Math Industry Workshop

COMPRESSING THE TRANSACTION DATA OF BLOCKCHAIN

IVAN LAU, SHANG LI, EVAN MACNEIL, ALEXANDRA MCSWEEN,
ABHISHEK KUMAR SHUKLA, AND YANHONG XU

Abstract. Since its inception in 2009, the Bitcoin blockchain has grown in size to more than 295 GB and continues to grow by approximately 50 GB per year. The decentralized, trustless framework of Blockchain requires participating nodes to store the entire blockchain data. This keeps smaller computing devices from participating fully in the network.

In this paper, we present methods to compress the blockchain in a lossless fashion. Our main observations rely on finding redundancies in the Bitcoin transaction data which can be leveraged to decrease the blockchain size. We are able to achieve a compression rate of approximately 20% by applying various compression schemes in concert. Further compression might be possible by using generic compression algorithms on top of our compression scheme.

1. Introduction

Bitcoin, the most famous cryptocurrency, was first introduced in Satoshi Nakamoto's white paper in 2008 [Nak08]. Philosophically, the idea is to have a peer-to-peer network of electronic cash without a centralized financial institution. In this decentralized system, we can operate (mostly) anonymously but we cannot trust anybody.

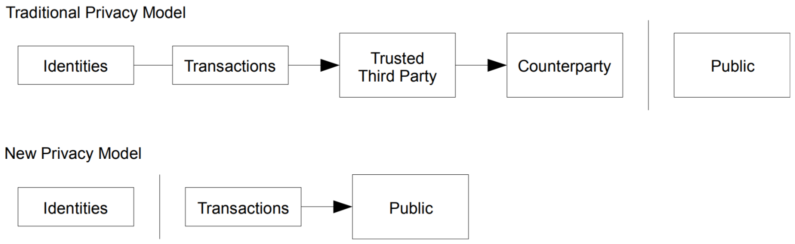

Figure 1. From [Nak08]

However, participating fully in the Bitcoin network remains inaccessible to most people. To verify the history of transactions, a node needs almost 300 gigabytes (GB) to store the blockchain [Blo]. To help "mine" the next block in the blockchain, thereby confirming new transactions, substantial computing power is needed. Many "wallet" apps allow users to trade in Bitcoin on the network without requiring large amounts of storage space and CPU power, but they require the user to trust in third parties, violating Bitcoin's trustless model.

The Divi Project [Div] is a blockchain-based cryptocurrency and smart wallet that seeks to make the cryptocurrency market accessible to all. It offers the first and only genuinely one-click masternode deployment and five tiers of affordability.

The motivation behind our project is help improve this inaccessibility. The original problem was posed by Germàn Luna from the Divi Project. We hoped to answer the following questions:

- Determine to what extent a transaction graph can be compressed (for later decompression) or what obstructions exist to its compression.
- What compression ratio can we achieve for an ordered sequence of cryptographic hashes?

These questions were intentionally left vague to allow us the freedom to explore and develop the project ourselves. We explored many options but in the end decided to look at techniques and places to compress transaction data since they carry the bulk of the storage strain.

For the purposes of our project, we worked on Bitcoin transactions, though we expect our methods to be applicable to other cryptocurrencies, including the Divi Project, as well.

1.1. Anatomy of the Bitcoin Blockchain.

We can think of the blockchain as the ledger. It consists of a linked list of blocks, chronologically ordered, and each block contains a list of transactions. Besides transaction data, a block also contains some metadata regarding the block itself. In detail, the components of the block are as follows.

- The *size* of the block in bytes.
- The *block header*, which consists of:
  - The *version number* of the block.
  - The *hash* of the header of the previous block in the chain.
  - The *Merkle root* of the transactions in this block. This is a combined hash of all of the transactions.
  - A *timestamp* indicating approximately when the block was mined.
  - The *target* (also called "bits"). The smaller the target, the more difficult the block is to mine
  - A *nonce*. A value placed in the header so that the header's hash is smaller than the target.
- The *transaction count*, i.e. the number of transactions in the block.

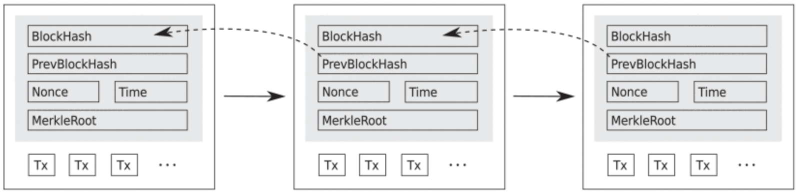

Figure 2. From [TS16]

- A list of *transactions*.

To mine a new block, a miner collects a number of transactions into a list and computes their Merkle root [Mer80]. The Bitcoin protocol dictates what the target value should be. The miner then iterates over several nonce values until one is found that causes the block header's hash to be lesser than the target value. It takes the entire network of miners working together about 10 minutes to mine a block; if miners work too fast or too slow, the Bitcoin protocol adjusts the target to keep this time around 10 minutes.

Note that a block header depends on the hashes of the blocks before it, which in turn depend on the transactions in those blocks. Therefore editing a previous transaction requires recomputing all subsequent block headers. Meanwhile, the mining network will be adding new blocks. Someone maliciously editing transactions will not be able to catch up unless they have more CPU power than the rest of the network combined.

1.2. **Anatomy of a Transaction.** Each block consists of one or more transactions. The first transaction in a block is the coinbase transaction, which rewards miners with BTC when adding a new block to the chain. Subsequent transactions reflect people sending money to one another. A transaction contains the following data in this order:

- The *version number*, either 1 or 2. Version 2 reinterprets the sequence numbers, below.
- A *flag* that, if present, is always equal to 1 and indicates that input script signatures are to be found in the witness section below[1].
- The *input count*, or number of inputs in the transaction.
- Several inputs, the number of which must match the count above. Each input in turn contains the following data:
  - The *previous transaction hash*, the hash of the transaction whose output this input will spend.
  - The *index* of the output within that transaction, since a transaction may have multiple outputs.
  - The *length* of the script signature to follow.
  - The *script signature*, a script that, when combined the output's script public key, must evaluate to true in order to spend the output.
  - The *sequence number*, related to when a transaction becomes final.
- The *output count*, the number of outputs in the transaction.
- Several *outputs*, each of which contain the following data:
  - A *value*, the amount of bitcoin to be transferred.
  - The *length* of the script public key to follow.
  - The *script public key*, a script that accepts an input and evaluate to true or false. Only the intended recipient should be able to produce the input that causes it to evaluate to true.

[1] "Segregated witness" (or "SegWit") is an upgrade to Bitcoin to address transaction malleability and block size limits. See [LLW]

- A list of *witnesses* (if the flag above was present) equal in number to the number of inputs. For each witness we have the number of items to push onto the stack (when evaluating scripts) followed by those items preceded by their lengths.
- A *lock time*, related to when a transaction becomes final.

1.3. **A Raw Transaction.** A transaction appears on the Bitcoin network as a serialized sequence of bytes. Here, is an example of one such serialized transaction, written in hexadecimal:

```
020000000001020df23cb58292fa49a1c14083e74b8f79d5dee5e32f75a96798438e
3be32ab68b0100000017160014b3026ad925e5c05d901493a733edfac535413f97fe
ffffff11e522fdf84b31800d1504d88b0bcd2fa36dcf83d3304222cc1ada77bd9b9d
1b00000000171600141ab4829400dd414ef49069b984d542847fb06f65feffffff02
808417000000000017a914d084a31c88c447cf6600b3cef10c63514bc0a91c878020
13000000000017a9148becb7c6ba1a3cde90cfa91dd28d3143c0c412428702473044
022054b3e206d43deb741f2e581d312bec9739d55c1fe8340a53be631a8fc818a269
022060fdec6eb9ef9bd9f0aadd0175018853670ad5f3cc7b33dd42668d31c82b9fbd
01210285687bd88db3039d5878e120a28b4f62e4726a2c08638d9181fa9233a20acb
4a0247304402203cea8cf98b019bc55c6d92c571b86c9a2bf9430cd340a487a7ffea
75e14b3ff602204c16854d229b883ee1009d961727d03479e917e675efa4b6331583
d2b3e66b7c01210268aded79b39ba3fd5dae2c335cfcaa676c73c7679ab1fd424fe6
748a079fe427c2dd0900
```

A transaction in this format is called a raw transaction. Reading left-to-right, all of the data from § 1.2 appear in order. If one knows the size in bytes of each piece of data, one may extract the meaning from the raw bytes. For instance, the first 4 bytes always correspond to the transaction version number. If the next byte is `00`, then it is followed by `01`, indicating the flag `0001` is present. Otherwise, if the next byte is non-zero, it belongs to the input count (which is necessarily non-zero). If the flag is present, it is then followed by the input count. The raw transaction above is parsed in full in Table 1 on next page.

Table 1: A parsed transaction

		Version	02000000
		Flag	0001
		Input count	02
Input 0		Prev. hash	0df23cb58292fa49a1c14083e74b8f79d5dee5e32f75a96798438e3be32ab68b
		Prev. index	01000000
		Script length	17
		Script sig.	160014b3026ad925e5c05d901493a733edfac535413f97
		Sequence no.	feffffff
Input 1		Prev. hash	11e522fdf84b31800d1504d88b0bcd2fa36dcf83d3304222cc1ada77bd9b9d1b
		Prev. index	00000000
		Script length	17
		Script sig.	1600141ab4829400dd414ef49069b984d542847fb06f65
		Sequence no.	feffffff
		Output count	02
Output 0		Value	8084170000000000
		Script length	17
		Script pub. key	a914d084a31c88c447cf6600b3cef10c63514bc0a91c87
Output 1		Value	8020130000000000
		Script length	17
		Script pub. key	a9148becb7c6ba1a3cde90cfa91dd28d3143c0c4124287
Witness 0		Item count	02
	Item 0	Length	47
		Data	3044022054b3e206d43deb741f2e581d312bec9739d55c1fe8340a53be631a8fc818a269022060fdec6eb9ef9bd9f0aadd0175018853670ad5f3cc7b33dd42668d31c82b9fbd01
	Item 1	Length	21
		Data	0285687bd88db3039d5878e120a28b4f62e4726a2c08638d9181fa9233a20acb4a
Witness 1		Item count	02
	Item 0	Length	47
		Data	304402203cea8cf98b019bc55c6d92c571b86c9a2bf9430cd340a487a7ffea75e14b3ff602204c16854d229b883ee1009d961727d03479e917e675efa4b6331583d2b3e66b7c01
	Item 1	Length	21
		Data	0268aded79b39ba3fd5dae2c335cfcaa676c73c7679ab1fd424fe6748a079fe427

Lock Time	`c2dd0900`

We note that many of the integer values in Table 1 (namely the version number, sequence numbers, indices, values, and lock time) are stored in little-endian format, meaning the least significant byte appear first. As such, `8084170000000000` would be read as the hexadecimal number `0x178480`, or the decimal number 1541248, indicating a transfer of 0.01541248 BTC.

2. Compression

2.1. Addresses.

Addresses in Bitcoin take the form of ECDSA (Elliptic Curve Digital Signing Algorithm) private key-public key pairs, on the elliptic curve Secp256k1. A private key is a randomly chosen 256-bit integer. The associated public key is obtained by multiplying the curve's standardized base point by the private key. Thus the public key consists of the x and y coordinates, each 32 bytes, of a point on the elliptic curve and can be used as a public address to which Bitcoins may be sent. However, advancements in quantum computing may lead to private keys being recoverable from public addresses using Shor's Algorithm. Bitcoin users now use 20-byte hashes of public keys, rather than the public keys themselves, as addresses.

Addresses appear in transaction scripts. For maximum privacy, a new address should be used for each transaction. Bitcoin software can generate and manage a multitude of addresses for the user. However, through our analysis we have found that some addresses have been reused in more than 50,000 transactions. Moreover, fewer than 2^{32} addresses appear in the Bitcoin blockchain, so that we can store addresses in a table and refer to them instead by a 4-byte index into this table. This saves us space as long as addresses are reused sufficiently often. Indeed, we find this to be the case.

We estimated the number of times an address is reused in the blockchain based on a sample of addresses. We queried `https://sochain.com` for the number of transactions involving a list of 74,591 addresses. Table 2 shows part of the data. The addresses are sorted lexicographically. We can see the first address was used 55,417 times in the sample, while the next two addresses were used 313 and 136 times, respectively.

Address	Frequency
`1111111111111111111114oLvT2`	55,417
`11112BvbV6fY4Y5rghDB1vnJtLGSjoB2n`	313
`1111VHuXEzHaRCgXbVwojtaP7Co3QABb`	136
`1111vP5eq5RCmRBeMwJGFW65owVtb3nM`	67
`11121FrRst9KCVrdM8SqLRzAFw3b1woSno`	1
...	...
`12pPUDiYU7JWejK6gT2Rr9NxS5QsGcF78f`	6
`12pPup7Pe1XGhpCLWHeGm9D9KBjLG4mvQv`	1

Table 2. Address reuse frequency

Table 3 groups addresses by the number of times there were reused. We can see that there were 53,967 addresses that were used exactly once each; 2,277 addresses that were used exactly twice each; 1,117 addresses used exactly three times teach; and so on. We found that the most frequently appearing address was reused 59,887 times in the sample. Furthermore, 72% of addresses were used only once while 28% of addresses were reused.

# of uses	# of addresses	% of sample
1	53,967	72.0021%
2	2,277	3.0379%
3	1,117	1.4903%
4	892	1.1901%
5	839	1.1194%
…	…	…
20,931	1	0.0013%
26,484	1	0.0013%
32,269	1	0.0013%
55,417	1	0.0013%
59,887	1	0.0013%

Table 3. Addresses by reuse frequency

If this sample is representative of the whole blockchain, then given that there are approximately 1.5×10^9 outputs [Bit], we can estimate the number of addresses as follows. Let x be the number of addresses in the Bitcoin blockchain. Then out of all the outputs $0.720021x$ are the number of addresses appearing exactly once, $0.030379x$ are the number of addresses appearing exactly twice, and so on. We solve for x in

$$x\,(1 \times 0.720021 + 2 \times 0.030379 + 3 \times 0.014903 + \ldots + 59887 \times 0.000013) = 1.5{\times}10^9$$

and find that there are approximately $x = 42,943,936$ distinct addresses appearing in the blockchain. This is much fewer than 2^{32} addresses, so we can store these addresses in a table and refer to them by a 4-byte index instead. Implementing such a table would cost $4x$ bytes, about 819 MB. However, replacing 20-byte addresses by 4-byte indices saves us 16 bytes per output, approximately 22.4 GB. The net savings are about 21.6 GB.

2.2. **Transaction Hashes.** Each transaction input refers to an unused output of an earlier transaction through the corresponding 32 bytes transaction hash (TXID). Therefore, transaction hashes will appear multiple times in the blockchain. Using data from [Bit] and [Tot], we find that there are approximately 1.4×10^9 inputs (or spent outputs), but only about 6.0×10^8 unique TXIDs. Therefore the average TXID is used 2.3 times.

Similar to § 2.1, we can store TXIDs in a table and refer to them instead by a 4-byte index. Implementing this table would cost 32 bytes per unique TXID, about

17.9 GB. However, we save 28 bytes per input, about 36.5 GB in total. The net savings are about 18.6 GB.

It is also worth mentioning that transaction hashes have a kind of "recursive structure" that can be exploited for further compression later on. Namely, since hashing is a deterministic operation and transactions depend on previous transactions' hashes, a different kind of lookup table might be possible where some hashes are stored and others are derived at decompression time by hashing decompressed data.

2.3. **Scripts.** A script is a list of instructions (also called script words, opcodes, or commands). It is a stack-based language processed from left to right. Every transaction input and output contains a script. In order for an input to spend an output, the input and output scripts are combined into one script, then evaluated. The spending is permitted if the script executes without error and leaves the value "true" on the stack.

Most scripts follow one of a few standardized forms, the most common of which is Pay to Public Key Hash (P2PKH). In P2PKH, the input script pushes a digital signature and public key onto the stack while the output script hashes the public key, compares it to an expected values, then verifies the signature.

In a script, data to be pushed to the stack is generally enclosed in angle brackets `<>` and data push commands are omitted while non-bracketed words are opcodes. For the sake of clarity, we include length of the script (`lenScript`) as well as push commands (`PUSHBYTES`) here. We now list five of the most common standardized scripts appearing in Bitcoin transactions, as well as how they appear in encoded in a raw transaction. Besides these five types, there are also multi-signature scripts (MULTISIG), unspendable outputs (OP_RETURN), and non-standard scripts.

PUBKEY	`lenScript PUSHBYTES[65] <pubKey> OP_CHECKSIG`
	`4341<pubKey>ac`
P2PKH	`lenScript OP_DUP OP_HASH160 PUSHBYTES[20] <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`
	`1976a914<pubKeyHash>88ac`
P2SH	`lenScript OP_HASH160 PUSHBYTES[20] <scriptHash> OP_EQUAL`
	`17a914<scriptHash>87`
P2WPKH	`lenScript OP_0 PUSHBYTES[20] <pubKeyHash>`
	`160014<pubKeyHash>`
P2WSH	`lenScript OP_0 PUSHBYTES[32] <witScriptHash>`
	`220020<witScriptHash>`

Table 4. Standard scripts

From Table 4, we observe that these five standard scripts have fixed formats except where the public keys may vary. For instance, a P2PKH script is always 26 bytes, starts with the bytes `1976a9a4`, and ends with the bytes `88ac`. One can replace this with a single byte indicating it is a P2PKH script, followed by the 20-byte hash, which may vary from script to script. We can do likewise for the other

Script type	Compressed script	Bytes saved
PUBKEY	`00<pubKey>`	2
P2PKH	`01<pubKeyHash>`	5
P2SH	`02<scriptHash>`	3
P2WPKH	`03<pubKeyHash>`	2
P2WSH	`04<scriptHash>`	2
Other	`05<script>`	-1

Table 5. Compressed scripts

standard script types. Table 5 summarizes how we may compress these scripts. There is a drawback in that we would be required to reserve a byte to indicate if a script is not of one of these five handled types, followed by the uncompressed script. This costs us one byte per unhandled script, but this cost is offset by the fact that these unhandled types are far outnumbered by the others.

The savings in Table 5 are per output with a script of the given type. Using data from [Bit], we compute the expected total savings by multiplying the per-output savings by the number of outputs with the given script type. For instance, at the time of writing, there are 1,044,567,464 P2PKH scripts, so that we can save 4.86 GB compressing P2PKH scripts alone. There are 46,788,036 scripts not of the above five types, costing us about 44.6 MB with our compression scheme. Putting it all together, we expect a net savings of 5.91 GB.

2.4. **Version Number, Flag, Lock Time, and Sequence Numbers.** Each transaction has a version number, stored in 4 bytes. However, there are presently only two transaction versions: version 1 and version 2. The transaction version can therefore be represented by a single bit until such a time as a new version is created. Compressed data must be written in whole bytes at a time to file, but we observe that there is other information in a transaction that can also be represented by a single bit — namely the flag and information pertaining to the lock time and sequence numbers — and we collect these all into one byte together.

A transaction sometimes includes a flag that, if present, takes 2 bytes and is always equal to `0x0001`. This flag is used to indicate whether the transaction includes any witness data (used in scripts such as P2WPKH and P2WSH). As this can be represented by one bit, we store this in the same byte as the transaction version number.

Each transaction has a 4-byte "lock time" and each input has a 4-byte "sequence number". The lock time and sequence number relate to when the transaction becomes final. The lock time is usually assigned the minimum value `0x00000000` while the sequence numbers are usually the maximum value `0xffffffff`. We use one bit to represent whether the lock time is zero or not and store this bit with the version number and flag. We write the lock time to the compressed file only when it is non-zero. Similarly, we use one bit to represent whether all sequence numbers are maximal or if one sequence number is non-maximal. If they are all maximal, then we do not write them to the compressed file. Moreover, if the sequence number is not maximal, then we also observe that it is usually some number close to `0xffffffff`. Indeed, we notice that the sequence number is usually between `0xffffff00` and `0xffffffff`. These we can represent with a single byte by storing instead their distance from `0xffffffff`.

In total, we save

- 3 bytes per transaction by compressing the version number down to 1 byte.
- 2 bytes per transaction with the flag present.
- 4 bytes per transaction with a lock time of `0x00000000`.
- 4 bytes per input in a transaction in which all sequence numbers are `0xffffffff`.
- 3 bytes per input in a transaction in which all sequence numbers are between `0xffffff00` and `0xffffffff`, inclusive, but where some sequence numbers is strictly below `0xffffffff`.

Based on a sample of $261,003$ transactions between August 17 and 18, 2020, we find that

- 20% of transactions had the flag present.
- 76% of transaction had a lock time of `0x00000000`.
- 68% of inputs were in transactions where all sequence numbers are `0xffffffff`.
- 31% of inputs were in transactions with a sequence number below `0xffffffff`, but where all sequence numbers are no less than `0xffffff00`.

If this sample is representative of the transactions in the blockchain, then given that there are now more than 560 million transactions and 1,380 million inputs (or spent outputs) [Bit], we can expect to save at least

$$560\,(3 + 2 \times 0.20 + 4 \times 0.76) + 1,380\,(4 \times 0.68 + 3 \times 0.31) = 8,643.4$$

8,643.4 million bytes, or about 8.05 GB.

2.5. **Values.** Each output in a transaction includes a "value" that represents the number of satoshis (100 millionths of a bitcoin) being transferred to the output address. This value is stored as an 8-byte signed integer, allowing up more than 9×10^{18} satoshis to be sent. However, Bitcoins protocols dictate how new coins enter the system and place a hard limit of 2.1×10^{15} satoshis. An 8-byte value allows one to send more satoshis than can ever exist. Of course, we also find that most of the values being sent are much smaller than even 2.1×10^{15} and the vast majority fit in a 4-byte unsigned integer. Hence, one approach to compressing the blockchain is to put these values in a data structure whose size is adaptable as needed.

We queried $54,947,133$ outputs from the Bitcoin blockchain to gather statistics on the sizes of values being sent. We found the following.

1. 45,402,811 values (82.6%) are larger than 2 bytes.
2. 23,631,735 values (43.0%) are larger than 3 bytes.
3. 2,600,279 values (4.7%) are larger than 4 bytes.
4. 18,532 values (0.03%) are larger than 5 bytes.
5. No values are larger than 6 bytes.

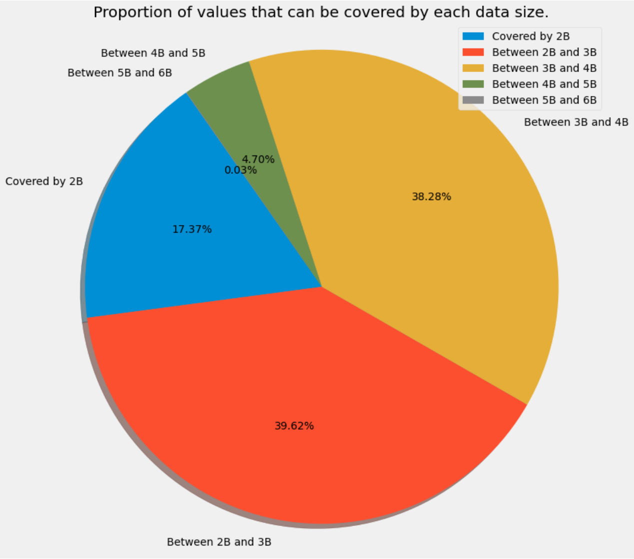

Figure 3. Proportion of values that can be covered by each data size.

We can see that over 95% of the values can be covered by 4-byte unsigned integers. Even for 3 bytes, 57% of the data can be covered. We propose the following strategies:

(1) Encode the values with 2 kinds of data types: 4 bytes and 8 bytes unsigned integers. In this case, we need one extra bit as flags for decompression indication.
(2) Encode the values with 4 kinds of data types: 3 bytes, 4 bytes, 6 bytes and 8 bytes unsigned integers. In this case, we need two extra bits as flags for decompression indication.
(3) Encode the values with 4 kinds of data types: 2 bytes, 3 bytes, 4 bytes and 8 bytes unsigned integers. In this case, we need two extra bits as flags for decompression indication.
(4) Encode the values with 8 kinds of data types: 1 byte, 2 bytes, . . ., 8 bytes unsigned integers. In this case, we need three extra bits as flags for decompression indication.

Assuming this sample is representative of the transactions in the blockchain, then given that there are approximately 1.5×10^9 total outputs [Bit], we can expect to save

- about 5.5 GB of data if we use method 1;
- about 6.5 GB of data if we use method 2;
- about 6.4 GB of data if we use method 3; or
- about 6.4 GB of data if we use method 4.

It seems that method 2 is the best among these approaches, which could give us an approximate 2% compression rate, based on the fact that the size of the current Bitcoin blockchain is about 300 GB.

3. Implementation

We have created a toy implementation of some of the compression schemes descrived in § 2, available at `http://github.com/emmacneil/btcompress`. The blockchain, when downloaded, is split up into multiple .dat files, approximately 128 MB each. Our toy implementation compresses and decompresses a single .dat file at a time. It implements compression of transaction hashes, version numbers, flags, lock times, sequence numbers, and values as outlined in § 2.2, § 2.4, and § 2.5. Due to time constraints, we did not implement compression of addresses and scripts as in § 2.1 and § 2.3.

We were able to compress the 128 MB file down to 117 MB, 91.4% of its original size. Of course, by implementing the rest of the methods described in this paper, we would bring that figure lower. We note also that the results are dependent on the choice of .dat file. Files corresponding to early blocks in the blockchain have proportionately fewer transactions and would yield worse results. Conversely, by compressing a single file at a time, we cannot take full advantage of the repetition of hashed values. By compressing multiple files at a time, we could achieve better results.

4. Conclusion

In § 2, we analyzed the repeated or redundant patterns in the Bitcoin blockchain. We then discussed approaches to compression based on those patterns. The compression rates we can get from each of those approaches are summarized in Table 6. If applied to to the whole blockchain, we expect a savings of about 58.1 GB, a compression rate of approximately 19.6%. In section § 3, we implement a few of the methods from § 2 to a portion of the blockchain and achieve a compression rate of 8.6%.

Patterns	Sections	Compression Amount	Compression Rate*
Indexing Repeated Addresses	2.1	21.6 GB	7.3%
Indexing Repeated Transaction Hashes	2.2	16.0 GB	5.4%
Indexing Scripts	2.3	5.9 GB	2.0%
Version Number, Flag, Lock Time, and Sequence Numbers	2.4	8.1 GB	2.7%
Compressing Redundant Values Data Type	2.5	6.5 GB	2.2%
Overall		58.1 **GB**	19.6%

* Compression rate is obtained as dividing the compression amount by the total block size 295 GB by the time of Aug 25, 2020 [Blo].

Table 6. Separated Compression Efficiency Achieved by Each Approach

Our methods of compression are applicable not only to the Bitcoin blockchain, but also other cryptocurrencies such as the Divi Project where transaction data includes repeated values and values are represented by more space than needed. Our methods may also be applied to data analyses that do not require the entire blockchain data. We consider the case of an analyst who wishes to study the graph structure of the blockchain's transactions and may be interested in storing transaction hashes and addresses of senders and receivers, but not scripts and version numbers. Applying some of our compression methods may mean the difference between their data fitting in RAM or not, which would greatly speed up the analysis.

5. Future Work

There are still more simple analyses that can be done. The witness section of a transaction may contain compressible data, but we have not explored this, due in part to not understanding what data was contained within until late in the project. Indeed, the witness section contains addresses and hashes much like input script signatures.

An output script typically contains a hashed value. An input that spends it has in its script the unhashed value. The former can be derived from the latter, meaning that we do not need to store both. A full analysis of how much can be saved has not been done.

In § 2.3, we focused on compressing output scripts. A P2SH output script may correspond to an input whose script, in turn, contains a P2PKH script. That is, for some input scripts, the methods of § 2.3 may be applicable.

Some output scripts may be classified as OP_RETURN scripts. These outputs are unspendable. It is possible to mark some bitcoins as permanently unspendable, a way of burning digital money. However, an OP_RETURN script usually comes with a value of 0 BTC, while the script itself contains metadata written in plain English, Chinese, or some other language. Natural plain text can be compressed effectively using variable-length encoding algorithms such as Huffman Coding, another avenue for compressing the blockchain. However, OP_RETURN scripts are only a small minority of outputs (about 3%), and we expect such compression to save hundreds of megabytes out of 295 GB.

We have focused primarily on exploiting the particular structure of Bitcoin's file format to achieve compression. We have proposed, among other things, to place frequently reused values such as transaction hashes and addresses into lookup tables. These tables, in turn, may be amenable to further compression by more general compression algorithms. This returns us to one of the original questions posed to us: "What compression ratio can we achieve for an ordered sequence of cryptographic hashes?" One could analyze how different generic compression algorithms perform on a sequence of hashes and whether reordering the hashes impacts the results.

Finally, even if we could achieve a compression ratio of 50%, the compressed block chain would still be over 100 GB in size, too large for small devices with storage space constraints. One might ask whether a cryptocurrency can be designed that allows even small devices to participate as full nodes on the network, requiring a blockchain on the order of at most a few gigabytes. We wonder whether a new cryptocurrency model can be defined whereby individual nodes can independently verify the validity of transactions without storing the entire history of transactions or depending on third parties. Perhaps an imaginative application of concepts such as zero-knowledge proofs, cryptographic set membership testing, homomorphic encryption, and Bloom filters would permit such a scheme.

Acknowledgments

The authors would like to thank our industry mentor, Germàn Luna, from the Divi Project for his guidance, support, and expertise. We'd also like to thank our academic mentor, Cuneyt Akcora, from the University of Manitoba for his helpful insight and advice. Finally, we'd like to thank Mitacs and the M2PI team and especially Kristine Bauer for organizing this amazing opportunity.

References

[Bit] Bitcoin transaction output statistics. https://bitaps.com/statistic/outputs. Accessed: 2020-08-25.

[Blo] Blockchain size. https://www.blockchain.com/charts/blocks-size. Accessed: 2020-08-23.

[Div] Divi project. https://diviproject.org.

[LLW] Eric Lombrozo, Johnson Lau, and Pieter Wuille. Segregated witness (consensus layer). https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki. Accessed: 2020-09-03.

[Mer80] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy,1980*, pages 122–134. IEEE Computer Society, 1980.

[Nak08] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[Tot] Total number of transactions. https://www.blockchain.com/charts/n-transactions-total. Accessed: 2020-08-23.

[TS16] F. Tschorsch and B. Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys Tutorials*, 18(3):2084–2123, 2016.

E-mail address: iplau@sfu.ca

E-mail address: shang.li1@ucalgary.ca

E-mail address: macneil.evan@ucalgary.ca

E-mail address: amcsw087@uottawa.ca

E-mail address: shukla2@ualberta.ca

E-mail address: yanhong.xu1@ucalgary.ca