Dixy: Transparent Payload Compression for Constrained Networks

Ichrak Kallala, Trifun Savic, Quentin Lampin, Marion Dumay, Stephane Coutant, Cédric Adjih, Paul Muhlethaler, Thomas Watteyne

Dixy: Transparent Payload Compression for Constrained Networks

Ichrak Kallala*, Trifun Savić‡, Quentin Lampin†, Marion Dumay†, Stephane Coutant†,
Cedric Adjih*, Paul Muhlethaler*, Thomas Watteyne*
* Inria, France
† Orange Lab, Meylan, France
‡ Wattson Elements/Falco, Paris, France
e-mail: first.last@{inria.fr*,orange.com†,wefalco.fr‡}

***Abstract*—This paper introduces Dixy, a solution specifically crafted for compressing the application payload of a low-power wireless networking protocol. Dixy is complementary and can be used in parallel to standardized header compression solutions such as 6LoWPAN or SCHC. Dixy operates as a dictionary compression by identifying a list of patterns and analyzing recently transmitted packets, but without ever having to explicitly share that dictionary with the receiver, as the receiver re-builds the same dictionary from the stream of packets it receives. We show how Dixy outperforms the closest related work by offering a compression factor 29% better by comparing the performance of multiple approaches on a dataset of 1,154,176 packets collected from 614 devices in a real-world commercial deployment. Dixy can be integrated seamlessly with payload security, and made standards-compliant.**

***Index Terms*—constrained networking, compression, SCHC.**

I. Introduction

Low-power wireless networks are composed of battery-powered devices which typically send data in a periodic fashion. The frames they exchange are typically short, in the order of 128 B at most, so reducing the length of the frame to transmit over the radio, even by a handful of bytes, results in a very valuable increase in battery lifetime. This is the reason why compression is so appealing in these networks.

We define "compression" as the process of removing bytes from the transmitted data while preserving all of the information it contains.

There are two primary types of compression: lossless and lossy. Lossless compression, which is the focus of this work, reduces the size of data packets without any loss of information, ensuring that the original data can be fully reconstructed from the compressed packets. On the other hand, lossy compression achieves higher compression ratios by permanently discarding certain non-essential data, resulting in smaller packet sizes but with some loss of information.

And we differentiate *header* compression from *payload* compression. The former has received lots of attention, in particular by the standardization bodies, as they are the ones that define the formats of those headers. As we review in detail in Section II, a standardization body such as the Internet Engineering Task Force (IETF) has standardized header compression solutions such as 6LoWPAN [1], 6LoWPAN-GHC [2] or SCHC [3]. Payload compression, however, hasn't received nearly as much attention, which is all the more surprising as the payload makes up a very significant portion of the frame.

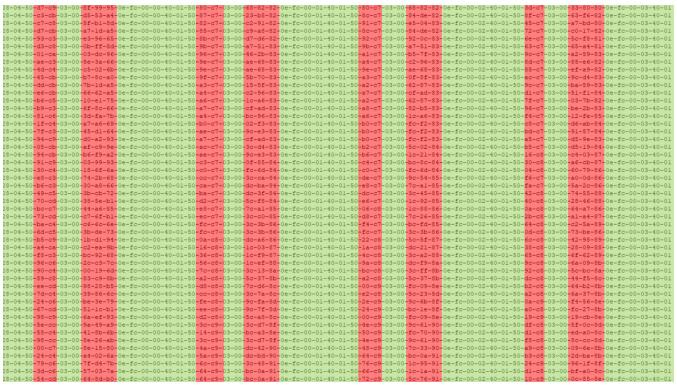

Fig. 1. Displaying successive payloads generated by a node on different lines show how most of the contents remains the same. Only the portions of the packets that are highlighted red are changing. Removing the green portions in this example reduces the number of bytes to transmit from 58 to 20, or a compression factor of 2.9. Dixy uses a dictionary-based technique to achieve exactly this: compress away the portions of the payload that remain the same across packets.

To illustrate the potential of compression, let's take Fig. 1. It shows successive payloads generated by a particular device (it is part of the Falco dataset we present in detail in Section IV). We show one packet per line. We color-code the bytes: only the red bytes change from packet to packet; all the green ones remain the same in each packet. This is very common: the payload consists of the information bundled into a data representation that specifies for example the format of the field, and is repeated for each packet. Protocols such as the JavaScript Object Notation (JSON) [4] or Concise Binary Object Representation (CBOR) [5] function like this. The result is that there is a lot of opportunity for compression: in Fig. 1, removing all the "green bytes" would reduce the number of bytes to transmit from 58 to 20, resulting in a compression factor of 2.9.

This paper introduces Dixy, a compression technique designed to achieve just that, using a form of dictionary-based compression. A Dixy transmitter maintains a buffer of previous payloads and identifies the patterns that often repeat. When transmitting a new packet, Dixy removes all the bytes that are

part of the patterns, leaving only the bytes that change from packet to packet, and pre-pending a 1-byte header to identify which pattern was added. The power on this technique is that the receiver side makes the same decisions on the same packet buffer: the pattern list (the "dictionary") never needs to be transmitted explicitly and always adapts.

The contributions of this paper are three-fold.

- We provide a survey of compression techniques for constrained networks, both for compressing the header and the payload.
- We describe the Falco dataset, composed of 1,154,176 packets collected from 614 devices in a real-world commercial "smart marina" deployment.
- We introduce Dixy, a transparent payload compression technique which achieve 29% higher compression than related work.

The remainder of this paper is organized as follows. Section II surveys the related work in compression for constrained networking. Section III introduces Dixy, and details how it works. Section IV describes the Falco dataset used to evaluate Dixy. Section V show how Dixy achieves 29% better compression that the closest related work. Section VI concludes the paper by summarizing its contributions and discussing current work on security, standards-compliance, and adapting to specific protocol stacks.

II. Related Work

Low-power wireless devices such as sensors tend to periodically send data. Because of their battery lifetime requirements, removing even a handful of bytes from each packet results in a significant lifetime increase. As a result, compression for low-power wireless communication has been a focus of standardization activities for a number of years. What's interesting is that the resulting solutions tend to only focus on *header* compression, typically the headers on the link, networking and transport layers. Very little related work has focused specifically on *payload* compression, which we believe is an oversight as the payload can take up very significant portion of the bytes making up the frame. In SmartMesh IP for example [6], the payload takes up to 90 of the 127 bytes of a frame.

This section surveys related work, both for header and payload compression.

Header compression has been a focus of research and development, particularly under the direction of the IETF, the standards group that underpins many of today's Internet protocols. Several header compression approaches have been developed for low-power, resource-constrained wireless networks with the goal of improving communication efficiency.

The 6LoWPAN protocol ("IPv6 over Low-Power Wireless Personal Area Networks") compacts IPv6 and UDP headers. In 6LoWPAN, a dedicated header replaces the IPv6 and UDP headers and contains instructions for recreating each field. This protocol employs a prefix/address dictionary, which allows a 16-byte IPv6 address to be substituted by a 4-bit index. Commonly used prefixes, such as `fe80::`, are compressed using single-bit flags. Although some data, such as the IPv6 length field, can be regenerated using other headers, others, such as Time-to-Live (TTL), stay constant.

TABLE I
Example SCHC rule, which compresses IPv6
(reproduced and adapted from RFC8724 [3], Fig. 27).

FID	FL	TV	MO	CDA
IPv6 Version	4	6	ignore	not-sent
IPv6 Diffserv	8	0	equal	not-sent
IPv6 Flow Label	20	0	equal	not-sent
IPv6 Length	16		ignore	compute-*
IPv6 Next Header	8	17	equal	not-sent
IPv6 Hop Limit	8	255	ignore	not-sent
IPv6 DevPrefix	64	[alpha/64, fe80::/64]	match-mapping	mapping-sent
IPv6 DevIID	64		ignore	DevIID
IPv6 AppPrefix	64	[beta/64, alpha/64, fe80::64]	match-mapping	mapping-sent
IPv6 AppIID	64	::1000	equal	not-sent

An extension to 6LoWPAN, known as 6LoWPAN-GHC [2] ("Generalized Header Compression") efficiently handles different types of headers without redefinition. This dictionary-based method substitutes matching header patterns with predefined indices, using a single static dictionary.

The Generic Framework for Static Context Header Compression and Fragmentation (SCHC) [3] was initiated due to the emergence of LPWAN technologies. The LPWAN networks in question have limited payload capacities, making it impractical to include 6LoWPAN-style flags. Additionally, these networks are often upstream-only, lacking the ability to negotiate context between the LBR and devices. SCHC relies on static contexts defined by a set of rules to compress packet headers. An example of a rule for compressing IPv6 and UDP headers can be seen in Table I. Each rule aims to match the entire packet header, with separate rows specifying how to handle each field in the packet.

To determine whether a rule applies to a given packet, the compressor checks if the field values match the "Target Value" (TV) provided in the corresponding row, using the specified "Matching Operator" (MO). If there is a match, the compressor follows the instructions outlined in the "Compression/Decompression Action" (CDA) column.

These compression actions might include omitting the field and reconstructing it on the receiver's end based on the TV (CDA `not-sent`), omitting the field and recalculating it using a known method based on other fields in the packet (CDA `compute-*`), or sending the index of the value from an array (CDA `mapping-sent`). By utilizing a set of tailored rules, SCHC customizes compression for each packet flow, providing more flexibility compared to 6LoWPAN that uses the same compression method for all UDP/IPv6 headers.

In summary, several header compression methods exist and have been standardized. Protocols like 6LoWPAN are specifically designed to compress the protocol they are associated with. On the other hand, 6LoWPAN-GHC is a more versatile option that allows compression of various protocols,

although it may not be as efficient. Notably, the recent SCHC distinguishes itself by separating policy (rules that form a compression context) from mechanism (compression process based on the context). For a comprehensive analysis of header compression, the interested reader is referred to [7].

Far less proposals have been made specifically for payload compression. This may be due to the fact that compression has been looked at most closely in the standardization organizations, and that those focus – by definition – on headers. Massey *et al.* did propose a protocol-agnostic compression approach as early as 2010 [8], [9]. The idea was to treat the compression routine as separate from the data format itself. In this approach, the protocol stack on a device generates a frame for transmission, which is then compressed just before being sent to the radio. This method is similar to how file compression algorithms like zip work, as the compressor doesn't require knowledge of the file's contents. This distinction sets it apart from the 6LoWPAN or SCHC header compression approaches mentioned earlier.

Massey's compression technique utilizes a dictionary-based approach to achieve compression. The compressor operates on a continuous stream of packets, processing them sequentially. It does a byte-wise comparison between a packet and each of previous ones in a recent packet buffer. If a sequence of consecutive bytes is found to be identical, it is identified as a "pattern". This pattern is then added to a pattern list, which acts as the dictionary, and is replaced by its corresponding index in the list. Compression stems from the index being much shorter than the pattern. The algorithm has three parameters: the minimal length of a pattern, the length of the pattern list and the length of the pattern buffer. By adjusting these parameters, the algorithm can be fine-tuned to achieve the highest possible compression while maintaining an acceptable memory footprint in Random-Access Memory (RAM).

The approach's strength lies in the fact that the pattern list (the dictionary) is not explicitly sent from the compressing device to the decompressing device. Instead, by running the same algorithm on the same packets, they both make identical decisions simultaneously, resulting in the creation of the same dictionary. Note that Massey *et al.* intended their technique as a replacement for other header compression techniques. In this paper, we build upon Massey's technique by using it for payload compression, and by adapting its internal routines to payload formats. The results is a 29% increase in compression factor when applied on a real-world dataset of 1,154,176 packets collected from 614 devices.

III. Dixy

Dixy is a compression routine which operates in a transparent manner on payloads. This is illustrated in Fig. 2. Dixy operates between the application layer and the remainder of the networking stack. That is, the application layer generates a datagram of data and wishes to transmit that over a network. Instead of handing that datagram directly to the protocol stack, it hands it to Dixy which compresses it, adds its own 1-byte header, and sends it to the networking stack. Similarly, on the receiver side, Dixy retrieves the payload as received by the networking stack, removes the Dixy-specific header, decompresses the data, and hands it to the application. This means that Dixy is entirely transparent to both the application and the networking layers: it can be introduced without requiring any changes to either.

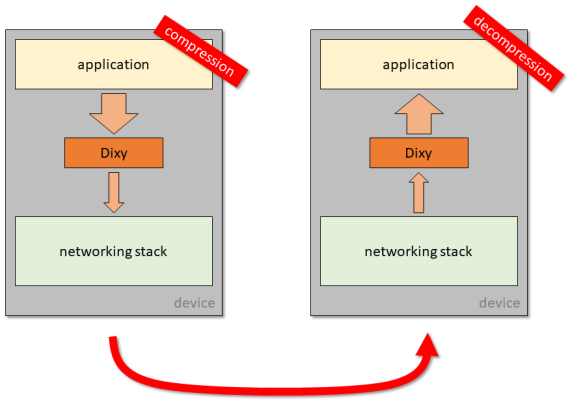

Fig. 2. Dixy operates in a transparent way between the application at the networking stack. That is, neither the application nor the networking layers need to change to support Dixy, or even be made aware Dixy is being used.

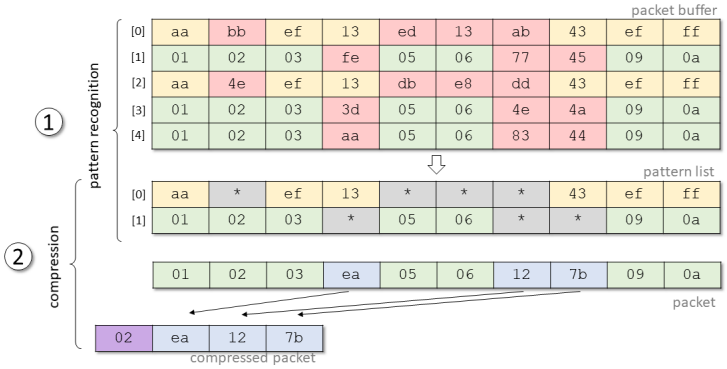

Fig. 3. Illustrating how Dixy works by presenting example contents for its internal "packet buffer" and "pattern list" structures, and how those are used to compression a 10 B packet into a 4 B compressed packet.

We use Fig. 3 as a visual aid when describing how Dixy works. A Dixy implementation maintains two structures in RAM memory: a packet buffer and a pattern list. The packet buffer holds the last P packets which the device has sent. In Fig. 3, $P = 5$. This structure hence has a memory footprint of $5 \times 128 = 640$ B in an IEEE802.15.4 systems, as IEEE802.15.4 frames are at most 128 B long. The pattern list holds a list of patterns that have been "recognised" in the packets as repeating. Each pattern is the same length as the longest packet.

On the transmitter side, Dixy operates in two steps: compression and pattern recognition.

When the application operating in a devices wishes to transmit a datagram, Dixy attempts to match it to each of the patterns in the pattern list, in order. Because every pattern is the same length as the longest packet, uniform handling and

compatibility are guaranteed. Dixy handles two primary tasks on the transmitter side: pattern identification and compression. Dixy tries to match a datagram against each pattern in the pattern list one after the other. This idea is illustrated visually in Fig. 3 with color-coded bytes. A series of bytes make up each pattern, some of which could act as "wildcards" (shown with a "⋆" character in Fig. 3) that can match any value. When a packet's bytes exactly match the fixed bytes of a pattern – wildcard bytes excluded – the packet is said to "match" the pattern. The packet that needs to be compressed in Fig. 3 corresponds to pattern at index `[1]` in the pattern list. Only the bytes from the packet at the wildcard places are retained if a match is found; the remaining bytes are discarded. As an example, only the three blue-colored bytes are kept in Fig. 3.

A 1-byte header is pre-pended to the packet; it consists of a bitmap indicating the pattern that was used. In the case of Fig. 3, the bitmap contains `0b00000010==0x02`, indicating the pattern at index `[1]` in the pattern list was used to compress. In the example in Fig. 3, the pattern was compressed from 10 B to 4 B.

Once a packet is compressed, the initial (uncompressed) packet is added to the packet buffer. It is done with a First-In-First-Out strategy: a new packet replaces the one that has been in the packet buffer the longest. In the implementation, duplicates in the packet buffer are avoided: if all bytes match between the recently compressed packet with one already in the packet buffer, it is not added again. Each time a packet is added to the packet buffer, the pattern recognition routine is run. The goal of the pattern recognition is to find two or more packets that "look the same". This is done by trying all combination of two packets, and remembering the bytes that are the same. We call those candidate patterns. The pattern list is then composed of the candidate patterns with the least wildcard bytes.

In Fig. 3, we have color-coded the bytes for easier reading. Packets at indexes `[0]` and `[2]` have six common bytes. Those become the pattern `[0]`. Packets at indexes `[1]`, `[3]` and `[4]` have seven common bytes: they result in pattern `[1]`.

Dixy is different from the proposal made by Massey *et al.* [9], [8] in two fundamental ways. First, Massey *et al.* designed their approach as a replacement for header compression; Dixy is targeted only at payload compression and can therefore be used in parallel to a header compression routine (see also the discussion in Section VI). Second, the patterns in Massey's proposal are short, typically only a handful of bytes long; Dixy is designed to identify patterns which span an entire packet, with wildcards in-between bytes. This makes it particularly good at compressing payloads formatted using data representations such as CBOR [5].

IV. Falco Dataset

Falco[1] is a start-up company that equips marinas with low-power wireless sensors to assist the marina operators and allow boat owner to know, in real time, how their boat is doing. There are different types of sensors generating different types of data. Some sensors are installed under the pontoons and detect whether a berth is occupied. Some are installed inside the electric pedestal and help ensure electricity is correctly used. Some are installed directly in the boats to raise an alert in case a fire starts, someone break in, or a mooring breaks, and to monitor the temperature and humidity inside the boat. This results in a large deployments with several very different sensors generating different types of data.

[1] https://wefalco.com/en_us/

TABLE II
The Falco dataset.

number of devices	614 devices
duration of the recording	3.909 days
number of packets collected	1,154,176 packets
number of packets per device (min / median / max)	1,050 / 1,118 / 11,684 packets
payload length (min / median / max)	4 / 29 / 58 B

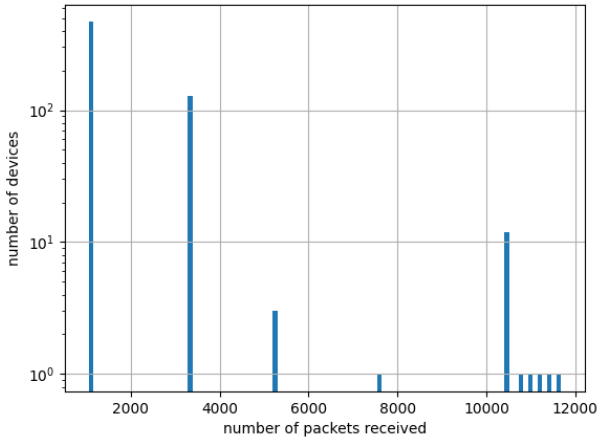

Fig. 4. Histogram of the number of packets received per device.

While all data is encrypted, we have instrumented one of the deployments to log all payloads of the wireless packets the device generated. This was done for approximately 4 days, during which the 614 devices have collectively generated over one million packets.

Table II provides high-level statistics about the Falco dataset. Over the 3.909 days that the logging happened, each of the 614 devices has generated between 1,050 and 11,684 packets, with a median of 1,118. Fig. 4 is a histogram of the number of packets generated by the devices.

Fig. 1 is generated by displaying successive packets from a given device, one the line, and coloring red the bytes that are different between successive packets, green the bytes that remain the same. Visually, Fig. 1 shows that removing the green portion of the payloads results in good compression.

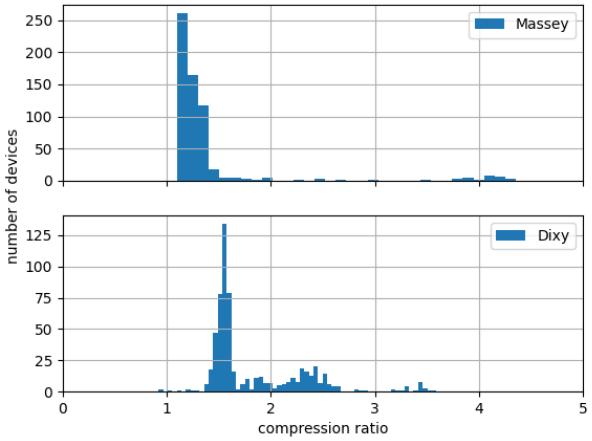

Fig. 5. Distribution of the compression factor for each of the 614 devices in the dataset.

TABLE III
COMPRESSION FACTOR ON THE FALCO DATASET.

	Massey *et al.* [9], [8]	Dixy
min	1.100	0.920
median	**1.224**	**1.582 (+29%)**
max	11.268	5.298

V. DIXY PERFORMANCE

The goal of this section is to quantify the main performance indicator of Dixy, its "compression factor", when using Dixy on the Falco dataset.

We call "compression factor" the ratio between the number of bytes in the initial (uncompressed) packet and the compressed packets, as computed by (1). A compression factor of 2.5 means that a 25 B packets was compressed down to 10 B. The compression factor is higher than 1 if the compression routine indeed removes bytes from the packet.

$$compression\ factor = \frac{num.\ bytes\ initial}{num.\ bytes\ compressed} \quad (1)$$

We implement Dixy in Python, as well as the proposal from Massey *et al.* [9], [8]. From all the related work surveyed in Section II, only the one from Massey *et al.* can be directly compared to Dixy, as it can be used to compressed an opaque set of bytes such as a payload. Other compression mechanisms (e.g. 6LoWPAN, 6LoWPAN-GHC, SCHC) all rely on the data to conform to a particular field format, which is the case for a protocol header but not for its payload. We run those implementations against the Falco dataset. For each solution, we record the resulting compression factor for each of the 614 devices.

Fig. III shows the histogram of the compression factor for each of the 614 devices in the Falco dataset, for both the proposal by Massey *et al.* (top) and for Dixy (bottom). A higher the compression factor is better. That is, a distribution more to the right of the histogram is better. We can see visually that Dixy outperforms Massey *et al.*. Table III provides a summary: while Massey *et al.* yield a compression factor of 1.224 for the median device, Dixy yields a compression factor of 1.582 for the median device, a 29% increase.

This means that, assuming a 127 B frame with a 90 B payload (the numbers used in Section II for SmartMesh IP), Massey *et al.* would compress the frame to 111 B, Dixy to 94 B.

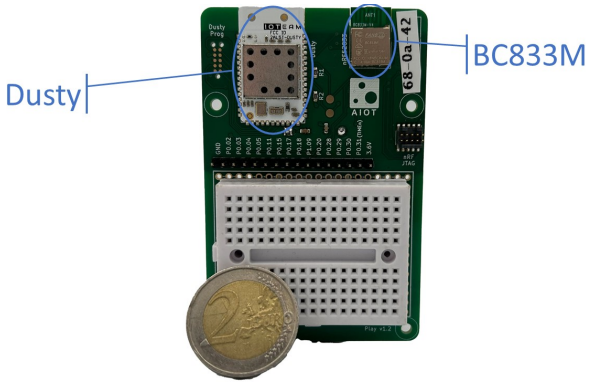

Fig. 6. The AIOT Play board features a BC833M module on which we implement Dixy, and a Dusty module which serves as an off-the-shelf low-power wireless networking modem.

TABLE IV
BATTERY LIFETIME WHEN USING DIXY.

duration of the experiment		13.88 h
number of packets generated		50,000
90 B packets (no Dixy)	num. bytes transmitted	4,500,000 B
	charge consumed	16,602 mC
	average current draw	332.25 μA
	battery lifetime (2xAA)	1 year
63 B packets (median compression of Dixy)	num. bytes transmitted	3,150,000 B
	charge consumed	15,083 mC
	average current draw	301.85 μA
	battery lifetime (2xAA)	1.1 years
20 B packets (max compression of Dixy)	num. bytes transmitted	1,000,000 B
	charge consumed	13,173 mC
	average current draw	263.62 μA
	battery lifetime (2xAA)	1.2 years

In addition, we conduct experiments using the `PkGen` application running on the AIOT Play board 6, generating 50,000 packets at a rate of one per second, with varying payload sizes of 20 B, 63 B, and 90 B, alongside approximately 30 B of headers. We analyze the impact of Dixy compression on charge consumption, goodput, and overall battery lifetime within a SmartMesh environment. Results are obtained through a comprehensive set of measurements, including a table showcasing battery lifetime when using Dixy (Table IV) and a graph illustrating the total charge consumed across different payload sizes (Fig. 7). These findings provide valuable insights into the efficiency of Dixy compression in optimizing power consumption and extending the operational lifespan of AIOT Play devices.

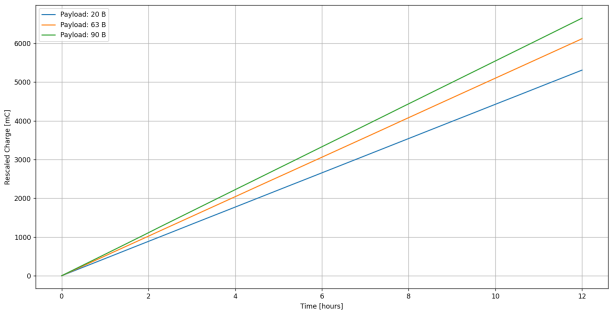

Fig. 7. Total charge consumed with the payload sizes of 90 B, 63 B and 20 B. AIOT Play sends a packet every 1,000 ms, for 12 h.

VI. Discussion

This paper introduces Dixy, a compression technique targeting specifically the payload of a low-power wireless frame. While header compression has received a lot of attention, in particular from standardization organizations, compressing the payload has received very little attention, even if that represents a significant portion of the frames. Dixy works by identifying patterns in a buffer of past payloads, then compressing a packet by removing all bytes common with one or more patterns. Because the receiving node makes the same decision on the same buffer of packet, the pattern list never needs to be explicitly transmitting. Also, Dixy operates transparently for both the application and the networking stack. We compare the performance of Dixy against previous proposal by Massey *et al.* by applying both on a dataset of 1,154,176 packets collected from 614 devices. We find that Dixy outperforms Massey *et al.*'s proposal by 29%.

We are working on three research paths for Dixy.

First, integration with security. In constrained networks, one part of securing a frame is encrypting the payload. A good cipher will transform the bytes in a the payload so it looks like a random series of bytes, which makes compression inefective. We are working on applying Dixy *before* encryption. On the transmitter side, the cleartext is compressed, then encrypted. On the receiver side, the encrypted payload is decrypted, then decompressed. This allows for both security and generic compression.

Second, standards compliance. Protocols such as SCHC define the generic `compute-*` Compression/Decompression Action (CDA). The flexibility of this CDA allows Dixy to run in a standards-compliant manner: the compression of the payload (using Dixy) is "just" another row in a list of rules such as the one shown in Table I.

Third, adaptation to specific network stacks. Some networks with not offer wire-like reliability. Others come with more or less communication overhead. We are working on hands-on evaluation Dixy on several established networking stacks to identify the right adaptation.

References

[1] J. Hui and P. Thubert, *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*, Internet Engineering Task Force (IETF) Std. RFC6282, 2011.

[2] C. Bormann, *6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*, Internet Engineering Task Force (IETF) Std. RFC7400, 2014.

[3] A. Minaburo, L. Toutain, C. Gomez, D. Barthel, and J. Zuniga, *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*, Internet Engineering Task Force (IETF) Std. RFC8724, 2020.

[4] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, Internet Engineering Task Force (IETF) Std. RFC8259, 2017.

[5] C. Bormann and P. Hoffman, *Concise Binary Object Representation (CBOR)*, Internet Engineering Task Force (IETF) Std. RFC8949, 2020.

[6] T. Watteyne, L. Doherty, J. Simon, and K. Pister, "Technical Overview of SmartMesh IP," in *International Workshop on Extending Seamlessly to the Internet of Things (esIoT)*, Taiwan, 3-5 July 2013.

[7] M. Tömösközi, M. Reisslein, and F. H. P. Fitzek, "Packet Header Compression: A Principle-Based Survey of Standards and Recent Research Studies," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 1, pp. 698–740, 2022.

[8] T. Massey, A. Mehta, T. Watteyne, and K. Pister, "Protocol-Agnostic Compression for Resource-Constrained Wireless Networks," in *IEEE Global Telecommunications Conference (GLOBECOM)*, Miami, Florida, USA, 6-10 December 2010.

[9] ——, "Packet Compression for Time-Synchronized Wireless Networks," in *IEEE Conference on Sensor, Mesh and Ad Hoc Communcations and Networks (SECON)*, Boston, MA, USA, 21-25 June 2010.