SoK: Compression in Rollups

Roshan Palakkal
Quantstamp, Inc.
roshan@quantstamp.com

Jan Gorzny
Quantstamp, Inc.
jan@quantstamp.com

Martin Derka
Quantstamp, Inc.
martin@quantstamp.com

***Abstract*—A *rollup* is a scaling solution built on top of an existing blockchain. Rollups separate execution from consensus, but are required to post the data used for state updates to the underlying blockchain. This data is required to ensure that execution of state updates are performed correctly. As writing data to a public blockchain is not free, rollups are incentivized to minimize the amount of data they post on-chain. Rollups therefore aggregate and compress the data used for executions in order to save on fees associated with writing data to the blockchain. In this work, we explore the methods for posting data on-chain and the compression techniques used by real-world rollups. We explore differences in implementations and contrast the approaches used by both optimistic and zero-knowledge rollups. We also explore approaches which enable domain-specific compression, consider upcoming changes to data storage on Ethereum, and suggest improvements for rollup compression.**

I. Introduction

Modern blockchains like Ethereum [1] support transactions that may move digital assets or make calls to programs that reside on the network (a.k.a. *smart contracts* [2], which might implement other digital assets like ERC-20 tokens [3]). However, blockchains like Ethereum suffer from limited throughput of transactions; Ethereum itself is currently only capable of somewhere between 30 and 40 transactions per second [4]. This limitation has motivated the study of various scalability solutions [5].

One such approach is the use of a *side-chain*, which is a blockchain that operates parallel to a main blockchain network. Users can use cross-chain communication protocols (a.k.a. *bridges*) that send messages between the chains to unlock digital assets or their representations and make cross-chain calls to smart contracts on either end [6]. However, these approaches do not inherit much security from the main blockchain: in essence, the side-chain is free to do what it wants as it is its own network. An ideal scaling solution improves the throughput of the network but does not introduce new sources of risk.

Therefore, scaling solutions that are governed by the underlying blockchain are desirable. Such solutions are called "layer 2" solutions as they build entirely on top of a main "layer 1" blockchain, like Ethereum [5]. These layer 2 protocols allow transacting parties to utilize a high-throughput blockchain ecosystem and fall back to the underlying layer 1 if there is a need to dispute behavior on layer 2 (e.g., with respect to their cryptocurrency balances), interact with decentralized applications on layer 1, or transact with accounts on other layer 2 solutions. Throughout this work, we will focus on layer 1 blockchains like Ethereum that support smart contracts, as smart contracts are required to implement these solutions.

Several layer 2 solutions have been studied, and each offer interesting tradeoffs between various concerns like interoperability, liquidity, security, and privacy (see e.g., [5]). Bitcoin [7] layer 2 networks favor payment channel based solutions [8], [9], [10], [11] as they are easy to implement on that network. The "Plasma" [12] family of protocols were initially popular on Ethereum, but have since been largely replaced by so-called "rollups" (also called *commit-chains* [13] or *validating bridges* [14]). This is in part because rollups overcome data availability issues with Plasma (see e.g., [15], [16]), and more easily support smart contracts.

Rollups require transaction data to be posted on the underlying layer 1. The posted data can be used to initiate transactions (e.g., withdrawals) on the underlying blockchain. However, this also means that actors watching the published data on the underlying layer 1 can detect fraudulent blocks. How fraudulent blocks are handled depends on the type of rollup. Rollups come in two main types: *optimistic* and *zero-knowledge* (ZK). Optimistic rollups post data on-chain and watchers can challenge invalid data within some period using the underlying layer 1. ZK rollups post cryptographic proofs that their state updates are valid, which are verified by smart contracts on the underlying layer 1 directly. Rollups get their names from the fact that they can batch, or *roll up*, transaction data into a single message posted onto the underlying chain. Section II introduces the concept of a rollup more formally, including important components.

Batching is important to reduce costs for the system. Posting data on-chain, even temporarily, costs *gas*. Gas is the resource which is metered by smart contracts to limit executions, and gas in turn costs money when it is paid for using a cryptocurrency like Ether (the native asset on Ethereum). Oversimplifying, the more data that is posted, the higher cost. Instead of posting a state root (the resulting state of the rollup; defined in Section II) for every transaction, a batch of transactions is executed, and only the final state root is posted. This reduces the number of state roots posted on-chain.

Recall that rollups also need to post the transaction data necessary to verify that the state root is correct. Techniques like compression can be used to reduce the amount of transaction data posted. Note that state roots are a (small) fixed size and stored on-chain in a fixed type, so there is little or no benefit in compressing them. Posting less transaction data means that the cost of operating the rollup decreases. As the cost of posting layer 2 data may be shared among users of the rollup, this results in a better user experience on rollups.

In this work, we study the use of compression[1] within rollups. The variety of rollup implementations means that various pieces of data are compressed in various ways, depending on the implementation. This functionality is also not necessarily well documented. However, even when the use of compression is well documented, users may not have any insight as to why particular approaches are used over others. We illustrate the state-of-the-art in rollup compression and justify why some approaches are useful by answering the following general research questions.

Q1 What compression algorithms are used in rollups? (Section III-A)

Q2 Do rollups use customized techniques to compress or reduce data that are specific to this domain? Do these techniques change if data is of a particular kind or if the data is uniform? (Section III-B)

Q3 How and where are compression algorithms used in rollups? For example, some rollups appear to compress batches of transactions only; do others batch compressed transactions? (Section III-C)

Q4 What does it mean for one compression algorithm to be better than another in this particular domain? (Section III-D)

We answer these questions using real-world examples. We will also evaluate existing approaches to validate claims by rollup developers about their compression.

Next, we consider compression in ZK rollups. These systems may introduce delays to data posting (e.g., they may wait for a proof to be generated), or may need to directly handle compressed data within the ZK proof engines; as a result, they may have different requirements for compression. We therefore also ask the following research question.

Q5 Do the required properties of compression algorithms differ for ZK rollups and optimistic rollups? (Section IV)

Lastly, we consider two questions related to the cost of data on rollups and the future of data availability for rollups on Ethereum. The first explicates to end-users how they are charged for rollup transactions, while the second discusses how compression may change in light of new methods for storing data on Ethereum via EIP-4844 [17] (a.k.a. *proto-danksharding* [18]).

Q6 How much should one charge users for the data of their rollup transaction? What schemes are possible, and what schemes are in use? (Section V-A)

Q7 How will the use of compression change in light of upcoming developments on Ethereum related to data availability? (Section V-B)

Finally, we conclude in Section VII, which also summarizes improvements that may be possible given the answers to these questions.

[1]Every effort was made to be accurate during the time of study, which was from April 2023 to June 2023. However, rollups develop quickly and as a result, this document may not be entirely accurate at the time it is read.

II. Preliminaries

We first introduce rollups and then introduce various concepts related to the data that they make available.

A. Rollups

We start by introducing rollups. A rollup can be broken down into several components[2]: a *sequencer*, *state proposer*, and *verifier*. A *sequencer* is responsible for ordering layer 2 transactions. Often a rollup only has a single sequencer, though a decentralized sequencer may be desirable; for simplicity, we will assume there is a single sequencer. Typically the sequencer publishes the data for layer 2 transactions onto layer 1. Possibly separate from the sequencer, the rollup will have an *state proposer*, responsible for executing the transactions on layer 2, creating layer 2 blocks and posting the resulting state of the network to layer 1. This component executes the transactions ordered by the sequencer. Finally, a rollup has a *verifier*, which is responsible for checking (in)correctness of layer 2 state updates.

The *verifier* differs depending on the type of rollup. There are two main types of rollups: *optimistic* and *zero-knowledge* (ZK) rollups.

In an optimistic rollup, the state proposer is bonded (i.e., it stakes some funds) in order to incentivize honest behaviour. Other users watching the posted data can verify that the data published is correct, and if it is not, they can challenge the data on layer 1. Such a challenge requires a *fraud proof* (a.k.a. *fault proof*) which is a trace of the computation in the block that differs from the one posted by the state proposer. A successful challenge is rewarded with the state proposer's bond, and invalidates the state update. Actors who provide these challenges are verifiers in these rollups, because unchallenged state is assumed to be correct after some time. Often, verifiers for an optimistic rollup suggest a new state via these proofs; in such cases they are both state proposers and verifiers, and we call them *validators*.

For ZK rollups, state proposers post *validity proofs* to layer 1 alongside the state they propose. In a ZK rollup, the state proposer performs state transitions within a zero-knowledge proof framework (e.g., [20]) which generates the validity proof: an artifact that proves that a particular function was executed with particular inputs which resulted in the new state. These validity proofs are checked using layer 1 smart contracts; the layer 1 is therefore the implicit verifier in these systems. ZK rollups may also have a (separate) *prover* component to generate proofs that the state proposer correctly computed the state update.

Both optimistic and ZK rollups are operating or in development on the Ethereum blockchain. Arbitrum is one example of an optimistic rollup [21], while zkSync is an example of a ZK rollup [22]. We note that the *zero-knowledge* aspect of ZK rollups is sometimes helpful for privacy, but mostly these systems are used because the proofs are also *succinct*. That is,

[2]Other work like [19], [14] use different terms for these components, but each rollup has some component that performs these actions.

the proofs can be verified in a fraction of the time required to run the computation in the first place, enabling efficient verification directly on a layer 1 blockchain.

The source of layer 2 transactions may be the layer 1 smart contracts of the rollup (as in a deposit transaction) or a user of the rollup (as in a layer 2-to-layer 2 transaction or a withdrawal transaction). The sequencer of the rollup and the smart contracts implementing the rollup are involved in any transaction (except possibly a forced withdrawal or so-called *escape hatch*; see e.g. [19]).

Rollups therefore involve *cross-chain* communication, from layer 1 to layer 2, and vice versa. A *bridge* is a system or protocol for taking assets or blockchain state from one blockchain to another [6]. As a result, rollups may be considered to be a bridge (as in [14]) or merely rely on one developed alongside the network infrastructure.

For simplicity and concreteness, we will assume our source blockchain and layer 1 is Ethereum throughout this work. Without loss of generality, this work should apply to other blockchains, as well as to layer 3 rollups built on layer 2 rollups with similar features, and beyond.

B. Data Availability

Rollups publish data in order to benefit from the security of the underlying blockchain. Although transactions are executed off-chain, rollups post the minimum amount of data required to locally verify rollups transactions. This enables anyone to to detect fraud, and as such, only one honest party watching the data is necessary to ensure security.

The state of the layer 2 chain is is updated as transactions are processed using transaction data. Transaction data is an encoding of a valid layer 2 transaction; often this data looks like layer 1 transactions.

A transaction is executed on the Ethereum Virtual Machine (EVM) given the current *state root* for the rollup chain, which results in another state root after the execution. The state root of the chain is the root of a state trie [23] for the blockchain, typically a Merkle Patricia Trie (MPT). A MPT is a combination of Merkle Trees (see e.g., [24]) and Patricia Trees (see e.g., [25]) [26]. The state trie contains the following data for every account: nonce (transaction count), balance (in Ether), code and storage (if the account contains a smart contract), and anything else that may be relevant to record the rollup state [27]; the root of the trie provides a signature of the state at a given point in time.

It is not necessary to explicitly record the entire state everytime a layer 2 block is produced, assuming knowledge of the previous state is known. In such a case, only some data is necessary is compute a new state root, and this is the data that needs to be published onto the underlying layer 1. As a concrete example, according to the Privacy and Scaling Explorations (PSE) team's specifications to build a ZK implementation of the EVM [28], the following data is required to reconstruct the state trie, assuming the state of the previous knowledge is known. Each field is also presented with its size.

- For each transaction, the following data is required:
  - `GasPrice` and `Value`: 256 bits each
  - `Gas`: 64 bits
  - `CallerAddress` and `CalleeAddress`: 160 bits each
  - `CallData`: `CallDataLength` number of bytes
- Block fields that affect EVM execution:
  - `Coinbase`: 160 bits
  - `Difficulty` and `BaseFee`: 256 bits each
  - `Number`, `GasLimit`, and `Time`: 64 bits each
- Extra fields that affect EVM execution:
  - `block[-1..-257].Hash`: 256 bits; the hash is computed using the `Keccak 256` algorithm [29].
  - `ChainID`: 64 bits

Note that the transaction signature is not needed to synchronize the state trie as the signature itself is not stored in the state; it is verified at block creation and stored in the block data. As noted by both the PSE team [28] and Buterin [27], on a rollup the transaction nonce can be inferred from the state trie of the previous block, and so it is unnecessary. Account specific values like the `balance` or code deployed at the address (`codeHash`) can be inferred from a transaction and knowledge of the previous state. Note that not all rollups necessarily follow this specification, and the specific data stored in a state trie may differ. Some ZK rollups call this data a *state diff* or *state delta* (e.g., zkSync Era [22], [30]) though the specific fields may differ for each rollup.

The state root should be stored on-chain so that there is consensus on what the recent state of the chain is. Moreover, if a rollup operator needs to be challenged because they are posting incorrect state roots (as might happen in an optimistic rollup), the state root used in the challenge is available on-chain and the data that was allegedly used to create it was also published on the blockchain.

The data used to recreate the state root may be published to underlying blockchain as `calldata`. This data is included in the transaction state trie, which is included in Ethereum blocks [31]. It is not accessible by the blockchain except for when it is used; for example, smart contracts cannot look up or reference that data as part of subsequent transactions. As the `calldata` is included in the blockchain as part of a Merkle Patricia trie, it may not be easily readable and may need to be decoded. Note that it is not necessary to record every state root update; all intermediate updates can be reconstructed via the published `calldata` and a suitable state root used as a starting point. In the case of a ZK rollup, the proof that a state root is correct may also be posted to the layer 1 as `calldata`, as long as it is verified when it is used to update the state root on layer 1.

However, even posting `calldata` costs money. As a result, some rollup operators choose to post this data off-chain. This is particularly effective for ZK rollups, who can post a commitment to the data (e.g., a hash of the data) to the layer 1. Matching off-chain data with the commitment, as well as the validity proof that uses the commitment, enables end-users to be convinced of correct execution. Such a ZK rollup is often called a *validium* to differentiate it from one that posts all necessary data on-chain. Some similar ZK scaling solutions let users decide where to the put their data: either on-chain (as in a ZK rollup) or off-chain (as in a validium); such a system may be called a *volition*. Other rollups still use decentralized storage solutions or post different data all together. These topics are explored in Section III-C.

C. Data Set

We collected (pre-Bedrock) Optimism batches posted on Ethereum from (Ethereum) block 16250000 (Dec 23, 2022) to (Ethereum) block 16350000 (Jan 6, 2023). The data footnoteThe data we used is available at https://github.com/quantstamp/l2-compression-data, and the scripts used to analyze the data can be found at https://github.com/quantstamp/l2-compression. contains 5.3 GB of batches stored as hexadecimal strings. Note that the repository may be much larger as it contains metadata and formatting characters (tabs and newline characters).

The data contained 42,888 batches which were decompressed; a decompressed batch consists of a list of transactions. The data had a total of 7,139,544 transactions. A sample batch is too large to put in this document (about 157 KB as a hexadecimal string), but a sample transaction contained within a batch is presented in Figure 1. All evaluation for this work was done on an Amazon AWS `t2.large` instance, a general purpose EC2 instance with 2 vCPUs and 8.0 GiB of memory.

III. Rollup Compression Survey

In this section, we survey compression algorithms and techniques used for rollups. Some rollups do not yet implement compression, while others are more mature and utilize some form of it. We reference existing implementations when possible, in order to bring differences to light and to guide our survey. These reference implementations allow others to understand the existing technology when considering new methods or building new rollups. Table I lists several rollups and their compression algorithms, which will be discussed the following subsections. Introductions to the algorithms discussed in this work are provided in Section III-A. Domain-specific heuristics are described in Section III-B. Rollups which apply compression in non-standard ways (e.g., combinations of approaches) are discussed in detail as part of Section III-C. Compression algorithms are applied to real-world data and evaluated in Section III-D.

A. Compression Algorithms

In this section, we review the compression algorithms used explicitly by the rollups in Table I. There are three common compression algorithms in Table I: Brotli [59], [60], zlib [61], and Zstandard ("zstd") [62]. Note that Brotli is a combination of (a variant of) the LZ77 algorithm [63], Huffman coding [64] and second order context modeling (see e.g., [65]). Similarly zlib only supports the `DEFLATE` algorithm [66], which is also a combination of (a variant of) the LZ77 algorithm along with

Rollup	Kind & DA	Compression Algorithm	Built On
ApeX [32]	ZK	*None*	StarkEx
Arbitrum Nova[a] [33]	O	Brotli	
Arbitrum One [34]	O ✓	*None*	
Aztec [35]	ZK ✓	*None*[*]	
Aztec Connect [36]	ZK✓	*None*	
Base [37]	O ✓	(zstd)	Bedrock
Boba Network [38]	O✓	(zlib)	Optimism
Boba Anchorage [39]	O✓	(zstd)	Bedrock[b]
Canvas Connect [40]	ZK	*None*	StarkEx
DeGate [41]	ZK ✓	LZ77[*]	Loopring
dYdX [42]	ZK ✓	*None*	StarkEx
Fuel (v1) [43]	O✓	None	
ImmutableX [44]	ZK	*None*	StarkEx
Layer2.Finance (Celer) [45]	O✓	*None*	
Loopring [46]	ZK✓	LZ77[*]	
Metis Andromeda[a] [47]	O	(zlib)[*]	Optimism
Myria [48]	ZK	*None*	StarkEx
Optimism [49]	O ✓	zlib	
Optimism Bedrock [49]	O ✓	zstd	
rhino.fi [50]	ZK	*None*	StarkEx
Polygon zkEVM [51] (Hermez)	ZK✓	*None*	
Scroll [52]	ZK	*None*	
Sorare [53]	ZK	*None*	StarkEx
StarkEx[c] [54]	ZK	*None*	
StarkNet [55]	ZK ✓	*None*	
ZKSpace [56]	ZK✓	*None*	
zkSync Era [22], [30]	ZK ✓	*None*[*]	
zkSync Lite [22], [57]	ZK ✓	*None*[*]	

[a]L2Beat.com lists these as "chains" rather than rollups; indicating that they do not partially or fully derive their state from the underlying layer 1.
[b]Custom implementation conforming to the Bedrock specification only [39].
[c]Not a standalone rollup; used to build other entries.

TABLE I: Compression algorithms for various layer 2 solutions, built on the list available at L2Beat.com [58] as of May 2023. A "ZK" in the second column indicates that the rollup is a zero-knowledger rollup, while an "O" indicates it is an optimistic rollup. We also indicate if a rollup stores data directly on chain via a ✓ in the second column. The third column indicates the main compression algorithm used when publishing data, if any. The last column indicates if a rollup was developed based on an different codebase: "StarkEx" ones are built using StarkEx code, "Optimism" ones use (pre-Bedrock) Optimism code, and "Bedrock" rollups are built on the Bedrock version of Optimism (later renamed to OP Mainnet). DeGate is built on the Loopring codebase. Algorithms in parenthesis are used as a result of the codebase the rollup is built on; for example, Base uses zstd because Optimism Bedrock does, and Base is built on that codebase.

```
"raw": "0xf9012e45830f42408305d468948add31bc901214a37f3bb676cb90ad62b24fd9a586b5e620f
    48000b8c48a182319000000000000000000000000000000000000000000000000000000000000010
    0000000000000000000000000000000000000000000000000000000b8c63f000000000000000000000
    000000000000000000000000000000000000005f5e1000000000000000000000000000000000000000
    0000000000000000000000000010000000000000000000000000000000000000000000000000000001
    c63c169000000000000000000000000000000000000000000000000000000000000004e2037a0652a2
    31e64666e9d07301ae326a005e144c0c7a07712e9be01cd8cff99c7f3bea02b37658f8cc5134fcb409
    6fefafc479eb620cc42c7c0a7b9eaedf67ee2a0ed79",
"nonce": 69,
"gasPrice": {
    "type": "BigNumber",
    "hex": "0x0f4240"
},
"gasLimit": {
    "type": "BigNumber",
    "hex": "0x05d468"
},
"to": "0x8adD31BC901214A37f3bb676Cb90AD62B24fd9a5",
"value": {
    "type": "BigNumber",
    "hex": "0xb5e620f48000"
},
"data": "0x8a182319000000000000000000000000000000000000000000000000000000000000000100
    000000000000000000000000000000000000000000000000000000b8c63f0000000000000000000000
    00000000000000000000000000000000000005f5e10000000000000000000000000000000000000000
    00000000000000000000000001000000000000000000000000000000000000000000000000000000001c
    63c16900000000000000000000000000000000000000000000000000000000000004e20",
"chainId": 10,
"v": 55,
"r": "0x652a231e64666e9d07301ae326a005e144c0c7a07712e9be01cd8cff99c7f3be",
"s": "0x2b37658f8cc5134fcb4096fefafc479eb620cc42c7c0a7b9eaedf67ee2a0ed79",
"from": "0x180920D0613954cC0A6150600e2EAf55beDC6D78",
"hash": "0xdc4280dd4e263ad37a24a22034e02ff6a303b1a938196972a1677fa19a78ffc7",
"type": null
```

Fig. 1: A sample Optimism transaction, posted to Ethereum. The `raw` field is posted on-chain, and some other fields (`gasPrice`, `gasLimit`, `value`, `data`, `r`, and `s`) are found within this field. The types `BigNumber` are a result of using the Web3 library to print the transaction; the data type used when executing the transaction is known and not transmitted. A batch contains many transactions.

Huffman coding. A custom version of LZ77 is also used by Loopring. Table I and the discussion above answer **Q1**.

In the remainder of this section, we first provide high level descriptions of both the LZ77 algorithm and Huffman coding, as these are common to many of the approaches implemented. Then we note other compression algorithms that may be particularly well suited to rollup data.

Some of the algorithms described in this section (e.g., Brotli, zlib, and zstd) use *dictionary encoders* in their algorithm design. A dictionary encoder is one where the compression algorithm creates a list, or dictionary, of strings and occurrences of those strings in the to-be compressed text are replaced with references to the string inside the dictionary. In some cases, the dictionary may be shared across executions; in other cases, the dictionary is created depending on the particular input string. For example, zstd is a dictionary-based approach for compression developed by Collet and Kucherawy [62] that also uses LZ77. The zstd algorithm uses Huffman coding and a method called "finite-state entropy" coding based on so-called asymmetric numeral systems [67].

1) LZ77 Compression: LZ77 compresses a string by replacing repeated substrings with references to their first occurrence. The input string is traversed character by character starting at the beginning, and a sliding window is maintained, storing a fixed amount of previously traversed (consecutive) characters. Whenever a repeated substring of consecutive characters is found, the repetition stored as a pointer to the location of the previous occurrence of the substring in the sliding window and the number of repeated characters in the symbol. For example, if the input string is `Ether Ethereum` and the buffer was suitably long (say, 11 characters), the result of the LZ77 would be something like `Ether <6,5>eum`, where `<6,5>` is the encoding of a pointer. This pointer encoding says to replace the pointer with the length 5 substring that starts 6 characters before the pointer in the text. The actual implementation details of the pointer are more complicated, and we refer the readers to the source of the algorithm for a complete description. LZ77 has an $O(n)$ run-time, where n is the size of the input string, provided the sliding window has a fixed size independent of n.

2) Huffman Coding: Huffman coding first finds the frequency of all characters in a string (i.e., symbols), and then based on each characters frequency, assigns a binary code to it. The highest frequency character gets the shortest binary code, and the lowest frequency character gets the longest binary code. Then, the string is replaced by the encoding of the characters. For example, if the input string is `aba`, which is 3 characters at 7 bits each using the original ASCII encoding [68]; 21 bits in total. The Huffman encoding of just this string would encode `a` as 1 and `b` as 0, and represent the string as 1 0 1, which is only 3 bits long.

It is important that the codes for each symbol are chosen so that a resulting code is never a prefix of any other symbol ([69], [70] provide example methods to compute such prefix-free codes), and this is achieved by building a *prefix tree* from the input based on the frequency of the characters seen. The codes generated by the prefix tree on a particular input string are optimal in the sense that no other code performs better. Generating the codes via the prefix tree method time $O(k \log k)$, where k is the size of the alphabet, given an input string and the frequencies of each element in the string.

Note that one may need to also transmit the prefix tree, but if it is re-used, this does not need to be transmitted alongside every compressed string. However, the resulting codes for a re-used prefix tree may not be optimal since the probability distribution across all input strings was not known when the tree was constructed.

3) Run Length Encoding (RLE) & Zero Length Encoding (ZLE): These related algorithms are based on techniques that have been around since the 1960s [71]. RLE performs the following operation: for a string, replace every consecutive substring of the same character with the length of the substring followed by the unique character within the substring. For example, the string `aabcccd` would would turn into `2a1b3c1d`. It is not hard to see that RLE may not always compress the string: consider `abcd`; after applying the RLE transform, the resulting string is `1a1b1c1d`. One can restrict RLE to only encode substrings consisting of the `0` character without changing any others, and this change results in the ZLE transformation. These algorithms can be implemented to run in time $O(n)$ where n is the size of the input string.

We explore the use of ZLE in Tables II and III. Table II illustrates the distribution of the *largest* run of zeroes in a batch. Note that there are no batches with no length-one run

Largest Run of Zeros	Count	Percent
1	285592	2.20%
2-5	767323	5.91%
5-25	2376701	18.31%
25-50	967724	7.46%
50-75	5448702	41.98%
75-100	143356	1.10%
100-150	1778170	13.70%
150-200	666255	5.13%
200-500	529623	4.08%
500-1000	17174	0.13%
1000-10000	9	0.00%

TABLE II: Analysing the effectiveness of ZLE by counting the number of batches with non-trivial runs of zeros.

Number of Zeros	Count	Percent
0-2	330096024	74.51%
2-5	25144992	5.68%
5-25	21752536	4.91%
25-50	10059366	2.27%
50-75	48162452	10.87%
75-100	1383302	0.31%
100-150	4781704	1.08%
150-200	869192	0.20%
200-500	738650	0.17%
500-1000	22074	0.00%
1000-10000	38	0.00%

TABLE III: Analysing the effectiveness of ZLE by counting the number of non-trivial runs of zeros across all batches.

of zeroes. For example, 967724 batches have a largest run of zeroes which is somewhere between 25 and 50 zeroes long. This table illustrates how often applying ZLE to a batch makes sense: the majority of batches (91.89%) include large runs of zeros. Table III illustrates the distribution of the size of runs of zeroes across all runs of zeros in the batch data. This table illustrates the how often the use of ZLE will have impact: more than 25% of all runs of zero have more than two zeroes.

4) Burrows Wheeler Transform (BWT): The Burrows-Wheeler Transform (BWT) [72] is used in the Bzip2 compression algorithm [73]. The BWT aims to rearrange a string into runs of similar characters. For example, the string `^BANANA|` (with "^" used to signify the start and "|" used to signify the end) will be transformed into "BNN^AA|A". The process is reversible, but requires the explicit indication of which character starts the string. Runs of similar characters are then suitable for RLE and ZLE based encoding (Section III-A3). The BWT be executed in time $O(n)$ where n is the size of the input string.

5) Move to Front Transform (MTF): This transform [74] replaces a string's character with indices in an alphabet string. The alphabet string is initially provided, but the last character considered in the string is always moved to the front of the last alphabet string, producing another one. As an example, consider the string `bca` with the (initial) alphabet string `abc`. First, the "`b`" is replaced with with its index in the alphabet string, 1, and then `b` is moved to the front of the alphabet string. This results in a new string `1ca` and a new alphabet string `bac`. The final alphabet string is necessary to reverse the transformation. Assuming the length of the input string is much larger than the size of the alphabet string, this does not add a lot of overhead. If there are many similar characters close to each other, final output results in many zero indices. This transform is often used alongside other algorithms, like the BWT (Section III-A4). The MTF transform can be executed in time $O(k \cdot n)$ where n is the size of the input string and k is the size of the alphabet.

B. Domain Specific Heuristics

In this section, we study ways to reduce the cost of publishing rollup data that are specific to these systems. These include optimizations and heuristics that developers can use in order to reduce necessary rollup data further. These heuristics are collected from various documents and real-world implementations. They answer question **Q2** in the affirmative: there are special considerations for rollup data.

1) Indexing Addresses [27]: A rollup can track which addresses have balances or a contract deployed. Buterin suggests that by giving an arbitrary ordering to the addresses which have some non-zero state, one can replace the `CallerAddress` and `CalleeAddress` fields of transactions, each 160 bits long, with indices to an `address` [27]. One natural ordering is when addresses are added to the state root for the rollup.

An `address` on Ethereum is typically 20 bytes (40 hexadecimal characters), but takes 21 bytes to store on Ethereum due to the Recursive-Length prefix (RLP) [75], [1] encoding used by Ethereum. RLP is a way to standardize data transfers on Ethereum, by encoding arbitrarily nested arrays of binary data. RLP adds the `0x80` prefix to data that are 0-55 bytes long (other prefixes are used for data of different sizes).

With this approach, one can reduce the data for these fields from 20 bytes (address description) $+1$ byte (RLP overhead) to $2 \cdot k$ bytes, for $k = \lceil \log_2(n) \rceil$ where there are at most n leaves of the state root. With just $k = 4$ bytes, you can represent $2^{4 \text{ bytes}} - 1 = 2^{4*8 \text{ bits per byte}} - 1 = 2^{32} - 1$ unique addresses using their index in a tree that stores all addresses active on a rollup. Thus you can reduce the size of the data to $2k = 8$ bytes in this example: k bytes for each of the `CallerAddress` and `CalleeAddress` fields. Note that one cannot directly recover the `CallerAddress` from the transaction signature, so it is necessary to record the index associated to the `CallerAddress`.

2) Scientific Notation [27]: The `value` field, which takes 256 bits to store on Ethereum, can be stored in scientific notation. Buterin suggests that transfers may only need 1-3 significant digits of precision [27].

3) Reducing Gas Data [27]: Buterin suggests that the `GasPrice` value, also stored on Ethereum as 256 bits, and the total gas value for a transaction, "could be limited to discrete values." Buterin considers consecutive powers of 2 and suggests that in this case, even 8 bits could be enough to capture most gas cost uses.

4) Inverse Huffman coding [76]: One can apply so-called *Inverse Huffman coding* to bias data to favor zero bytes. This is particularly helpful when a compression algorithm output appears random. As described by Felton [76], the coding "recodes the data so that it includes a larger fraction of bytes that are zero, and a correspondingly smaller fraction of bytes that have other values." Felton expects that this transformation increases the overall size by 18%, but reduces the gas cost by 3.5%, since the result has more (cheaper) zero bytes (until this is changed; see Section V-B).

5) Data Availability Committees: Both StarkEx and variants of Arbitrum use a so-called *Data Availability Committee* (DAC), another method to reduce the cost of posting rollup data. This is not a compression technique, but a method to avoid writing data to the underlying layer 1 by allowing some parties to attest that they have seen the data for a state transition for the corresponding rollup. The central concept is the same in both rollups and results in posting a signed hash, rather than the transaction data itself, to layer 1.

A rollup that uses a DAC for data availability maintains a committee of actors who are responsible for storing the data for a rollup off of its layer 1. At times, users can request the data for the rollup from the committee, e.g., if a rollup operator does not process a withdrawal transaction. At least two actors should be honest so that if all but one person provides data, there is still an honest actor present.

A sequencer in a rollup that uses a DAC can post data as normal, via `calldata` on the layer 1, or post a *Data Availability Certificate* (or *DACert*). This certificate should contain the hash of relevant layer 2 block data be digitally signed by a sufficient number of members of the committee, and should not be expired. Layer 1 smart contracts for the rollup can verify the signatures and accept the certificate if they are valid and the certificate is not expired. Any invalid certificates are discarded and ignored. Due to their construction, data availability certificates are much smaller to post on the underlying layer 1 than all of the transaction data.

C. Approaches to Compression

In this section, we answer **Q3** by discussing where compression algorithms (Section III-A) and heuristics (Section III-B) are used within rollups.

There are at least two obvious ways to compress data posted by rollups:

A1 Apply compression to individual transaction strings within a batch and concatenate the resulting strings to use as the batch data.

A2 Concatenate individual transaction strings to form batch data and compress the results.

As we will show in Section III-D, when comparing applying a general purpose compression algorithm (any of the algo-

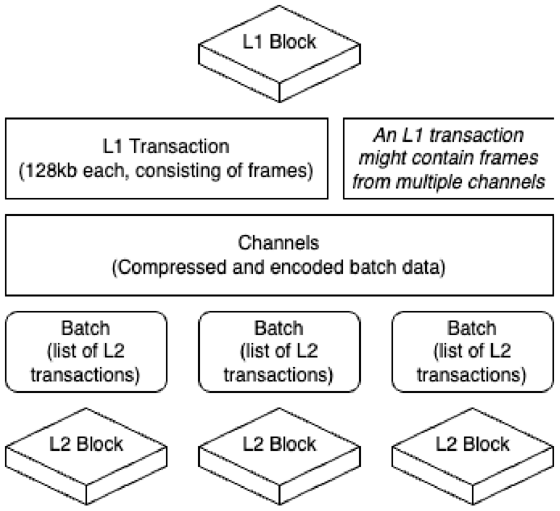
L1 Block
L1 Transaction (128kb each, consisting of frames)
An L1 transaction might contain frames from multiple channels
Channels (Compressed and encoded batch data)
Batch (list of L2 transactions)
Batch (list of L2 transactions)
Batch (list of L2 transactions)
L2 Block
L2 Block
L2 Block

Fig. 2: Simplified block derivation in Optimism Bedrock [77]. Note that compression is only applied to one level of the derivation process. Compression is applied after transactions are used to build layer 2 blocks, which in turn determine batches that are finally compressed as so-called channels. Compressed channels may be split into multiple layer 1 transactions to maximize layer 1 block space utilization.

rithms listed in Section III-A) the approach **A2** appears more successful in general than approach **A1**.

These two approaches disregard the heuristics described in Section III-B. Some of these heuristics can be applied at the transaction level. For example, the `value` field of a transaction is 256 bits, but its content may not require 256 bits of precision; it may be beneficial to apply ZLE & RLE (Section III-A3) to this field first. Combing these heuristics with the first approach described may yield further compression. In other cases, these heuristics may not be particularly helpful: the `CallerAddress` (160 bits) and `CalleeAddress` (160 bits) fields have high, hard-to-compress entropy as they are the result of hash functions. Regardless, the specific way data is compressed should be described by a rollup so that users can decompress the data as necessary.

We now consider the specific approaches taken by rollups outlined in Table I.

1) Optimism-Based Implementations: Optimism and Boba post data compressed by the zlib algorithm. Optimism evaluated the use of Brotli, zlib, and zstd as potential algorithms [78]. They determined that zlib and zstd were the most effective, and zlib was chosen over zstd for implementation convenience. Brotli was ruled out as too slow, and zstd was chosen for their Bedrock upgrade (Section III-C2). They use approach **A2** and note that this approach

> is better because users tend to interact with some contracts significantly more than other contracts. In addition, certain fields (like chain ID and gas price) tend to be similar across transactions. Compression algorithms rely on these similarities to do their job.

Although Metis Andromeda is Optimism-based, they have unique plans to use decentralized storage (Section III-C7).

2) Optimism Bedrock-Based Implementations: Optimisms's Bedrock version, along with the rollups built on it (Boba Anchorage and Base), writes lists of transaction data to layer 1 by breaking the data up into so-called *channel frames* [77]. A *channel* is a sequence of batches of layer 2 transactions tied to a particular time frame in the layer 2's execution (i.e., a layer 2 epoch). Channels are compressed and have a maximum size of about 9.5Mb [79], but may still be too large to post to Ethereum; they are therefore further divided into channel frames which are finally written to Ethereum. A transaction writing to Ethereum may contain several channel frames. This complicated process is illustrated in Figure 2. This enables Bedrock to post data in parallel and to fill up all available space on the layer 1. Bedrock-based implementations use zstd. As compression is applied to batches, which are lists of uncompressed transactions, Bedrock implementations use approach **A2**.

3) Loopring & DeGate: Loopring and DeGate appear to use a version of approach **A1**. They apply ZLE to transactions in order to include more transactions in blocks posted to layer 1 [80], [81]. Both projects have LZ77 implementations in their source code, but this does not appear to be used at this time.

4) Aztec: Aztec network does not explicitly use compression, but occasionally employed some small heuristics to simplify `calldata`. In particular, they used a single 24-bit unsigned integer to encode three concatenated 8-bit unsigned integers [82]; a variant of approach **A1**.

5) Arbitrum: The Arbitrum One rollup (built on the Arbitum *Nitro* design) uses the Brotli compression algorithm to compress batches. This is approach **A2**, though it is noted that the original version of Arbitrum used compression on individual transactions (approach **A1**) [83]. The Nitro design also supports the use of Inverse Huffman Coding (Section III-B4) to read messages in a zero-heavy encoding.

Abritrum Nova uses a DAC (Section III-B5), but falls back to the Arbitrum rollup as a backup.

6) zkSync: zkSync Lite compresses large values with floating point arithmetic [84] (see also Section III-B2), but no other compression. zkLink Era posts data through so-called *State Differences* (or *State Diffs*) [85]. These post the outputs of the state transition (e.g., some storage slot now has a new value of x), rather than the inputs. This data is not compressed at this time, but since only the changes to the storage are recorded, the data posted on-chain is typically less than that of a system which posts the full transaction data.

7) MEMO: Metis Andromeda plans to upgrade their Optimism-based implementation to use MEMO [86], a decentralized storage solution. Since this may not be sufficient — what if the data is withheld or cannot be accessed? — Metis plans to implement a system where the sequencer can be forced to post data on-chain via a layer 1 smart contract call [87]. The Metis Andromeda network will replace the sequencer if the data is not made available by the original sequencer in a short period of time.

The use of compression was not evaluated as the source code for this system is not available.

8) StarkEx & Data Availability Committees: According to L2Beat.com [58], a DAC approach (Section III-B5) is taken by the following StarkEx-based projects: ApeX, Canvas Connect, Immutable X, Myria, Rhino.fi, and Sorare. *AnyTrust* [88] is a variant of the Arbitrum Nitro design which uses a DAC.

As these systems do not post data directly on-chain (they post a hash of it), we did not investigate the compression used by the DAC, which may not have the same cost concerns (e.g., because they store data on cheaper centralized machines). As each DAC is a fixed-size hash or digital signature, they are unlikely to benefit from compression.

D. Algorithm Comparison

In this section, we compare the results of the algorithms in Section III-A. We evaluate claims made by rollup developers (e.g., [78]) and aim to answer **Q4**. We study two metrics: the size of the compressed data, and the algorithm run time.

The first metric we are concerned with is the size of the data after compression. More accurately, the resulting gas cost to post the data is the most important metric. Counter-intuitively, if an algorithm increases the overall size of the data but its output contains many zeros, the *larger* data might be cheaper, as Ethereum currently discounts zero-byte data. However, since some rollups may take this into account and others do not (Section V-A), it is easier to measure the resulting size. With EIP-4844 (see Section V-B), this metric is more relevant since all bytes will cost the same amount on Ethereum.

The second metric that we are concerned with is the time required for each algorithm to compress the data. To ensure that sequencers that post data do not suffer from performance issues, these algorithms should be sufficiently efficient to compress the data without becoming a bottleneck in the system. The algorithms in Section III-A have polynomial time implementations, assuming a fixed alphabet. For small rollup batches, this theoretical time complexity is not too important as the constant overhead may be a much bigger factor.

We show the compressed size and required time for the algorithms in Section III-A on our data set in Table IV. For each approach, we measure four values for a particular compression algorithm: (1) the average (mean) compressed size, which indicates the average size of a batch after compression by the algorithm; (2) the average (mean) compression ratio obtained by the algorithm, which is computed as the compressed size divided by the total size of a batch; (3) the average (mean) time to compress a batch using the algorithm; and (4) the total time to compress all batches using the algorithm. The compression sizes are in number of characters in the resulting hexadecimal string representing the output (hexadecimal strings for batches are converted to bytes and compressed, and the length of the result converted back to a hexidecimal string is used). In all cases, lower values are better than higher ones. Note that we also evaluate the compression of batches using the LZMA algorithm (see e.g., [89]), another dictionary-based compression algorithm, used in the 7Zip compression tool. In addition to Table IV, we illustrate the results of compression for each algorithm. Each chart has a line $y = x$, and points above this line are the result of an algorithm increasing the size of original data, while points below it are the results of successful compression; points on the line have the same size before and after the algorithm is run on the data.

Figure 3 shows the results of running Brotli for both approaches. Brotli was used with the maximum compression setting, using 11 as the "quality level" (levels 1-11 are possible). Approach **A2** has better compression on average, reducing the overall size to only 37% of its original size, when compared to approach **A1**, which only reduced the overall size to 57% of its original size. Moreover, Table IV indicates that the average (and total) time to compress batches in approach **A2** is less than half of that of approach **A1**.

The results of zlib are shown in Figure 4. zlib was also executed with the maximum compression level, 9 (levels 0-9 are possible). Again, approach **A2** is superior to approach **A1**. Approach **A2** reduced the overall size by 61% (100%-39%) while **A1** only reduced by the size by 55%. There is little difference in the average and total time for compression when zlib is used; it is clear that zlib is a very fast algorithm compared to other algorithms in this table.

Figure 5 shows the results of running zstd with maximum compression level 22 (levels 1-22 are possible; levels 19 and above are cautioned to require additional memory). Once again, approach **A2** is superior to approach **A1**, though this time only by an additional 2% of compression. The time necessary to compress batches with this approach is also much closer than the time required to compress using Brotli.

In Figures 6 and 7, we can see the results of applying RLE and ZLE, respectively. Using approach **A1**, RLE fails to compress the data, actually making the result larger than the original data. For approach **A2**, RLE does manage to compress some data, but only results in a 7% reduction in total size. It is clear that the use of RLE should be with caution and only if the result is checked to be an improvement over the initial data, or if there is another reason to use RLE. On the other hand, ZLE performs fairly well. In both approaches, ZLE manages to reduce the data set by at least 44%. While this is not as good as zlib, this is not too far from the results of Brotli. ZLE likely performs well as many transactions (and therefore batches) contain strings of zeros (c.f. Tables II and III). The time required for ZLE is comparable to others, but slower.

Figure 8 shows the results of evaluating bzip2 with compression quality 9 (levels 1-9 are possible) on our data set. Again, approach **A2** wins out over approach **A1**, as bzip2 is able to save 30% more space using approach **A2**. Approach **A2** is also faster than approach **A1** in this case, unlike for Brotli; however, both total times are relatively fast.

Finally, Figure 9 shows the results of evaluating LZMA with compression quality 9 (levels 1-9 are possible) on our data set. The results for LZMA mirror those of Brotli in terms of compression. However, LZMA is considerably slower in

	Approach **A1**				Approach **A2**			
Algorithm	Average Compressed Size	Average Compression Ratio	Average Time	Total Time	Average Compressed Size	Average Compression Ratio	Average Time	Total Time
Brotli	69739	57%	0.56s	23843s	44015	37%	0.17s	7212s
zlib	54039	45%	0.02s	782s	46389	39%	0.02s	853s
zstd	46843	39%	0.05s	1965s	44112	37%	0.05s	2173s
RLE	213560	173%	1.5s	66240s	114117	93%	0.32s	13851s
ZLE	69765	56%	0.2s	8521s	67319	55%	0.11s	4640s
bzip2	90269	74%	0.05s	2108s	48226	41%	0.02s	711s
lzma	65967	55%	11.2s	480102s	44505	37%	0.08s	3616s

TABLE IV: The results of evaluating the compression algorithms in Section III-A with approaches **A1** and **A2**.

approach **A1** than any other algorithm and approach in our experiments. It is so slow that LZMA usage in approach **A1** would most likely incur noticeable delays on a rollup network.

Figure 10 shows the resulting overall compression for each algorithm for both approaches, visualizing the values in columns 3 and 7 of Table IV. Smaller bars are better. In all cases, approach **A2** wins out over approach **A1** for compression results. There are situations where approach **A1** may be desirable — for example, if speed is of utmost importance and a particular compression algorithm must be used (e.g., Brotli) — but the improved compression of approach **A2** seem worthwhile. The zstd algorithm appears to be the best overall single compression algorithm: it saves the most space in approach **A1** (an mount which is also comparable to the best results in approach **A2**), ties for the best compression achieved in approach **A2**, and has the fastest compression time among all algorithms with the same resulting compression size.

Finally, we discuss claims made by rollup developers. Recall from Sections III-C1 and III-C2 that Optimism ruled out Brotli as a compression algorithm because it was too slow, considering instead zlib and zstd. Optimism ultimately favored zstd after initially choosing zlib in their first version for implementation convenience. Table IV can be used to confirm that in fact, Brotli is among the slowest compression algorithms we studied, and definitely the slowest of the three algorithms considered by Optimism. The slowdown from using Brotli would have been particularly evident for Optimism which uses approach **A2**, where it is more than two times slower than zstd. The table supports the claims made by Optimism.

IV. Zero-Knowledge Concerns

In this section, we aim to answer **Q5**, investigating if the requirements of ZK rollup compression differ from those of optimistic rollups.

In a ZK rollup, the bottleneck for block finality is not execution, but generating validity proofs. Typically, generating the validity proof for a given block is much slower than executing it (on the order of minutes, rather than seconds [90]). It is natural to ask if we can use slower compression algorithms when publishing data, as long as the compression still complete prior to proof generation. However, it seems unlikely that there are any algorithms that can perform better than the "state of the art" algorithms in Section III-A, even if they are given additional time. Either way, the compressed data may cause other problems.

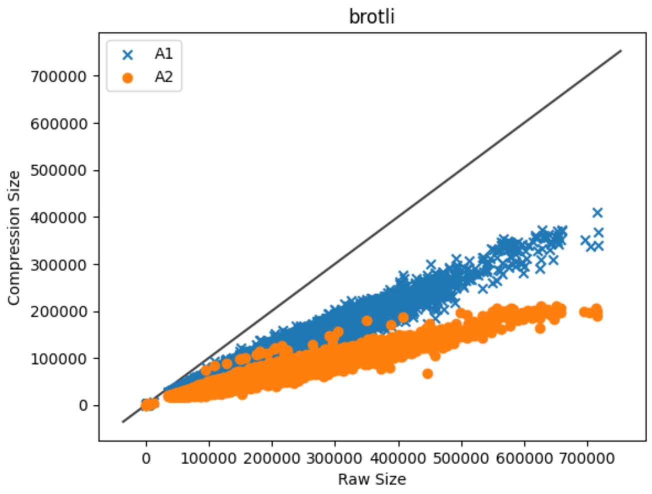

Fig. 3: Brotli results for both approaches. Although both charts look similar, approach **A1** results is a larger compressed size on average than approach **A2**. In particular, the largest batches are compressed to a smaller size in approach **A2** compared to approach **A1**.

The data published should be tied to the (inputs of the) proof posted for a block, which complicates the use of compression. Specifically, when submitting a proof on-chain, the inputs to the circuit should be provided in some form. Typically this is the block data for the proof of the block (or batch of blocks). This data may be public or private, depending on whether or not the ZK rollup is concealing transactions. We assume that it is public, as otherwise the data is not posted on-chain and there is nothing to compress. A ZK circuit can publish and use the data via a *commitment* to the data or directly.

A commitment is a cryptographic primitive that binds the prover to using the data within the circuit without explicitly exposing it [91]. For example, a hash of a transaction may be used as a commitment, and the prover's circuits must verify

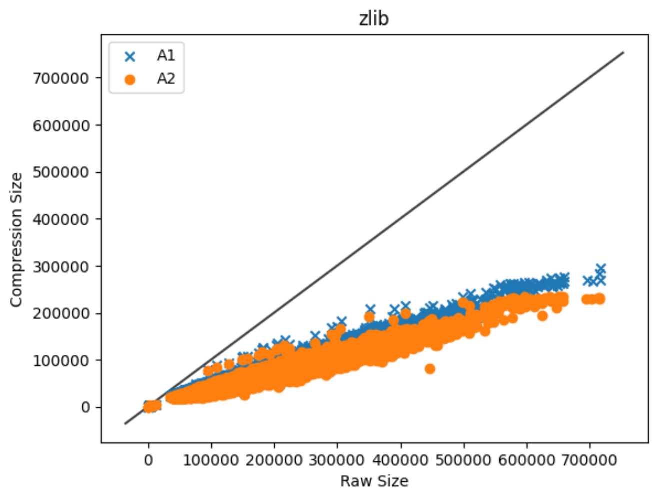

Fig. 4: zlib results for both approaches. Both approaches yield almost the same amount of compression for the batches, though approach **A2** does yield slightly better results.

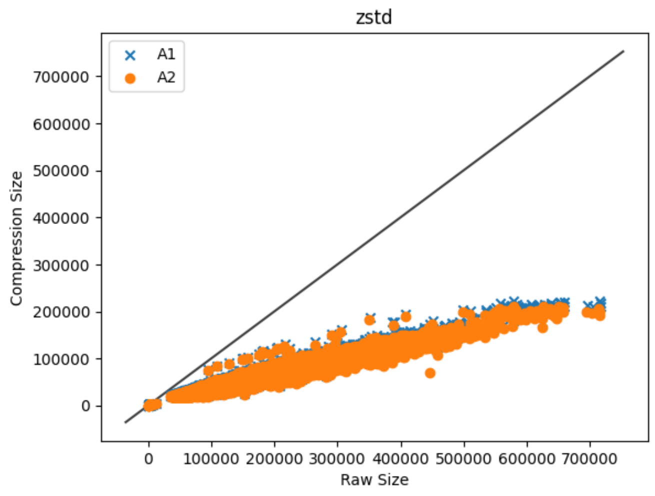

Fig. 5: zstd results for both approaches. Both approaches achieve approximately the same amount of compression on average.

that it knows the pre-image of the hash within the circuit (i.e., the transaction itself). In these cases, the commitments are a small fixed size, and do not benefit from compression. Note that commitments reduce the data posted on chain, but do not enable users to determine what data was processed using only the commitment; they are typically one-way functions.

This can be problematic, as a rollup's sequencer may be responsible for constructing the commitment (representing the block and relevant transactions) for a ZK circuit to use, but it is not running a ZK circuit. In this case, end-users need the data itself to check that the commitment was correctly computed and to verify that the ZK circuit uses the public input commitment correctly. Otherwise, a sequencer could publish a commitment and the circuit may take it as input, but not

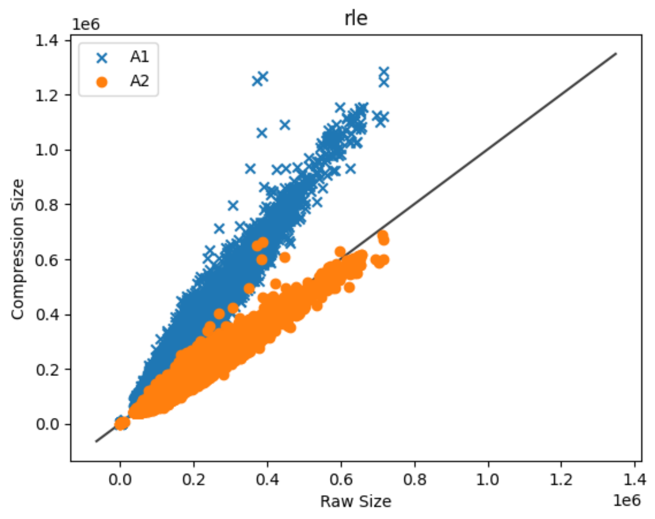

Fig. 6: RLE results for both approaches. Both approaches must be used with care: the use of RLE alone can often increase the size of the data that was to be compressed. This is concern is minimized, but not removed, by using approach **A2**. Compare this to the case where ZLE is used (Table 7).

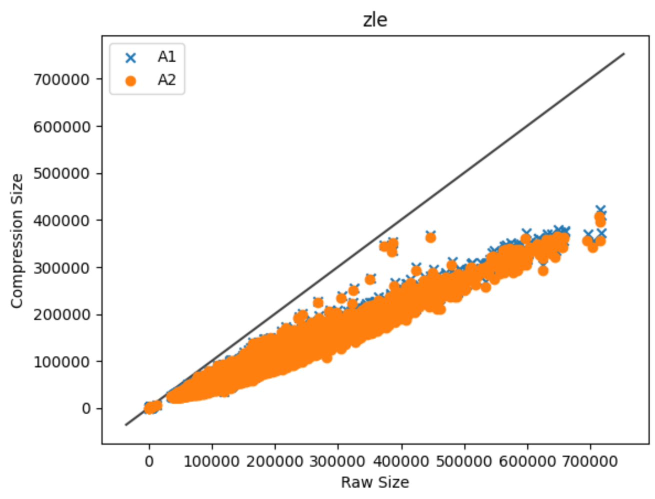

Fig. 7: ZLE results for both approaches. Unlike in the case of general RLE (Table 6), ZLE does not increase the size of the data. This can be attributed to the prevalence of runs of zeroes in the data. Both approaches achieve approximately the same amount of compression, though the average compression is imperceptibly better using approach **A2**.

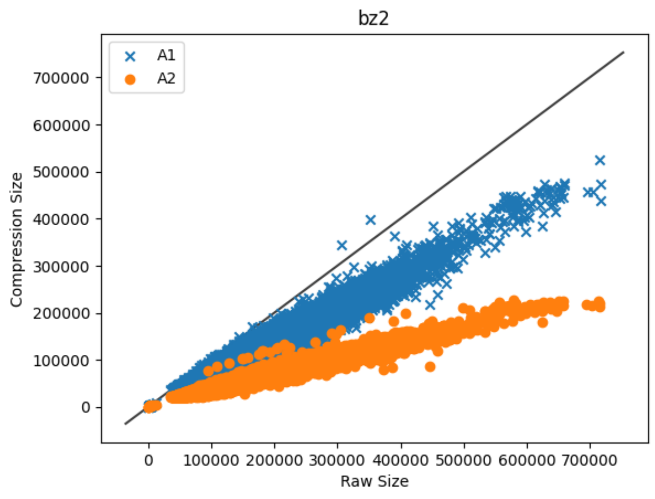

Fig. 8: bzip2 results for both results. Approach **A2** is the superior method for this algorithm, achieving much greater compression. Surprisingly, bzip2 occasionally increases the size of the compressed data in approach **A1**.

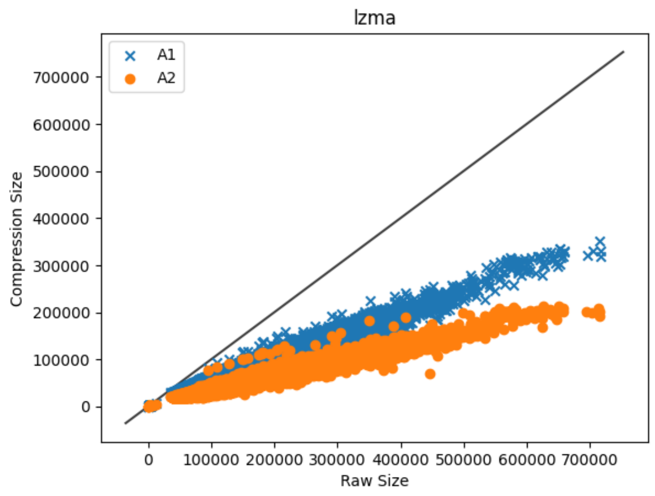

Fig. 9: LZMA results for both results. Approach **A2** is the superior method for this algorithm, achieving much greater compression.

use it in a meaningful sense. One way to overcome this is to open-source the circuits so that users can be convinced of correct results and verification smart contracts.

Another way to overcome this would be to generate the commitment to the data in a ZK proof system, and to verify the proof that this commitment is correctly computed before the state transition function in a ZK rollup uses the commitment. However in order to do that, the inputs to the proof system responsible for generating the commitment proof is necessary, and that is exactly the data that would be posted on-chain in the first place. Since a ZK rollup's sequencer is typically not the same service or actor as its prover, having the sequencer compute commitments provides some level of

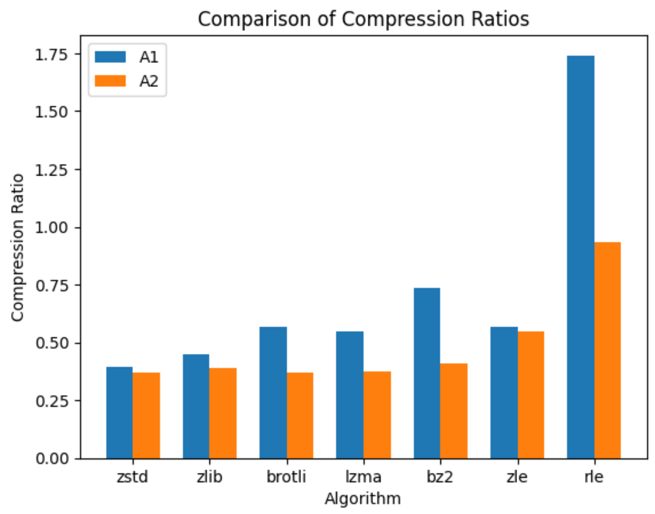

Fig. 10: Compression ratios from Table IV visualized. Lower bars are better. The zstd algorithm performs the best, though most algorithms achieve similar results: the resulting size is about 40% of the original size in approach **A2**. RLE is the notable exception, which is not worth using (in approach **A2**) or is actually counter-productive (as in approach **A1**).

coupling between these components, which may increase trust in the system. Such a sequencer should still post the data itself in addition to the commitment, so that users can verify it (this may be easier or quicker than understanding the ZK circuits for an average user). The layer 1 verification contracts may not consider the state update invalid if the sequencer is dishonest in this manner and the prover discards or fails to use the commitment correctly.

Therefore it makes sense to just use the posted data directly for the state transition proof. In this case, the input to the circuit should use exactly the same data posted on-chain. If data is compressed when posted on-chain, decompression would need to be handled when the proof is verified. This is necessary as otherwise users would also need to verify the compression. There are at least two ways this could happen: (1) decompress the data on-chain before passing it to the verification call, or (2) handle decompression *within* the circuit. Option (1) is likely too expensive (decompressing the data would be more costly than just using uncompressed data). Option (2) requires writing decompression algorithms inside the ZK circuit itself, which may be problematic.

A new combination of "circuit-friendly" primitives would need to be constructed to optimize for compression ratio and circuit complexity. Many of the algorithms in Section III-A (e.g., Brotli, zlib, and zstd) rely on dictionary based compression and involve some bitwise arithmetic. If a dictionary can be pre-computed over all prior collected transactions, some level of compression may be possible by ensuring that the dictionary is contained in a lookup table (see e.g., [92]). However, if the table is not complete, the in-circuit compression will still need to be executed for input which cannot be mapped to table entries, resulting in reduced savings. Furtheremore, bitwise arithmetic may be problematic within an arithmetization: the ZK circuits typically operate over finite fields and integers, rather than bitwise representations. Performing bitwise operations becomes expensive in this setting, increasing the cost or time of proof generation for compressed data.

Finally, one could imagine compressing the proof itself when posting it to the underlying layer 1. However, since the verification logic on-chain for a proof system requires uncompressed proofs, the contract receiving the compressed proof would need to decompress the data prior to verifying the data. This is not likely to be beneficial as these proofs are large numbers (polynomial coefficients and points on which to evaluate relevant polynomials) which resemble random numbers, and compression is therefore not likely to help. The decompression of the proof data would likely add more to the verification gas cost than the savings gained by posting compressed data.

We can now comment on answers to **Q5**. If data is published on-chain (i.e., inputs to the circuits are not kept private), the properties of compression algorithms for ZK rollups may differ. They may differ if they enable compression on either the public inputs, which requires efficient decompression inside of a circuit itself, or if they can be used to decompress proofs on-chain with great efficiency. It appears that both of these properties are not well studied at this time, and are expected to be more harmful than beneficial. Further work is necessary to check that this is indeed the case. Otherwise, the state of the art algorithms presented in Section III-A are likely to be suitable for ZK rollups which also choose to publish transaction data, rather than just commitments.

V. Other Considerations

In this section, we consider the issues of pricing data on-chain and future updates to the Ethereum blockchain. We consider the issue of pricing data on-chain as it is related compression: currently some bytes are cheaper than others, affecting the algorithm choice (sub-optimal compression algorithms may result in cheaper data if they have many zeroes). These considerations are presented in Section V-A and aim to answer question **Q6**. The future of data availability is important as there are major changes planned to the Ehereum network, e.g., EIP-4844 (a.k.a. *proto-danksharding*) [18], that will affect how data is stored on-chain. Such concerns are discussed in Section V-B and aim to answer question **Q7**.

A. Resource Pricing

We consider the issue of pricing data on-chain as it is related to compression. The cost of data may have many parts; for example, in Arbitrum there are costs related to posting data on-chain and rewards for other participants in the system. We focus on the former, as it is universal to rollups which post data to layer 1 networks. The latter may be unique to particular systems; in Aribtrum, these rewards may go to the sequencer or DAC (see Section III-C8) for the network. Other fees, like sequencer operator fees or layer 1 fees incurred by cross-chain function calls are not considered in this section.

A simple way way to price `calldata` for a layer 2 transaction t is as follows. First, calculate the total uncompressed batch size x and the compressed batch size $\hat{x}$. Using the compressed batch size $\hat{x}$, compute the expected gas cost for the entire compressed batch $g(\hat{x})$. To find out how much to charge a particular user for a transaction with `calldata` size t, find what fraction of the uncompressed batch their transaction took up t/x and multiply that by the expected gas cost $g(\hat{x})$ of the compressed batch. The resulting cost of a transaction on the layer 2 is $c(t) = t/x \cdot g(\hat{x})$.

The previous approach is not ideal at for at least two reasons. First, computing $g(\hat{x})$ requires the knowledge of the compressed size of the batch containing the transaction t. However, the batch containing t (and therefore its compressed size) is not known at the time the transaction t is submitted by the user. In some situations it may be possible to charge users at a later time for their transaction, but in general this complicates the process of collecting fees. Moreover, Felton [83] notes that the approach of computing cost according to uncompressed data may not be fair:

> If your [transaction] is in a batch of 100, and you make your [transaction] twice as compressible, your cost will only decrease by 0.5%. You'll have to share the benefits of your improvement with 99 other senders. Ideally, we'd like to give stronger incentives to senders to contribute to better compressibility.

Second, the pricing for data posted to Ethereum is not uniform, and the result may not be as low as possible for the user.

The current pricing for posting `calldata` to Ethereum is not uniform: zero bytes are priced more cheaper (4 gas) than non-zero bytes (16 gas since the Istanbul upgrade [93] implemented EIP-2028 [94] which reduced the cost from 68 gas initially). Since users are charged per unit of gas used, zero bytes are cheaper than non-zero bytes. As a result, algorithms resulting in the most compression may not be necessary or cheapest: algorithms which favor the presence of zero-bytes in the output may result in lower costs.

Arbitrum appears to be the only rollup with compression that leverages this fact to price layer 1 data. Arbitrum prices layer 1 data using an algorithm called `brotli0` [95], [83]. The `brotli0` algorithm is applied to the transaction data, rather than the transaction batch data; the cost function is therefore applied in a manner similar to approach **A1**. Note however that this is only for cost estimation and to charge the fees; the use of compression (Brotli; Section III-C5) is applied to batches rather than transactions. As Arbitrum appears to the rollup with a resource pricing scheme based on compression, we have answered **Q6**.

However, it even without taking advantage of non-uniform pricing, compression reduces costs for end-users by reducing the overhead of posting rollup `calldata`. It may be difficult to fairly price each transaction, but reductions in `calldata` can mean that the average batch size decreases and fixed-cost fees can be reduced. Figure 11 shows the gas savings on the

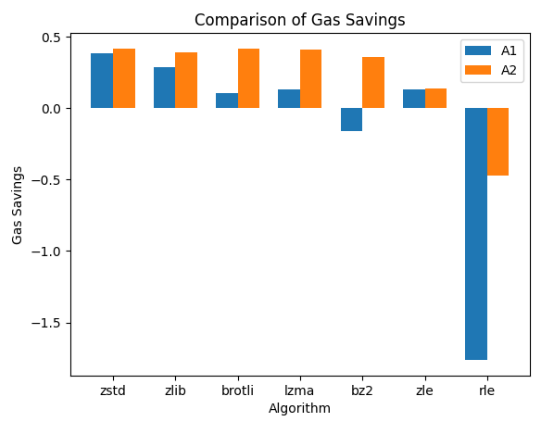

Fig. 11: Gas savings for each approach. Both approaches result in gas savings for almost every algorithm (RLE is the notable exception; c.f. Figure 6), which actually increases the required gas. Approach **A2** results in almost 40% less gas required for most algorithms, compared to about 10% for the most performant algorithm in approach **A1**.

data set for each algorithm. Applying any of the algorithms considered in this work in either approach will almost always result in less gas costs for posting batches on-chain. The use of only RLE should be avoided in either approach, which often increased the size of the data (Figure 6), and in turn increases the cost for posting data. Surprisingly, the use of bz2 in approach **A1** is also to be avoided; bz2 only occasionally increased the size of the data (Figure 8) but this was enough to negate the savings of any compression.

B. *Future of Data Availability*

The future of data availability is important as there are major changes planned to the Ehereum network. In this section we aim to answer question **Q7** by considering the impact of e.g., EIP-4844 (a.k.a. *proto-danksharding*) [18].

EIP-4844: Shard Blob Transactions [18] introduces changes to both the consensus and execution layers of Ethereum. These changes introduce a new transaction type: a *blob carrying transaction* which stores 125kB of data that is not accessible by the EVM. Instead, the EVM can only use view a KZG commitment [91] to the data (see also Section IV). Blobs are stored on the consensus later and are pruned periodically (documents suggest some time between two weeks and thirty days). Blobs are also expected to be cheaper than `calldata` because of this pruning. Also as a result of pruning, the data does not live on the blockchain forever, so interested parties must watch and record the data as it is processed, if it is necessary to keep it for long periods of time.

Discussions on gas price previously were centered around ratio of zero to non-zero bytes before and after compression (since a user's transaction may made up a smaller fraction of the batch before compression compared to after if the transaction had many zero bytes), but this is no longer a consideration after EIP-4844. If one needs to consider compressibility of transactions pre- and post-compression, then using the fraction of a batch gas that a transaction took up pre-compression is a better way to assign final gas cost. This does not penalize people who have bigger batches than may be comprised of more zeros – which may aid in compression – compared to just caring about size. After EIP-4844, the price of `calldata` for blobs is uniform for all bytes. Therefore, after EIP-4844, optimizing for compression ratio (compressed size over raw size) is the most important aspect of compression. Specific approaches like Inverse Huffman coding (Section III-B) will be irrelevant after EIP-4844.

EIP-4844 also introduces a multi-dimensional EIP-1559 price market [96] to price transactions. This will change how much data costs from the constant amount of 16 or 4 gas per byte. Blobs will be paid for in gas, and the amount of gas per blob will adjust so that the average blobs in a block hits some target amount.

In the ZK setting, changes may be desirable. EIP-4844 blobs will not be accessible on the execution layer; only their KZG commitments will be. It may be beneficial to build ZK circuits that can verify KZG batch opens to then do computations over the batch. This may be necessary if the verifier contract only has access to the KZG commitment on-chain but requires the entire batch to actually verify a batch was correctly executed.

VI. Related and Future Work

We restricted our analysis to algorithms in Table I in order to evaluate real-world approaches. Naturally, the algorithms considered may also perform differently on other data sets; Gupta et al. [97] evaluate `DEFLATE` (zlib) and other algorithms on a more general corpus of data, and others (e.g., [98], [99], [100]) have evaluated compression in other domains. However, they do not necessarily consider the same algorithms and to the best of our knowledge, this is the first work that considers compression for data posted on blockchains, especially data required for rollups.

Other applications of compression to blockchain technologies do exist, but are limited to reducing state space growth of these networks, (e.g., [101], [102]) or using compression on blockchain-enabled devices to compress data stored (e.g., [103]).

The general purpose algorithms described in work have been studied for various reasons. For example, zstd and zlib have been studied with hardware acceleration [104], [105]. Theoretical bounds on some compression techniques in this work have also been studied (e.g., [106]).

More generally, some related work exists for domain-specific concerns discussed in this work. For example, ideas for unequal costs of symbols are discussed in e.g., [107], [108]; these approaches may be beneficial considering the currently unequal cost of zero bytes on Ethereum. Exploring these ideas for further compression is a direction for future work.

Another interesting direction for future work is to compress dynamically using combinations of the algorithms presented in this work. The use of `brotli0` is an initial step in this direction, but additional work can reduce costs further. Given the latency for finality of both optimistic (typically several days) and ZK (typically minutes) rollups, these systems can likely afford try combinations of compression techniques for a given data set, and take the best one. It would also be interesting to see if there is structure within some transactions that might enable rollup sequencers to build blocks and batches in order to maximize compressibilty (subject to some minimum liveness requirements).

Finally, there is at least one more static approach to compression that can be used:

A3 Apply compression to individual transaction strings within a batch and concatenate the resulting strings to use as the batch data, *and* then compress the results.

This is a combination of approaches **A1** and **A2**. Given the apparent advantage of approach **A2** over **A1** and the diminishing returns of compressing already compressed data, this was not explored in this work. However, there may be cases (especially with some specific algorithm combinations), where this has benefits.

VII. Conclusion

In this work, we have surveyed the use of compression in rollups for the Ethereum ecosystem. We first identified which algorithms are in use by modern rollups and enumerated heuristics used in the space. Then, we identified where these algorithms and heuristics are used within these systems and how they can be evaluated. We found that these systems typically use off-the-shelf state of the art algorithms with a handful of heuristics to further reduce the amount of data posted on-chain and in turn the cost for end-users. Turning to the specific types of rollups, we investigated the helpful properties of compression in zero knowledge rollups, and found that they may not benefit so readily from compression techniques. We surveyed the approaches to utilize compression to impact resource pricing and found that not many exist. Then we considered how all of these findings may change in light of major upcoming changes to the Ethereum network. Finally, we outlined directions for future work to further utilize compression in rollups and reduce both end-user and operating cost for these systems.

References

[1] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. Ethereum Project Yellow Paper, https://ethereum.github.io/yellowpaper/paper.pdf.

[2] Nick Szabo. Smart contracts: Building blocks for digital markets. 1996.

[3] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, 2015. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[4] Santeri Paavolainen and Christopher Carr. Security properties of light clients on the ethereum blockchain. *IEEE Access*, 8:124339–124358, 2020.

[5] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. SoK: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 201–226. Springer, 2020.

[6] Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. SoK: Not quite water under the bridge: Review of cross-chain bridge hacks, 2022.

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

[8] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure multi-hop payments without two-phase commits. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 4043–4060. USENIX Association, 2021.

[9] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 455–471. ACM, 2017.

[10] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[11] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^2l: Anonymous atomic locks for scalability in payment channel hubs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1834–1851. IEEE, 2021.

[12] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017. https://www.plasma.io/plasma.pdf.

[13] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-chains: Secure, scalable off-chain payments. Cryptology ePrint Archive, Paper 2018/642, 2018. https://eprint.iacr.org/2018/642.

[14] Patrick McCorry, Chris Buckland, Bennet Yee, and Dawn Song. SoK: Validating bridges as a scaling solution for blockchains. Cryptology ePrint Archive, Paper 2021/1589, 2021. https://eprint.iacr.org/2021/1589.

[15] Vitalik Buterin. Sidechains vs Plasma vs sharding, 2017. https://vitalik.ca/general/2019/06/12/plasma_vs_sharding.html.

[16] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud and data availability proofs: Detecting invalid blocks in light clients. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2021.

[17] Vitalik Buterin, Dankrad Feist, Diederik Loerakker, George Kadianakis, Matt Garnett, Mofi Taiwo, and Ansgar Dietrichs. EIP-4844: Shard blob transactions, 2022. https://eips.ethereum.org/EIPS/eip-4844.

[18] Vitalik Buterin. Proto-danksharding faq, 2022. https://notes.ethereum.org/@vbuterin/proto_danksharding_faq.

[19] Jan Gorzny, Lin Po-An, and Martin Derka. Ideal properties of rollup escape hatches. In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, DICG '22, page 7–12, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[21] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1353–1370. USENIX Association, 2018.

[22] Alex Gluchowski. Introducing zkSync: the missing link to mass adoption of ethereum, 2021.

[23] Kamil Jezek. Ethereum data structures. *CoRR*, abs/2108.05513, 2021.

[24] Barbara Carminati. Merkle trees. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[25] Wojciech Szpankowski. Patricia tries again revisited. *J. ACM*, 37(4):691–711, 1990.

[26] Kiyun Kim. Modified Merkle Patricia trie — how Ethereum saves a state. https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd.

[27] Vitalik Buterin. An incomplete guide to rollups, January 2021. https://vitalik.ca/general/2021/01/05/rollup.html.

[28] Public inputs. https://github.com/privacy-scaling-explorations/zkevm-specs/blob/9eedc87b7ec44dfad6ffea8b2d7a010773de4d41/specs/public_inputs.md.

[29] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Keccak. Cryptology ePrint Archive, Paper 2015/389, 2015. https://eprint.iacr.org/2015/389.

[30] zkSync Era. https://zksync.io/.

[31] Clarification on how calldata persists on the blockchain and how optimistic rollups use it. https://ethresear.ch/t/clarification-on-how-calldata-persists-on-the-blockchain-and-how-optimistic-rollups-use-it/8136/2.

[32] Apex. https://www.apex.exchange/.

[33] Arbitrum Nova. https://nova.arbitrum.io/.

[34] Arbitrum One. https://bridge.arbitrum.io/.

[35] Ariel Gabizon, Zac Williamson, and Tom Walton-Pocock. Aztec Network yellow paper, 2021. https://hackmd.io/@aztec-network/ByzgNxBfd.

[36] Jon Wu. Private DeFi with the Aztec Connect bridge, 2021. https://medium.com/aztec-protocol/private-defi-with-the-aztec-connect-bridge-76c3da76d982.

[37] Will Robinson. Introducing Base, February 2023. https://www.coinbase.com/blog/introducing-base.

[38] Turing hybrid compute. https://boba.network/turing-hybrid-compute/.

[39] Engineering roadmap AMA recap, September 2022. https://bobanetwork.medium.com/engineering-roadmap-ama-recap-f942acb8aea4.

[40] CANVAS connect. https://canvas.co/.

[41] DeGate. https://docs.degate.com/.

[42] Antonio Juliano. dYdX: A standard for decentralized margin trading and derivatives, 2018. https://whitepaper.dydx.exchange/.

[43] Fuel (v1). https://www.fuel.network/.

[44] Immutable X. https://www.immutable.com/.

[45] Layer2.Finance. https://app.l2.finance/.

[46] Loopring. https://loopring.org/.

[47] Metis. https://www.metis.io/.

[48] Myria. https://myria.com/.

[49] Optimism. https://www.optimism.io/.

[50] Rhino.fi. https://rhino.fi/.

[51] Polygon ZK: Deep dive into Polygon Hermez 2.0, 2022. https://blog.polygon.technology/zkverse-deep-dive-into-polygon-hermez-2-0/.

[52] Scroll. https://scroll.io/.

[53] Sorare. https://sorare.com/.

[54] StarkEx. https://starkware.co/starkex/.

[55] StarkNet. https://starkware.co/starknet/.

[56] ZKSpace. https://zks.org/.

[57] zkSync Lite. https://lite.zksync.io/.

[58] L2beat. https://l2beat.com/.

[59] Jyrki Alakuijala and Zoltan Szabadka. Brotli compressed data format. https://datatracker.ietf.org/doc/html/rfc7932.

[60] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. *ACM Trans. Inf. Syst.*, 37(1):4:1–4:30, 2019.

[61] Jean loup Gailly and Mark Adler. zlib. https://www.zlib.net/.

[62] Yann Collet and Murray Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021.

[63] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[64] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[65] Debra A Lelewer and Daniel S Hirschberg. An order-2 context model for data compression with reduced time and space requirements. Technical Report 90-33, Information and Computer Science, University of California, Irvine, 1990.

[66] deflate. https://refspecs.linuxbase.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/zlib-deflate-1.html.

[67] Jarek Duda. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540, 2013.

[68] Charles E Mackenzie. *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[69] M.J. Golin and G. Rote. A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs. *IEEE Transactions on Information Theory*, 44(5):1770–1781, 1998.

[70] Phil Bradford, Mordecai J Golin, Lawrence L Larmore, and Wojciech Rytter. Optimal prefix-free codes for unequal letter costs: Dynamic programming with the monge property. *Journal of Algorithms*, 42(2):277–303, 2002.

[71] Internet FAQ Consortium. Run Length Encoding Patents, March 1996. http://www.ross.net/compression/patents_notes_from_ccfaq.html.

[72] Michael Burrows. A block-sorting lossless data compression algorithm. *SRC Research Report, 124*, 1994.

[73] bzip2. http://www.bzip.org/.

[74] Boris Yakovlevich Ryabko. Data compression by means of a "book stack". *Problemy Peredachi Informatsii*, 16(4):16–21, 1980.

[75] Recursive-length prefix (RLP) serialization, 2023. https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp/.

[76] Ed Felton. Zero-heavy encoding #409, March 2022. https://github.com/OffchainLabs/nitro/pull/409.

[77] L2 chain derivation specification, 2023. https://github.com/ethereum-optimism/optimism/blob/develop/specs/derivation.md.

[78] The road to sub-dollar transactions, part 2: Compression edition, March 2022. https://medium.com/ethereum-optimism/the-road-to-sub-dollar-transactions-part-2-compression-edition-6bb2890e3e92.

[79] Here's how bedrock will bring significantly lower fees to optimism mainnet, March 2023. https://blog.oplabs.co/heres-how-bedrock-will-bring-significantly-lower-fees-to-optimism-mainnet/.

[80] Loopring 3. https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/DESIGN.md.

[81] Batch spot trade. https://docs.degate.com/advanced/batch-spot-trade.

[82] ACE.sol specification, 2023. https://github.com/AztecProtocol/aztec-v1/blob/a47d3d9ea38cd1363ede730998145da663df6091/packages/documentation/styleguide/categories/Specification/ACE.md.

[83] Compression in Nitro, March 2022. https://research.arbitrum.io/t/compression-in-nitro/20.

[84] zkSync rollup protocol. https://github.com/matter-labs/zksync/blob/master/docs/protocol.md.

[85] gm zkEVM! https://blog.matter-labs.io/gm-zkevm-171b12a26b36.

[86] Memolabs. https://www.memolabs.org/.

[87] The tech journey: Lower gas costs & storage layer on Metis, 2022. https://metisdao.medium.com/the-tech-journey-lower-gas-costs-storage-layer-on-metis-867ddcf6d381.

[88] Introducing AnyTrust chains: Cheaper, faster L2 chains with minimal trust assumptions, March 2022. https://medium.com/offchainlabs/introducing-anytrust-chains-cheaper-faster-l2-chains-with-minimal-trust-assumptions-31def59eb8d7.

[89] Lasse Collin. A quick benchmark: Gzip vs. bzip2 vs. lzma. *Web site: http://tukaani. org/lzma/benchmarks. html [Last accessed: 2 October 2012]*, 2005.

[90] zkEVM FAQ. https://docs.zksync.io/zkevm/.

[91] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[92] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[93] The history of Ethereum, 2023. https://ethereum.org/en/history/#istanbul.

[94] Alexey Akhunov, Eli Ben Sasson, Tom Brand, Louis Guthmann, and Avihu Levy. EIP-2028: Transaction data gas cost reduction, 2019. https://eips.ethereum.org/EIPS/eip-2028.

[95] l1pricing.go. https://github.com/OffchainLabs/nitro/blob/ce5eb97644e03586175d0b4b3f6ed9966ac42f70/arbos/l1pricing/l1pricing.go.

[96] Vitalik Buterin. Multidimensional EIP 1559. https://ethresear.ch/t/multidimensional-eip-1559/11651.

[97] Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–8, 2017.

[98] Somayeh Sardashti and David A. Wood. Could compression be of general use? evaluating memory compression across domains. *ACM Trans. Archit. Code Optim.*, 14(4), dec 2017.

[99] Zaid Bin Tariq, Naveed Arshad, and Muhammad Nabeel. Enhanced LZMA and BZIP2 for improved energy data compression. In Markus Helfert, Karl-Heinz Krempels, Brian Donnellan, and Cornel Klein, editors, *SMARTGREENS 2015 - Proceedings of the 4th International Conference on Smart Cities and Green ICT Systems, Lisbon, Portugal, 20-22 May, 2015*, pages 256–263. SciTePress, 2015.

[100] Oswald C., E. Haritha, A. Akash Raja, and B. Sivaselvan. An efficient and novel data clustering and run length encoding approach to image compression. *Concurr. Comput. Pract. Exp.*, 33(10), 2021.

[101] Beltran Borja Fiz Pontiveros, Robert Norvill, and Radu State. Recycling smart contracts: Compression of the ethereum blockchain. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2018.

[102] Ulfah Nadiya, Kusprasapta Mutijarsa, and Cahyo Y Rizqi. Block summarization and compression in bitcoin blockchain. In *2018 International Symposium on Electronics and Smart Devices (ISESD)*, pages 1–4, 2018.

[103] Teasung Kim, Jaewon Noh, and Sunghyun Cho. Scc: Storage compression consensus for blockchain in lightweight iot network. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–4, 2019.

[104] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. FPGA acceleration of zstd compression algorithm. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*, pages 188–191. IEEE, 2021.

[105] David Zaretsky, Gaurav Mittal, and Prithviraj Banerjee. Streaming implementation of the ZLIB decoder algorithm on an FPGA. In *International Symposium on Circuits and Systems (ISCAS 2009), 24-17 May 2009, Taipei, Taiwan*, pages 2329–2332. IEEE, 2009.

[106] Travis Gagie and Giovanni Manzini. Move-to-front, distance coding, and inversion frequencies revisited. *Theor. Comput. Sci.*, 411(31-33):2925–2944, 2010.

[107] Mordecai J. Golin, Claire Kenyon, and Neal E. Young. Huffman coding with unequal letter costs. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 785–791, New York, NY, USA, 2002. Association for Computing Machinery.

[108] Y. Perl, M. R. Garey, and S. Even. Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters. *J. ACM*, 22(2):202–214, apr 1975.