applied sciences

Article

A Framework for Reversible Data Embedding into Base45 and Other Non-Base64 Encoded Strings

Marco Botta * and Davide Cavagnino

Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italy; davide.cavagnino@unito.it

* Correspondence: marco.botta@unito.it; Tel.: +39-011-6706721; Fax: +39-011-751603

Featured Application: Embedding of extra information, like a signature or a Message Authentication Code, in printable string encodings, for example, QR codes.

Abstract: Printable string encodings are widely used in several applications that cannot deal with binary data, the most known example being the mail system. In this paper, we investigate the potential of some of the proposed encodings to hide and carry extra information. We describe a framework for reversibly embedding data in printable string encodings, like Base45. The method leverages the characteristic of some encodings that are not surjective by using illegal configurations to embed one bit of information. With the assumption of uniformly distributed binary input data, an estimation of the expected payload can be computed easily. Results are reported for Base45 and Base85 encodings.

Keywords: data hiding; watermarking; authentication; integrity protection; printable encoding; binary-to-text conversion

check for updates

Citation: Botta, M.; Cavagnino, D. A Framework for Reversible Data Embedding into Base45 and Other Non-Base64 Encoded Strings. *Appl. Sci.* **2022**, *12*, 241. https://doi.org/10.3390/app12010241

Received: 24 November 2021
Accepted: 24 December 2021
Published: 27 December 2021

1. Introduction

Data hiding is a family of techniques aimed at embedding information into digital objects for various application contexts. Digital watermarking, for copyright protection or authentication, and steganography, for covert communication or information storage, are typical applications of data embedding and hiding (see [1–4]).

Many algorithms have been devised for watermarking in different kinds of objects like images [5], audio [6], video [7], neural networks [8] and text data [9]. These algorithms can be classified according to various properties and characteristics; one of the most important properties is reversibility, i.e., the possibility to obtain the original object after the extraction of the embedded data.

On the other hand, printable string encoding of binary data is a well-known and widely used technique to cope with systems designed to manage only bytes representing printable characters: in other terms, printable string encoding is an encapsulation method to process, in a transparent manner, any possible bit string by systems able to run only printable strings (for example, some mail servers).

Even though printable string encodings and watermarking are widely used techniques in several applications, to the best of our knowledge there are no attempts to combine these techniques for data hiding. The main questions we try to answer are:

(a) Can we hide information in string encodings?
(b) How much information can we embed in such string encodings?
(c) Can we make it reversible?

In this paper, we present a framework defining data and procedures for embedding and retrieving a bit string, which will be called a watermark, from a printable character string built using a binary-to-higher-base representation, like Base45 [10] and/or

Base85 [11,12]. The embedding-retrieving process is reversible. Indeed, the original printable character string is restored after the watermark extraction.

The paper is structured as follows: firstly, a brief notation and terminology subsection is presented, then some related works on the topic of printable encodings are briefly discussed. Section 3 introduces the proposed framework for reversible data embedding and the results of its application are shown in Section 4. The last section draws some conclusions and proposes some future works.

Notation

In this subsection the notation and terminology used throughout the paper is presented.

Small italic letters denote scalar values, e.g., a, b, p.

Boldface italic capital letters denote sets, e.g., $\boldsymbol{T}$, $\boldsymbol{C}$. The cardinality of a set $\boldsymbol{T}$ is denoted as $\mathrm{card}(\boldsymbol{T})$.

An alphabet is denoted with Greek capital letters (e.g., Ψ).

The terms *symbol* and *character* are applied interchangeably trying to use the most meaningful in the sentence context.

A word composed from symbols of an alphabet Ψ is called a *sequence*. A sequence of n symbols from the alphabet $\Psi = \{0, 1\}$ is called binary string.

Capital letters (e.g., B, S, W) denote sequences of symbols from an alphabet.

Greek letters represent functions, e.g., β, φ.

Specific characters or strings (sequences) of characters are enclosed inside single quotes, e.g., 'D' or 'string<My Header>!'.

The floor operation is denoted as r, with $r \in \mathbb{R}$, and returns the largest integer not greater than r.

2. Related Works

To define the possible applications of the proposed framework to an encoding method, it is useful to recall some of the binary-to-text mappings presently available and used in computer systems.

Base64 was introduced many years ago and standardized in 1992: it encodes 3 bytes (24 bits) with 4 printable characters each encoding 6 of the 24 bits. The characters are thus chosen from a set of $2^6 = 64$ symbols, namely the 26 letters of the English alphabet, uppercase and lowercase, the 10 decimal digits and 2 special characters that may differ among the various standardized applications. The most recent definition of the Base64 encoding may be found in [13]: in that RFC also the Base32 and Base16 representations are specified. Base32 encodes groups of 5 bytes (40 bits) in 8 printable characters each representing 5 bits: thus, 32 possible characters are needed (the 26 uppercase letters of the English alphabet and the digits from 2 to 7). For Base64 and Base32 the character '=' is used for padding the encoded string when the number of input bytes is not a multiple of 3 or 5, respectively. Base16 is simply the hexadecimal representation of the input bytes using the decimal digits and the uppercase letters from 'A' to 'F' (some implementations may differ from [13] and allow the use of lowercase letters from 'a' to 'f'); by construction this encoding does not need any padding.

Base45 is defined in the work in progress [10] and is developed for encoding data as text in QR codes. The symbols it uses are the 10 decimal digits, the 26 letters of the English alphabet, the space and the other 8 special characters. The encoding takes pairs of bytes and converts the number in Base45 using 3 digits (in a little-endian way, i.e., the leftmost character is the least significant); in case of a binary string having odd length then the last byte is converted in 2 base 45 digits. It is interesting to note that not all 3 digit numbers in base 45 can be converted in a binary number having at maximum 16 bits: in fact, from the sequence 'GGW' (representing $2^{16} = 65{,}536$) to the sequence ':::' more than 16 bits are needed to write the corresponding number in binary format; these sequences are considered unacceptable by the standard and accordingly rejected.

Base85, also called Ascii85 in [11], encodes 4 bytes using 5 characters from an alphabet of 85 symbols (a subset of the ASCII characters, from the code 33, '!', to the code 117, 'u'). An exception is made for the binary value 0 which, instead of being encoded as '!!!!!', is encoded as 'z'. The version of this coding presented in [11] uses a delimiter (namely '~>') to mark the ending of the character sequence. Moreover, to cope with binary strings having a length that is not a multiple of 4, a particular padding method in encoding and decoding is used. During decoding, a 'z' character in a 5-character sequence is an error. Also, as in the analogous case of Base45, a 5-character sequence decoding to a value greater than or equal to 2^{32} is regarded as an error and not accepted. The base 85 is also used in an encoding of IPv6 addresses [12].

Encodings using Base91 are presented in [14,15]. In [14], the authors propose an encoding that splits the input binary string into 13-bit words and encodes the resulting binary numbers in pairs of characters using an alphabet of 91 symbols (the printable ASCII characters from '!' to '~' excluding '-', '=' and '.'). To cope with bit strings having a length not being a multiple of 13, 12 pairs are reserved to encode how many unused bits are present in the last 13-bit word: this encoding leaves $91^2 - 2^{13} - 12 = 77$ unused Base91 pairs. The encoding used in the source code in [15] makes use of all the 91^2 character pairs; in fact, 13 bits are encoded with 2 Base91 characters unless the value to be encoded is less than 89: in that case 14 bits are encoded adding as a significant bit one more bit from the binary string to encode. This implies the use of $2^{13} + 89 = 91^2$. pairs of Base91 characters saturating all the possible configurations.

Base58 is an encoding introduced for the Bitcoin cryptocurrency (as referred to in [16]) by S. Nakamoto and described in the work in progress [17]. The objective of this encoding is to represent meaningful data types of the protocol in a human readable format that would not allow any ambiguities when written, thus, starting from the Base64 alphabet, the special characters '+' and '/' are taken out (to avoid ambiguities in URLs or file system paths [17]) along with '0' (zero), 'O' (uppercase o), 'I' (uppercase i), 'l' (lowercase L) for possible ambiguities when reading or writing data by humans. Thus, the Base58 alphabet described in [17] totals the following 58 symbols: the 9 digits from '1' to '9', the 24 uppercase letters of the English alphabet (i.e., without 'I' and 'O') and the 25 lowercase letters of the English alphabet (i.e., without 'l'). The data represented as a byte string is interpreted as a sequence of symbols (bytes) in base 256 which is transformed in Base58 with a base conversion algorithm: to avoid the loss of the leading zeros (if present, e.g., '00A23') in the encoding, a string of '1' symbols represents how many null bytes compose the prefix of the byte string. As a side note, some applications developed a Base56 encoding where also the characters '1' (one) and 'o' (lowercase O) are removed from the Base58 alphabet.

The Base62 encoding uses as symbols the 10 decimal digits and the 26 letters from the English alphabet both uppercase and lowercase. In [18], the author presents UTF-62, a transformation for ISO 10646 (the Universal coded character set, UCS). When a character code is contained into 2 bytes (UCS-2) it is represented with 3 Base62 symbols (the leftmost one having its Most Significant Bit, MSB, valued 0); instead, for encoding UCS-4 characters (31 bits) 6 Base62 symbols are used (the leftmost one having its Most Significant Bits valued 1000). In [19] a data stream is encoded examining groups of 6 bits at a time using Base62 characters. The first 60 binary configurations (namely from 000000 to 111011) are mapped directly to the Base62 alphabet in the corresponding character position. The 5 bit configuration 11110 is represented with the second to last Base62 character whilst the 11111 configuration is encoded with the last Base62 character: in both cases the sixth bit becomes the first bit of the next group. Note that this encoding operates one Base62 character at a time thus all the Base62 characters sequences are possible while in [18] even keeping the MSBs according to the proposed representation (0 for UCS-2 and 1000 for UCS-4):

- when using UCS-2 encoding the possible Base62 sequences are 32×62^2 = 123,008 to represent 2^{16} = 65,536 different binary strings, and
- for UCS-4 the possible sequences are 4×62^5 = 3,664,531,328 to represent 2^{31} = 2,147,483,648 different binary strings.

This fact leads to a redundancy in the resulting representation allowing extra bits (e.g., data like a watermark) to be stored without impacting on the reversibility of the method: this data hiding capacity is summarized for the main encodings discussed so far in the column Available configurations of Table 1.

Table 1. Summary of the main encodings' capacity.

Base	No. of Encoded Binary Strings	No. of Representable Binary Strings	Available Configurations
Base16 [13]	2^8	2^8	0
Base32 [13]	2^{40}	2^{40}	0
Base45 [10]	2^{16}	45^3	25,589
Base62 [18]	2^{16}	32×62^2	57,472
Base62 [18]	2^{31}	4×62^5	1,517,047,680
Base62 [19]	2^6	62	0
Base64 [13]	2^{24}	2^{24}	0
Base85 [11,12]	2^{32}	85^5	142,085,829
Base91 [14]	$2^{13} + 12$	91^2	77
Base91 [15]	$2^{13} + 89$	91^2	0

3. The Proposed Framework

Suppose one wants to represent binary strings of fixed length n with sequences of fixed length s using the a symbols from an alphabet Ψ (for example, the 45 printable symbols of the Base45 system [10]). Then, the following constraint must be satisfied:

$$2^n \leq a^s. \tag{1}$$

The passive redundancy (i.e., the number of sequences of s symbols from Ψ not representing any binary string of length n) of this encoding according to [20] is:

$$r = a^s - 2^n. \tag{2}$$

Note that in the rest of the paper we will assume that the binary strings of length n have all the same probability as is the case for compressed and/or encrypted data.

Thus, the exceeding r sequences that do not represent one of 2^n binary strings may be used to embed extra data bits in the flow of symbols from Ψ: these bits may carry, for example, a watermark, a signature, a message authentication code, a cyclic redundancy check, or any extra information an application may need.

Let us consider a generic sequence of s symbols from the alphabet Ψ, and define the following sets:

- $\boldsymbol{T}$, the set of all possible sequences of s symbols, having cardinality a^s;
- $\boldsymbol{C}$, the subset of $\boldsymbol{T}$ containing the sequences used to encode the 2^n binary strings (obviously having cardinality 2^n): the mapping is defined with a bijective function β;
- $\boldsymbol{E}$, the subset of $\boldsymbol{T}$ containing the exceeding r sequences that are not used to represent the binary strings.

The sets $\boldsymbol{C}$ and $\boldsymbol{E}$ constitute a partition of $\boldsymbol{T}$. To exploit the redundancy in this coding the set $\boldsymbol{C}$ is further partitioned into two subsets:

- $\boldsymbol{E}'$, a subset of cardinality r containing the sequences put in one-to-one correspondence with the sequences in $\boldsymbol{E}$ by a bijective mapping function φ;
- $\boldsymbol{N}'$, the subset of the remaining sequences in $\boldsymbol{C}$ having no corresponding sequence in $\boldsymbol{E}$.

In the present work every sequence in $\boldsymbol{C}$ is put in relation with at most one sequence in $\boldsymbol{E}$. Moreover, we assume that the encoding is frugal in the sense that $\mathrm{card}(\boldsymbol{C})$ is not lower than $\mathrm{card}(\boldsymbol{E})$. The extension to a mapping allowing for more than one sequence in $\boldsymbol{E}$ to be associated to some sequence in $\boldsymbol{C}$ is left as a further development, as it will be discussed in the conclusions.

The structure and the relationship among these sets is shown in Figure 1.

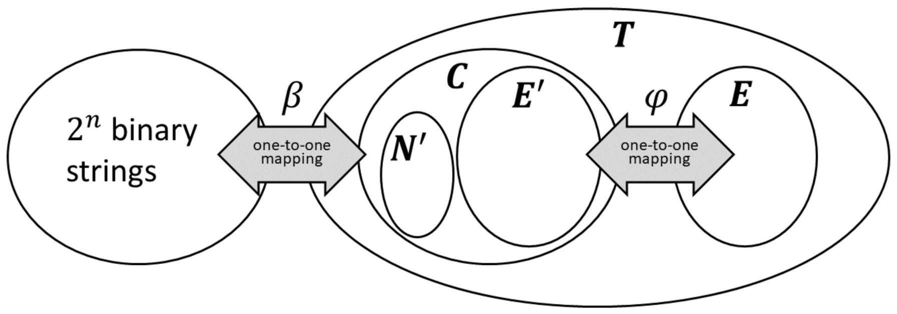

Figure 1. Relationship among the sets composed by the sequences of s symbols from the alphabet Ψ.

When encoding an n bits binary string B mapped through β to an s symbols (belonging to Ψ) sequence S, if $\mathrm{S} \in \boldsymbol{E}'$ it is possible to embed one extra bit b of information saving S (for $b = 0$) or $\varphi(\mathrm{S})$ (for $b = 1$) depending on the bit value (if $\mathrm{S} \notin \boldsymbol{E}'$, i.e., $\mathrm{S} \in \boldsymbol{N}'$, no bit can be embedded).

The decoding can be performed by reversing the previous operations: if the sequence $\mathrm{S} \in \boldsymbol{N}'$ then the original binary sequence is restored with $\beta^{-1}(\mathrm{S})$, if $\mathrm{S} \in \boldsymbol{E}'$ then a 0 valued bit is extracted and the original binary sequence is restored with $\beta^{-1}(\mathrm{S})$, otherwise $\mathrm{S} \in \boldsymbol{E}$, a 1 valued bit is extracted and the original binary sequence is restored with $\beta^{-1}\left(\varphi^{-1}(\mathrm{S})\right)$.

The pseudo-code description of the encoding and decoding algorithms are presented in the following Algorithms 1 and 2.

Algorithm 1: Pseudo-code of the encoding algorithm

```
ENCODING ALGORITHM
INPUT: set of binary strings to be encoded, $C = N' \cup E'$, $E$, $\beta$, $\varphi$,   is a binary string of
            bits to be embedded.
WHILE there are binary strings to be encoded select next one and call it B
        $S = \beta(B)$;
        IF $S \in E'$ THEN
            Get next bit $b$ from W ;
            IF $b = 1$ THEN $S = \varphi(S)$;
        OUTPUT S;
```

Algorithm 2: Pseudo-code of the decoding algorithm

```
DECODING ALGORITHM
INPUT: set of sequences to be decoded, $C = N' \cup E'$, $E$, $\beta$, $\varphi$.
W = {};
WHILE there are sequences to be decoded select next one and call it S
        IF $S \in E$ THEN
            W = concat(W, 1);
            $S = \varphi^{-1}(S)$;
        ELSE IF $S \in E'$ THEN W = concat(W, 0);
        $B = \beta^{-1}(S)$;
        OUTPUT B;
OUTPUT W;
```

By assuming a uniform distribution for the input binary strings, an average payload of

$$p = \frac{a^s - 2^n}{2^n} = \frac{a^s}{2^n} - 1 \tag{3}$$

bits per binary string (bpbs) may be embedded in the output sequence. The corresponding payload per byte (bit per byte, bpb) can be computed as:

$$q = \frac{8p}{n}. \tag{4}$$

The payload referred to the output symbols (bit per symbol, bps) is:

$$o = \frac{p}{s}. \tag{5}$$

For example, Base45 as defined in [10] encodes $n = 16$ bits in $s = 3$ symbols (with $a = 45$) thus $p = 0.390457$ bpbs, $q = 0.195229$ bpb, $o = 0.130152$ bps. In case of Base85, $n = 32$, $s = 5$, $a = 85$ leading to $p = 0.033082$ bpbs, $q = 0.008270$ bpb, $o = 0.006616$ bps.

For Base64 encoding [13] all the output sequences are used to encode the input binary strings thus it is not possible to apply the proposed method for data embedding ($a^s = 64^4 = 2^{24}$, $2^n = 2^{24}$, $p = q = o = 0$).

4. Experimental Results

In this section, we report on experimental results by applying the proposed framework to Base45 and Base85 encodings: as previously said, the set $\boldsymbol{E}'$ is built with φ assuming a uniform distribution of the 2^n binary strings. This choice was made because it is general and makes no assumptions on the possible distribution of the binary strings: we will discuss the possibility to increase the payload with φ mapping on the set of the most probable sequences in $\boldsymbol{C}$ for specific domain contexts, like encodings of TIFF or JPEG images.

Anyway, we tested three different mappings called Map 1, Map 2 and Map 3. When Map 1 is used the set $\boldsymbol{E}'$ consists of the sequences mapped to the $a^s - 2^n$ binary strings counting from $\{0\}^n$ with an increment of $\lfloor 2^n/(a^s - 2^n) \rfloor$. Map 2 maps the sequences in $\boldsymbol{E}'$ to the first $a^s - 2^n$ binary strings while Map 3 puts in correspondence $\boldsymbol{E}'$ with the last $a^s - 2^n$ binary strings.

Tables 2–4 report the averaged results from embeddings using Base45 and Base85 encodings into a set of 500 colour images of size 768 × 576 in JPEG, TIFF and PNG formats, respectively. The first column reports the string encoding, the second and third columns report the average payload in bpb and in bps, respectively. The fourth column reports the used mapping, while the fifth column reports when the experimental average payload is greater (+, ++) or less (−, −−) than the expected one. As it can be seen from the values shown in columns 2 and 3, the results are in line with the expected theoretical values (p and q) computed in the previous section.

Table 2. Results of the Base45 and Base85 encodings applied to 500 JPEG colour images (average image size 109 KB).

Base	**Average (±std dev) Payload per Input Byte [bpb]**	**Average (±std dev) Payload per Output Symbol [bps]**	**Map**	**Comparison with Expected Payload**
Base45	0.19905 ± 0.00415	0.13270 ± 0.00277	1	+
Base45	0.19820 ± 0.00397	0.13213 ± 0.00265	2	+
Base45	0.19208 ± 0.00371	0.12805 ± 0.00247	3	−
Base85	0.00838 ± 0.00029	0.00670 ± 0.00024	1	+
Base85	0.00915 ± 0.00080	0.00732 ± 0.00064	2	++
Base85	0.00733 ± 0.00067	0.00586 ± 0.00054	3	-

Table 3. Results of the Base45 and Base85 encodings applied to 500 TIFF colour images (average image size 1297.4 KB).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.22824 ± 0.02025	0.15216 ± 0.01350	1	++
Base45	0.25218 ± 0.08942	0.16812 ± 0.05961	2	++
Base45	0.11456 ± 0.06031	0.07638 ± 0.04021	3	−
Base85	0.00844 ± 0.00099	0.00675 ± 0.00082	1	+
Base85	0.01287 ± 0.01747	0.01031 ± 0.01402	2	++
Base85	0.00660 ± 0.01039	0.00528 ± 0.00831	3	−−

Table 4. Results of the Base45 and Base85 encodings applied to 500 PNG colour images (average image size 1017.8 KB).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.19754 ± 0.00337	0.13169 ± 0.00224	1	+
Base45	0.19779 ± 0.00539	0.13186 ± 0.00359	2	+
Base45	0.19090 ± 0.00313	0.12727 ± 0.00209	3	−
Base85	0.00834 ± 0.00038	0.00667 ± 0.00031	1	+
Base85	0.00687 ± 0.00579	0.00550 ± 0.00469	2	−−
Base85	0.00812 ± 0.00360	0.00650 ± 0.00297	3	-

In almost all cases, Map 1 and Map 2 mappings result in a higher average payload than the expected one for image files. It should be pointed out that for Base85, Map 2 mapping behaves quite differently for TIFF compared to PNG image formats: in the former case, the average payload is much greater ($\sim$ 1.55 times) than the expected one, while in the latter is much smaller (0.84 times).

Similar results for compressed files (in .zip, .bz2, .gz and .tgz formats) of various sizes are reported in Tables 5–8. By contrast with the image case, it seems that the best mapping depends on the compressed file format: for .zip and .bz2 file format, Map 1 and Map 2 give better payload than expected, while for .gz file format Map 3 is the best.

Table 5. Results of the Base45 and Base85 encodings applied to 75 compressed files (format .zip).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.19630 ± 0.01242	0.13087 ± 0.00828	1	+
Base45	0.19785 ± 0.01709	0.13190 ± 0.01139	2	+
Base45	0.19191 ± 0.01683	0.12794 ± 0.01122	3	−
Base85	0.00871 ± 0.00150	0.00697 ± 0.00123	1	+
Base85	0.01066 ± 0.00734	0.00855 ± 0.00593	2	++
Base85	0.00858 ± 0.00189	0.00687 ± 0.00150	3	+

Table 6. Results of the Base45 and Base85 encodings applied to 109 compressed files (format .bz2).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.20293 ± 0.01094	0.13528 ± 0.00729	1	++
Base45	0.20224 ± 0.01068	0.13482 ± 0.00712	2	++
Base45	0.18910 ± 0.01059	0.12606 ± 0.00706	3	−
Base85	0.00917 ± 0.00180	0.00734 ± 0.00146	1	+
Base85	0.01040 ± 0.00285	0.00833 ± 0.00230	2	++
Base85	0.00781 ± 0.00139	0.00625 ± 0.00111	3	−

Table 7. Results of the Base45 and Base85 encodings applied to 28 compressed files (format .gz).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.19213 ± 0.00248	0.12809 ± 0.00165	1	−
Base45	0.19203 ± 0.00285	0.12802 ± 0.00190	2	−
Base45	0.19857 ± 0.00284	0.13238 ± 0.00189	3	+
Base85	0.00818 ± 0.00011	0.00654 ± 0.00009	1	−
Base85	0.00787 ± 0.00042	0.00629 ± 0.00033	2	−
Base85	0.00897 ± 0.00073	0.00718 ± 0.00058	3	+

Table 8. Results of the Base45 and Base85 encodings applied to 112 compressed files (formats: .zip, .bz2, .gz).

Base	Average (±std dev) Payload per Input Byte [bpb]	Average (±std dev) Payload per Output Symbol [bps]	Map	Comparison with Expected Payload
Base45	0.19574 ± 0.01060	0.13049 ± 0.00707	1	=
Base45	0.19669 ± 0.01436	0.13113 ± 0.00957	2	+
Base45	0.19334 ± 0.01420	0.12890 ± 0.00947	3	−
Base85	0.00855 ± 0.00125	0.00685 ± 0.00103	1	+
Base85	0.00985 ± 0.00612	0.00789 ± 0.00495	2	+
Base85	0.00863 ± 0.00160	0.00691 ± 0.00128	3	+

We also performed an evaluation for the embedding into Base45 encoded files (for Base85 encoded files we did not perform such a test, because the statistics required a very large number of data and a histogram for 2^{32} entries). By using a mapping φ onto the set of the most probable sequences in $\boldsymbol{C}$ for the set of 500 JPEG images, the resulting payload is 0.262 bpb, that is significantly larger than with the uniform distribution. Analogously, an optimized mapping for a set of 212 compressed files resulted in a payload of 0.2038 bpb. In this case, it is much closer to the theorical payload (0.1952 bpb) for uniform input data distribution. Obviously, the function φ specifying the mapping onto the most probable sequences in $\boldsymbol{C}$ needs to be shared between encoder and decoder.

Possible Applications

Several possible applications of the presented data hiding framework can be envisaged. Here, we describe a commercial application. Let us suppose we have a set of different kinds of objects, each one described by a multimedia record: for example, the products sold in a shop may be described by their name, producer, weight/capacity, price and, possibly, a small icon. These data can be compressed, e.g., in zip format, and encoded, through Base45, in a sequence of symbols represented by a QR (Quick Response) code that can be printed on or near the object (like the shelf where the object is displayed). To avoid forging of QR codes, the zip compressed data can be signed with ECDSA [21,22] and the signature embedded in the Base45 sequence using the proposed framework (Figure 2).

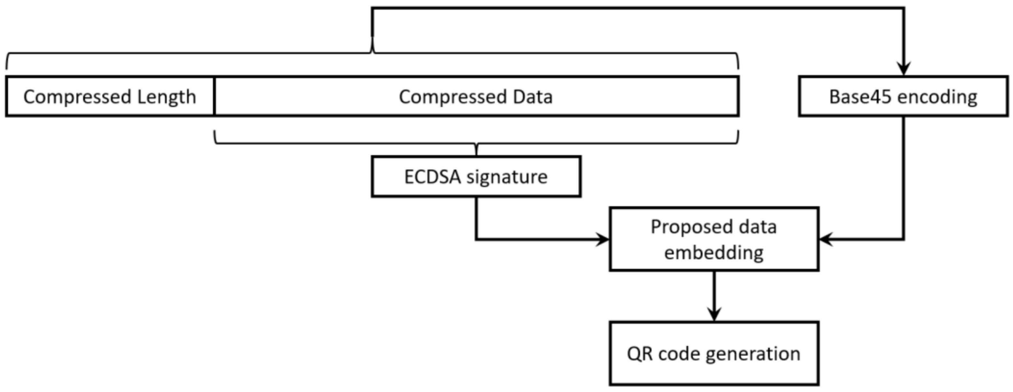

Figure 2. The data flow for the Quick Response (QR) code generation.

We performed some tests showing the possibility to embed an ECDSA signature of 448 bits (computed from a key of length 224 bits) into a compressed Base45 encoded string. In case the compressed string is not long enough for storing the signature, padding bytes are added to complete the signature embedding (the padding bytes are binary strings mapped by β to the set $\boldsymbol{E}'$ to maximize the payload). To correctly restore the original compressed string two bytes containing its length are prefixed before Base45 encoding.

Thus, the QR code may be read (for example, with a smartphone app), the signature and the compressed data extracted from the Base45 encoding, and finally the signature can be verified: if the signature is correct the compressed data are decompressed and the original record is shown to the user to give her the information on the tagged object.

Another application of the proposed method is the embedding of extra data into Base85 encoded files: metadata, signatures, integrity protection data are examples of information that can use the extra space obtained with the proposed framework.

5. Conclusions

In this paper we presented a framework to exploit the intrinsic redundancy present in some binary-to-printable string encodings with the aim of embedding additional data into the output stream. We showed that by using the illegal output configurations it is possible to embed extra information mapping these illegal sequences to some legal ones. The process is reversible, allowing the original cover data and the embedded bit string to be recovered. The payload available depends on the ratio between the number of illegal configurations and the legal ones and on the statistical distribution of the input binary strings: assuming a uniform distribution, which is reasonable when binary-to-printable string encodings are employed like compressed or encrypted data, we have shown that the experimental results are close to the theoretical bounds.

We have applied the proposed framework to two widely used binary-to-printable string encodings, namely Base45 and Base85, showing the applicability of the method and its payload capacity. In particular, we have shown how the method may be applied to data encoded and stored in QR codes.

As a further development we plan to adapt the methodology to distributions of the binary input strings that are not uniform allowing us:

- to use the most probable input strings for bit embedding; and
- to encode several bits for every input string depending on its probability by mapping more than one output sequence.

Author Contributions: Both authors equally contributed to the research development and the writing, proofreading and editing of this manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research has been supported by the Italian Ministero dell'Università e della Ricerca.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cox, I.J.; Miller, M.L.; Bloom, J.A.; Fridrich, J.; Kalker, T. *Digital Watermarking and Steganography*, 2nd ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2008.
2. Pomponiu, V.; Botta, M.; Cavagnino, D. Data Hiding in the Wild: Where Computational Intelligence Meets Digital Forensics. In *Surveillance in Action: Technologies for Civilian, Military and Cyber Surveillance*; Karampelas, P., Bourlai, T., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 301–331. ISBN 978-3-319-68533-5. [CrossRef]
3. Khan, A.; Siddiqa, A.; Munib, S.; Malik, S.A. A recent survey of reversible watermarking techniques. *Inf. Sci.* **2014**, *279*, 251–272. [CrossRef]
4. Megías, D.; Mazurczyk, W.; Kuribayashi, M. Data Hiding and Its Applications: Digital Watermarking and Steganography. *Appl. Sci.* **2021**, *11*, 10928. [CrossRef]
5. Mahto, D.K.; Singh, A.K. A survey of color image watermarking: State-of-the-art and research directions. *Comput. Electr. Eng.* **2021**, *93*, 107255. [CrossRef]
6. Jain, R.; Trivedi, M.C.; Tiwari, S. Digital Audio Watermarking: A Survey. In *Advances in Computer and Computational Sciences. Advances in Intelligent Systems and Computing*; Bhatia, S., Mishra, K., Tiwari, S., Singh, V., Eds.; Springer: Singapore, 2018; Volume 554, pp. 433–443. [CrossRef]
7. Banyal, S.; Sharma, S. Survey on Digital Video Watermarking Techniques. *Int. J. Adv. Res. Comput. Commun. Eng.* **2016**, *5*, 100–103. [CrossRef]
8. Li, Y.; Wang, H.; Barni, M. A survey of deep neural network watermarking techniques. *arXiv* **2021**, arXiv:2103.09274v1. [CrossRef]
9. Kamaruddin, N.S.; Kamsin, A.; Por, L.Y.; Rahman, H. A Review of Text Watermarking: Theory, Methods, and Applications. *IEEE Access* **2018**, *6*, 8011–8028. [CrossRef]
10. Fältström, P.; Ljunggren, F.; van Gulik, D.-W. The Base45 Data Encoding. Internet-Draft. IETF. 2021. Available online: https://datatracker.ietf.org/doc/html/draft-faltstrom-base45-07 (accessed on 29 September 2021).
11. Adobe Systems Incorporated. *PostScript®Language Reference*, 3rd ed.; Addison-Wesley Publishing Company: Boston, MA, USA, 1999.
12. Elz, R. A Compact Representation of IPv6 Addresses. RFC 1924, RFC Editor. 1996. Available online: https://rfc-editor.org/rfc/rfc1924.txt (accessed on 29 September 2021).
13. Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor. 2006. Available online: https://rfc-editor.org/rfc/rfc4648.txt (accessed on 29 September 2021).
14. He, D.; Sun, Y.; Jia, Z.; Yu, X.; Guo, W.; He, W.; Qi, C.; Lu, X. A Proposal of Substitute for Base85/64–Base91. In Proceedings of the SUMMER 8th International Conference on Computing, Communications and Control Technologies: CCCT 2010, Orlando, FL, USA, 29 June 2010.
15. Henke, J. Base91 Encoding. Available online: http://base91.sourceforge.net/ (accessed on 22 November 2021).
16. Antonopoulos, A.M. *Mastering Bitcoin*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017; ISBN 9781491954386.
17. Nakamoto, S.; Sporny, M. The Base58 Encoding Scheme. Internet-Draft. IETF. 2021. Available online: https://datatracker.ietf.org/doc/html/draft-msporny-base58-03 (accessed on 29 September 2021).
18. Wu, P.-C. A base62 transformation format of ISO 10646 for multilingual identifiers. *Softw. Pract. Exp.* **2001**, *31*, 1125–1130. [CrossRef]
19. He, K.; Xu, X.; Yue, Q. A secure, lossless, and compressed Base62 encoding. In Proceedings of the 2008 11th IEEE Singapore International Conference on Communication Systems, Guangzhou, China, 19–21 November 2008; pp. 761–765. [CrossRef]
20. Rocchi, P. How 'unused' codewords make a redundant code. In Proceedings of the 45th Annual Southeast Regional Conference ACM-SE 45, Winston-Salem, NC, USA, 23–24 March 2007; pp. 407–412. [CrossRef]
21. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*; ANSI X9.62-2005; American National Standards Institute: New York, NY, USA, 2005.
22. Pornin, T.; Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, RFC Editor. 2013. Available online: https://www.rfc-editor.org/info/rfc6979 (accessed on 29 September 2021).