Compcrypt – Lightweight ANS-based Compression and Encryption

Seyit Camtepe[2], Jarek Duda[3], Arash Mahboubi[4], Paweł Morawiecki[1], Surya Nepal[2], Marcin Pawłowski[1], and Josef Pieprzyk[1,2]

[1] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[2] Data61, CSIRO, Sydney, Australia
[3] Institute of Computer Science and Computer Mathematics, Jagiellonian University, Cracow, Poland
[4] School of Computing and Mathematics, Charles Sturt University, Port Macquarie, Australia

Abstract. Compression is widely used in Internet communication to save communication time and bandwidth. Recently invented by Jarek Duda asymmetric numeral system (ANS) offers an improved efficiency and a close to optimal compression. The ANS algorithm has been deployed by major IT companies such as Facebook, Google and Apple. Compression by itself does not provide any security (such as confidentiality or authentication of transmitted data). An obvious solution to this problem is an encryption of compressed bitstream. However, it requires two algorithms: one for compression and the other for encryption.

In this work, we investigate natural properties of ANS that allow to incorporate authenticated encryption using as little cryptography as possible. We target low-level security communication such as transmission of data from IoT devices/sensors. In particular, we propose three solutions for joint compression and encryption (compcrypt). All of them use a pseudorandom bit generator (PRGB) based on lightweight stream ciphers. The first solution applies state jumps controlled by PRGB. The second one employs two ANS algorithms, where compression switches between the two. The switch is controlled by a PRGB bit. The third compcrypt modifies the encoding function of ANS depending on PRGB bits. Security and efficiency of the proposed compcrypt algorithms are evaluated.

1 Introduction

It is a common knowledge that a majority of Internet transmission is highly redundant. Popular video/audio streaming applications such as radio, TV, Skype/Zoom/Webex teleconferencing, Netflix/Stan entertainment providers, Facebook social platforms, medical remote diagnosis and monitoring, and remote teaching are all good examples Internet applications, which transmit and process highly redundant data. To save communication bandwidth and make transmission faster, a redundant stream is compressed before sending to a receiver. Upon reception, the receiver recovers the original (redundant) stream from the compressed one. Note that we are talking about lossless compression, where the receiver is able to recreate the original/uncompressed data. Note that video/audio compression is usually lossy.

Theoretical underpinning of compression is deeply rooted in Information Theory initiated by Shannon's seminal work [9]. The first compression algorithm invented by Huffman [5] and known as the Huffman code (HC) shows optimal compression for symbol streams, whose probabilities follow very specific patterns (i.e. natural powers of $\frac{1}{2}$). Arithmetic coding (AC) (and many its variants – see [8]) offers compression of symbols with an arbitrary probability distribution and is close to optimal. The main drawback of AC is its low efficiency as it requires complex arithmetics and heavy computational overhead. In contrast, asymmetric numeral system (ANS) invented by Jarek Duda [2] gives a close to optimal and efficient compression. The efficiency gain is achieved by representing coding/decoding operations by their tables that define corresponding finite-state machine transition functions. Note that this variant is called the tabled ANS or simply tANS. Note that expensive arithmetics for AC is replaced by lookup operation for ANS. Since 2015, tANS has been applied in Facebook Zstandard, Linux kernel and Android operating system to name a few.

Motivation: One of emerging applications of ANS is compression of data gathered and transmitted by Internet of things (IoT) devices. It is predicted that by 2025, the IoT infrastructure is going to include more than 75 billion devices. This allows users to access their data from everywhere any time. Note that because of limited computing and storage resources, IoT devices use ANS compression that does not protect transmitted data against unauthorised reading and/or modification. This work addresses the security issues when ANS compression is applied by IoT devices with limited resources. The idea is to identify natural properties of ANS, which together with lightweight cryptographic tools, can provide a "decent" level of security for confidentiality and integrity. The topic has been investigated in [3] by Duda and Niemiec. The authors consider a plain ANS with a (pseudo)randomly chosen encoding function. In other words, they assume that the secret is the encoding function and compression is performed by a plain ANS. In contrast, we cryptographically change behaviour of an underlying ANS during symbol processing.

Contribution: This work investigates joint compression and encryption for low security IoT communications. In particular, the work

- analyses confidentiality and integrity of data provided by a plain ANS (without any cryptography). The analysis is done for ciphertext-only and known-plaintext attacks. It also discusses integrity of output streams,
- defines our requirements for lightweight compcrypt algorithms, i.e. (1) minimal use of cryptography, (2) security against ciphertext-only adversaries and (3) integrity checking mechanism,
- provides three compcrypt solutions. First one is based on state jumps. The second one applies two plain ANS algorithms with transition between the two controlled by PRGB bits. And the third one uses PRGB bits to modify ANS encoding function,
- evaluates security and efficiency of the proposed compcrypt algorithms.

The rest of the work is structured as follows. Section 2 introduces the plain ANS. We first give a bird-eye view of ANS followed by a formal description of its algorithms. The section is complemented by an example of a toy ANS. Section 3 analyses confidentiality and integrity of the plain ANS under ciphertext-only and known-plaintext attacks. Section 4 describes our three lightweight compcrypt algorithms. Section 5 evaluates security and efficiency of the proposed algorithms. Section 6 concludes the work.

2 Description of Asymmetric Numeral System (ANS)

Let m be the size of an alphabet $\mathbb{S}$, n – the number of symbols in a sequence, and N – the number of bits in a sequence. Given a source that generates a sequence $\mathcal{S} = \{s_j\}_{j=1}^{n}$ of symbols with their probabilities $\Pr(s_i) = p_i \approx |\{j : s_j = i\}|/n$, where $|\mathbb{A}|$ is cardinality of the set $\mathbb{A}$. In entropy coding, we would like to uniquely translate $\mathcal{S}$ into bit sequence $\mathcal{B} = \{b_j\}_{j=1}^{N}$. Shannon defines entropy of the source as $H(p) = \sum_{i=1}^{m} p_i \lg_2(\frac{1}{p_i})$. Roughly saying, entropy gives the average number of bits per symbol for a given probability distribution. This implies that ideally $N/n \approx H(p)$ when $(n \to \infty)$. The Huffman code (HC) is the first attempt to encode a symbol sequence $\mathcal{S}$ into a binary sequence $\mathcal{B}$. Note that HC works well for probability distributions described by integer powers of $\frac{1}{2}$. Otherwise, N/n moves away from $H(p)$. Both AC and ANS address the problem of encoding symbols with an arbitrary probability distribution. They allow to achieve encoding that is as close to Shannon entropy as needed. The reader interested in ANS details is referred to [2, 7]

2.1 Bird-eye View of ANS

ANS [2] allows to achieve a close to optimal compression for a source of an arbitrary probability distribution. The ANS encoding and decoding can be done very efficiently. When describing ANS operations, it is helpful to think about ANS as a finite state machine (FSM) optimized for a given probability distribution, whose states are labelled by integers. In this context, ANS is also a Moore machine as a binary encoding and the corresponding symbol are uniquely determined by the machine state.
The main data structure is determined by the number of states $L = 2^R$, where $R \in \mathbb{N}^+$ is a parameter, which determines the number of states. Let L_s denote the number of occurrences of symbol s, where $\sum_s L_s = L$ and L_s is an approximation of $Pr(s) = p_s$. Define the following sets $\mathbb{L} = \{L, \ldots, 2L-1\}$ and $\mathbb{L}_s = \{L_s, \ldots, 2L_s - 1\}$, where $s \in \mathbb{S}$. Assume that the current state is x, then ANS processes a symbol s in two steps:

1. re-normalises the current state x by truncating enough least significant bits (LSB) of x so the truncated integer belongs to $\mathbb{L}_s$,
2. calculates a new state by applying an encoding function $C(\cdot)$ or $\mathbb{L}_s \xrightarrow{C(\cdot)} \mathbb{L}$ and outputs the binary sequence b_s=LSB(x), which is a binary encoding of s.

The crux of ANS is its encoding function $x = C(s, y)$ that assigns a state/integer $x \in \mathbb{L}$ that encodes $s \in \mathbb{S}$ using the integer $y \in \mathbb{L}_s$. A *symbol spread function* $\bar{s} : \mathbb{L} \to \mathbb{S}$ is closely connected to the encoding function $C(s, y)$. It determines the symbol s that is encoded in x or $\bar{s}(x) = s$. The encoding function $C : \mathbb{L}_s \to \mathbb{L}$ is constructed so the following conditions hold:

- The approximation $\frac{L_s}{2^R} \approx p_s$ determines quality of compression. This means that there are L_s different integers $x \in \mathbb{L}$ that encode s. Also the probability distribution of integers $x \in \mathbb{L}$ is as flat as possible.
- By construction for a given symbol s, $C(s, y)$ accepts integers $y \in \mathbb{L}_s$. The function $C(s, y)$ can be represented by a table, whose rows are indexed by a symbol s and columns by an integer $y \in \mathbb{L}_s$ – see below.

$C(\cdot,\cdot)$	$\cdots$	$\overbrace{L_s \quad \cdots \quad 2L_s - 1}^{\mathbb{L}_s}$	$\cdots$
$\vdots$	$\vdots$		
s		$\underbrace{\text{Symbol Spread for } s}_{\mathit{\Gamma}_s}$	

- The integers $x \in \mathit{\Gamma}_s$ (called also the symbol spread for s) can be chosen at random from $\mathbb{L}$ as long as any symbol spread pair does not have any integers in common, i.e. $\mathit{\Gamma}_s \cap \mathit{\Gamma}_{s'} = \emptyset$ as long as $s \neq s'$, where $\mathit{\Gamma}_s = \{x \in \mathbb{L} | x = C(s, y); y \in \mathbb{L}_s\}$.

A decoding function $D : \mathbb{L} \to \mathbb{S} \times \mathbb{L}_s$ takes an integer $x \in \mathbb{L}$ and returns the corresponding symbol s and an integer y, which is a re-normalised state from $\mathbb{L}_s$. In fact, $D(x)$ can be seen as the inverse of $C(s, y)$. By construction, the integer x points out a unique pair (s, y). Note that if $x \in \mathit{\Gamma}_s$, then it cannot occur in any other $\mathit{\Gamma}_{s'}$, otherwise the condition $\mathit{\Gamma}_s \cap \mathit{\Gamma}_{s'} = \emptyset$ is violated.

Encoding – given a state $x \in \mathit{\Gamma}_s$ that is an encoding of s, the number of bits of b_s is computed as

$$k = k_s(x) = \lfloor \log_2(x/L_s) \rfloor \longrightarrow b_s = x \mod 2^k$$

The binary string b_s is sent to the output. Now for the next symbol $s' \in \mathbb{S}$, the state x is updated as follows

$$x \longrightarrow x' = C(s', \lfloor x/2^k \rfloor)$$

Note that $x \in \mathit{\Gamma}_s$ but $\lfloor x/2^k \rfloor \in \mathbb{L}_s$.
Decoding – for a state $x \in \mathbb{L}$ and an output binary string $\mathcal{B}$, the decoding function $D(x) = (s, y)$ determines the symbol s and integer $y \in \mathbb{L}_s$. Next the number of bits that needs to be read from $\mathcal{B}$ is calculated as

$$k = k(x) = R - \lfloor \log_2 x \rfloor$$

The k-bit string is read from $\mathcal{B}$ or $b_s = MSB(\mathcal{B})_k$, where MSB stands for the most significant bits. The string $\mathcal{B}$ is updated by removing b_s and the state is modified

$$x' = 2^k \cdot y + b_s$$

The full description of ANS is given below.

2.2 ANS Algorithms

The ANS compression can be seen as a triplet $\langle \mathbf{I}, \mathbf{C}, \mathbf{D} \rangle$, where $\mathbf{I}$ is an initialization algorithm executed once before compression by communicating parties. $\mathbf{C}$ is a compression algorithm performed by a sender and $\mathbf{D}$ is a decompression algorithm used by a receiver.

Initialisation I

Input: A set $\mathbb{S}$ of symbols, their probability distribution $p : \mathbb{S} \to [0, 1]$, $\sum_s p_s = 1$ and a parameter $R \in \mathbb{N}^+$.
Output: Instantiation of

- the encoding functions $C(s, x)$ and $k_s(x)$ and
- the decoding functions $D(x)$ and $k(x)$.

Steps: Initialisation proceeds as follows:

- calculate the number of states $L = 2^R$;
- determine the set of states $\mathbb{L} = \{L, \ldots, 2L - 1\}$;
- for each symbol $s \in \mathbb{S}$, compute integer $L_s \approx Lp_s$, where p_s is probability of s;
- define the symbol spread function $\overline{s} : \mathbb{L} \to \mathbb{S}$, such that $|\{x \in \mathbb{L} : \overline{s}(x) = s\}| = L_s$;
- establish the coding function $C(s, y) = x$ for the integer $y \in \mathbb{L}_s = \{L_s, \ldots, 2L_s - 1\}$, which assigns states $x \in \mathbb{L}$ according to the symbol spread function;
- compute the function $k_s(x) = \lfloor \lg(x/L_s) \rfloor$ for $x \in \mathbb{L}$ and $s \in \mathbb{S}$. The function shows the number of output bits generated during a single encoding step;
- construct the decoding function $D(x) = (s, y)$, which for a state $x \in \mathbb{L}$ assigns its unique symbol (given by the symbol spread function) and the integer $y \in \mathbb{L}_s$. Note that $D(x) = C^{-1}(x)$.
- calculate the function $k(x) = R - \lfloor \lg(x) \rfloor$, which determines the number of bits that need to be read out from the bitstream in a single decoding step.

The algorithm $\mathbf{C}$ takes a sequence of symbols further called a *symbol frame* and generates a stream of bits also called *binary frame.*

Frame Encoding C

Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_n)$ and an initial state $x = x_n \in \mathbb{L}$; where $n = |\mathcal{S}|$.
Output: A binary frame $\mathcal{B} = (b_1, b_2, \ldots, b_n)$, where b_i is a binary encoding of s_i; $|b_i| = k_{s_i}(x_i)$ and x_i is the state.

```
Steps: For $i = n, n-1, \ldots, 2, 1$ do                (encoding has to be in opposite direction)
    {
    $s := s_i$;
    $k = k_s(x) = \lfloor \lg(x/L_s) \rfloor$;           (compute the number of bits to be extracted)
    $b_i = x \mod 2^k$;                                 (send k LSB of current state $x = x_i$ to the output)
    $x := C(s, \lfloor x/2^k \rfloor)$;                  (update the state $x_i \to x_{i-1}$)
    };
    Store the final state $x_0 = x$;
```

The next algorithm takes a binary frame and the final state and produces symbols of the corresponding frame.

Stream Decoding D

```
Input: A binary frame $\mathcal{B}$ and the final state $x = x_0 \in \mathbb{L}$ of the encoder.
Output: A symbol frame $\mathcal{S}$.
Steps: while $|\mathcal{B}| \neq 0$
    {
    $(s, y) = D(x)$;                          (produce the corresponding symbol $s$ and integer $y$)
    $k = k(x) = R - \lfloor \lg(x) \rfloor$;   (compute the number of bits to be read from $\mathcal{B}$)
    $b_s = MSB(\mathcal{B})_k$;               (extract k MSB from $\mathcal{B}$)
    $\mathcal{B} := LSB(\mathcal{B})_{|\mathcal{B}|-k}$;   (update the stream of bits to be processed)
    $x := 2^k y + b_s$;                       (update the state $x_{i-1} \to x_i$)
    }
```

Note that $LSB(\mathcal{B})_n$ and $MSB(\mathcal{B})_n$ stand for the n least and most significant bits of $\mathcal{B}$, respectively.

2.3 Example

Design compression and decompression algorithms for a symbol source $\mathbb{S} = \{s_0, s_1, s_2\}$, where $p_0 = \frac{3}{16}$, $p_1 = \frac{8}{16}$, $p_2 = \frac{5}{16}$ and a free parameter $R = 4$. The number of states $L = 2^R = 16$ and the state set $\mathbb{L} = \{16, 17, \ldots, 31\}$. We follow the initialisation.

- Determine symbol spread function $\overline{s} : \mathbb{L} \to \mathbb{S}$ such that

$$\overline{s}(x) = \begin{cases} s_0 & \text{if } x \in \{18, 22, 25\} = \Gamma_0 \\ s_1 & \text{if } x \in \{16, 17, 21, 24, 27, 29, 30, 31\} = \Gamma_1 \\ s_2 & \text{if } x \in \{19, 20, 23, 26, 28\} = \Gamma_2 \end{cases}$$

  where $L_0 = |\{18, 22, 25\}| = 3$, $L_1 = |\{16, 17, 21, 24, 27, 29, 30, 31\}| = 8$ and $L_2 = |\{19, 20, 23, 26, 28\}| = 5$.
- Write the encoding function $C(s, y)$ as the following table

$s \backslash y$	3	4	5	6	7	8	9	10	11	12	13	14	15
s_0	18	22	25	–	–	–	–	–	–	–	–	–	–
s_1	–	–	–	–	–	16	17	21	24	27	29	30	31
s_2	–	–	19	20	23	26	28	–	–	–	–	–	–

  The top row of the table defines $\mathbb{L}_0 = \{3, 4, 5\}$, $\mathbb{L}_1 = \{8, 9, 10, 11, 12, 13, 14, 15\}$ and $\mathbb{L}_2 = \{5, 6, 7, 8, 9\}$.
- Construct the encoding table $\mathbb{E}(x_i, s_i) = (x_{i+1}, b_i) \overset{def}{\equiv} \binom{x_{i+1}}{b_i}$ as follows:

$s_i \backslash x_i$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
s_0	$\binom{22}{00}$	$\binom{22}{01}$	$\binom{22}{10}$	$\binom{22}{11}$	$\binom{25}{00}$	$\binom{25}{01}$	$\binom{25}{10}$	$\binom{25}{11}$	$\binom{18}{000}$	$\binom{18}{001}$	$\binom{18}{010}$	$\binom{18}{011}$	$\binom{18}{100}$	$\binom{18}{101}$	$\binom{18}{110}$	$\binom{18}{111}$
s_1	$\binom{16}{0}$	$\binom{16}{1}$	$\binom{17}{0}$	$\binom{17}{1}$	$\binom{21}{0}$	$\binom{21}{1}$	$\binom{24}{0}$	$\binom{24}{1}$	$\binom{27}{0}$	$\binom{27}{1}$	$\binom{29}{0}$	$\binom{29}{1}$	$\binom{30}{0}$	$\binom{30}{1}$	$\binom{31}{0}$	$\binom{31}{1}$
s_2	$\binom{26}{0}$	$\binom{26}{1}$	$\binom{28}{0}$	$\binom{28}{1}$	$\binom{19}{00}$	$\binom{19}{01}$	$\binom{19}{10}$	$\binom{19}{11}$	$\binom{20}{00}$	$\binom{20}{01}$	$\binom{20}{10}$	$\binom{20}{11}$	$\binom{23}{00}$	$\binom{23}{01}$	$\binom{23}{10}$	$\binom{23}{11}$

To illustrate calculations in the table, assume that we have $x_i = 25$ and input symbol is s_0. First we determine the number of bits that need to be extracted $k = \lfloor \lg(x_i/L_0) \rfloor = \lfloor \lg(25/3) \rfloor = 3$ and compute

$$x_{i+1} = C(s_0, \lfloor \frac{x_i}{2^k} \rfloor) = C(s_0, 3) = 18$$
$$b_i = x_i \mod 2^k = 25 \mod 8 = 1 \longrightarrow 001$$

Given an initial state $x_0 = 19$, compress the following symbol frame

$$\mathcal{S} = (s_1, s_1, s_2, s_1, s_2, s_1, s_1, s_0, s_2)$$

Applying the encoding table for consecutive symbols, we get

$$\begin{array}{ccccccccccccccccccc} & \binom{19}{s_1} & & \binom{17}{s_1} & & \binom{16}{s_2} & & \binom{26}{s_1} & & \binom{29}{s_2} & & \binom{23}{s_1} & & \binom{24}{s_1} & & \binom{27}{s_0} & & \binom{18}{s_2} & \\ (19) \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow (28) \\ & 1 & & 1 & & 0 & & 0 & & 01 & & 1 & & 0 & & 011 & & 0 & \end{array}$$

The output bits are $\mathcal{B} = 110001100110$ and the final state is 28.

- Build the decoding table. The decoding function $D(x) = (s, y)$, where the integer y is given by the following table

x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
y	8	9	3	5	6	10	4	7	11	5	8	12	9	13	14	15

The decoding table $\mathbb{D}(x_i, b_i)$ can be represented as follows

x_i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
s_i	s_1	s_1	s_0	s_2	s_2	s_1	s_0	s_2	s_1	s_0	s_2	s_1	s_2	s_1	s_1	s_1
k	1	1	3	2	2	1	2	2	1	2	1	1	1	1	1	1
x_{i+1}	$16{+}b_i$	$18{+}b_i$	$24{+}b_i$	$20{+}b_i$	$24{+}b_i$	$20{+}b_i$	$16{+}b_i$	$28{+}b_i$	$22{+}b_i$	$20{+}b_i$	$16{+}b_i$	$24{+}b_i$	$18{+}b_i$	$26{+}b_i$	$28{+}b_i$	$30{+}b_i$

Note that $x_{i+1} = 2^k y + b_i$, where b_i is an integer that corresponds to binary encoding of s_i. Given the binary frame $\mathcal{B} = 110001100110$ and the final state 28, we recover the corresponding sequence of symbols, where the binary string needs to be read from right to left.

$$\begin{array}{ccccccccccccccccccc} & \binom{28}{0} & & \binom{18}{011} & & \binom{27}{0} & & \binom{24}{1} & & \binom{23}{01} & & \binom{29}{0} & & \binom{26}{0} & & \binom{16}{1} & & \binom{17}{1} & \\ (28) \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow (19) \\ & s_2 & & s_0 & & s_1 & & s_1 & & s_2 & & s_1 & & s_2 & & s_1 & & s_1 & \end{array}$$

Let us check how optimal is our compression algorithm. The source entropy is

$$H(p) = \frac{3}{16} \log_2 16/3 + \frac{1}{2} \log_2 2 + \frac{5}{16} \log_2 16/5 \approx 1.48 \text{ bits}$$

Assuming that probability distribution of $x \in \mathbb{L}$ is close to uniform, the average length ℓ of binary (compressed) string is

$$\ell = p_0 \cdot \ell_{s_0} + p_1 \cdot \ell_{s_1} + p_2 \cdot \ell_{s_2} = p_0 \cdot 2.5 + p_1 + p_2 \cdot \frac{28}{16} \approx 1.51 \text{ bits},$$

where ℓ_{s_i} is the average length of the output if algorithm compresses the sequence of the same symbol s_i.

3 Analysis of Plain ANS

A symbol frame $\mathcal{S}$ is compressed at the sender side. A binary frame $\mathcal{B}$ together with the final state x_F is sent to a receiver who decompresses the stream back to the symbol frame $\mathcal{S}$. Both the sender and receiver know a symbol source statistics and parameters of the ANS including an encoding function $C(s, y)$. It is important to distinguish between different views of binary frames, namely

- a view of a receiver who knows $C(s, y)$. It sees sequence of encodings for consecutive symbols, i.e. $\mathcal{B} = (b_1, \ldots, b_n)$, where b_i is an encoding of s_i. In other words, it knows how to divide a binary frame into encodings b_i. As each b_i may have a different length, we can define a *window frame* $\mathcal{W} = (k_1, \ldots, k_n)$, where k_i indicates the number of bits in b_i or $k_i = |b_i|$. In other words, the receiver knows both frames $\mathcal{B}$ and $\mathcal{W}$,
- a view of an adversary $\mathcal{A}$ who does not know $C(s, x)$. $\mathcal{A}$ deals with a binary frame $\mathcal{B}$ and it does not know how to extract particular encodings b_{s_i}. In other words, $\mathcal{A}$ knows $\mathcal{B}$ but does not know the window frame $\mathcal{W}$. Note that, in general, a window frame does not determine symbols as the same symbol s can be encoded into b_s of different lengths.

Note that we ignore active adversaries who may access to an ANS encoder/decoder (or to oracles $\mathcal{O}_{\mathbb{E}}/\mathcal{O}_{\mathbb{D}}$). In this case, an adversary is able to extract states and encodings by inputting short symbol frames to $\mathcal{O}_{\mathbb{E}}$ and reconstruct the encoding function $C(s, x)$. In our analysis, we assume that an adversary is passive and can observe behaviour of ANS. The table below shows attack scenarios investigated in this section.

Attack	$\mathcal{A}$'s *Knowledge/Ability*	$\mathcal{A}$'s *Goals*
ciphertext-only	– symbol statistics $\{p_s : s \in \mathbb{S}\}$ – parameters of ANS – n number of symbols in frame – $\mathcal{A}$ observes binary frame $\mathcal{B}$ and x_F	– guessing a window frame $\mathcal{W}$ – finding full/partial symbol frame
known-plaintext	– As above + – observes symbol frame $\mathcal{S} = (s_i)_1^n$; $s_i \in \mathbb{S}$	– guessing a window frame $\mathcal{W}$ – finding encoding function $C(s, y)$
Integrity	– As for ciphertext-only attack – $\mathcal{A}$ can inject bits to binary frames	– acceptance of forged binary frames as genuine

The above scenarios are most common in IoT applications. The ciphertext-only attack is relevant to any adversary who is able to see the traffic generated by an IoT device. The known-plaintext attack can be launched if an adversary has additionally access to source of symbols. For instance, it is easy to determine symbols for a temperature sensor by installing an adversarial sensor nearby that hopefully replicates the temperature readings.

3.1 Ciphertext-only Attack against ANS

A majority of IoT devices that use ANS for compression communicates with their servers via broadcasting channels (such as Bluetooth or WiFi). This makes them vulnerable to eavesdropping (alternatively called ciphertext-only attacks). The main difficulty for an adversary is to guess a window frame. After she has guessed it, she can upload an observed binary frame $\mathcal{B}$ into consecutive windows and recover a sequence of encodings. Our task here is to determine an upper bound for probability of guessing the window frame and evaluate a lower bound of security provided by a plain ANS.

Symbol versus Widow Statistics A typical source includes all 2^8 ASCII symbols. Its statistical properties are approximated by geometric probability distribution truncated to 2^8 events. To recall, geometric probability distribution is defined as $P(j) = (1-p)^j p$, where $0 < p < 1$ is a parameter and $j = 1, 2, \cdots$ are the events. In practice, instead of infinite number of events, j has to be equal to the number of all symbols produced by the source. It is important to note that, in general, a symbol can be assigned to binary encodings of different lengths. In our example, s_2 can be compressed into either 1-bit or 2-bit encoding. Consequently, the statistics of windows of different lengths is different from the source statistics. This is illustrated below.

$$\begin{array}{ccc} \textit{Source Statistics} & & \textit{Window Statistics} \\ P(s_1) = p_1 & & P(W=0) = P_0 \\ \vdots & \rightarrow ANS \rightarrow & \vdots \\ P(s_m) = p_m & & P(W=\alpha) = P_\alpha \end{array}$$

$P(W = i) = P_i$ gives probability that ANS produces a window of the length i, where $i = 1, \cdots, \alpha$ and α is the longest window used by ANS. In our example, ANS translates the source probabilities $(\frac{3}{16}, \frac{1}{2}, \frac{5}{16})$ into window probabilities $(P_1, P_2, P_3) = (\frac{37}{64}, \frac{21}{64}, \frac{6}{64})$.

Remark 1. Given ANS window probabilities P_i; $i = 1, \cdots, \alpha$ and the number n of symbols processed by ANS, then an adversary can guess the window frame for the symbols with probability no better than

$$P_{\text{GWF},n} \leq 2^{-n \cdot H_W},$$

where $H_W = \sum_{i=1}^{\alpha} P_i \log_2 P_i^{-1}$ is an entropy of W.

Guessing Window Frames Assume that n symbols are generated according to the source statistics and processed by ANS. The resulting window frame is a random variable, which is a described by concatenation of n window random variables. In practice, ANS accepts 2^8 possible symbols described by an appropriate probability distribution and processes it into a window probability distribution $\{P(W = i) = P_i | i = 1, 2, \ldots, \alpha\}$. To simplify our considerations, we assume that the window variable W is defined for three events $\{1, 2, 3\}$ only, where $P(W = 1) = P_1$, $P(W = 2) = P_2$ and $\hat{P}_3 = P(W = 3) = \sum_{i=3}^{\alpha} P_i$. To enumerate possible window frames and find the probability of guessing the right one, we use the binomial theorem [6] that says that

$$(x+y)^r = \sum_k \binom{r}{k} x^k y^{r-k}$$

Let us illustrate the connection between the theorem and our problem. Assume that we are dealing with window frames that are built from n window variables. We have 3^n possible window frames (events) containing 1-bit, 2-bit and 3-bit windows. The binomial theorem asserts us that

$$3^n = (1+2)^n = \sum_{k=0}^{n} \binom{n}{k} 2^k = \binom{n}{0} \cdot 2^0 + \binom{n}{1} \cdot 2^1 + \ldots + \binom{n}{n-1} \cdot 2^{n-1} + \binom{n}{n} \cdot 2^n$$

Note that the term $\binom{n}{k} 2^k$ gives the number of window frame events that consists of k either 2-bit or 3-bit windows and $(n-k)$ 1-bit windows. The probability that a randomly chosen window frame contains $(n-k)$ 1-bit windows and k either 2 or 3-bit windows is

$$\binom{n}{k} 2^k \cdot P_1^{n-k} \cdot \left(\frac{P_2 + \hat{P}_3}{2}\right)^k$$

The total length of window frames ranges from $n + k$ to $n + 2k$, where $k = 0, \dots, n$.

Remark 2. It is reasonable to assume that an adversary knows the length n of symbol frame and the length N of binary frame. This helps the adversary as the space of events is restricted to N-bit window frames. She knows that

$$\begin{aligned} n_1 + n_2 + \cdots + n_\alpha &= n \\ n_1 + 2 \cdot n_2 + \cdots + \alpha \cdot n_\alpha &= N, \end{aligned} \tag{1}$$

where $n_i \in \mathbb{N}$ is the number of i-bit windows; $i = 1, \dots, \alpha$. It is easy to determine a space of solutions $(n_1, \dots, n_\alpha)$ for which Equation (1) holds. For a given $(n_1, \dots, n_\alpha)$, the adversary needs to look through

$$\binom{n}{n_1, n_2, \dots, n_\alpha} = \frac{n!}{n_1! n_2! \cdots n_\alpha!} \tag{2}$$

equally probable events (window frames). Equation (2) represents a multinomial coefficient [6].

Let us make the following observations about ANS resistance against ciphertext-only attack.

- The adversary has a "good" chance to guess relatively short window frames. She may attempt to determine such frames at any position of binary stream as symbols are independently generated. As the number n of symbols grows, the probability of success quickly becomes negligible. Note that this observation is consistent with the conclusion made by Gillman et al. [4] about cryptanalysis of compression with Huffman codes that is "surprisingly difficult".
- If the length N of binary frame is known and very close to either n or $\alpha \cdot n$, then it is possible to guess the window frame with a non-negligible probability. Note however that probability of such events is negligible for a large enough n.
- Guessing a window frame allows $\mathcal{A}$ to correctly allocate binary encodings. Moreover, she is able to determine the most frequent symbols (as they are assigned to 1-bit windows with a high probability) and the least frequent symbols (as they are assigned to α-bit windows with probability 1).

3.2 Known-Plaintext Attack against ANS

For a given symbol, ANS assigns binary encodings/windows of different lengths. The following observation can be used to determine the window lengths for each symbol.

Fact 1 *Given a symbol $s \in \mathbb{S}$ and its $L_s \approx 2^R \cdot p_s$, then the window length k_s satisfies the following condition*

$$\lfloor \log_2 \frac{2^R}{L_s} \rfloor \le k_s \le \lfloor \log_2 \frac{2^{R+1} - 1}{L_s} \rfloor$$

When the approximation $L_s \approx 2^R \cdot p_s$ can be replaced by equality $L_s = 2^R \cdot p_s$, the above condition can be re-written as

$$\lfloor \log_2 p_s^{-1} \rfloor \le k_s \le \lfloor \log_2 p_s^{-1} (2 - \frac{1}{2^R}) \rfloor$$

□

A closer look at the above conditions reveals the following properties of window lengths:

- If $p_s = (1/2)^i$, then ANS assigns a window of the length i.
- If $p_s > 1/2$, then ANS assigns a window of the length either 0 or 1.
- Otherwise, ANS assigns a window of the length $k_s \in \{i, i+1\}$, where $i = \lfloor \log_2 p_s^{-1} \rfloor$.

From now on, we assume that for each symbol $s \in \mathbb{S}$, ANS assigns an encoding b_s, whose length is either k_s or k_s+1 with the probabilities $P(k_s) = \beta_s$ and $P(k_s+1) = 1-\beta_s$, respectively. In cases, where there is only one length k_s, the probability distribution becomes trivial, i.e. $P(k_s) = 1$ and $P(k_s+1) = 0$ (or vice versa). A probabilistic model of ANS is illustrated below. Note that symbols are listed according to decreasing order of their probablities, i.e. s_1 is the most probable while s_m – the least.

Symbol s	*Length k_s*	*Probability β_s*
s_1	$\{0,1\}$	β_{s_1}
s_2	$\{1,2\}$	β_{s_2}
$\vdots$		
s_{m-1}	$\{m-2, m-1\}$	$\beta_{s_{m-1}}$
s_m	$\{m-1, m\}$	β_{s_m}

Consider ANS from our example. For s_1, it assigns a window of length 1 with probability 1. For s_2, it allocates a window of the length either 1 or 2, where $\beta_{s_2} = 1/4$. For s_0, it points a window of the length either 2 or 3, where $\beta_{s_0} = 1/2$.

Guessing Window Frames This time our adversary knows both a symbol frame $\mathcal{S} = (s_i)_{i=1}^n$ and a binary frame $\mathcal{B}$ of N bits. To determine a corresponding window frame, the adversary

- finds a space of all solutions of the following relation

$$k_1 + k_2 + \ldots + k_n = N, \tag{3}$$

  where k_i is the length of a window used by ANS to encode s_i; $i = 1, \ldots, n$. Note that k_i can take on two values only so we can write that $k_i = c_i + \gamma_i$, where a constant c_i is known to the adversary and $\gamma_i \in \{0,1\}$ is unknown. Equation (3) can be re-written as

$$\sum_{i=1}^{n} \gamma_i = N - \sum_{i=1}^{n} c_i$$

  The integer $\sum_{i=1}^n \gamma_i$ is the number of times when $\gamma_i = 1$ and it is known to the adversary,
- enumerates all possible patterns of $(\gamma_i)_1^n$, whose weight is $N - \sum_{i=1}^n c_i$. It is obvious that the number of patters is

$$\binom{n}{N - \sum_{i=1}^n c_i}$$

  To maximise chances, the adversary tries from most probable patterns. This can be done as she knows probabilities β_s.

In general, guessing of window frames can be difficult or even impractical for some ANS instances. There is, however, a word caution. If some probabilities of symbols are powers of (1/2) or close to it, then ANS assigns to them a window with a single length. This increases chances of guessing a widow frame. In an extreme case, when all probabilities are powers of 1/2, the adversary can determine a window frame with probability 1.

Adaptive Attack against ANS We assume that an adversary knows a symbol frame $\mathcal{S} = (s_i)_{i=1}^n$ together with the corresponding binary frame $\mathcal{B}$ and a guessed (correctly) window frame. In other words, $\mathcal{A}$ knows all encodings $(b_i)_{i=1}^n$ and a final state x_F. Her goal is to find an encoding function $C(s, y)$. However, one can argue that instead of finding $C(x, y)$, the adversary can design (adaptively) her own $\mathrm{ANS}_{\mathcal{A}}$, which is fully/partially "isomorphic" to the analysed ANS. In other words, the adversary intends to find a function that translates output bits of the original (attacked) ANS into output bits of the adversary $\mathrm{ANS}_{\mathcal{A}}$. Note that the adversary does not known the current state of the original ANS. In fact, the adversary does not need to know the original ANS as long as her $\mathrm{ANS}_{\mathcal{A}}$ produces a bitstream that can be translated to bitstream generated by the original ANS. In this sense, both ANS and $\mathrm{ANS}_{\mathcal{A}}$ are isomorphic. In other words, we are looking for a function F such that

$$\begin{tikzcd}
\mathrm{ANS} \arrow[d] & \mathrm{ANS}_{\mathcal{A}} \arrow[d] & \\
b_i \arrow[r, leftrightarrow, "F"] & b'_i & \text{for } i = 1, 2, \cdots
\end{tikzcd}$$

The adaptive attack proceeds along the following steps:

1. The adversary $\mathcal{A}$ designs her $\mathrm{ANS}_{\mathcal{A}}$ applying the same parameters as the original ANS.
2. $\mathcal{A}$ chooses an initial state x'_1 at random. For the first observation (s_1, b_1), she finds b'_1 from the encoding table of $\mathrm{ANS}_{\mathcal{A}}$. She records
$$(s_1, x_1, b_1) \xrightarrow{F} (s_1, x'_1, b'_1)$$
3. $\mathcal{A}$ continues with subsequent observations and builds the function (table) F. This process is successful if the function fully determined for all symbols and states. If the original ANS or $\mathrm{ANS}_{\mathcal{A}}$ contain cycles then the algorithm fails. If a cycle occurs in the original ANS, $\mathcal{A}$ needs to "re-design" $\mathrm{ANS}_{\mathcal{A}}$ by introducing the cycle of an appropriate length. On the other hand, if $\mathrm{ANS}_{\mathcal{A}}$ hits a cycle, it needs re-design to remove the cycle.

3.3 Integrity of ANS Binary Frames

ANS is normally represented by its encoding table $\mathbb{E}(x_i, s_i)$. Equivalently, it can be described by a directed graph with 2^R vertices that correspond to states and edges that are labelled by symbols. An edge s from a vertex x_i to x_{i+1} shows transition determined by the encoding function $x_{i+1} = C(s, \left\lfloor \frac{x_i}{2^{k_s}} \right\rfloor))$. For a fixed symbol $s \in \mathbb{S}$, the function $C(s, \cdot)$ assigns one of $\mathbb{L}_s$ states. This implies that the following sequence of transitions

$$x_i \xrightarrow{s} x_{i+1} = C(s, \left\lfloor \frac{x_i}{2^{k_s}} \right\rfloor)) \xrightarrow{s} x_{i+2} = C(s, \left\lfloor \frac{x_{i+1}}{2^{k_s}} \right\rfloor)) \xrightarrow{s} \cdots \xrightarrow{s} x_{i+j} = C(s, \left\lfloor \frac{x_{i+j-1}}{2^{k_s}} \right\rfloor))$$

has to be periodic for $j \geq L_s$. This also means that the ANS graph has to be cyclic. For each fixed symbol $s \in \mathbb{S}$, there may be a single cycle of up to the length L_s or a collection of shorter ones. The cycle includes different binary encoding of s.

Consider ANS from our Example. Assume that ANS starts from an initial state $x = 19$ and processes a long sequence of s_2. ANS produces the following (periodic) sequence of binary stream:

$$\begin{array}{ccccccccccccc}
 & \binom{28}{s_2} & & \binom{23}{s_2} & & \binom{19}{s_2} & & \binom{28}{s_2} & & \binom{23}{s_2} & & \binom{19}{s_2} & \\
(19) \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow \cdots \\
 & 1 & & \underbrace{00 \quad\quad 11 \quad\quad 1}_{cycle} & & & & & & 00 & & 11 &
\end{array}$$

The periodic nature of ANS has the following security and design implications.

- Cycles in ANS are unavoidable. A designer of ANS can avoid loops (cycles of the length 1) making sure that for each state x_i and any symbol $s \in \mathbb{S}$
$$x_{i+1} = C(s_j, \left\lfloor \frac{x_i}{2^{k_i}} \right\rfloor) \neq x_i.$$
Getting rid of longer cycles requires more and more computation overhead as the designer has to consider different combinations of states and symbols. This also means that the entropy of state selection drops, which means that an adversary does not need to enumerate encoding functions $C(s, y)$ that have short cycles.
- Cycles are easy to identify by searching binary frame for repeating sequences. A detection of a concatenation of two or more bit patterns allows the adversary to remove or insert arbitrary number of times the bit pattern without detection by the receiver. This is true as injection/removal of bit pattern repetition correspond to adding/removing a cycle without disturbing decoding process for other parts of the binary frame (before and after injection/removal).
- If a ciphertext-only adversary can remove/inject binary patterns from/into the binary frame, then a decoder recovers an incorrect symbol frame. A typical integrity check applied in ANS that checks correctness the final state fails.
- For an observed binary cycle in $\mathcal{B}$, a known-plaintext adversary can ensemble a relation for encoding function $C(s, y)$. This reduces entropy of the encoding function.

4 Lightweight Encryption with ANS

The analysis given in Section 3 identifies strengths and weaknesses of ANS and is a major driver for our design of a cryptographically strengthened ANS-based compcrypt. Note that it is easy to design a very secure compcrypt algorithm when one can use a full range of cryptographic tools. A price to pay for increase of security is a heavy resource overhead, which discourages potential users from using them. This is true if ANS is applied for a relatively low-security communication (such as collecting data from IoT devices). Our constructions are guided by the following design principles:

- Minimal application of cryptographic tools so compcrypt preserves its efficiency and compression quality. In other words, our designs must be lightweight avoiding "heavy" cryptography and encouraging potential user to adopt the designs for protection of data collected by IoT devices.
- Secure against a ciphertext-only adversary who additionally can modify binary frames by injecting/removing bit cycles. In other words, detection of a cycle in a binary frame is a "false positive" with overwhelming probability.
- Repair of the existing ANS authentication/integrity checking mechanism so any bit stream modification is detected with probability $\approx (1 - 2^{-R})$. Note that the plain ANS allows to check equality of a (pre-agreed) encoding initial states on both communicating sides. As discussed in Section 3, this may involve a careful selection of encoding function $C(s, y)$ with no short cycles.

Interestingly enough, our analysis indicates that there is no need for encryption of bit stream under the assumption of ciphertext-only adversary. The main security feature already provided by plain ANS is a variable length of binary encodings, which are glued together when sending to the decoder. So any attempt to recover symbol frame amounts to guessing a correct window frame. As shown in Section 3, probability of a successful guess is negligible even for short sequences of symbols and decreases exponentially with the number of compressed symbols.

4.1 Compcrypt with State Jumps

The main cryptographic tool used here is a pseudorandom bit generator (PRGB), whose seed K is a secret cryptographic key that is shared between encoder and decoder. PRGB is used to produce sequence of integers *state_ cor*, where $0 \leq \mathit{state\_ cor} \leq 2^R$. The integer *state_ cor* determines a jump from the current state $x \in \mathbb{L}$ to a new one

$$x := (x + \mathit{state\_ cor}) \bmod 2^R + 2^R. \qquad (4)$$

The integer $\mathit{state\_ cor} := PRGB(i, K)$ is a state correction at the ith iteration. To make implementation easier, we assume that the distance between two consecutive jumps denoted by an integer *length* is fixed for the duration of frame encoding. The integer should be kept secret and known to the communicating parties. Below there is a pseudocode for frame coding. A pseudocode for frame decoding can be easily reconstructed.

Algorithm 1: Frame Coding **C** for State Jumps

```
Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_n)$, an initial state $x = x_n \in \mathbb{L}$ and a secret key $K$ for $PRGB$.
Output: A binary frame $\mathcal{B} = (b_1, b_2, \ldots, b_n)$, where $|b_i| = k_{s_i}(x_i)$ and $x_i$ is state at $i$-th step.
begin
    $offset := n \mod length$;                              (jump if offset is zero)
    $no\_jumps := \lfloor n/length \rfloor$;                (number of jumps during the frame processing)
    state_cor := $PRGB(no\_jumps, K)$;                      (state correction for the last jump
    for $i = n, n-1, \ldots, 2, 1$ do
        $s := s_i$;                                         (new symbol to be compressed
        if $offset \neq 0$ then
            $offset - -$;                                   (decrease the variable by 1)
        else
            $x := (x + state\_cor) \mod 2^R + 2^R$;         (state jump)
            $offset := length$;                             (reset offset)
            $no\_jumps - -$;                                (decrease the variable by 1)
            $state\_ cor := PRGB(no\_jumps, K)$;            (next state correction)
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;         (compute the number of bits to be extracted)
        $b_i := x \mod 2^k$;                                (send k LSB of current state $x = x_i$ to the output)
        $x := C(s, \lfloor x/2^k \rfloor)$;                 (update the state $x_i \to x_{i-1}$)
    Store the final state $x_0 = x$;
```

A simple illustration of compcrypt with state jumps is given below. Note that addition for state jump is given by Equation (4).

$$\cdots \xrightarrow{s_{i-1}} \boxed{x_{i-1}} \xrightarrow{s_i} \boxed{x_i \xleftarrow{jump} x_i + state\_cor} \xrightarrow{s_{i+1}} \boxed{x_{i+1}} \xrightarrow{s_{i+2}} \cdots$$

Implementation of the algorithm seems to introduce a relatively light overhead. Few points are relevant here.

- State jumps tend to have a negative impact on quality of compression. This implies that jumps should not occur too often. Consequently, very short cycles of output bits may be observable. To avoid such cycles, ANS should be carefully designed to exclude short cycles.

- Consider a state jump. Note that a binary encoding b_i has to be computed for the state after jump, i.e. $x_i + \mathit{state\_cor}$. Otherwise, decoding fails.
- The only cryptographic component used is *PRGB*. It could be as simple as a linear feedback shift register (LFSR), whose seed (or initial state) is K. It could be also cryptographically strong *PRGB* based on nonlinear feedback shift register (NFSR) or block cipher or hashing.
- Generation of integers $\mathit{PRGB}(i, K)$ for state correction should be easy in both directions: backward (for encoding where i decreases) and forward (for decoding where i increases).

4.2 Compcrypt with Double ANS

The idea is to design two copies of ANS$_i$ with their encoding functions $C_i(s, y)$, where $i = 1, 2$. So we have two symbol encoding tables $\mathbb{E}_i(x, s)$. Consider entries (s, x_i) from $\mathbb{E}_1(x, s)$ and $\mathbb{E}_2(x, s)$. They can be merged as shown below:

$$\boxed{\begin{array}{c} x_{i+1} = C_1(s, \lfloor \frac{x_i}{2^{k_s}} \rfloor) \\ b_i = x_i \mod 2^{k_s} \end{array}} + \boxed{\begin{array}{c} x_{i+1} = C_2(s, \lfloor \frac{x_i}{2^{k_s}} \rfloor) \\ b_i = x_i \mod 2^{k_s} \end{array}} \xrightarrow{merge} \boxed{\begin{array}{c} x_{i+1} \overset{\$}{\leftarrow} \{C_1(s, \lfloor \frac{x_i}{2^{k_s}} \rfloor), C_2(s, \lfloor \frac{x_i}{2^{k_s}} \rfloor)\} \\ b_i = x_i \mod 2^{k_s} \end{array}}$$

Note that k_s and b_i for both ANS copies (and the merged ANS$_D$) are the same. Compcrypt selects the next state (pseudo) randomly from two possibilities. As before, we use a pseudorandom bit generator controlled by a seed K that is a secret key shared by both encoder and decoder. A pseudocode for compcrypt with double ANS is given below.

Algorithm 2: Frame Coding **C** for Double ANS Compcrypt

```
Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_n)$, an initial state $x = x_n \in \mathbb{L}$ and a secret key $K$ for $PRGB$.
Output: A binary frame $\mathcal{B} = (b_1, b_2, \ldots, b_n)$, where $|b_i| = k_{s_i}(x_i)$ and $x_i$ is state at $i$-th step.
begin
    for $i = n, n-1, \ldots, 2, 1$ do
        $s := s_i$;                                  (new symbol to be compressed
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;   (compute the number of bits to be extracted)
        $b_i := x \mod 2^k$;                         (send k LSB of current state $x = x_i$ to the output)
        if $PRGB(i, K) = 0$ then
            $x := C_1(s, \lfloor x/2^k \rfloor)$;       (update the state $x_i \to x_{i-1}$ using $C_1$)
        else
            $x := C_2(s, \lfloor x/2^k \rfloor)$;       (update the state $x_i \to x_{i-1}$ using $C_2$)
    Store the final state $x_0 = x$;
```

We assume that *PRGB* generates a single bit for each call $\mathit{PRGB}(i, K)$. The pseudocode is written for the case when compcrypt chooses next state from two possibilities for each symbol. Clearly, we can allow compcrypt to run a single encoding function (single ANS) for longer sequence of symbols before the next pseudorandom toss. Let us make the following observations.

- Intuitively, switching encoding functions should not have an impact on compression quality.
- Compared to a single ANS$_i$, compcrypt requires larger memory (twice as much) to store two encoding functions $C_1(s, y)$ and $C_2(s, y)$. The same size of memory is enough to store encoding function $C(s, y)$ for ANS with a double number of states, which allows better approximation of symbol statistics and consequently better compression.
- It is possible to reduce storage requirements by selection of encoding functions that are the same for least probable symbols, i.e. $C_1(s, y) = C_2(s, y)$ for $s \in \mathbb{S}$, where s is one of least probable symbols.
- An adversary who detects a cycle in the bit stream is unlikely to succeed in injecting it into the stream without detection.

4.3 Compcrypt with Encoding Function Evolution

Compcrypt based on two ANS can be seen a graph built from two subgraphs. Each subgraph represents a plain ANS. Compression is done by using both subgraphs, where transition between them is controlled by *PRGB*. As already noted that may be perceived as a waste of resources. An option could be to modify a encoding function $C(s, y)$ after a few steps of compression. To make the presentation simpler, we assume that we modify the function after processing a single symbol. In practice, the function modification can be done less frequently. The idea is depicted below.

$$\boxed{x_{i-1} \xrightarrow[C(s,y)]{s_{i-1}} x_i} \quad \boxed{x'_i = x_i + PRGB(i,K) \xrightarrow[C'(s,y)]{s_i} x_{i+1}}$$

The symbol s_{i-1} is processed using $C(s, y)$, where $D(x)$ is its inverse. Next compcrypt generates a pseudorandom integer $PRGB(i, K)$ that modifies x_i according to the following equation

$$x'_i = x_i + PRGB(i, K) \mod 2^R + 2^R,$$

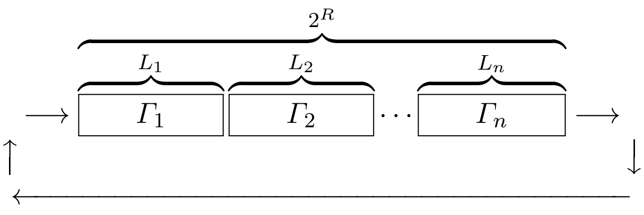
where $0 \leq PRGB(i, K) \leq 2^R$. The states x_i, x'_i are swapped and the resulting encoding function is denoted by $C'(s, y)$. Its inverse $D'(x_i)$ satisfies two relations, namely $D'(x'_i) = D(x_i)$ and $D'(x_i) = D(x'_i)$. Otherwise, $D(x) = D'(x)$ for $x \notin \{x_i, x'_i\}$. A sketch of pseudocode for compression is given below.

Algorithm 3: Frame Coding **C** for Compcrypt with Encoding Function Evolution

```
Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_n)$, an initial state $x = x_n \in \mathbb{L}$ and a secret key $K$ for $PRGB$.
Output: A binary frame $\mathcal{B} = (b_1, b_2, \ldots, b_n)$, where $|b_i| = k_{s_i}(x_i)$ and $x_i$ is state at $i$-th step.
begin
    for $i = n, n-1, \ldots, 2, 1$ do
        $s := s_i$;                                        (new symbol to be compressed
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;        (compute the number of bits to be extracted)
        $b_i := x \mod 2^k$;                               (send k LSB of current state $x = x_i$ to the output)
        $x_{new} := x + PRGB(i, K)$;                       (new state generated pseudorandomly)
        $C(s, y)$ with $x$ and $x_{new}$ swapped;          (update encoding function)
        $x := C(s, \lfloor x_{new}/2^k \rfloor)$;          (update the state $x_i \to x_{i-1}$ )
    Store the final state $x_0 = x$;
```

This variant has interesting properties. Let us discuss some of them.

- As the encoding function is constantly updated, it seems to be difficult to extend attacks, whose goal is its recovery. Additionally, insertion/deletion of binary cycles into/from binary frame is very likely to be detected with high probability.
- Quality of compression could suffer and this aspect needs more investigation.
- As we have already noted, the $C(s, y)$ update does not need to be done for every symbol. It looks reasonable to allow longer runs of compression without $C(s, y)$ update. If the interval between two consecutive updates is too long, then one can expect that short cycles could be detectable. However, we do not know how this can be exploited by an adversary.
- It is possible to get rid of *PRGB* all together and rely on the internal structure of ANS. $C(s, y)$ can be stored as the following cyclic register:



The register stores all 2^R ANS states. Each state is allocated to a unique symbol s by the spread function. The above structure resembles the data structure used by the well-known RC4 cipher [10]. We can modify the RC4 key scheduling algorithm to swap states using the cryptographic key K only with a few instructions.

5 Security and Efficiency of Lightweight Encryption with ANS

Our goal is to strengthen a plain ANS using as little cryptography as possible. In our three compcrypt versions we use a cryptographically strong PRGB. This is the only cryptographic tool needed. Note that we assume that the adversary knows our ANS algorithm details except the cryptographic key K that is applying in PRGB to extract pseudorandom bits. For security evaluation, we normally assume that the adversary may have access to compcrypt algorithms for compression/encryption and for decompression/decryption. For simplicity we call them encryption and decryption oracles. Recall that encryption/decryption oracles are abstract concepts that allow the adversary to interact with cryptographic algorithms by asking an oracle to generate outputs for given inputs.

5.1 Security against Known-Plaintext Attacks

Observe that our adversary $\mathcal{A}$ is able to recover the window frame with high probability. The probability becomes 1 if different symbols are assigned to encoding with different lengths. We assume that this is the case. This removes one of important and natural strengths of a plain ANS. Consider our three versions.

- Compcrypt with state jumps – $\mathcal{A}$ may target the least probable symbols, which have been encoded into longest binary strings. She can recover pseudorandom bits by looking up the encoding table. To illustrate the attack, take into account example from Section 2. Assume $\mathcal{A}$ observes that the i-th symbol is s_0 with its encoding is 010. The next symbol is also s_0 and is compressed into 101. $\mathcal{A}$ knows from the ANS encoding table that the state $x_i = 18 \equiv 10010$ and $x_{i+1} = 29 \equiv 11101$, which means that the pseudorandom string has been 01111. This allows $\mathcal{A}$ to replace the symbol s_0 by other most frequent symbols.
- Compcrypt with double ANS – $\mathcal{A}$ knows the two ANS encoding tables and as above she targets two consecutive occurrences of the least probable symbol. She can identify a (hopefully) unique states for both ANS copies. Now she has to consider four possible cases: (1) both symbols are encoded by ANS_1, (2) both symbols are encoded by ANS_2, (3) first by ANS_1 and the second by ANS_2 and (4) first by ANS_2 and the second by ANS_1. If the tables are "sufficiently" different, $\mathcal{A}$ can identify the PRGB bit.
- Compcrypt with encoding function evolution – $\mathcal{A}$ knows the initial ANS and her goal is to recover the pseudorandom bits. As this compcrypt algorithm needs recalculation of encoding table every time the states are swapped, it is reasonable to expect that the swapping is not frequent. This assumption allows the adversary to launch the following attack. $\mathcal{A}$ starts from the initial and known encoding table and identify the state just before the first swap. After the first state swap, she guesses the second state (or equivalently PRGB bits). For each guess, she recalculates the encoding table and checks if it is consistent with sequence of observed symbols and their encodings. This costs her 2^{R-1} guesses on the average. The probability of success depends on the number of symbols encoded between two consecutive swaps.

5.2 Security against Ciphertext-only and Integrity Attacks

Here we assume that the adversary observes outputs from an encryption oracle or she knows a binary frame (that does not reveal window frame lengths). This time, to analyse the encryption, $\mathcal{A}$ needs first to guess symbols (knowing its probability distribution) of the corresponding symbol frame. Next for each symbol s_i, it has to guess the length $|b_i|$ of the corresponding output bits b_i. After correct guesses, $\mathcal{A}$ may apply the known-plaintextext attack scenario. To achieve confidentiality with λ bit security, one needs to modify our algorithms so they do not generate binary frames for small number of symbols so the probability of correct guesses of symbols and lengths of their compressions is smaller than 2^{λ}. Integrity checks are done on the receiving side. In our algorithms, the receiver verifies if the final state is the prearrange one (could be determined by both sides using PRGB). Note that probability of detecting tampering with output bit stream is equal to 2^{-R}.

5.3 Efficiency Evaluation

Our implementation of tabular version of ANS was written in the Go language (version 1.15.2). Throughout our experiments, we have used an OpenBSD 6.8-current installed on a Lenovo Thinkpad T450s with 12 GB of RAM and an i7-5600U CPU with 2 physical cores running at 2.6 GHz and hyper-threading enabled, which makes 4 threads available in total. All our compcrypt algorithms invoke PRBG. The impact of the PRBG on the execution time of the encoding and decoding heavily depends on its implementation. Our implementation use standard *Go* function provided by *math/rand*.

Let us discuss briefly some implementation details of our comcrypt algorithms with:

- state jumps – our experiments assume that state jumps are performed for each input symbol. The initial encoding/decoding tables are created precisely as in the plain ANS.
- double ANS – there are two plain ANS algorithms. The switch between the two is done by PRBG for each input symbol. The execution time should not be much different from the previous algorithm. A significant difference relates to an extra memory needed to store two encoding/decoding tables. Consequently, loading time may impact overall execution time. This may be noticeable when processing short streams of symbols.
- encoding function evolution – the algorithm is initialised to a plain ANS and then its encoding table is modified for each symbol by swapping the current state with a random one (chosen by PRBG). The swap might look like a computationally cheap operation but, in fact, each non-trivial swap involves recomputation of the encoding table. This means that for each symbol, we may expect up to 2^R table operations.

Our experiments are performed for three geometric probability distributions denoted by $p \in \{0.5, 0.7, 0.9\}$. The number of states in ANS are 2^R, where $R \in \{10, 12, 14\}$ and the parameter $m = 10$, which indicates the number of symbols in the source alphabet. For each of the above setting, we have processed 1024, 2048, 4096, 8192, 16394 randomly generated symbols and counted the output bits and execution times. Each experiment is executed 200 times with a random initial state. Average numbers of both output bits and encoding execution time have been computed. Figure 1 compares efficiency of our three compcrypt algorithms with the plain ANS. Clearly, compcrypt with encoding function evolution is the least efficient. The efficiency loss is attributed to state swaps and as expected, it is especially noticeable when processing a large number of symbols. On the average, compared to a plain ANS, compcrypt with double tables incurs extra

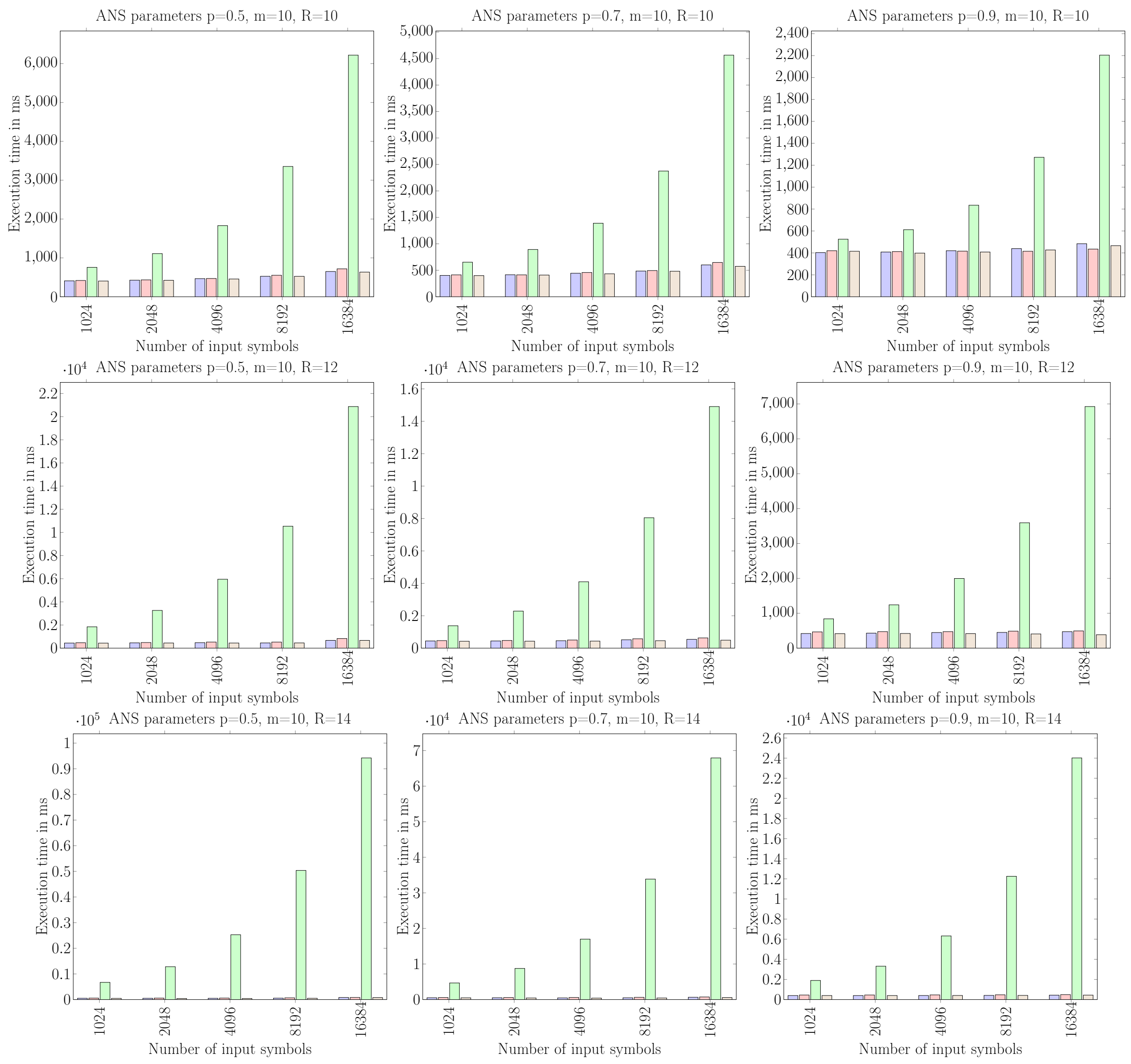


Fig. 1. Execution times of plain ANS (blue 🟦) and compcrypt algorithms with: double tables (red 🟥), encoding function evolution (green 🟩) and state jumps (brown 🟫)

overhead of 50 ms. It is caused by processing the second table. As one can expect, efficiency of compcrypt with state jumps is comparable to the one offered by a plain ANS.

Let us consider quality of compression provided by the three compcrypt algorithms. We use a plain ANS as a reference. Figure 2 describes our results. We observe that compcrypt with encoding function evolution lengthens output stream by $< 10\%$ in comparison to the plain ANS. Compcrypt with double tables increases the length of output bits by less than 1%. Compression quality of compcrypt with state jumps is similar to the one of a plain ANS.

6 Conclusions and Future Research

The work investigates joint compression and encryption for lightweight applications, where natural behaviour of ANS is enhanced using as little cryptography as possible. Consequently, resulting compcrypt algorithms offer low-security level for both confidentiality and integrity (against ciphertext-only adversaries). The only cryptographic tool used is PRBG, which can be chosen

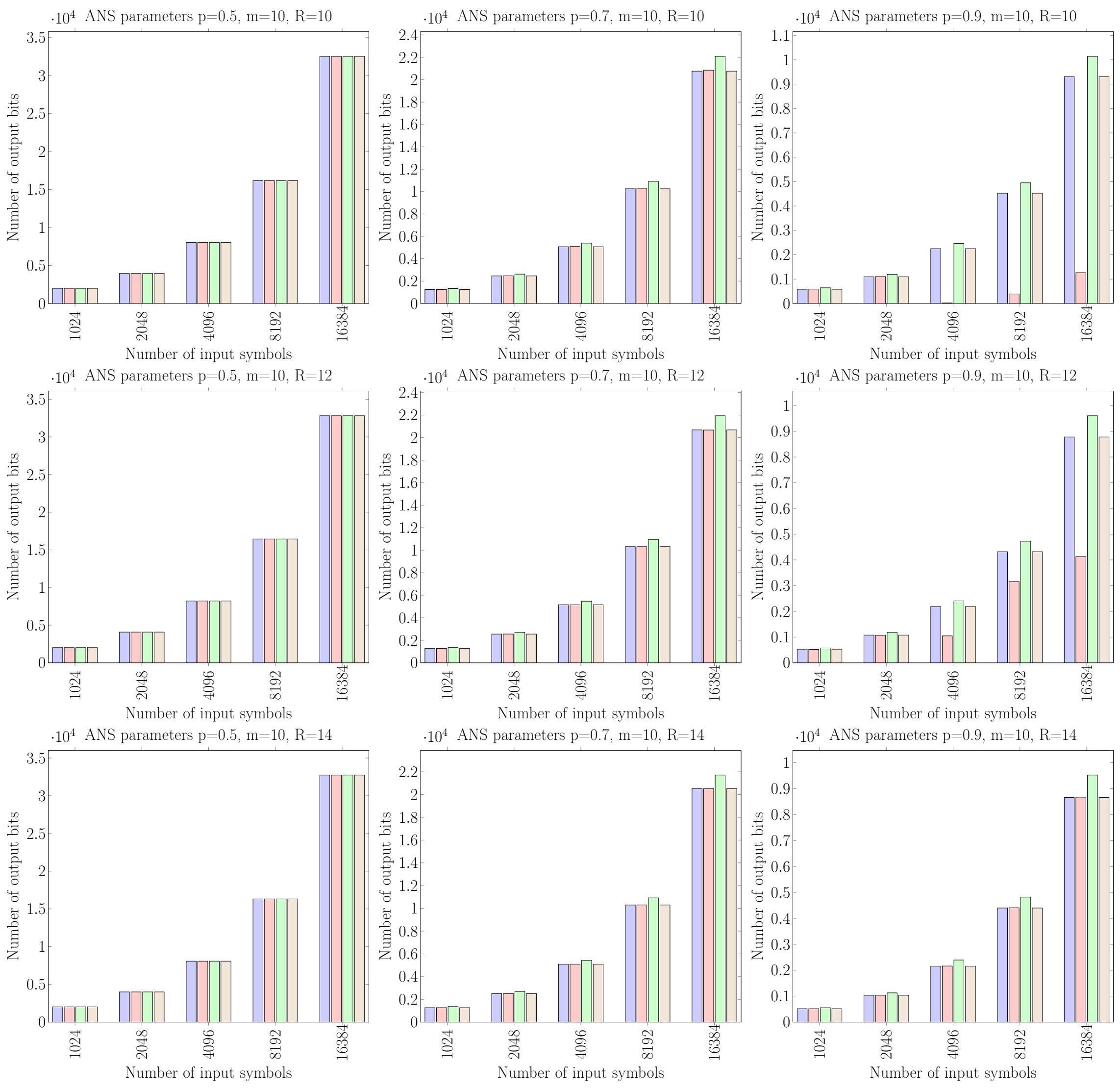


Fig. 2. Quality of the compression (measured by the number of output bits) for plain ANS (blue □) and compcrypt algorithms with: double tables (red □), encoding function evolution (green □), state jumps (brown □)

depending on efficiency and security requirements. For applications that require a decent security level, a PRBG based on a good quality stream cipher (such as Trivium [1]) is recommended. As hinted in the work, PRBG can be removed all together and replaced by a cryptographic key and make the encoding table dynamic (using encoding function evolution). This is an attractive direction for future research (connection with the RC4 cipher).

We propose three compcrypt algorithms. The first one applies a single ANS with state jumps controlled by PRBG. The second one uses two copies of ANS, where PRBG manages transition between copies. The third compcrypt deploys encoding function evolution that modifies encoding tables. Assuming a ciphertext-only adversary, the security level for confidentiality is mainly determined by the probability of guessing input symbols. It is significant for small number of symbols but diminishes exponentially when the number grows. This is true for all three algorithms. But when the guess is correct we deal with a known-plaintext attack. Under the attack, compcrypt with encoding function evolution offers best security. With the exception of compcrypt with encoding function evolution, the algorithms offer similar efficiency and compression quality as the plain ANS.

Note that compcrypt with encoding function evolution can be slightly modified so it preserves good security features and has “almost” the same efficiency and compression quality as the plain ANS. Instead of swapping states after processing any single symbol, compcrypt starts as the original algorithm (swapping states frequently) and then it gradually increases number of symbols between two consecutive swaps.

Acknowledgments

Paweł Morawiecki and Marcin Pawłowski have been supported by Polish National Science Center (NCN) grant 2018/31/B/ST6/03003. Josef Pieprzyk has been supported by Australian Research Council (ARC) grant DP180102199 and Polish National Science Center (NCN) grant 2018/31/B/ST6/03003.

References

[1] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 171–186, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] Jarek Duda. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540, 2013.

[3] Jarek Duda and Marcin Niemiec. Lightweight compression with encryption based on asymmetric numeral systems, 2016.

[4] David W. Gillman, Mojdeh Mohtashemi, and Ronald L. Rivest. On breaking a Huffman code. *IEEE Transaction on Information Theory*, 42(3):972–976, 1996.

[5] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[6] Donald Knuth. *The art of computer programming, Vol. 2*. Addison-Wesley, 1973.

[7] A. Moffat and M. Petri. Large-alphabet semi-static entropy coding via asymmetric numeral systems. *ACM Transactions on Information Systems*, 38(4):1–33, 2020.

[8] J.J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[9] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[10] Wikipedia. RC4. `https://en.wikipedia.org/wiki/RC4`. Accessed Dec 12, 2020.