applied sciences

MDPI

Communication

Escaping Printable Encoded Streams to Embed Out-of-Band Data

Marco Botta and Davide Cavagnino *

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

Abstract: In this paper, we propose to exploit the unused configurations of a printable encoding such as Base41, Base45 or Base85 to create a side channel that can store extra data such as error detection or correction codes, integrity verification and authentication information or application defined data. After introducing the encoding of binary octet strings in printable form, we present some case studies that show possible applications of the unused configurations.

Keywords: data embedding; data hiding; error correction; integrity verification; printable encoding; string encoding; unused configurations.

1. Introduction

Printable string encoding of binary data is a well-known and widely spread technique used by systems designed to manage only printable characters: in other words, printable string encoding is an encapsulation method to process, in a transparent way, any possible bit string by systems able to treat only printable strings (for example, some mail servers).

Typical examples of this are old mail servers that are not able to directly accept binary data and the QR-code representation of the European Union Digital COVID Certificate.

To allow the storage and transmission of binary data with printable strings, many encodings have been proposed: in Section 3, some of these works will be recalled, emphasizing the use of the legal strings and computing the space of unused configurations that allow the embedding of additional information.

In general, the approach is to define an alphabet made of symbols that are used to compose strings, each one associated with a binary configuration to be encoded. For example, Base41 [1,2] uses three symbols from an alphabet of 41 printable characters to encode a pair of octets: in this case the possible printable strings are $41^3 = 68{,}921$, while the configurations of pair of octets are $2^{16} = 65{,}536$, leaving $41^3 - 2^{16} = 3385$ free printable strings that, in general, are considered "illegal" but may be employed to store additional data such as a Cyclic Redundancy Check (CRC) value, a cryptographic hash, a Message Authentication Code (MAC), or a digital signature.

The main contributions of this paper are as follows:

- Showing the redundancy present in some printable encodings;
- Developing a methodology to embed extra data in a printable encoded stream;
- Showing the effectiveness of embedding these data for security purposes;
- Showing some other applications that leverage unused printable strings.

The following Section 2 will establish a uniform nomenclature and notation to be used throughout the paper, after which Section 3 will present some printable encoding methods and applications. Section 4 will give details on some of the encodings that allow the embedding of extra (i.e., payload) data and will present some methods for performing this operation. Section 5 will present a numerical analysis for some embodiments of the case studies introduced in Sections 4 and 6 will discuss some conclusions on the proposed methodology and its applications.

check for updates

Citation: Botta, M.; Cavagnino, D. Escaping Printable Encoded Streams to Embed Out-of-Band Data. *Appl. Sci.* **2023**, *13*, 6926. https://doi.org/10.3390/app13126926

Academic Editor: Paolino Di Felice

Received: 8 May 2023
Revised: 23 May 2023
Accepted: 3 June 2023
Published: 8 June 2023

2. Nomenclature and Notation

In this section, we briefly recall some nomenclature and notation to have a uniform and clear definition and representation of the entities involved in this paper.

A *symbol* or *character* is a graphical representation of an abstract or real entity or concept.

An *alphabet* is an ordered, finite size collection of distinct symbols.

A *set* is a collection of distinct items, or elements, that in the present context will be symbols or characters.

A *sequence* or *string* will refer to an ordered collection of symbols from an alphabet. In particular, a sequence of symbols from an alphabet having cardinality 2 is called *binary string*. A string of characters is written within double quotes, e.g., "ABC" represents the string of the first three symbols of the Latin alphabet (capital letters).

To refer to instances of the previous entities, variables, or their properties, we denote them with the following rules:

- Alphabet: uppercase boldface italic letter, e.g., $\boldsymbol{A}$;
- Sequence of symbols from an alphabet: uppercase letter, e.g., S, S_1;
- Set: uppercase calligraphic letter, e.g., $\mathcal{B}$;
- Cardinality of a set: the function **card**() counting the number of elements in a set, e.g., **card**($\mathcal{B}$);
- Floor operation: the operator $\lfloor\ \rfloor$ defining the integer number not greater that its argument, e.g., $\lfloor \pi \rfloor = 3$;
- Bijection between sets: the symbol $\Leftrightarrow$, e.g., $\mathcal{A} \Leftrightarrow \mathcal{B}$;
- Constant or single scalar value: lowercase italic letter, e.g., v.

Throughout this document, BaseYY will denote an encoding method based on an alphabet of YY symbols: for example, Base41 refers to an encoding method based on 41 symbols.

3. Related Works

In the first part of this section, we will recall some printable encodings, giving emphasis to those that have unused configurations that can be exploited to represent extra data. Then, given that the present proposal is concerned to the employment of printable sequences to stuff extra data, the second part will discuss some works that embed payload information into textual data.

One of the most widely used printable encodings is Base64 [3], which represents three binary octets with four base 64 symbols, also dealing with an input length not multiple of three using the special symbol "=". The same paper [3] presents a base 32 encoding that represents five octets with a sequence of eight symbols taken from an alphabet of 33 (32 plus "=" that is used for padding when the input sequence has a length that is not a multiple of five). Furthermore, Ref. [3] discussed the base 16 that is essentially the well-known hexadecimal representation.

The use of base 41 is the core of [1,2,4]. The proposed encodings emplxoy two different alphabets of 41 symbols; moreover, Refs. [1,2] also discussed bit strings of arbitrary length, i.e., not necessarily having a length of a multiple of eight. In particular, Refs. [1,2] encoded a pair of octets, a single octet, and a string of length from 0 to 7 bits with three Base41 symbols and, as previously cited, leaves 3385 free (unused) printable Base41 strings.

Base 45 iswas used in [5] for encoding data to be represented by QR codes. The proposed method expresses a pair of binary octets with three symbols from an alphabet of 45. A special coding is reserved for a single octet in case the original stream has an odd number of octets. This code uses only $2^{16} = 65{,}536$ triplets of the $45^3 = 91{,}125$ available: this redundancy was used in [6,7] to reversibly embed data in a Base45 encoded stream.

Base 85 was used in two works: Ref. [8] used an alphabet of 85 printable symbols to obtain an efficient and compact representation of IPv6 addresses, while [9] used the alphabet made of 85 ASCII characters, from code 33 to code 117, to define an encoding called Ascii85 that represents a quadruple of octets with five Base85 symbols. Given the

mapping performed by [9], there are $85^5 - 2^{32} = 142{,}085{,}829$ unused configurations for this encoding.

Ninety-one printable characters are employed in two Base91 encodings [10,11]. The software available at [10] encodes blocks of 13 bits with pairs of symbols from a Base91 alphabet: given that $91^2 = 8281$ Base91 pairs exceed by 89 the configurations of 13 bits ($2^{13} = 8192$), if the block has a value not greater than 88, then one more bit is encoded; in this way, all the 91^2 Base91 pairs are used for encoding a bit stream in printable form. On the other hand, Ref. [11] encoded groups of 13 bits with two Base91 characters but used 12 pairs of the exceeding 89 to indicate the length of the last group of bits (in case it has a length different from 13): in this encoding, $89 - 12 = 77$ pairs are left unused.

With the proposal of this paper, any of the printable encodings that leave unused configurations may be utilized to embed out-of-band data.

In the field of data hiding in textual data many works have been developed to store a payload into a text written with a word processor: in general, non-printable and empty characters or various kinds of white spaces are used to encode binary information.

For example, Ref. [12] developed UniSpaCh that works on Microsoft® Word documents, inserting different Unicode spacing characters between words, sentences, and paragraphs: this method adds (non-visible) characters to the file increasing its size as our method does. The feature of change tracking in Microsoft® Word documents was exploited in [13] to hide a secret message for steganographic communication.

In [14], the two non-printing characters *zero-width joiner* (ZWJ) and *zero-width non-joiner* (ZWNJ) were employed to store information in a text: a binary information can be embedded if ZWJ and ZWNJ are used to represent the two states, but the paper also proposes an encoding that exploits longer sequences of ZWJ and ZWNJ to save characters from the Latin alphabet.

Ref. [15] merged the approach in [12] with the use of a *zero-width character* (ZWC): different combinations of Unicode spaces are used to embed bit pairs between words, sentences, lines, and paragraphs, and the payload is increased considering also the possibility to store ZWCs between words and sentences.

Modifications to the colors of printed characters were employed in [16] to embed a message in a document that will be printed and successively scanned to extract the hidden information: the paper discusses text color modulation (TCM), defining a model for the process of printing and successive scanning (PS model) and defines embedding and detection methods that save the information in the channels red and blue with respect to the value of the green channel.

4. Printable Encodings and Case Studies of Payload Data Embedding

Consider the set $\mathcal{B}$ of all binary strings of length n bits; thus, $\mathbf{card}(\mathcal{B}) = 2^n$. Furthermore, having an alphabet A of t printable symbols compute the value v such that:

$$vs. = \min\left\{k \mid 2^n \leq t^k\right\} \tag{1}$$

and define the set $\mathcal{S}$ of all sequences of v symbols from the alphabet A: obviously, $\mathbf{card}(\mathcal{S}) = t^v$.

Using 2^n different sequences from $\mathcal{S}$, it is possible to encode all the bit strings in $\mathcal{B}$ using only symbols from A. It follows that there will be a subset $\mathcal{E}$ of $\mathcal{S}$ ($\mathcal{E} \subseteq \mathcal{S}$), whose elements are in one-to-one correspondence with the binary strings of $\mathcal{B}$, that is, there is a bijection between $\mathcal{E}$ and $\mathcal{B}$, $\mathcal{E} \Leftrightarrow \mathcal{B}$.

Table 1 reports the characterizing values for some printable encodings.

The set $\mathcal{U} = \mathcal{S} - \mathcal{E}$, which contains the unused sequences of $\mathcal{S}$, will have $\mathbf{card}(\mathcal{U}) = t^v - 2^n$. From Table 1, it may be observed that this set $\mathcal{U}$ is non-empty for Base41 [1], Base45 [5], Base85 [9], and Base91 [11].

Table 1. Some printable encodings with their main parameters and number of unused sequences.

Encoding	Encoded Bit Length n	Number of Encoding Symbols v	Alphabet Cardinality t	Number of Unused Sequences $t^v - 2^n$
Base16 [3]	8	2	16	0
Base32 [3]	40	8	32	0
Base64 [3]	24	4	64	0
Base41 [1]	16	3	41	3385
Base45 [5]	16	3	45	25,589
Base85 [9]	32	5	85	142,085,829
Base91 [11]	13	2	91	77

As previously said, in [6], the sequences in $\mathcal{U}$ are employed for reversibly embedding data into a Base45 or Base85 encoded stream.

Here, we propose a general framework for exploiting the unused sequences in several contexts, allowing applications to choose the most appropriate setting for their own purposes. Therefore, every application must define the meaning assigned to every unused sequence and how to process it. Suppose to encode binary sequences of n bits with v symbols belonging to an alphabet $\boldsymbol{A}$ (v is determined as in Equation (1)). If $\mathcal{U} \neq \varnothing$ (see, for example, the encodings with a non-zero value in the last column of Table 1), an application selects a set of sequences $\mathcal{Z} \subseteq \mathcal{U}$ and assigns a meaning to every sequence $\mathrm{S} \in \mathcal{Z}$. The semantics of each sequence must be known to both the encoder and decoder and agreed upon to have a correct transmission and extraction of the encoded data.

As will be shown later on, a sequence $\mathrm{S} \in \mathcal{Z}$ may represent:

a. A string of bits encoding the whole or part of a Cyclic Redundancy Check (CRC) code;
b. A prefix indicating that a fixed number of following sequences encode a CRC, a Message Authentication Code, or a digital signature;
c. One or more bits to be transmitted separately from the data encoded by the sequences belonging to $\mathcal{E}$;
d. A separator to split portions of the data stream encoded by the sequences in $\mathcal{E}$;
e. An identifier specifying the characteristics of a portion of following sequences;
f. A context defining the meaning of the following sequences $\mathrm{S} \in \mathcal{Z}$.

For instance, an application that uses Base41 printable encodings can decide that the sequence "zxx" is a prefix indicating that the next two sequences represent a 32 bit CRC. Note that different applications can assign different meanings to the same sequence from $\mathcal{Z}$.

The next subsections will present some possible embodiments using the previously introduced representations.

4.1. Error Detection and Correction Information Embedding

The stream of printable encoded data may be stuffed with sequences belonging to $\mathcal{U}$ that encode a Cyclic Redundancy Check (CRC) [17] of a portion of data that has to be controlled for errors.

It is possible to encode a CRC of length $l \leq \lfloor \log_2(t^v - 2^n) \rfloor$ bits using a subset $\mathcal{C}$ of 2^l sequences in $\mathcal{U}$ associating every CRC binary string of length l to one sequence in $\mathcal{C}$ (Figure 1a). In this case, the proposed framework is instantiated with $\mathcal{Z} = \mathcal{C}$.

The maximum values of l for the encodings in Table 1 are 11 for Base41, 14 for Base45, and 27 for Base85. Longer CRC codes may be stuffed by simply concatenating more unused sequences (Figure 1b) and also in this case $\mathcal{Z} = \mathcal{C}$ or, considering a single unused sequence S_c, $\mathcal{Z} = \{\mathrm{S}_c\}$, as a preamble for a fixed number of legal sequences belonging to $\mathcal{E}$ each carrying n bits of the CRC (Figure 1c) (see [18] for a comprehensive list of CRC polynomials).

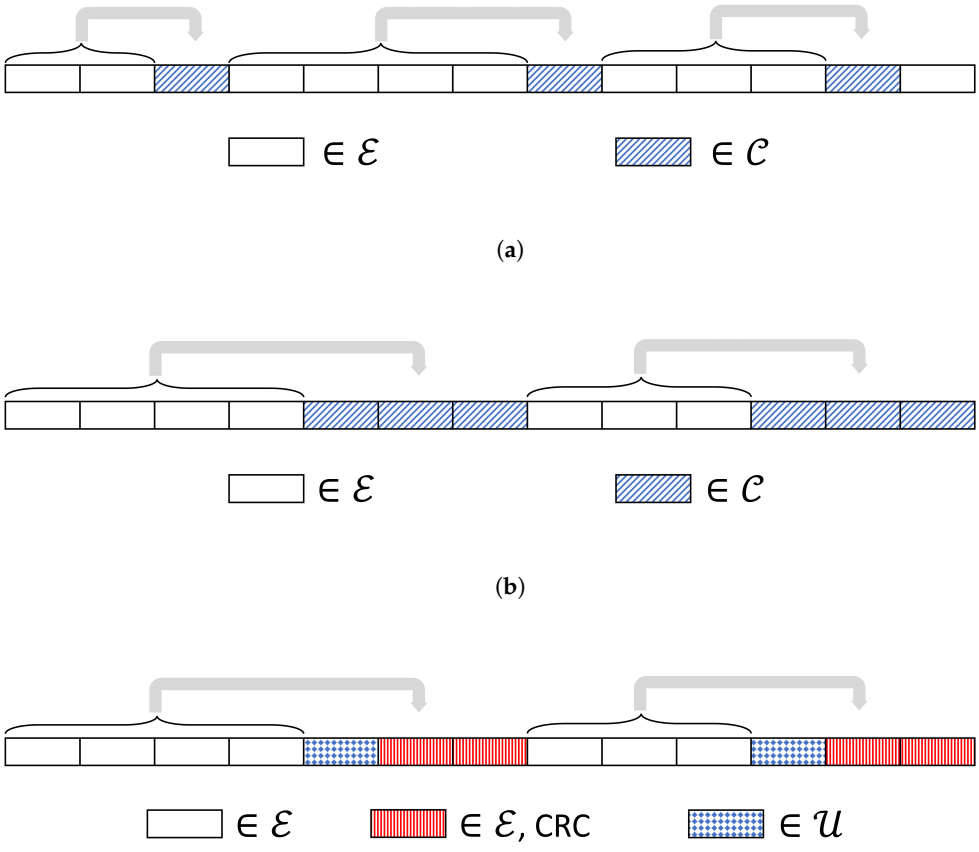

Figure 1. Simple schemes to show possible encodings of CRC codes (gray arrows show the sequences covered by the CRC). (**a**) CRC encoded in a single unused sequence. (**b**) CRC encoded in multiple unused sequences. (**c**) CRC encoded in an unused sequence and multiple legal sequences.

Example 1. *Considering the Base41 encoding [1], an implementation of Figure 1a is to employ* 2048 *of the* 3385 *unused sequences available to stuff CRCs of length* $l = \log_2 2048 = 11$ *bits computed on the previous bit string for error detection.*

Example 2. *Using the same Base41 encoding [1], an implementation of Figure 1b is to employ* 2048 *of the* 3385 *unused sequences available and concatenate* three *of them to stuff CRCs of length* $3\,l = 3 \log_2 2{,}048 = 33$ *bits computed on the previous bit string for error detection.*

Example 3. *A possible implementation of Figure 1c with Base45 [5] is to employ one of the* 25,589 *unused sequences available (see Table 1) to specify that the following two sequences belonging to* $\mathcal{E}$ *(each one encoding* 16 *bits) will encode a* $2 \times 16 = 32$ *bits CRC.*

4.2. Integrity Information, Message Authentication Code, and Digital Signature Embedding

The printable encoded data may be stuffed and/or terminated with security information such as a cryptographic hash, a Message Authentication Code (MAC), or a signature covering the whole or a portion of the encoded data. Due to the bit length of these binary strings, it is more efficient to employ three unused sequences S_h, S_m, S_{ds} from $\mathcal{U}$ to specify the type of security information, respectively, hash, MAC, and signature, encoded in the following sequences and then use a fixed number of sequences in $\mathcal{E}$ to store the hash, the MAC, or the signature (Figure 2). In this case, $\mathcal{Z} = \{S_h, S_m, S_{ds}\}$.

Figure 2. Simple scheme to show possible encodings of security information for data protection (gray arrows show the sequences covered by the hash, MAC, or digital signature).

Example 4. *As shown in Figure 2, a single unused sequence S_{h1} of the Base41 encoding [1] may be employed to specify that the following eight sequences belonging to $\mathcal{E}$ (each one encoding* 16 *bits) will store a* $8 \times 16 = 128$ *bits hash, such as MD5 [19]. Furthermore, another unused sequence S_{h2} of the Base41 encoding can be utilized to indicate that the following sixteen sequences belonging to $\mathcal{E}$ (each one representing* 16 *bits) will encode a* $16 \times 16 = 256$ *bits hash such as SHA3-256 [20]. In this case,* $\mathcal{Z} = \{S_{h1}, S_{h2}\}$.

4.3. Secondary Data Channel

It is possible to create a second data channel that carries information, such as a watermark, using the sequences in the previously defined set $\mathcal{C}$ ($\mathcal{Z} = \mathcal{C}$): every sequence represents l bits of information and may be interleaved anywhere in the encoded data stream being recognizable and distinguishable from data transformed in printable form (Figure 3).

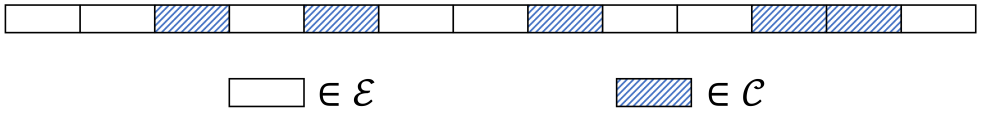

Figure 3. Secondary channel information interleaved in printable encoded data.

Example 5. *Suppose a desire to store extra data in a Base85 [9] encoded stream. Exploiting the* 142,085,829 *unused sequences (see Table 1), it is possible to encode* $l = \lfloor \log_2 142{,}085{,}829 \rfloor = 27$ *bits with an unused sequence of five characters. These can be inserted anywhere in the normal flow of Base85 sequences creating a secondary channel that, for example, can carry RGB colors (expressed with* 8 *bits per channel for a total of* 24 *bits).*

4.4. Parameter Separation

A printable encoding may be also employed to encode parameters passed to a function in a context where binary data cannot be directly transmitted, for example, in the query string of a Web address. To separate the various encoded parameters, it is possible to use a single sequence S_d belonging to the previously defined set $\mathcal{U}$ and another sequence S_t from the same set to indicate the end of the parameters (Figure 4). The framework is instantiated with $\mathcal{Z} = \{S_d, S_t\}$.

Another possibility is to identify the data types of the various parameters employing sequences from the set $\mathcal{U}$ (Figure 5): for example, it is possible to use a use sequence $S_i \in \mathcal{U}$ to identify an integer, another sequence $S_f \in \mathcal{U}$ to specify a float, then $S_o \in \mathcal{U}$ to specify an octet string, $S_p \in \mathcal{U}$ to express a binary pointer, and two sequences $S_{rs}, S_{re} \in \mathcal{U}$ to indicate the beginning and the end of a record made of fields in turn identified with these delimiters (with a possible recursive structure). The parameter's list can be terminated with the sequence S_t from the same set $\mathcal{U}$. In this case, the framework is instantiated with $\mathcal{Z} = \{S_i, S_f, S_o, S_p, S_{rs}, S_{re}, S_t\}$.

Nonetheless, the encodings proposed in Sections 4.1 and 4.2 may be used as an additional data protection feature for the parameters, taking care to choose S_i, S_f, S_o,

S_p, S_{rs}, S_{re}, S_d, and S_t among the sequences in $\mathcal{U}$ **not** encoding a CRC (Figure 1) **nor** a type of hash, MAC, or digital signature (Figure 2). The proposed framework has $\mathcal{Z} = \left\{S_i, S_f, S_o, S_p, S_{rs}, S_{re}, S_t, S_c, S_h, S_m, S_{ds}\right\}$.

Example 6. *Assume having a program running on a Web server that needs a (variable) set of parameters in binary form. In this case, the various data can be encoded with Base41 [1] and sent as a query string to the program, separating the various parameters with a single sequence from $\mathcal{U}$ and terminating the parameter list with another sequence in $\mathcal{U}$. At the receiving side, the program can split the data using the separator and recover the original binary values decoding the Base41 strings.*

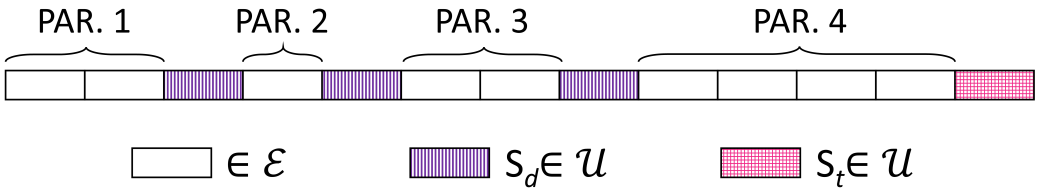

Figure 4. Possible encoding of parameters with separators S_d and S_t.

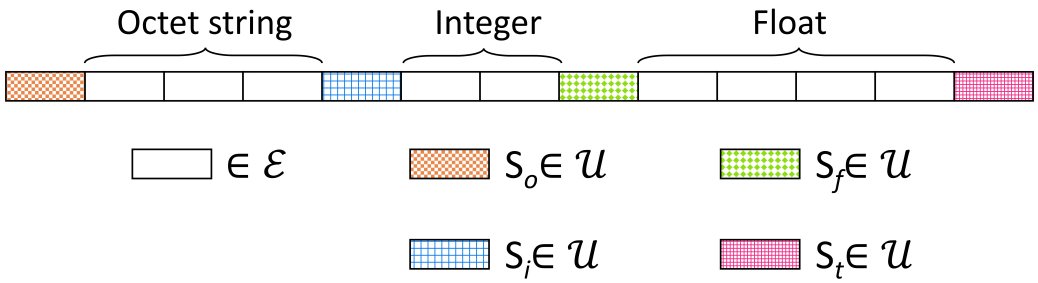

Figure 5. Identifying types of parameters with separators.

5. Discussion and Results

In this section, we perform some numerical computations on some possible practical applications of the proposed method to printable encoded streams.

5.1. CRC Embedding

In the first run of tests, we considered adding an 11 bits CRC to blocks of data encoded in printable form with Base41 [1]. The method adds three octets to the Base41 encoding of the block; thus, if the block has size n octets ($8\,n$ bits), then the Base41 encoding inflates it to $1.5\,n$ octets, adding the CRC leads to $1.5\,n + 3$ octets with an overload of $\frac{3}{1.5\,n+3} \times 100\%$. On the other hand, an 11 bits CRC on a block of $8\,n$ bits represents an overload of $\frac{11}{8\,n+11} \times 100\%$. Analogous formulas can be derived for 14 bits CRC and employing Base45 unused sequences.

We performed the computation of the overload for blocks of sizes 128, 256, 512, and 1024 bits (or 16, 32, 64, and 128 octets, respectively). Table 2 shows the resulting overloads for CRCs embedded into Base41 and Base45 encodings as proposed, comparing them with the classical overload had when embedding a CRC of (11 and 14 bits, respectively). From these data, it may be seen that the increase in overload is quite limited and feasible for an application level error detection and data protection from unintentional modifications.

Table 2. Computation and comparison of CRC overloads for Base41 and Base45 encoded CRCs (11 and 14 bits, respectively).

Block Size [bits]	Block Size [octets]	Encoded Block Size [octets]	Encoded Block Size with CRC [octets]	CRC Overload (Proposed Method) %	CRC Overload (Standard Method, 11 bits) %	CRC Overload (Standard Method, 14 bits) %
128	16	24	27	11.2	8.0	9.9
256	32	48	51	5.9	4.2	5.2
512	64	96	99	3.1	2.2	2.7
1024	128	192	195	1.6	1.1	1.4

5.2. Hash Embedding

Let us now examine a Base41 or a Base45 encoding: three printable characters encode two octets (apart from a single octet encoded when the stream length is not even). An MD5 hash [19] has a length of 16 octets, and thus, $3 + 3 \times 16/2 = 27$ octets may encode a file MD5 hash. Furthermore, a SHA-1 hash [21] has a length of 20 octets, and thus, $3 + 3 \times 20/2 = 33$ octets may encode a file SHA-1 hash.

Considering Base41 [1], we may assign the unused sequence S_{MD5} = "`zzM`" to indicate that the following 24 characters encode an MD5 hash and the unused sequence $S_{SHA\text{-}1}$ = "`zzS`" to indicate that the following 33 characters encode a SHA-1 hash (in this embodiment, the framework is instantiated with $\mathcal{Z} = \{S_{MD5}, S_{SHA\text{-}1}\}$). The impact on the size of the resulting encoding is, in both cases, only of three octets due to the escaping sequence (in this case, "`zzM`" or "`zzS`").

5.3. Extra Data Attachment

As a practical instance of Example 5, let us consider the use of Base85 to represent the pixels of an RGB color image to be appended to an Ascii85 encoded stream. Building $\mathcal{Z}$ with 16,777,216 sequences, and thus, $\mathbf{card}(\mathcal{Z}) = 16{,}777{,}216$ of the 142,085,829 unused ones, it is possible to printable encode the pixels of the image: if the image dimensions are $320 \times 240 = 76{,}800$ pixels, then the size (inflated with a ratio 5:3) of the uncompressed image will be $76{,}800 \times 5 = 384{,}000$ or octets.

5.4. Client-Server Parameter Passing

Let us consider passing a variable number of parameters from a Web client to a server. As a concrete example, suppose conveying a 16 bit integer valued 41, a 16 bit integer valued 65,535 and a character string valued "BASE". Having built $\mathcal{Z}$ with the unused sequences "`xBA`", "`xBB`", "`xBC`", "`xBD`", "`xBF`", "`xBG`", "`xBH`", and "`xBJ`" to represent S_i, S_f, S_o, S_p, S_{rs}, S_{re}, S_d, and S_t, respectively, to perform an encoding that follows the proposal shown in Figure 5, the resulting printable stream will be:

`xBA ABA xBA vzV xBC MDk Qiv xBJ`

Corresponding to:

S_i 41 S_i 65,535 S_o "BA" "SE" S_t

It is obvious that the resulting Base41 string can be immediately and unambiguously decoded by a procedure aware of the Base41 encoding symbols assignment and expecting the corresponding parameters.

One disadvantage is that the insertion of extra data in the encoding increases the size of the processed stream and this might be limiting the application on low-capacity links or small-capacity devices.

Concerning the security issues of the proposed framework, it should be pointed out that any printable encoding presents the same security issues, being just an encoding. We merely present a way in which an application can make use of unused configuration to insert extra information in the encoding. When this extra information is a MAC or a signature, the encoded data are protected against modification attacks. Transferring the resulting encoding in a secure way is out of the scope of the present work, and mainly relies on the use of proper security measures (for instance, cryptography, secure protocols such as https, SSL, and TLS, etc.).

6. Conclusions

In this paper, we proposed to exploit the unused configurations of some printable encodings to carry extra information that may be employed for the following:

- Error detection and correction of data at application level;
- Carrying a cryptographic hash, a Message Authentication Code, or a digital signature for integrity protection and origin authentication;
- Carrying extra payload data building a secondary communication channel;
- Transferring parameters in (remote) function calls.

The data hiding method in [6,7] is a special case of the proposed framework (case c. in the list). In particular, the sequences in $\mathcal{Z} = \mathcal{U}$ carry a watermark bit valued '1', and each of them is associated with one, and only one, sequence of $\mathcal{E}$ carrying a '0'-valued watermark bit.

As we already implemented the payload embedding in [6,7], we plan to develop functions that allow the storage/extraction of integrity/security data and the secure communication of parameters in function calls in Web browsers.

Author Contributions: All the authors gave the same contribution in all aspects of this paper. All authors have read and agreed to the published version of the manuscript.

Funding: This work 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: Data sharing not applicable.

Conflicts of Interest: The authors declare that they have no conflict of interest.

References

1. Botta, M.; Cavagnino, D. Base41: A proposal for printable encoding of bit strings. *Eng. Rep.* **2023**, *5*, e12606,
2. Botta, M.; Cavagnino, D. Base41: A Method for Bit String Encoding in Printable Form. 2023. Available online: https://watermarking.di.unito.it/base41.html (accessed on 2 May 2023).
3. Josefsson, S. *RFC 4648*; The Base16, Base32, and Base64 Data Encodings; RFC Editor: Phoenix, AZ, USA, 2006. [CrossRef]
4. Veljkovic, S. Base41. 2014. Available online: https://github.com/sveljko/base41 (accessed on 27 March 2023).
5. Fältström, P.; Ljunggren, F.; van Gulik, D.W. *RFC 9285*; The Base45 Data Encoding; RFC Editor: Phoenix, AZ, USA, 2022. . [CrossRef]
6. Botta, M.; Cavagnino, D. A Framework for Reversible Data Embedding into Base45 and Other Non-Base64 Encoded Strings. *Appl. Sci.* **2022**, *12*, 241. [CrossRef]
7. Botta, M.; Cavagnino, D. Improving data embedding capacity into Base45 encoded strings. *Eng. Rep.* **2023**, e12622.
8. Elz, R. *RFC 1924*; A Compact Representation of IPv6 Addresses; RFC Editor: Phoenix, AZ, USA, 1996. [CrossRef]
9. Adobe Systems Incorporated. *PostScript Language Reference*, 3rd ed.; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.
10. Henke, J. basE91 Encoding. 2006. Available online: https://base91.sourceforge.net/ (accessed on 28 April 2023).
11. 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 Proceedings of the SUMMER 8th International Conference on Computing, Communications and Control Technologies: CCCT, 2010, Orlando, FL, USA, 29 June–2 July 2010.
12. Por, L.Y.; Wong, K.; Chee, K.O. UniSpaCh: A text-based data hiding method using Unicode space characters. *J. Syst. Softw.* **2012**, *85*, 1075–1082. [CrossRef]

13. Liu, T.Y.; Tsai, W.H. A New Steganographic Method for Data Hiding in Microsoft Word Documents by a Change Tracking Technique. *IEEE Trans. Inf. Forensics Secur.* **2007**, *2*, 24–30. [CrossRef]
14. Ali, A.E. A New Text Steganography Method By Using Non-Printing Unicode Characters. *Eng. Tech. J.* **2010**, *28*, 72–83.
15. Aman, M.; Khan, A.; Ahmad, B.; Kouser, S. A hybrid text steganography approach utilizing Unicode space characters and zero-width character. *Int. J. Inf. Technol. Secur.* **2017**, *9*, 85–100.
16. Borges, P.V.K.; Mayer, J.; Izquierdo, E. Robust and Transparent Color Modulation for Text Data Hiding. *IEEE Trans. Multimed.* **2008**, *10*, 1479–1489. [CrossRef]
17. Peterson, W.W.; Brown, D.T. Cyclic Codes for Error Detection. *Proc. IRE* **1961**, *49*, 228–235. [CrossRef]
18. Koopman, P. Best CRC Polynomials. 2018. Available online: https://users.ece.cmu.edu/~koopman/crc/ (accessed on 2 May 2023).
19. Rivest, R.L. *RFC 1321*; The MD5 Message-Digest Algorithm; RFC Editor: Phoenix, AZ, USA, 1992. [CrossRef]
20. Dworkin, M. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2015. [CrossRef]
21. *FIPS Pub 180-1*. Secure Hash Standard. National Institute of Standards and Technology: Gaithersburg, MD, USA, 1995.