Attaching Semantic Metadata to Cryptocurrency Transactions

Luis-Daniel Ibáñez, Huw Fryer, and Elena Simperl

University of Southampton, Southampton, UK,
[l.d.ibanez|h.fryer|e.simperl]@soton.ac.uk

Abstract. Cryptocurrencies like Bitcoin have provided a platform for parties to transfer digital value in a decentralised way. However, to support scalability, the amount of extra data that can be attached to a transaction is very limited, greatly limiting the possibility of attaching descriptive metadata to them with the same guarantees that the Blockchain In this paper we study the problem of attaching metadata in RDF to Bitcoin-style transactions, where extra data is limited to 83 bytes per transaction, linking the problem to the one of RDF-Compression. However, the motivating scenarios for RDF compression (IOT devices and Big Data Publish/Exchange) do not have the same requirements as ours, as memory is extremely limited, but the input data is small and there is no need for energy efficiency or providing a query interface. We evaluated the size reduction of 7 RDF compression algorithms on two small documents (4 and 10 triples), finding that only two of them on 4 triples and none on 10 triples were able to improve over a naive gzip compression. Our findings motivate the need of further research on adequate representations for this particular use case.

Keywords: Blockchain, Cryptocurrency, Bitcoin, RDF Compression

1 Introduction

Cryptocurrencies have revolutionized the way we transact digital assets and value. Instead of using a bank as trusted intermediary to transfer value from a sender to a receiver, cryptocurrencies provide a decentralized and dis-intermediated environment in which a network of peers that do not necessarily trust each other can keep a ledger of the transactions between them. The protocols designed for this purpose provide the following desirable features: (I) Increases the difficulty for an attacker to maliciously delete or inject transactions in the Ledger, from needing only to compromise the central trusted node, to needing to compromise a large number of peers and/or computing power in the network. (II) Thanks to the fact that the ledger is fully replicated across a large subset of the peers in the network, transactions recorded in the ledger are more persistent, that is, for them to become lost, all nodes with a full replica will need to disappear, in contrast with only the trusted node in a central scheme.

However, whenever a transaction is made, we are often interested in its context: What is the purpose of the transaction? Is a transaction related to other transactions, how? What are the links of a transaction with entities outside the Blockchain? The Resource Description Format (RDF) provides the means to write both descriptions and links in a standardized way, but how to attach an RDF document to a cryptocurrency transaction?

Unfortunately, one of the sacrifices that cryptocurrency designers had to make to achieve the above properties was to reduce as much as possible the size of transactions. This means that only few bits of metadata about the transaction are recorded: a transaction identifier, the recipient addresses, the amount transferred and the status of the transaction in the network. There is generally, very little space to attach any metadata to the transaction. For example, in Bitcoin, and in all its derivate or inspired-by cryptocurrencies, this space is limited to one field of 83 bytes, which is in many cases not enough for even one RDF triple.

Services like CoinSpark [1], that work on top of Bitcoin, use this field to store a cryptographic hash of a document describing the context of the transaction that is then stored off-blockchain. The point is that the external storage does not need to be trusted in not tampering the document, as anyone could easily check integrity against the hash stored with the transaction. However, the external storage needs to be trusted in preserving the document, as in the case it gets lost or corrupted, it cannot be reconstructed from the hash. With this approach, the context does not have the same persistence guarantees than the transactions.

In this paper we aim at answering the question: How to store the RDF description of a cryptocurrency transaction in the same Ledger as the transaction itself? Different cryptocurrencies have different transaction metadata sizes and policies, we chose to focus on the Bitcoin parameters, a single 83 bytes field free of charge per transaction, for two reasons: 1) Bitcoin is currently the most used cryptocurrency [2] 2) Many other popular cryptocurrencies derived or inspired from Bitcoin (Litecoin, Dogecoin, Namecoin, Zcash) use the same parameters.

Our contributions are summarized as follow: 1) We identify the problem as a variant of compression of RDF documents, and compare it with two other scenarios that motivated it: adding semantics to IoT devices and publication and exchange of Big Semantic Data. We discover that the our scenario is not equivalent to any of the other two, having an extremely limited memory (less than a IoT device), but a small input data size (like IoT devices) and relaxed constraints on energy consumption and computational resources (similar to the Publish/Exchange scenario) 2) On the light of 1), we compare 7 approaches for RDF compression on two small documents (10 and 4 triples) describing the provenance of a transaction to determine how well they generalise to the characteristics and size of documents that we expect to be attached to transactions, and to an scenario where compression ratio is the most important criterion. Somewhat surprisingly for us, we found that none of them performed better

[1] http://coinspark.org/

[2] measured in market capitalisation on 15th July 2017 as for https://coinmarketcap.com

than a naive gzip compression over Ntriples for 10 triples, and only two of them (RDSZ and EXI with concrete grammar) performed better for 4 triples.

The paper is organised as follows: section 2 provides an overview of Blockchains, Cryptocurrencies and Bitcoin transactions; section 3 describes previous efforts on embedding data in Bitcoin, carried out by users and commercial service providers; section 4 compares the Cryptocurrency transaction motivation for RDF-compression with the the IoT and Publish/Exchange scenarios; section 5 describes our experimental setup and results; section 6 concludes the paper and provides future work directions.

2 Blockchains and Bitcoin-like cryptocurrencies

A *Blockchain* is a reversed linked list of sets of transactions called *Blocks*. Each Block stores a pointer to the immediately preceding one. Blocks model the pages of a *ledger*, while the links represent the order on which sets of transactions are written in the ledger. In the case of currency transactions, banks are trusted to maintain the Blockchain[3] such that no transaction is lost, no fake transactions are registered, and that no one can *double-spend*, that is, spend the same unit of currency twice.

Bitcoin [10], the first cryptocurrency, was developed to answer the question: *How to maintain a ledger without relying on a single central organisation?* Bitcoin's approach combines a game-theory argument with cryptographic hashing techniques. Instead of having the bank verifying the validity of transactions and deciding the order on which they are committed to the ledger, Bitcoin opens that task to any peer in the network willing to do so, in exchange of a transaction fee. Peers that take this task are known as *miners*. To overcome the possibility of a Sybil attack, where an attacker creates several different identities to control the transactions committed to the ledger, miners need to prove that they solved a CPU-intensive cryptographic puzzle that is attached to each transaction (so-called Proof-of-Work) to earn the transaction fee. The first miner that solves the puzzle claims the fee, creating competition among them. This way, to gain control of the ledger, an attacker needs to have a substantial amount of computing power with respect to the rest of the miners in the network, to be able to consistently solve the puzzles before them. A detailed analysis of the Bitcoin protocol for ledger maintenance is presented in [7].

To better understand how transactions work, the following preliminary definitions are required. Some of them are borrowed from the Bitcoin Wiki[4], while others are of our own[5]

Definition 1 (Base Address). *A base address is a pair of ECDSA public/private keys.*

[3] For simplicity, we assume that banks store the record of transactions in a Blockchain

[4] http://en.bitcoinwiki.org

[5] We credit `https://www.cryptocoinsnews.com/bitcoin-transaction-really-works/` as a resource that helped with the development of our definitions

Definition 2 (Address). *An address is a hashing of the public key of a Base Address.*

In Bitcoin, addresses are computed with a sequence of hashing operations described in `http://en.bitcoinwiki.org/Bitcoin_address`

Definition 3 (Value Container). *A value container is a pair* $(Address, Value)$*, where value is a real number representing the amount of cryptocurrency assigned to Address.*

A value container can be seen as analogous of a bank note represented as a pair (serial number,face value)

Definition 4 (Lock Script). *A lock script is a boolean function that takes a finite number of arguments as input. We call the Pre-Image of* $True$ *in the lock script a* combination*.*

In Bitcoin, lock scripts are written in a special language called *Script*[6].

Definition 5 (Value Lockbox). *A value lockbox is a pair* $(ValueContainer, LockScript)$*. To be able to use the value container as input for a transaction, one needs to provide a combination for the lock script. This can be seen as a voucher that requires a code to be redeemed.*

Definition 6 (Transaction). *A transaction is a transfer of value to be appended to the Blockchain. It is comprised by:*

1. *An unique identifier* txid
2. *An address R, called the Receiving Address. In Bitcoin, R is generated from a Base Address created by the receiving peer.*
3. *A set of value lockboxes* $In = I_1, ..., I_n$ *called the* inputs *of the transactions*
4. *A set of combinations* $C = C_1, ..., C_n$ *such that* C_1 *is a combination of* I_1 *and so on.*
5. *A set of value lockboxes* $Out = O_1, , O_n$ *called the* outputs *of the transaction. At least one of the outputs must have R as address of its value container.*

To be appended to the ledger, a transaction must be *valid*. In Bitcoin, the validation check-list is quite long and includes several items related to efficiency. For the sake of simplicity, we highlight the two critical checks: first, the sum of the values of the value containers of the inputs must be greater than the sum of the values of the value containers of the outputs. This is analogous to a merchant checking that one did not attempt to pay a 15 euro bill with a 10 euro note; second, no input must appear as input of a transaction already recorded in the ledger, *i.e.*, reject double spending attempts. Figure 1 shows a simplified graphical representation of a transaction that receives as input two value lockboxes, each one containing one coin, creating one output value lockbox containing two coins.

[6] https://en.bitcoin.it/wiki/Script

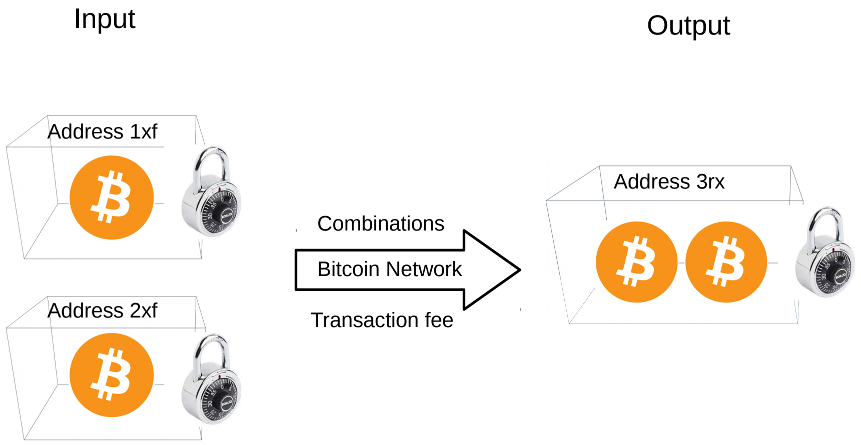

Fig. 1. Graphical representation of a Bitcoin transaction

3 Embedding data in the Bitcoin Blockchain

After Bitcoin became a mass phenomenon, adopters started to ask themselves if it was possible to record in the ledger data beyond transactions to satisfy two use cases:

1. Blockchains are perceived as secure, trustable, permanent and decentralized store of transactions. I have a very important piece of data that I would like to save in a store having exactly those characteristics.
2. Attach the context of transactions (e.g., what is being paid for, who is paying, etc). One could store the context locally and have a link to the transaction identifier, however, this means that the context will not be stored with the same security, trust, permanence and decentralization of the transaction. Following the previous use case, some users considered that transaction context is important enough to be stored together with the transaction.

[2] summarizes it as follows: *small payloads with high value need to be publicly broadcasted and permanently recorded in an asynchronous and pay-as-you-go way.*

Bitcoin adopters started to devise “creative ways” of realizing these use cases by making use of the elements at their disposal, that is, encoding data into

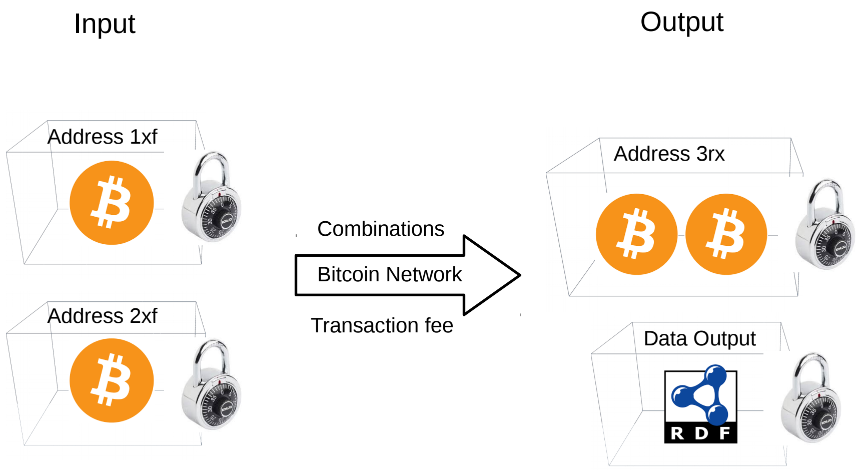

Fig. 2. Graphical representation of a Bitcoin transaction including a data output

addresses, values and lock scripts (see [3] for a compilation of techniques). After a long internal debate, the Bitcoin community reached consensus on letting users attach data to a transaction through a special kind of output in the transaction, commonly known as *OP_RETURN*. *OP_RETURN* is a single instruction of the Script language that implements a lock script that always returns False. As such, any value container locked with *OP_RETURN* cannot be used as input in any other transaction. In our formalisation, we define this as a special output.

Definition 7 (Data Output). *A data output is a value lockbox of the form* $((null, data), OP\_RETURN)$*, where data is a hex value.*

Figure 2 extends Figure 1 to include a data output. Note that the data output lock script only prevents its reuse as input to another transaction, anyone can read the data.

The Bitcoin protocol limits by default the number of data outputs to one per transaction, however, miners may choose to ignore transactions including data outputs or accept to validate transactions with more than one data output (up to the maximum limit of 11 outputs). For the rest of the paper, we assume that all miners implement the default behaviour. A recent study[1] analyses the metadata embedded in the blockchain using the *OP_RETURN* code, finding that 1% of transactions make use of it, representing 0.3% of the size of the Ledger.

In general, there are two strategies to use the small amount of data allowed per transaction. The first, that we call *max-compression*, is to compress data as much as possible, ideally to fit in the transaction being described. If this is not possible, split the compressed data in several chunks and create transactions (for example from the sender to itself, to minimise the currency cost) that carry the rest of the chunks. The sender keeps references to the set of transactions carrying data chunks, to be able to reconstruct the data from the payloads on different transactions, sharing them when necessary.

The second strategy, that we call *hash-out*, uses a single data output to store a hash of the data. The hash is then used both as a key to retrieve the document from a external store, and to verify that data has not been altered by the manager of the external store. Several companies that offer services on top of Bitcoin use either strategy to link transactions to several objects, random short string messages (Eternity Wall[7]), pdf documents (CoinSpark[8]), or other abstract digital assets that use Bitcoin as an underlying transaction layer (Open Assets [9], CounterParty[10])

Table 1 compares both strategies. Those who choose to maximize compression are sure that data outputs have the same guarantees that the transactions they are attached to, but are either limited to 83 bytes or forced to pay a transaction fee for every extra chunk required. Those who choose to hash-out have space bounded by the limit of the external storage (for sure more than 83 bytes), but only the hash has the same guarantees than the transaction. If the external storage fails and data gets corrupted, there is no way to recover it, one can only use the hash to certify the corruption.

Table 1. Comparison of strategies to attach metadata to transactions

	Max-Compress	Hash-Out
Write-Latency	Time to confirm all transactions carrying a chunk of data	Time to confirm transaction being described
Write-Cost	Sum of fees of all transactions carrying a chunk of data	Fee of transaction being described
Integrity	Same as transaction	Same as transaction
Persistence	Same as transaction	Same as transaction for hash Same as external storage for data

In this paper, we aim at exploring the limits of the max-compression strategy for the special case of data encoded in RDF.

[7] `https://eternitywall.it/`

[8] `http://coinspark.org/`

[9] `http://www.openassets.org/`

[10] `https://counterparty.io/`

4 RDF compression for Bitcoin

Several compression algorithms have been developed for RDF data, but none of them for the Cryptocurrency transaction scenario. RDF compression algorithms can be divided in two categories according to their purpose: IoT devices and Publication and Exchange. Compression for IoT devices is motivated by the semantization to the Internet of Things, to make devices compatible with the Web of Data. As IoT devices have limited computing power, communication, memory and energy resources, a whole corpus of research has been devoted to the most appropriate data formats, processing algorithms and protocols to manage Semantic data for IoT (see [12] for an overview).

In the Publication and Exchange scenario, semantic sata publishers are looking to optimize the way on which they archive, store and serve Semantic Data. As opposed to the IoT device scenario, publishers have far more computing, communication, memory and energy resources available, but need to serve large amounts of data to a large number of clients. The focus in this scenario is the binary representation of Big Semantic Data, the development of streamable formats that improve data transference between publishers and consumers, and the provision of query interfaces on top of these compressed formats.

To provide insight on what should be the most appropriate compression algorithm for Bitcoin transaction, we compare side by side the scenarios of IoT devices, Publish and Exchange and Bitcoin transactions across the key parameters of both use cases

- **Input data**: In an IoT device, one is interested in storing the description of the device and its capabilities, and to store/send measures in RDF format. A Semantic Publisher usually store a large RDF-Graph (e.g. DBpedia) or large collections of content-heterogeneous data (e.g. LOD Laundromat). In Bitcoin, we are interested on small description of a transaction, therefore, closer to the IoT scenario.
- **Memory**: A typical IoT device has 64kb of RAM and 128Kb of ROM, where besides data, all software required for the device functioning needs to be stored. Semantic publishers have large amount of space, but want to optimize them to be able store more and larger datasets. The Bitcoin scenario allows only for 83 bytes per transaction, a restricted version of the IoT scenario.
- **Who compresses?**: This refers to which actor (client or server) has to perform data compression and de-compression. In both IoT and publish/exchange context, there are use cases on which both the client and the server/device have to perform both tasks. In Bitcoin, there is currently no way to add to the system a compression functionality, and is probably unreasonable to think that the miners would also take that task. As such, both tasks are always responsibility of the client.
- **Communication** In Semantic Publishing, communication is bound by http, however, as large datasets are involved, there is motivation to develop streamable formats. In IoT devices, communication across devices is often through

a 6LowPan network, that limits packet-size to 80 bytes. In Bitcoin, communication is through http via an API, as data is small, there is no need for streamability. However, note that the packet-size constraint in IoT devices is similar to the space constraint in Bitcoin.
- **Energy**: For IoT devices, a critical design aspect is to keep the energy consumption as low as possible. For semantic publishers, this is not an issue per-se, but connected to the amount of computational resources (CPU) used to serve data. In Bitcoin, this is not an issue, as the difference in energy consumption between a transaction with or without an *OP_Return* is negligible.
- **Query Interface** We refer here to the means to provide querying capabilities on the stored data, both from the expressiveness of the programming language and computational resources point of view. In IoT devices, these resources are limited. In Semantic Publishing, they are less limited, but still a concern, as the size of the underlying data and the number of clients is much larger than in IoT devices (see [13] for a comparison of approaches to limit the query interface to increase availability). Bitcoin does not offer any possibility to implement a query interface on the server side, clients need to download, decompress, load and query data themselves. Following the classification provided in [13], Bitcoin falls in the category of the most limited servers, that only provide dumps to the clients.
- **Read Cost** This refers to the cost of reading data in each scenario. In IoT devices, reads are paid with energy of the device. In publish/exchange, this is paid in computational resources of both server and client. In Bitcoin, getting a transaction through its ID is unexpensive, and as the amount of data requested is small, the de-compression cost is negligible.
- **Write Cost** This refers to the cost of writing data in each scenario. The cost is the same than for reading for IoT devices and publish/exchange: energy and computational resources respectively. In Bitcoin, this is included with the transaction fee for the first 83 bytes, but any further data would need to be written through separate transactions, each of them incurring in a transaction fee to be paid in cryptocurrency

Table 4 summarizes the comparison between scenarios. The Bitcoin scenario shares with the IoT device scenario the small input data size, the limited memory (very limited for Bitcoin), and with the public/exchange scenario the non-importance of the communication and energy parameters. Regarding query interface, we can consider Bitcoin as the extremely limited case of the publish/exchange scenario. The key difference among the three scenarios lies on the dominant cost for write, which, in combination with the differences along the memory dimension, impact the design of compression algorithms. For example, in the IoT context, it is possible to sacrifice some compression ratio for energy savings in the processing device, which does not make sense in the Bitcoin scenario. Along the same lines, some algorithms for Publish/Exchange add extra data to enable the implementation of query interfaces on top of them without decompressing, again, something that does not make sense in Bitcoin.

Table 2. Comparison of RDF compression motivating scenarios

	IoT Device	Publish/Exchange	Bitcoin-like transaction
Input data size	Small	Very Large	Small
Memory	64kb RAM + 128kb ROM	GBs of RAM + TBs of ROM	83 bytes
Who processes?	Client and Server	Client and Server	Only Client
Communication	6LowPan bound (limited)	Http/FTP bound	Http bound
Energy	Limited	Plenty	Not an issue
Query Interface	Limited	Expressive	Very Limited (get transaction)
Dominant cost for read	Energy	Computational resources	Negligible
Dominant cost for write	Energy	Computational resources	Cryptocurrency

5 Experimental Study

In this section, we test current state-of-the-art RDF compression algorithms on the Bitcoin transaction scenario. Following from table 4, the key parameter for the Bitcoin scenario is the compression ratio. As such, we focus our experiment in answering the question *which algorithm offers maximal compression for input documents that we might want to attach to transactions?*

We believe that documents that one would attach to transactions differs from the ones used to benchmark current algorithms. Algorithms designed for Publish/Exchange are tested against Big Data RDF documents, while algorithms for IoT devices are tested against documents expressed with ontologies describing sensing, with a large amount of numeric literals representing the measures of the sensors. As such, we created two documents based on the PROV ontology, providing metadata about a fictional transaction. We assume that transactions have URIs minted from the namespace *http://bitcoin.org/* plus a transaction id.

The first document (see Listing 1.1) describes the transaction as a *prov:activity* and contained a reasonable set of information about it. The second (Listing 1.2) contained the minimum amount of information for valid RDF, including two statements (besides type), a $prov : informedBy$ and a $rdfs : seeAlso$ declaration.

Listing 1.1. The complete RDF/XML document used, referred to as ”Full Document”

```
<rdf:RDF
    xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
    xmlns:prov=”http://www.w3.org/ns/prov#”>
  <prov:Activity rdf:about=”http://bitcoin.org/56754644” >
    <prov:wasDerivedFrom>
      <prov:Entity rdf:about=”http://bitcoin.org/41565751”/>
    </prov:wasDerivedFrom>
    <prov:wasStartedBy>
```

```
      <prov:Agent rdf:about="http://example.com/bob-from-finance"/>
    </prov:wasStartedBy>
    <prov:wasAssociatedWith>
      <prov:Agent rdf:about="http://example.com/alice-from-finance"/>
    </prov:wasAssociatedWith>
    <prov:wasInformedBy>
      <prov:Activity rdf:about="http://example.com/procurement-ticket"/>
    </prov:wasInformedBy>
    <rdf:type rdf:resource="http://www.w3.org/ns/prov#Entity"/>
  </prov:Activity>
</rdf:RDF>
```

Listing 1.2. Stripped down document with one piece of information, and a seeAlso reference. Referred to as ‘Small Document’

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:prov="http://www.w3.org/ns/prov#"
    xml:base="http://example.com/">
        <prov:Activity rdf:about="http://bitcoin.org/56754644">
                <prov:wasInformedBy>
            <prov:Activity rdf:about="/procurement-ticket"/>
        </prov:wasInformedBy>
          <rdfs:seeAlso rdf:resource="http://bitcoin.org/41565751"/>
        </prov:Activity>

</rdf:RDF>
```

We now briefly describe the compression algorithms considered in the literature.

HDT [5] is a binary representation designed for the Publish/Exchange use case. HDT decomposes an RDF dataset in three parts: a *header* that holds metadata describing an HDT semantic dataset using plain RDF. It acts as an entry point for the consumer, who can have an initial idea of key properties of the content even before retrieving the whole dataset. The main motivation behind HDT is the ability to provide a simple query interface to large datasets that otherwise wouldn’t fit in memory, as such, we expected to not perform well in this use case. For the experiment, we used the HDT CPP Docker container with version 1.1.1, commit 421165e.

SHDT [8] is a simplified version of HDT, adapted for working on IoT devices, that improves the memory and energy footprint of HDT in exchange of lower compression ratio. As is the case with HDT, this tradeoff is in principle not appropriate for our scenario. This application is part of the Wiselib library[11] which targets embedded devices, and as such difficulties were encountered in compiling for a PC so this algorithm was not run.

[11] `https://github.com/ibr-alg/wiselib`

RDF4J Binary RDF (Formerly Sesame) [12] was the algorithm for binary encoding in the Sesame library, which is now continued in RDF4J[13]. We used the Java RDF4J library version 2.2.0.

RDSZ [6] is another approach focused on reducing communication and processing overhead when streaming RDF data. Is based on the combination of differential encoding with the wide-known ZLib compression algorithm. We used RDSZ from the bitbucket repository [14] at commit c4da3b2. We used the default size for batch (5), cache (128), and buffer (32 * 1024), and used the Zlib compression algorithm.

ERI [4] aims at improving over RDSZ by exploiting structural information that is known before hand between publisher and consumer. ERI and RDSZ have similar performance, one being better than the other depending on the underlying distribution of predicates and entities of the input RDF dataset. We used the prototype release of the GitHub repository[15],at commit a99ff03, without additional configuration. Both RDSZ and ERI were only tested with large input datasets, therefore, we did not know what to expect on our setup.

RDF-Thrift [16] is a binary encoding for fast machine encoding and decoding based on Apache Thrift, and Google's Protocol Buffers. The main goal is to reduce the communication and compression-decompression overhead between publishers and consumers or between co-operating processes. We used the Apache Jena 3.3.0 implementation of Apache Thrift for RDF.

EXI for RDF [9] leverages W3C's Efficient XML Interchange (EXI) format [11] to achieve an efficient binary representation of RDF from its XML representation. EXI uses a grammar-driven approach to represent XML-based data in an efficient binary form and vice versa. The grammar is derived from given XML Schema where each defined complex type is represented as a deterministic finite automaton. [9] explores the use of two type of grammars: a generic one that allows the encoding of an RDF using several vocabularies, but with limited compression ratio; and with *concrete* grammars. The generic grammars follow EXI by using string tables, to map unknown elements and attributes, as well as strings to an ID, which is then managed to ensure consistency with the repository. This store allows the a triple to be represented in a compact form based on these IDs. Using this format allows a compressed RDF to be queried by the client, without the client having pre-existing knowledge of the RDF graph[9]. The concrete grammars remove the need for this, since the available elements are almost entirely known from the schema, and can be defined accordingly, removing the need for storage and processing of strings [9]. EXI with a concrete

[12] `http://rdf4j.org/`

[13] `http://docs.rdf4j.org/rdf4j-binary/`

[14] `https://bitbucket.org/norbertofdz/rdsz`

[15] `https://github.com/webdata/ERI`

[16] `https://afs.github.io/rdf-thrift/rdf-binary-thrift.html`

grammar was reported to compress 20 triples in 78 bytes, orders of magnitude better than Thrift. [17]. As such, we expected it to be the best performer in our setup.

We used the Exificient GUI[18] obtained on 24 July 2017. The GUI provides the ability of creating grammars from XML Schemas. We re-used the generic schema of the uRDF store[19]. To generate a concrete grammar, we input to the GUI the XMLSchema representation of the PROV ontology [20].

Table 3. The size of the input document in different RDF formats (bytes)

RDF Representation	**Full Document** (10 triples)	**Small Document (4 triples)**
RDF/XML	769	502
Turtle	911	529
NTriples	1190	476
JSON LD	1316	670

Prior to testing the specific algorithms, we tested standard Linux command line compression applications: GZip, BZip2, and xz in order to provide a baseline for the ability for naive compression without underlying knowledge of the structure of the data. These were run to be the most aggressive compression available at the expense of higher memory and CPU usage. Since the file sizes were small, and the processing environment was not constrained, this is not a problem for our use case.

Table 3 reports the size of the compressed Full and Short Documents achieved with each approach. For the standard compression applications, we report the compression over the Ntriples format, that was the best. Figures 3 and 4 show data in Table 3 as a bar chart. The horizontal line represents the the uncompressed size of the smallest format for each case. To better understand the differences among the approaches that improve over the baseline, we set the length of the y-axis to the next round value above the baseline.

The majority of compression techniques performed poorly on our setup, and none came close to the required 83 bytes. Notably, for Full Document, neither technique was able to improve over the naive gzip compression over Ntriples. For Small Document, RDSZ and EXI for RDF using the grammar generated from the public PROV XML schema were able to improve over the gzip baseline.

Poor performance was expected for the techniques that add metadata to enable a query interface (HDT, RDF4JBinary). On the other hand, our expec-

[17] Unfortunately, neither the ontology or the data used in the evaluation is openly available

[18] `http://exificient.github.io/java/exificient-gui-jar-with-dependencies.jar`

[19] `https://github.com/vcharpenay/urdf-store-exp/blob/master/lubm/schema/basic_rdf_for_exi_v03.xsd`

[20] `https://www.w3.org/TR/prov-xml/`

Table 4. Size of compressed document and (compression ratio) of RDF compression techniques when applied to our test documents. The compression ratio is computed using the smallest size representation as uncompressed size: NTriples for the small document and RDF/XML for the full document.

Compression Technique	**Full Document**	**Small Document**
gzip over NT	267 (2.88)	256 (1.85)
bzip2 over NT	327 (2.35)	297 (1.6)
xz over NT	304 (2.52)	288 (1.65)
HDT	2305	2329
RDF-Thrift	942	563
RDF4J Binary	1386	1169
RDSZ	279 (2.75)	215 (2.21)
ERI	654 (1.175)	573
EXI for RDF (schemaless)	397 (1.93)	261 (1.82)
EXI for RDF (PROV schema)	306 (2.51)	217 (2.19)

tation was that EXI for RDF would outperform the others, due to the results obtained by [9], who succeeded in compressing 20 triples down to 78 bytes. The results we obtained are less impressive, although we have two possible areas for improving this:

- The use case for the EXI format is focused on sensors, and is heavily biased in favour of numerical data representing sensor measures
- The concrete grammar was more comprehensively implemented than we were able to with the PROV XML schema. By taking a subset of the ontology, or including stricter rules, we might increase compression efficiency.

In regards to the first item, we have some indication that this is the case, with some preliminary testing. By changing the *rdf:about* to numbers, that is, crafting a document with triples with literal numeric values as objects, the compression ratio improved from 44% of the size of XML to 19% of the XML (down to 120 bytes). This indicates that in this setup, where every byte counts, the characteristics of the input document become critical.

The good performance of RDSZ came as a surprise to us. RDSZ was not compared against EXI due to being unsuitable for embedded devices, owing to the loss of the RDF triple structure, and the use of the energy-inefficient Zlib[9]. However, as energy is not a factor in our setup, our results suggest that sacrificing triple structure is beneficial to our scenario. The good result of the naive gzip seems to reinforce this hypothesis.

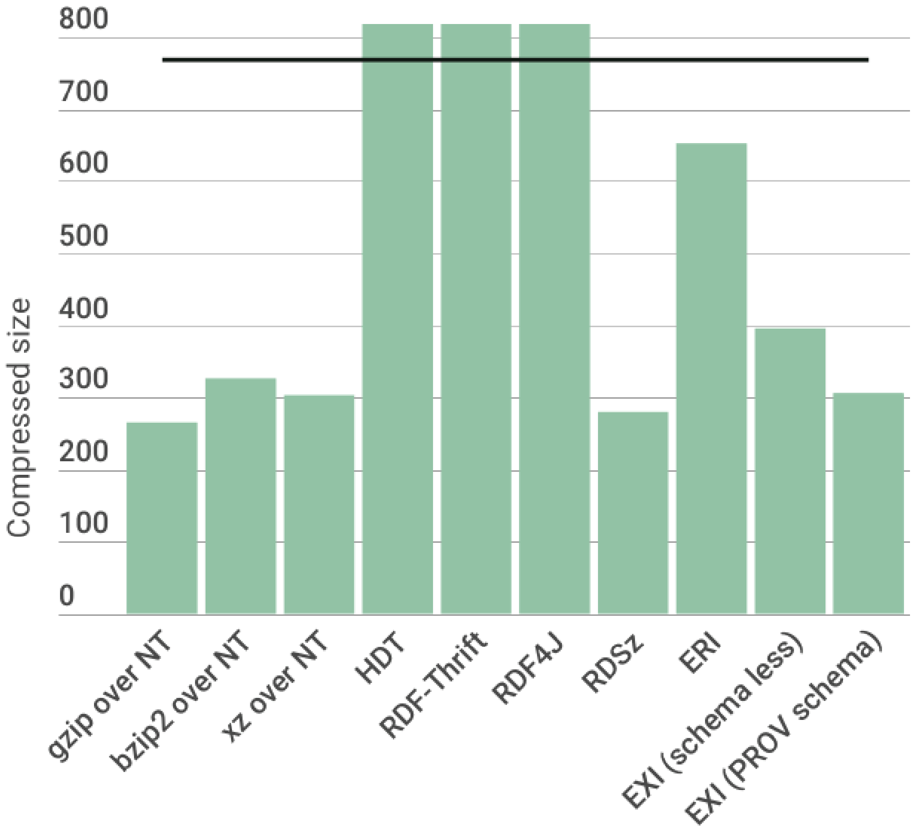

Fig. 3. Compression sizes achieved for the Full Document. The horizontal line represents the uncompressed size of the smallest format (RDF/XML)

6 Conclusion and Future Work

In this paper we have studied the problem of attaching RDF metadata to transactions in Cryptocurrency Blockchains, so that metadata and transaction are stored in the same Blockchain, with special emphasis on the Bitcoin family, where each transaction can carry by default 83 bytes of data. The problem is related to the RDF-Compression problems motivated by Publish/Exchange of Big Semantic Data collections and for Semantic Data management in IoT devices. We compared the key dimensions considered for the design of those algorithms with the Bitcoin transaction scenario, uncovering that it is not completely aligned with any of the other two. In Bitcoin transactions, the critical aspect is the compression ratio on a small size input, with factors like energy or provision of a query interface being irrelevant.

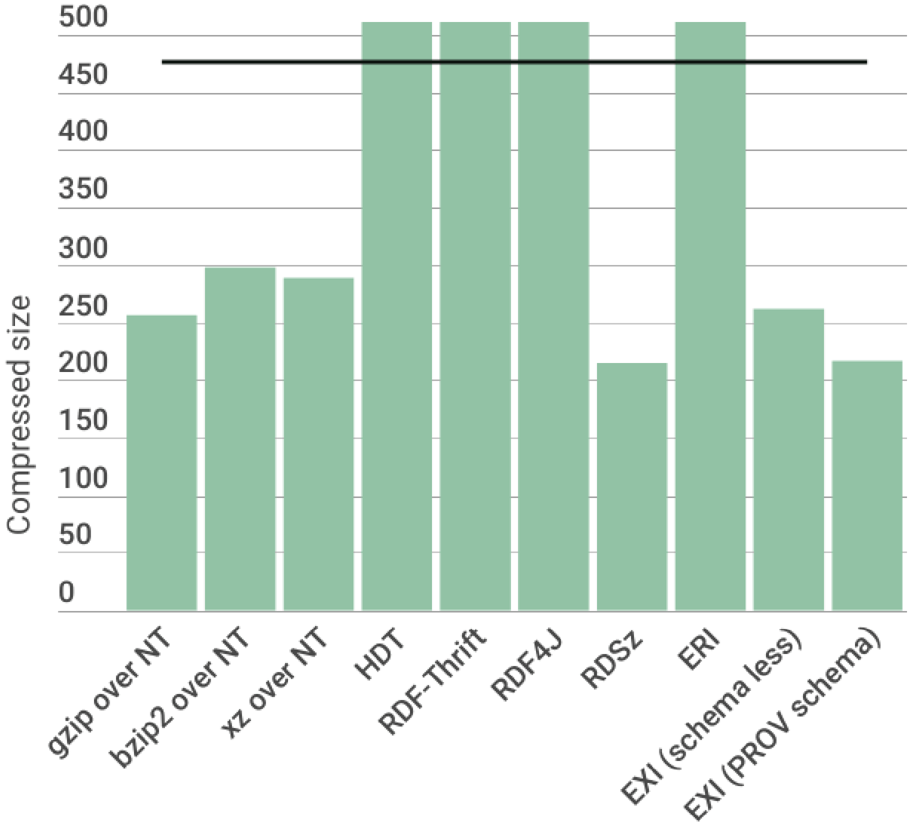

Fig. 4. Compression sizes achieved for the small Document. The horizontal line represents the uncompressed size of the smallest format (NT)

Finally, we tested seven state-of-the-art RDF compression algorithms on two sample documents describing the provenance of a transaction in 10 and 4 triples respectively, to test how well they generalize to the Bitcoin scenario. The results are largely negative, as most approaches did not improve over the naive approach of compressing the NTriples representation with gzip. Only RDSZ and EXI with concrete grammar were able to improve in the document with 4 triples. Our results also suggest that, in the current state of the art, only a very limited class of RDF Documents could be attached to a Bitcoin transaction without requiring additional transactions. We believe this motivates research for new RDF compression algorithms specifically tailored to this use case.

As future work, besides the aforementioned development of specifically tailored RDF-Compression algorithms, we consider of interest the study of which classes of RDF documents can be better compressed by which algorithm, and the optimal way to split and compress a document to minimize the number of required extra transactions. An intelligent client would then be able to combine different approaches depending on the structure of the document to be attached to the transaction. Furthermore, domain-specific vocabularies designed with compressibility in mind, perhaps derived from existing vocabularies and from which mappings exist, enabling intelligent clients to bridge between the two worlds.

Finally, another interesting direction is repeating this same analysis for the case of Smart Contract Blockchains like Ethereum, where additional data registers are available and is possible to create a server-side query/update interface.

Acknowledgements We thank Javier D. Fernández for making the code of ERI available on github. We thank Victor Charpenay for kindly answering inquiries about the EXI format and the EXIficient GUI.

References

1. Bartoletti, M., Pompianu, L.: An analysis of Bitcoin OP_return metadata (2017)
2. Coin Sciences Ltd: Metadata in the Blockchain: The OP_return Explosion, https://www.slideshare.net/coinspark/bitcoin-2-and-opreturns-the-blockchain-as-tcpip
3. Colored Coins Team: Data storage on the blockchain, https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification/wiki/Data-Storage-Methods
4. Fernández, J.D., Llaves, A., Corcho, O.: Efficient RDF Interchange (ERI) Format for RDF Data Streams. In: The Semantic Web – ISWC 2014. pp. 244–259. Lecture Notes in Computer Science, Springer, Cham (Oct 2014), https://link.springer.com/chapter/10.1007/978-3-319-11915-1_16
5. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web 19, 22–41 (Mar 2013), http://www.sciencedirect.com/science/article/pii/S1570826813000036
6. Fernández, N., Arias, J., Sánchez, L., Fuentes-Lorenzo, D., Corcho, O.: RDSZ: An Approach for Lossless RDF Stream Compression. In: The Semantic Web: Trends and Challenges. pp. 52–67. Lecture Notes in Computer Science, Springer, Cham (May 2014), https://link.springer.com/chapter/10.1007/978-3-319-07443-6_5
7. Garay, J., Kiayias, A., Leonardos, N.: The Bitcoin Backbone Protocol: Analysis and Applications. In: Advances in Cryptology - EUROCRYPT 2015. pp. 281–310. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Apr 2015), https://link.springer.com/chapter/10.1007/978-3-662-46803-6_10
8. Hasemann, H., Kröller, A., Pagel, M.: RDF provisioning for the Internet of Things. In: 2012 3rd IEEE International Conference on the Internet of Things. pp. 143–150 (Oct 2012)
9. Käbisch, S., Peintner, D., Anicic, D.: Standardized and Efficient RDF Encoding for Constrained Embedded Networks. In: The Semantic Web. Latest Advances and New Domains. pp. 437–452. Lecture Notes in Computer Science, Springer, Cham (May 2015), https://link.springer.com/chapter/10.1007/978-3-319-18818-8_27

10. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008)
11. Schneider, J., Kamiya, T., Peintner, D., Kyusakov, R.: Efficient XML Interchange (EXI) Format 1.0. Tech. rep., W3C (2014), https://www.w3.org/TR/2014/REC-exi-20140211/
12. Su, X., Riekki, J., Nurminen, J.K., Nieminen, J., Koskimies, M.: Adding semantics to internet of things. Concurrency and Computation: Practice and Experience 27(8), 1844–1860 (Jun 2015), http://onlinelibrary.wiley.com/doi/10.1002/cpe.3203/abstract
13. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. Web Semantics: Science, Services and Agents on the World Wide Web 37, 184–206 (Mar 2016), http://www.sciencedirect.com/science/article/pii/S1570826816000214