Smart Meter Data Compression using Generalized Deduplication

Marcell Fehér[1], Niloofar Yazdani[1], Morten Tranberg Hansen[2], Flemming Enevold Vester[2], Daniel E. Lucani[1]
[1]Agile Cloud Lab, Department of Engineering, DIGIT, Aarhus University, Aarhus, Denmark
[2] Kamstrup A/S, Skanderborg, Denmark
{sw0rdf1sh,n.yazdani,daniel.lucani}@eng.au.dk, {mtr,flev}@kamstrup.com

***Abstract*—Utility providers are relying more often on smart, wirelessly connected smart meters to collect consumption information of their customers. The sheer amount of connected smart meters and the growing requirements to provide more frequent reports from each device are putting a large strain on existing systems and protocols. In this paper, we propose three novel lossless compression schemes that significantly reduce the size of standard DLMS data messages uploaded by smart electricity meters. Using real life data sets, we show that these methods can achieve compression rates of over 90% while being transparent to the DLMS protocol and eliminating the privacy concerns of the existing compression methods.**

***Index Terms*—compression, IoT, generalized deduplication, smart meters**

I. Introduction

Utility providers have started to switch collecting consumption measurements from technicians manually reading electricity meters to installing smart meters with network connectivity to report periodic readings. Besides saving the cost of manual readings, this move also allows providers to collect data more frequently. This opens up new opportunities like real-time monitoring, reporting customer usage with higher granularity, dynamic pricing or better anomaly and fraud detection. While these new abilities bring plenty of business advantages, they also come with technical challenges to efficiently collect, store and analyze all this additional data coming from a multitude of smart meters [1], [2]. This is particularly challenging in protocols using highly configurable and descriptive data formats, e.g., Device Language Message Specification (DLMS), which introduces a considerable header for data units containing measurements. In particular, as fewer measurements are transmitted, the header can become prohibitively large, which is problematic for real-time and higher frequency monitoring and reporting.

In this paper, we address this growing problem of transferring large volumes of data from the smart meters to a Head-End System (HES, server back-end of the utility provider), by proposing compression schemes that work particularly well with this type of data (and headers). We also evaluate them using a real-life data set provided by a major smart meter manufacturer, Kamstrup A/S. By reducing the amount of data sent by each smart meter, we aim to bring multiple improvements. First, reduce congestion on the (typically cellular) network carrying measurement data by significantly decreasing the bandwidth requirement of smart meters. Second, reduce the storage costs of the HES and/or Cloud service storing historical smart meter data, i.e., reducing data storage costs for the utility provider. This is due to the fact that the compressed representation we propose for improving data transfers can also be used to store the data persistently. These advantages ultimately allow utility companies to measure quantities of interest with higher granularity and report them more frequently.

The paper is organized as follows. Section II reviews existing efforts in the field of compressing smart meter data. Section III introduces the standard data format used by electricity meters, and presents the available/supported compression methods in the protocol. We also briefly discuss generalized deduplication as the core technique used for our compression methods. Section IV introduces our novel compressor algorithms. Each method is evaluated using a real life data set (Section V). We summarize our findings in Section VI.

II. Related Work

Strategies to handle bandwidth and storage limitations of smart meters can be classified into edge processing and lossy/lossless data compression algorithms. Sirojan *et al.* [3] proposed an embedded edge computing paradigm based on event detection, feature extraction and classification techniques to enable the meter itself to generate consumer energy savings feedback. Aside from the additional computational complexity introduced by these techniques to the smart meters, a considerable amount of data still needs to be transmitted to the HES, which make data compression algorithms a good choice. Lossy compression algorithms such as [1] and [4] provide a high compression ratio at the cost of accurate data reproducibility. In [1] an adaptive data reduction algorithm using compressive sampling is proposed, which exploits the sparsity in a dataset. [4] has adapted singular value decomposition technique.

There are few works in the field of lossless compression techniques for smart meter readings. Kraus *et al.* [5] introduced a compression scheme based on predictive modeling and the LZMA algorithm. The technique needs a preprocessing step and a dictionary. In [6] a compression approach using the notation of compressive sensing is proposed. They have considered an empirical modeling of the aggregated power signal of home appliances. However, different home appliances have different power signal model. In [7] a compression algorithm with the target of representing the readings in few parameters using Gaussian approximation based on

dynamic-nonlinear learning mechanism is proposed. A resumable data compression approach based on differential coding with sampling intervals in the range of few seconds or less is proposed in [2]. Despite all the efforts, compressing smart meter readings effectively with low computational complexity, low memory requirement, and being independent of the data model remains practically challenging.

III. Background

To better understand some of the unique aspects of our proposed compression algorithms, we first introduce the data format used by smart meters to upload readings to the utility's back-end system. Then, we give an overview of data reduction methods that are supported by the standard out of the box and introduce a novel technique (Generalized Deduplication), which we use in our proposed algorithms.

A. Data Format

A very general data format called Device Language Message Specification (DLMS) has been standardized by an association formed by meter manufacturers, various utility providers and other stakeholders. It is used as the communication protocol of smart meters measuring electricity, gas and water. Electricity meters can measure several physical quantities at programmable time intervals. DLMS allows defining profiles for collecting and pushing sets of these measurements, and rules about when to push them to the HES. When upload is triggered, the meter generates an application data unit (APDU) message with the data rows it holds in its buffer, and uploads it to the server.

The important aspect of the messages we evaluated is that they consist of two main parts; a relatively large header and the measurement data. The header encodes information about what is being measured and with what frequency, as well as additional object definitions. The data part contains the measurement records in a serialized form. Because the data format is very general, DLMS messages are quite verbose. For example, an APDU that contains four readings of four quantities each is 527 bytes long, out of which only 152 bytes contain the actual reading values, the rest is the header (331B) and the type information (44B).

B. Built-in Compression Methods

The DLMS standard includes built-in techniques to decrease message size.

Null data compression replaces repeating values in the data records with a single marker byte if they are identical (e.g. the same active energy unit was measured twice in a row) or the difference between consecutive values is predictable (typically used for timestamps and counters). This method can achieve higher compression rate when more data is present. Since the large header is still uncompressed, fewer measurements per APDUs provide less ground for the compressor to decrease the message size.

Delta array compression is a new technique that aims to use a smaller data type for values that can be represented

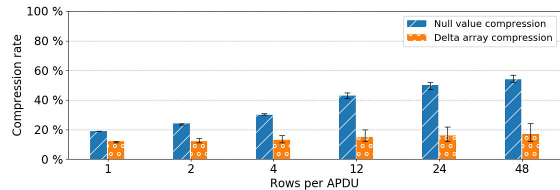

Fig. 1: Average compression rates of Delta array compression compared to Null data compression, applied to APDUs of five smart meters.

in fewer bytes than reserved for them. For example, if the measurement identifier is defined as a 4-byte unsigned integer but the actual value fits in a single byte, then delta array compression down-casts the value and saves 3 bytes. This compression method is not restricted to the data part of a DLMS message but can also shrink values in the header, therefore the total compression gains are not dependent on the ratio of header and data, as they were with Null compression. Figure 1 shows the average compression rate of both built-in compression methods when applied to APDUs of five smart meters, with error bars showing the minimum and maximum.

Null and Delta compression are effective packet size reduction methods, but some utility providers hesitate to employ them because they leak information about the consumption. An attacker who can determine the size of consecutive APDUs compressed with these techniques is able to infer the usage pattern of the consumer.

C. Generalized Deduplication

Deduplication (DD) is a traditional compression technique that reduces size by eliminating repeating chunks of the data. In recent years a new, more effective variant of this algorithm has been introduced, which overcomes the biggest limitation of the original method. Generalized deduplication (GDD) [8] can reduce data size not only when chunks of the data are identical, but also when they are similar to each other. During compression, each segment is transformed to a pair of "basis" and a small "deviation" and each unique basis is only stored once. The decompressor is able to reconstruct the original sections without any loss in precision. A good transformation function produces the same basis for similar data chunks, allowing for higher compression gains. In [9] a multi source, lossless data transmission compression approach based on GDD is proposed which transmits the basis (large part of the data) only if it is not available at the sink node to reduce data transmission.

IV. Compression Schemes

In this paper we focus on compression techniques that can be used on top of the DLMS protocol in a transparent way. We propose methods that compress standard APDUs generated by smart meters, and decompress them in a lossless way at the Head End System, outputting the same APDU that would have been transmitted without the compressor. To ensure compatibility with existing configurations, our methods

are designed to work when either Null value, Delta array, or both built-in compression modes are turned on. However, we are aiming at replacing these compressors with new techniques that does not reveal usage patterns of consumers.

We propose three techniques to reduce the size of the transferred APDU packets. The first one targets the large and very redundant header, while the second and third compress the whole APDU using generalized deduplication, either from a raw or pre-compressed representation. All three methods assume relatively small APDU sizes, where the header takes up a considerable chunk of the whole message, and there are up to a few tens of readings in each APDU. This assumption is in line with the real life trend of utility providers wanting to monitor consumption closely and have the ability to report usage to consumers with lower latency.

A. Header Hash Compression

A major source of overhead in APDU messages is the header, which contains a very verbose encoding of a small number of attributes that are actually relevant for each message. Our first approach to decrease the message size is to replace the whole header with a hash value.

In this scenario, smart meters construct APDUs as usual, but instead of uploading the whole message, the header bytes are deleted and replaced by their own hash. The decompressor at the Head End System side stores a list of active measurement profile headers and their corresponding hash values. When a compressed APDU is received, the hash is looked up from this table and the original APDU header is reconstructed.

Based on how many different measurement profiles (therefore headers) are in use, hash functions of different lengths may be used. In real life this is usually not higher than a few hundreds, therefore a short 4-byte hash is enough to keep the probability of hash collisions at an acceptably low rate. The compression scheme allows using any hash function, as long as the meters and the decompressor at the Head End System use the same one.

Header hash compressor supports the built-in Null value and Delta array compressor naturally. Null value compression does not change the header, and since our decompressor can store any header-hash pairs in its lookup table, it's possible to keep both an uncompressed and a delta-compressed version of the same header. Both compression and decompression work the same as before: the compressor replaces the header with its hash, and the decompressor reconstructs the original APDU using its lookup table.

This algorithm uses very little resources on the smart meter. To replace the header with the hash, we only need a few hundred bytes of memory to compute the hash value. Depending on the hash function, the computational requirements range from extremely low (CRC32) to moderate (SHA-256).

To store the lookup table of hash values and full headers, the decompressor on the Head End System side needs a relatively small amount of space, as seen in Table I. In the usual range of a few hundred different profiles, the lookup table takes less than a megabyte.

TABLE I: Lookup table sizes at the Head End System

Profiles	CRC32	MD5	SHA-1	SHA-256
100	33 KB	34 KB	35 KB	36 KB
500	166 KB	172 KB	174 KB	180 KB
1.000	331 KB	343 KB	347 KB	359 KB
5.000	1.7 MB	1.7 MB	1.7 MB	1.8 MB
10.000	3.3 MB	3.4 MB	3.5 MB	3.6 MB

A disadvantage of this compression scheme is that the lookup table must be filled with the headers of profiles before the first compressed APDU is received. Without a matching hash, the decompressor is unable to reconstruct the original message, and the Head End System cannot parse and process the data records. This limitation can be overcome using different approaches like asking the meter about an unknown hash, or periodically sending the whole header.

Overall, the header hash compression scheme produces extremely good compression when a small hash function like CRC32 is used, and the APDU message contains few measurements.

The next two compression schemes treat the whole APDU as a continuous byte array, without assuming anything about its internal structure, e.g. separation of header and data, or where are the data values in the encoded message. Therefore, these methods work without modifications on uncompressed or pre-compressed APDUs, using any combination of Null Value, Delta Array or Header Hash compressors.

B. GDD Block Compression

A straightforward way to utilize Generalized Deduplication on APDUs is to compress each message separately, as an independent block (hence the name "block compression"). Each APDU is broken into a series of chunks, which are each transformed into a pair of basis and a small deviation. Unique bases of an APDU are only stored once, repeating bases are replaced by the index of their first occurrence. Deviations are stored without modifications. Figure 2 illustrates the process and Algorithm 1 describes it in detail.

Algorithm 1 Compress an APDU with Block compression

Input: Original APDU $apdu_in$, chunk size $chunk_len$, GDD transformation function $gdd_transform(chunk)$
Output: Compressed APDU $apdu_out$
1: $bases \leftarrow \{\}$, $deviations \leftarrow \{\}$
2: $chunks \leftarrow slice(apdu_in, chunk_len)$
3: **for** $chunk$ in $chunks$ **do**
4: $basis, dev = gdd_transform(chunk)$
5: $i \leftarrow find_index(basis, bases)$
6: $bases.insert(i$ if $i \geqslant 0$, otherwise $basis)$
7: $deviations.insert(dev)$
8: **end for**
9: $apdu_out \leftarrow (bases, deviations)$
10: **return** $apdu_out$

The compressor maintains a list of bases that has already been produced by chunks of the same APDU. When a new

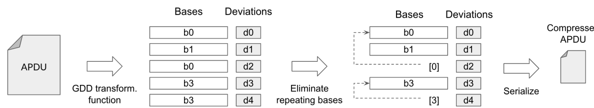

Fig. 2: Compressing each APDU as an independent block. Chunks are transformed to pairs of bases and deviations, and repeating bases are replaced by a base index.

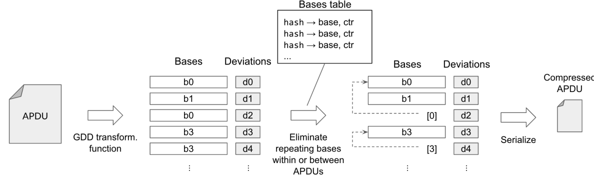

Fig. 3: Compressing APDUs as a stream. Bases from previous messages are replaced by their hash.

chunk is transformed, the algorithm tries to find the basis in this list and instead of storing the basis again, only its first index is added to the list of bases. The compressed APDU is a list of basis-deviation pairs, where each basis is either a unique byte array or an index.

Every repeating basis reduces the message size by the difference of index size and base size, therefore, the more chunks are mapped to the same basis, the higher compression is achieved. The overall compression rate primarily depends on two factors: the rate of repeating bases (higher is better) and the relative size of an index to a full base (smaller is better). To increase the ratio of identical bases, we can change the transformation function or its parameters, or decrease the chunk size, since smaller chunks are more likely to be similar to each other. At the same time, to decrease the relative size of the index, the chunk size must be increased. Fewer chunks become larger, resulting in larger bases and smaller indexes, both effects contributing to more bytes saved when a base is replaced by its first index. Finding the best chunk size, transformation function and its parameters for generalized deduplication is a challenging task due to the conflict of optimizing these two contradictory goals at the same time.

C. GDD Stream Compression

Block compression processes each APDU as a separate message, even though similarity is also present between them, especially if they belong to the same measurement profile. We can leverage this in the compressor, and introduce a new technique where repeating bases across APDUs are also not sent multiple times (see Figure 3).

Our GDD-based stream compressor works similarly to the block version, with a notable addition: it replaces bases from previous messages with their hash values. To do that, the compressor maintains another list where base hashes from previous APDUs are stored. When the next message is constructed, bases that are repeating within the current message are replaced by their first index within the message (same as Block Compression), while bases that were part of a previous message are replaced by their hash value. New bases which have not been seen in any of the previous APDUs are added to the message in their full representation and their fingerprint is inserted to the lookup table.

Since the meters have limited memory and storage, this table cannot grow indefinitely. When there's no space for a new hash, the meter has to select a victim row that is evicted. To increase efficiency of compression, we assign a counter to each base hash in the lookup table, which is incremented each time when the corresponding base is seen. Using this value as a measure of importance, the compressor can evict the least frequently used hash from the table. If an evicted base is seen again later, it is considered a new base and added to the message in its full representation.

The decompressor at the Head End System builds and maintains a list of hash-base pairs as APDUs are being received, so it can look up the original base value when a hash is representing it in the compressed message. Since identical bases from different meters produce the same hash value, it is enough for the decompressor to maintain a single lookup table for all meters, which requires significantly less memory than having one for each meter.

V. Performance Evaluation

We test the compression ratio of our proposed compression methods on smart meter readings provided by Kamstrup A/S. The data set consists of five CSV files which correspond to five smart meters. They contain thousands of measurements of the same four quantities with the resolution of 15 or 60 minutes (see Table II). The figures show the average compression rates of the APDUs from the five meters, error bars showing the minimum and maximum compression rate achieved.

TABLE II: Evaluation data sets

Meter ID	Resolution	Measurements	Data collection period
28982000	15 minutes	8.495	176 days
28981768	15 minutes	8.767	180 days
28034449	60 minutes	6.451	268 days
27028733	60 minutes	5.661	232 days
29481728	60 minutes	4.160	172 days

A. Comparison Schemes and Strategy

Header Hash Compression is considered as an optional step before Block or Stream compression using generalized deduplication. We compare the performance of Header Hash against the built-in Null and Delta compressors, and see what happens when all three are used at the same time.

GDD Block and Stream compressors are measured on raw and pre-compressed APDUs and compared against the commonly used Lempel–Ziv–Welch (LZW) algorithm. We apply LZW to each APDU separately when evaluating our Block compressor, and run it on all APDUs at once for a fair comparison against our Stream compression method. This technique simulates a **stream-optimized version of LZW** where the dictionary of unique substrings is built using all

APDUs instead of starting a new one for each. This way LZW can also leverage information sent in previous messages of the stream, just like our GDD-based stream compressor does. In our method however, the decompressor can operate with much less memory allocation, since a common lookup table can be used for all meters, while a streaming LZW implementation would have to build a separate dictionary per meter.

We evaluate our GDD-based compressors using two **transformation functions**: Hamming code with parity bits 5, 6, 7 and 9, and Reed-Solomon error correction code with chunk sizes of 8, 16, 32, 50, 64 and 100 bytes. The number of parity bits used in Hamming code also determine the chunk size of processing input APDUs. Table III summarizes the tested parameters of both algorithms. When measuring the compression rate with a GDD-based compressor, we run every configuration of both Hamming and RS transformation functions, and report the best one of each, which usually comes from a subset of the parameter space. For example, Hamming parity settings 6, 7 and 8 produce the best GDD Block compression results, as illustrated by Table IV.

Meter readings are converted to APDUs with different **time window lengths**. We evaluate our compression methods using six time periods for reporting: 1, 2 and 4, as well as 12, 24, and 48 readings per APDU. For example, 4 measurements in an APDU where the meter is recording once every 15 minutes can be interpreted as converting and uploading the buffer of th meter once every hour. In our tests we compressed all APDUs from each meter and report the average compression rate, as well as the minimum and maximum rates across the five meters as error bars.

TABLE III: Evaluated Hamming and RS code configurations

Hamming configurations			
Parity bits	**Chunk size**	**Base size**	**Deviation size**
5	4 bytes	26 bits	6 bits
6	8 bytes	57 bits	7 bits
7	16 bytes	120 bits	8 bits
8	32 bytes	247 bits	9 bits
9	64 bytes	502 bits	10 bits

Reed-Solomon configurations		
Chunk size	**Base size**	**Deviation size**
8 bytes	6 bytes	3 bytes
16 bytes	14 bytes	3 bytes
32 bytes	30 bytes	4 bytes
50 bytes	48 bytes	4 bytes
64 bytes	62 bytes	4 bytes
100 bytes	98 bytes	4 bytes

TABLE IV: Hamming parity bits setting with the best GDD Block compression

Rows per APDU	**1**	**2**	**4**	**12**	**24**	**48**
No pre-compression	8	8	7	7	6	6
Null value + Delta array	8	8	8	7	7	7
Header hash	6	6	6	6	6	6

B. Header Hash Compression

This compression technique is extremely effective when the header takes up most of the message. Since the data part

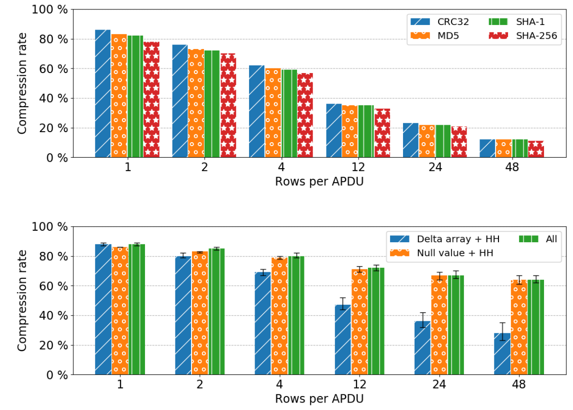

Fig. 4: Header Hash scheme with different hash functions (above) and combined with other compressors (below)

of the APDU is not compressed in any way, the gains are determined by the header-to-data size ratio within a message. The performance difference of using different hash functions is a few percents at best, and almost nothing when the header is relatively small. In the following, we will evaluate the header hash compressor with CRC32, since the collision rate of this hash function is acceptably low in real life scenarios, and its compression performance is always better than the other algorithms.

In our analysis we evaluated four widely used hash functions with different output lengths: CRC32 (4B), MD5 (16B), SHA-1 and SHA-256 (20B and 32B). Figure 4 shows the resulting compression rates of APDU messages. This compressor can be performed either on raw APDUs, or together with Null and/or Delta compression. Since Null compression works across the whole APDU, it helps keeping the compression rate high even when the header becomes relatively small compared to the data part of a message. Delta and Header hash compressors also improve each other, but not as much as Null compressor does. The combination of all three methods is marginally better than using just null and header hash together.

Overall, the header hash compression brings tremendous gains when meter readings are uploaded frequently, and the large protocol header dominates the message size. Since computing the CRC32 hash of a few hundred bytes is computationally very cheap, this compression method has high potential in the DLMS protocol.

C. GDD Block Compression

First, we measure compression rate of raw APDUs. As shown on Figure 5, the block compressor shows very different performance with Hamming and Reed Solomon transformation functions. While Hamming code shows a predictable compression rate of around 20%, RS code struggles to compress and even produces larger messages due to padding. LZW performance heavily depends on the size of the APDU: it can barely compress small messages, but even surpasses Hamming-based GDD on larger ones.

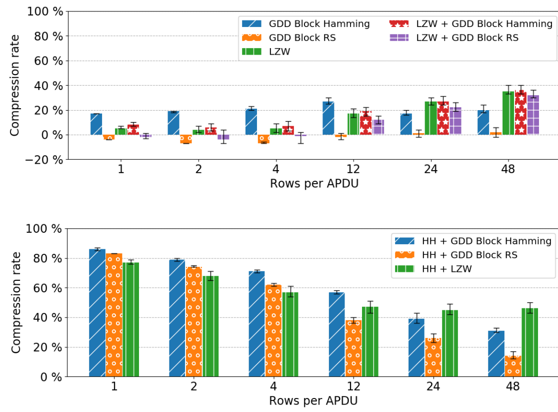

Fig. 5: GDD Block compressor compared to LZW, on raw APDUs (above) and on pre-compressed ones (below)

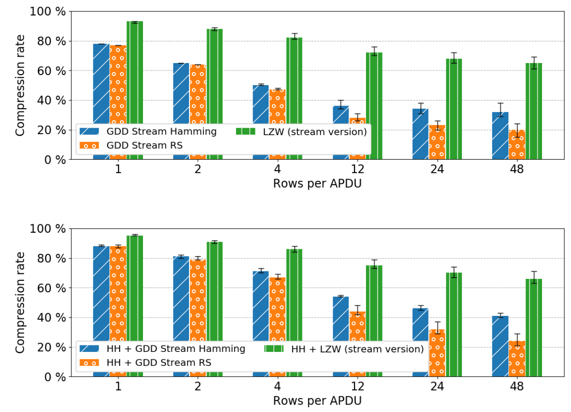

Fig. 6: GDD Stream compressor compared to LZW, on raw APDUs (above) and on pre-compressed ones (below)

It is also interesting to see what happens if we run the same algorithms on APDUs pre-processed with Header Hash compressor. The trend is similar to the raw APDU measurements: GDD is significantly better with small data, while LZW can achieve higher rates with more data. Remarkably, even RS performs bettern than LZW on APDUs with up to 4 measurements.

D. GDD Stream Compression

Our stream compressor is evaluated assuming all bases from previous APDUs fit into the lookup table on the meter and bases are always replaced by their CRC32 hash when they appear again in a later APDU. If a base is repeating both within and across APDUs, it is represented by a local index that points to the first occurrence within the same message.

Figure 6 shows that the GDD-based compressors perform very similarly. When the message contains a small number of readings, both Hamming and RS code achieve high gains, which is decreasing as there are more measurements in a single message. The compression rate of LZW also declines with larger message size, but it stays above 60% in all tested scenarios. Even though GDD compressors never reach as high compression as LZW, their decompressor counterpart has a significantly lower memory requirement. While the LZW decompressor has to maintain a separate dictionary for each meter, the GDD-based component can leverage the fact that the same base on different meters produce identical hash values. Therefore, a single global hash lookup table can be used for APDUs coming from different meters.

When applied on pre-compressed APDUs, both Hamming and RS-based GDD stream compressors improve the overall compression rate. Applying the stream version of LZW on top of Header Hash compression also results in extraordinary size reductions of over 90% with small APDUs, and around 70% with larger ones.

VI. Conclusions

The proliferation of smart electricity meters bring numerous advantages to the utility providers and come with a plethora of technical challenges. Increasing the volume and frequency of data being uploaded from the smart meters to the utility's Head-End Systems is getting more and more challenging. In this paper, we have shown three compression techniques which significantly decrease the amount of data that has to be sent through the network from the meters, while eliminating the privacy concerns of the existing compression methods. We have shown that the proposed methods can be applied transparently to the existing DLMS APDUs from Kamstrup smart meters.

References

[1] S. Tripathi and S. De, "An efficient data characterization and reduction scheme for smart metering infrastructure," *IEEE Trans. on Industrial Informatics*, vol. 14, no. 10, pp. 4300–4308, 2018.

[2] A. Unterweger and D. Engel, "Resumable load data compression in smart grids," *IEEE Trans. on Smart Grid*, vol. 6, no. 2, pp. 919–929, 2014.

[3] T. Sirojan, S. Lu, B. Phung, and E. Ambikairajah, "Embedded edge computing for real-time smart meter data analytics," in *Int. Conf. on Smart Energy Systems and Technologies (SEST)*. IEEE, 2019, pp. 1–5.

[4] J. C. S. de Souza, T. M. L. Assis, and B. C. Pal, "Data compression in smart distribution systems via singular value decomposition," *IEEE Trans. on Smart Grid*, vol. 8, no. 1, pp. 275–284, 2015.

[5] J. Kraus, P. Štěpán, and L. Kukačka, "Optimal data compression techniques for smart grid and power quality trend data," in *IEEE 15th Int. Conf. on Harmonics and Quality of Power*. IEEE, 2012, pp. 707–712.

[6] Y. Lee, E. Hwang, and J. Choi, "A unified approach for compression and authentication of smart meter reading in AMI," *IEEE access*, vol. 7, pp. 34 383–34 394, 2019.

[7] A. Abuadbba, I. Khalil, and X. Yu, "Gaussian approximation-based lossless compression of smart meter readings," *IEEE Trans. on Smart Grid*, vol. 9, no. 5, pp. 5047–5056, 2018.

[8] R. Vestergaard, D. E. Lucani, and Q. Zhang, "Generalized deduplication: Lossless compression for large amounts of small IoT data," in *European Wireless Conference*. VDE, 2019, pp. 1–5.

[9] N. Yazdani and D. E. Lucani, "Protocols to reduce CPS sensor traffic using smart indexing and edge computing support," in *IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2019, pp. 1–6.