's output is denoted by W_j Fig. 2, which gets reevaluated in the beginning of every loop iteration. After every 64 cycles eight 32-bit intermediate hash values are calculated, this happen *N* times. In the end, the

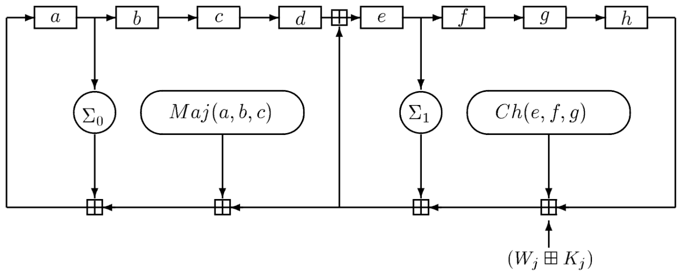


Fig. 1. SHA-256 Compression function on j^{th} step

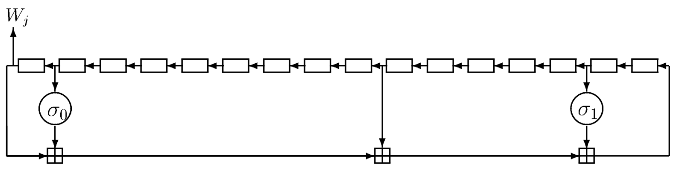


Fig. 2. SHA-256 message schedule

eight intermediate hashes concatenated into a 256-bit hash. Which is the result from the input M.[7]

To summarize this subsection, the computations start out with fixed initial hash value, $H^{(0)}$, which is computed by taking the first eight prime numbers, then taking the square roots of those eight prime numbers and finally taking the fractional parts of those square roots and then formatting those fractions into 32-bit words to produce $H_1^{(0)}, H_2^{(0)}, ..., H_8^{(0)}$. Then using the following formula, to consecutively compute

$$H^{(i)} = H^{(i-1)} + C_{M^i}(H^{(i-1)}).$$

Here, C stands for SHA-256 compression function, that was described above, ”+” denotes 32-bit wise $mod \ 2^{32}$ addition. $H^{(N)}$, where *N* denotes the number of blocks the padded message was parsed into, is the digest of M. [7]

B. Blockchain

The

III. Implementation

The section at hand is divided into two subsection. In subsection A the author describes the data and its' origin. Subsection B gives an overview of the code that was implemented for this report - what was done. Efficacy of the implementation is discussed in the next section, titled ”Performance”.

A. Data

The data, that was used for this report, came from data.gov.ie website, where Dublin City Council had published Dublin Bus GPS sample data in two zip files. Both zips contain several comma-separated values (CSV) format files. Out of the two available zip files, the one titled ”From 1st Jan 2013 to 31st Jan 2013” was chosen. The data covers geographically Dublin City. The data is released under the Creative Commons Attribution 4.0 license. [10] According to the latter, the author of this report is allowed to share and adapt the data at hand, as long as the changes, that are made are presented and appropriate credit is given. [9]

The CSV files contain 15 columns and a myriad number of rows. The exact content of the columns is as follows:

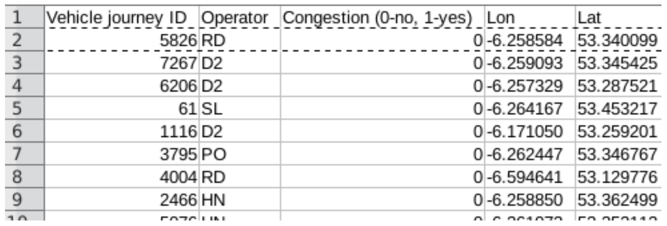
- Timestamp *e.g.* 1354233602000000
- Line ID *e.g.* 272
- Direction *e.g.* 0
- Journey Pattern ID *e.g.* 027B1002
- Time Frame *e.g.* 2012-11-29
- Vehicle Journey ID *e.g.* 331
- Operator *e.g.* HN
- Congestion *e.g.* 0
- Lon WGS84 *e.g.* -6.229033
- Lat WGS84 *e.g.* 53.409618
- Delay *e.g.* 68
- Block ID *e.g.* 272006
- Vehicle ID *e.g.* 33452
- Stop ID *e.g.* 675
- At Stop *e.g.* 0

It becomes clear from the list, that almost all data types are present *i.e.* integers, doubles, strings, signed integers, signed doubles - they are all present in one row. This created a good opportunity to see, if the implementation could handle several data types.

The author of this report felt like for the purpose of this report not all of the columns in the original data were necessary. Hence, the CSV files went through a slight preprocessing, which consisted of the following 2 steps:

- Total number of columns was brought down to five *i.e.* fields containing Vehicle journey ID, Operator, Congestion (0-no, 1-yes), Lon, Lat were kept for testing, like in Fig. 3.
- Data was separated into 10 files with the difference being the number of rows they contained - 100, 250, 500, 1000, 2500, ..., 100000 rows.

To summarize this subsection, the final row, that was ready for hashing had a format of [5826, ’RD’, 0, -6.258584, 53.340099] and data was split into several files, of different sizes (number of rows), in order to measure the performance. Different data types in the list were not a problem for Python 3.5.



1	Vehicle journey ID	Operator	Congestion (0-no, 1-yes)	Lon	Lat
2	5826	RD	0	-6.258584	53.340099
3	7267	D2	0	-6.259093	53.345425
4	6206	D2	0	-6.257329	53.287521
5	61	SL	0	-6.264167	53.453217
6	1116	D2	0	-6.171050	53.259201
7	3795	PO	0	-6.262447	53.346767
8	4004	RD	0	-6.594641	53.129776
9	2466	HN	0	-6.258850	53.362499

Fig. 3. Sample of the CSV file after preprocessing

B. *Code and modifications*

As mentioned previously the code was implemented and modified in Python version 3.5.3. The code was borrowed from Oscar Alsing’s github. The github README.md says it is for ”A simple blockchain for educational purposes”. [8] The author of this report also discovered that Oscar Alsing also has a YouTube channel and a corresponding video about the code. In the video [11] the author says ”you can experiment and you know do whatever you want with it” - this piece of information was just to clarify about the usage of the code.

The code for this report was divided up into two .py files. Class Block, which is responsible for the content of a hashed block, has the following fields:

- digest of the previous block
- data - payload to be stored
- timestamp - block creation time
- hash for the block that is being created currently

There is also a method for creating the genesis block also known as the very first block of the chain. The digest of the previous block and the data are set to equal to ”0” for the genesis block.

The last part of the Block class is the hashing method. Where the digest of the previous block, data and timestamp are converted into strings. Then the string gets turned into binary and then hashed. Oscar Alsing created the hashing method in a way, that the data gets hashed twice by the Python’s hashlib library’s sha256 function - inner hash, which gets turned into binary again and then hashed as outer hash. The author of this report feels as if the double hashing is redundant, but saw no harm in it either and kept it as was.

The blockchain.py file was however modified by the author of this report. Firstly, the code was made to handle and read CSV files. The separate data files, that were created in subsection A, were given as arguments to the blockchain.py file from the commandline. Each row from the CSV file was read in as a list of comma separated values and attributed to the data field of the block, that was to be created. The final modification to the blockchain.py file was done with the Python timeit library, which allowed the author of this report to measure the time for creating blocks.

As was it was mentioned in the subsection A, the author of this report made ten input files. That meant that the gathering performance data, in this case time, was carried out in the terminal of the author's laptop by inserting 10 commands *i.e.* python blockchain.py Data_for_testing*.csv. The asterisk symbol was replaced with the corresponding number of rows to be measured *i.e.* 100, 250, 500, 1000, ..., 100000.

IV. Performance

Hashing function performs in constant time, due to the size of the input for the function does not vary. Thus, adding new blocks to the blockchain is performed in constant time. *ergo* the current code performs in linear time. It is dependent on the size of the size of the input, number of rows that need to be turned into blocks.

The hashing function that was defined in the Block class performs the hashing of data in constant time. That is partly due to the fact, that the size of the data, that is hashed into block, is constant. There are always five fields in the row containing same types of data. Also, while the hashing process was ongoing the author of this report made sure, no other user related applications were running in the background. Since the hashing was done in constant time, the relationship between the amount of rows that were hashed and the time it required to complete the task was linear. As it was described in the beginning of the report the author focused on only implementing the simplest functionality of the blockchain. Therefore no other performance related data was gathered.

The table I below displays the gathered data. The left column contain row numbers form 100 till 100000 and the right column contains the corresponding time in seconds.

TABLE I

Time it took to hash and append data into the blockchain

Number of rows hashed	Time in seconds
100	0.001560926437
250	0.003134965897
500	0.008181095123
1000	0.01314806938
2500	0.03413701057
5000	0.05853295326
10000	0.1154618263
25000	0.2901489735
50000	0.5976510048
100000	1.213989019

The line graph, Fig. 4 displays relationship between time (in seconds), on the Y-axis and number of rows to be hashed, on the X-axis. From that it is obvious that the hashing function performed in linear time.

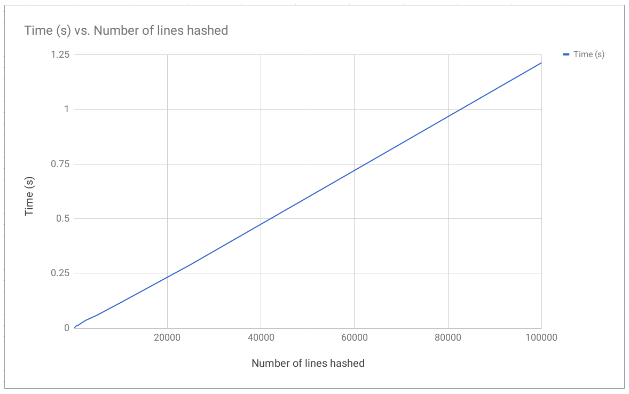


Fig. 4. Something about Cool

V. DISCUSSION AND FUTURE WORKS

While searching for example solutions of a simple blockchain code, the author came across several potential sources [12] [13]. The final decision to use Oscar Alsing's code was made thanks to Oscar Alsing's clear statement, that his code could be used as one needed for educational purposes. The author of this report has used the source code to gain more insight into blockchain and modified it to fit the goal of this report.

Only the simplest functionality has been implemented in the the work at hand. Next steps would be to go through and try all of the other sources the author had mentioned before. This would ensure that the knowledge base would grow even further and introduce new information regarding the basic functionality of blockchain technology in general. Additionally, it is important to familiarize oneself with all the possible ways to implement the basic blockchain to avoid possible confusion in the future.

After gaining more insight into blockchain, the next important functionality to implement would be a verifier, so that it would be possible to check the integrity of the blockchain. Checking for integrity is vital part of any functioning blockchain, what use is a blockchain and its' property of immutability if it can not be checked.

Then the consensus problem has to be handled in a way, that would allow for effective functioning of the blockchain. And the concept of a miner, the one that adds info to the blockchain, has to be introduced to this body of work. Finally, distributing a copy of blockchain to several hosts and maintaining it effectively should be done.

Regarding the performance of the current code, it should be noted, that the results that were obtained are not relatable to the real world. In the implementation that was carried out for this report, the hashing happens at the speed of the processor. In a real world application this kinds of speeds would be considered spamming the blockchain and would not be acceptable.

VI. CONCLUSIONS

There were two goals for this report. Firstly, to give an overview of SHA-256 hash function and blockchain technology. Secondly, implement a blockchain with basic functionality. The implementation was carried out in python language and using source code from Oscar Alsing's github [8]. The source code was modified to handle CSV data, that was taken from [10], which was free for use according to Creative Commons Attribution 4.0 International [9]. Hashing was done on 10 CSV files, all of which were of different size. The performance of the implementation was measured and presented in the report's section IV, titled ”Performance”. The implementation performed way better than was expected by the author, but also a caveat of the implementation was discussed in the V section of this report. It is clear there is more work to be done regarding the technology at hand. In general, all of the goals, the author set before starting this work, were met.

References

[1] ”About Kodak”, Company homepage, https://www.kodak.com/US/en/corp/company/default.htm

[2] KODAK and WENN Digital Partner to Launch Major Blockchain Initiative and Cryptocurrency, 9th of Jan. 2018, https://www.kodak.com/corp/press_center/kodak_and_wenn_digital_partner_to_launch_major_blockchain_initiative_and_cryptocurrency/default.htm

[3] Kodak announces its own cryptocurrency and watches stock price skyrocket, Shannon Liao, The Verge, 9th of Jan. 2018, https://www.theverge.com/2018/1/9/16869998/kodak-kodakcoin-blockchain-platform-ethereum-ledger-stock-price

[4] Byzantine fault tolerance, From Wikipedia, the free encyclopedia , used on 09.05.2018, https://en.wikipedia.org/wiki/Byzantine_fault_tolerance

[5] What is the Difference Between a Blockchain and a Database?, Nolan Bauerle, coindesk, used on 09.05.2018, https://www.coindesk.com/information/what-is-the-difference-blockchain-and-database/

[6] Hash function , From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Hash_function, Used on 06.06.18

[7] Descriptions of SHA-256, SHA-384, and SHA-512 , The Information Warfare Site, http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf, Used on 06.06.18

[8] oalsing, Simple-Blockchain, GitHub, https://github.com/oalsing/Simple-Blockchain, used on: 29th of May, 2019

[9] Attribution 4.0 International (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

[10] Dublin City Council, Dublin Bus GPS sample data from Dublin City Council (Insight Project), released: 2013-06-28, License: Creative Commons Attribution 4.0 https://data.gov.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insigh

[11] Blockchain in 15 MINUTES! Code Your Own Simple Blockchain, Oscar Alsing, YouTube, Published on: 12. nov 2017 https://www.youtube.com/watch?v=p4lw8CuQtq8&t=40s, Used on 15. may 2018

[12] Learn Blockchains by Building One, Daniel van Flymen ,https://hackernoon.com/learn-blockchains-by-building-one-117428612f46 Used on: 15.05.18

[13] Building a Simple Blockchain in Python, Gaurav Jain, http://www.pyscoop.com/building-a-simple-blockchain-in-python/, Used on: 15.05.18