Compcrypt–Lightweight ANS-Based Compression and Encryption

Seyit Camtepe, *Senior Member, IEEE*, Jarek Duda, Arash Mahboubi, Paweł Morawiecki, Surya Nepal, Marcin Pawłowski, and Josef Pieprzyk

***Abstract*— Compression is widely used in Internet applications to save communication time, bandwidth and storage. 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 and storage such as transmission of data from IoT devices/sensors. In particular, we propose three solutions for joint compression and encryption (compcrypt). The solutions offer different tradeoffs between security and efficiency assuming a slight compression deterioration. All of them use a pseudorandom bit generator (PRBG) based on lightweight stream ciphers. The first solution is close to original ANS and applies state jumps controlled by PRBG. The second one employs two copies of ANS, where compression is switched between the copies. The switch is controlled by a PRBG bit. The third compcrypt modifies the encoding function of ANS depending on PRBG bits. Security and efficiency of the proposed compcrypt algorithms are evaluated. The first compcrypt is the most efficient with a slight loss of compression quality. The second one consumes more storage but the loss of compression quality is negligible. The last compcrypt offers the best security but is the least efficient.**

Manuscript received January 14, 2021; revised April 18, 2021 and May 19, 2021; accepted June 25, 2021. Date of publication July 9, 2021; date of current version August 3, 2021. The work of Paweł Morawiecki and Marcin Pawłowski was supported by the Polish National Science Center (NCN) under Grant 2018/31/B/ST6/03003. The work of Josef Pieprzyk was supported in part by the Australian Research Council (ARC) under Grant DP180102199 and in part by the Polish National Science Center (NCN) under Grant 2018/31/B/ST6/03003. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Lejla Batina. *(Corresponding author: Josef Pieprzyk.)*

Seyit Camtepe and Surya Nepal are with Data61, CSIRO, Sydney, NSW 1466, Australia (e-mail: sayit.camtepe@data61.csiro.au; surya.nepal@data61.csiro.au).

Jarek Duda is with the Institute of Computer Science and Computer Mathematics, Jagiellonian University, 30-348 Krakow, Poland (e-mail: dudajar@gmail.co).

Arash Mahboubi is with the School of Computing and Mathematics, Charles Sturt University, Bathurst, NSW 2795, Australia (e-mail: amahboubi@csu.edu.au).

Paweł Morawiecki and Marcin Pawłowski are with the Institute of Computer Science, Polish Academy of Sciences, 01-248 Warsaw, Poland (e-mail: pawel.morawiecki@gmail.com; pawlowski.mp@gmail.com).

Josef Pieprzyk is with the Institute of Computer Science, Polish Academy of Sciences, 01-248 Warsaw, Poland, and also with Data61, CSIRO, Sydney, NSW 1466, Australia (e-mail: josef.pieprzyk@csiro.au).

Digital Object Identifier 10.1109/TIFS.2021.3096026

***Index Terms*— Asymmetric numeral system, compression, lightweight encryption, authentication.**

I. Introduction

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. Upon reception, the receiver recovers the original (redundant) stream. The focus of the paper is lossless compression, where a receiver is able to fully recreate the original/uncompressed data. However, video/audio compression is usually lossy.

Theoretical underpinning of compression is deeply rooted in Information Theory initiated by Shannon's seminal work [22]. Huffman codes (HC) [9] show optimal compression for symbol streams, whose probabilities follow very specific patterns (i.e. natural powers of $\frac{1}{2}$). Arithmetic coding (AC) (and its variants – see [14], [17], [21]) offers compression of symbols with an arbitrary probability distribution and is close to optimal. Its main drawback is a low efficiency as it requires complex arithmetics and heavy computational overhead. In contrast, an asymmetric numeral system (ANS) invented by Jarek Duda [5] and Duda *et al.* [6] 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. This allows to avoid expensive arithmetic operations. This variant is called the tabled ANS or simply tANS, where expensive arithmetics is replaced by lookup operation. Since 2015, tANS has been applied in Facebook Zstandard, Linux kernel and Android operating system to name a few.[1] A wide adoption of ANS by the IT industry makes it an attractive option for lightweight IoT applications. To the best of our knowledge, this paper is the first that deals with joint compression and encryption for ANS. There are, however, many published works that address the problem but for different compression algorithms (see for example [10], [11], [20], [23]).

The generic research question posed here is whether it is possible to design a single algorithm that simultaneously

[1]See https://en.wikipedia.org/wiki/Asymmetric_numeral_systems

compresses and encrypts (called compcrypt). The work addresses its variant when compression is based on ANS and there is no encryption per se but instead, internal states of ANS are controlled by random bits. To make implementation easy, we use PRBG to provide the requested randomness. We expect that

$$\begin{aligned} \text{cost(compcrypt)} &= \text{cost(ANS)} + \text{cost(PRBG)}, \\ \text{comp\_rate(compcrypt)} &\approx \text{comp\_rate(ANS)}, \\ \text{security(compcrypt)} &> \text{security(ANS)}, \end{aligned}$$

where comp_rate stands for compression rate.

Motivation: One of the emerging ANS applications is compression of data gathered, stored and transmitted by the Internet of things (IoT) devices. It is predicted that by 2025, the IoT infrastructure will include more than 75 billion devices [13]. Those Internet-enabled devices will be able to produce and consume a large amount of data. For example, one of the missions of 5G is to provide connectivity for data-intensive machines such as IoT devices forming the foundation of the fourth industrial revolution. Resource limitations on the most IoT devices restrict deployment of both compression and encryption algorithms, especially in real-time applications.

The work aims to extend ANS compression with cryptographic features to address this gap. The idea is to identify the natural properties of ANS, which, together with lightweight cryptographic tools, can provide a "decent" level of security for confidentiality and integrity [3]. This topic has been previously investigated in [7] by Duda and Niemiec. The authors consider a plain ANS with a (pseudo)randomly chosen encoding function. However, once it has been chosen (as a part of initialisation), it is fixed for the whole compression session. Due to an inherent cyclic nature of compression, this leaves a door ajar for possible integrity attacks, where an adversary may inject/remove parts of output bits without detection by the receiver. Note that in many application, data integrity is much more important than confidentiality. For example, environment sensing (temperature/pollution) or smart parking.

In contrast, we cryptographically change behaviour of an underlying ANS during symbol processing so the above mentioned integrity attacks fail with high probability. In particular, we investigate joint compression and encryption (compcrypt) for low-level security IoT communications. By low-level security, we mean IoT applications whose compromise causes relatively low damages or alternatively, an adversary attacking IoT communication has a limited computing resources. Our work is guided by the following requirements for lightweight compcrypt algorithms, i.e. (1) minimal use of cryptography, (2) security against ciphertext-only adversaries and (3) integrity checking mechanism,

Contributions: 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,
- provides three compcrypt solutions. First one is based on state jumps. This is a very basic solution that applies a plain ANS, where state transition is controlled cryptographically. It has a similar efficiency as a plain ANS with a slightly reduced compression quality. The second one applies two plain ANS algorithms with transition between the two controlled by PRBG bits. It is as efficient as the first solution but uses two encoding tables so it needs twice as much storage as the first compcrypt. Compression quality is almost the same as the one in ANS. The third one uses PRBG bits to modify ANS encoding function. This solution is the most secure but the least efficient,
- evaluates security and efficiency of the proposed compcrypt algorithms.

The rest of the work is structured as follows. Section II 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 IV analyses confidentiality and integrity of the plain ANS under ciphertext-only and known-plaintext attacks. Section V describes our three lightweight compcrypt algorithms. Section VI evaluates security and efficiency of the proposed algorithms. Section VII concludes the work.

II. Description of Asymmetric Numeral System

Let m be the size of an alphabet $\mathbb{S}$ (or $m = |\mathbb{S}|$), 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 $\Pr(s_i) = p_i \approx |\{j : s_j = i\}|/n$. 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 natural 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.

A. Bird-Eye View of ANS

ANS [5] 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), optimised for a given probability distribution, whose states are labelled by integers [16]. We describe building blocks of ANS without giving rationale for their design. This is enough to understand the encryption part of the paper. The reader interested in ANS details is referred to [5], [18].

The main data structure is determined by the number of states L. For simplicity, we assume that $L = 2^R$, where $R \in \mathbb{N}^+$ is a parameter, which determines the quality of compression. Let L_s denote the number of occurrences of symbol s, where $\sum_s L_s = L$ and L_s is an approximation of probabilities of $Pr(s) = p_s$. Define the following sets $\mathbb{L} = \{L, L+1, \ldots, 2L-1\}$ and $\mathbb{L}_s = \{L_s, L_s+1, \ldots, 2L_s-1\}$, where $s \in \mathbb{S}$. Given a state is x_i and a symbol s_i. ANS needs to determine the next state x_{i+1} and the output bits b_i. If one looks at ANS as FSM, then the state-transition function is defined by the following two steps:

1) the current state x_i is re-normalised by truncating enough least significant bits (LSB) so the truncated integer y belongs to $\mathbb{L}_{s_i}$,
2) calculates a new state x_{i+1} by applying an encoding function $C(s_i, y)$ and outputs the binary sequence b_i=LSB(x_i), which is a binary encoding of s_i.

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 $\frac{L_s}{L} \approx p_s$ determines quality of compression. This means that there are L_s different integers $x \in \mathbb{L}$ that encode s.
- 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 $y \in \mathbb{L}_s$. The columns are indexed by all consecutive integers starting from $y_{min} = \min_s L_s$. The last column index is $2^R - 1$. The entries of the s-th row for consecutive columns $L_s, L_s + 1, \ldots, 2L_s - 1$ create a set Γ_s of states arranged in an increasing order. Γ_s is also called symbol spread for s.
- The integers $x \in \Gamma_s$ can be chosen at random from $\mathbb{L}$ as long as any symbol spread pair is disjoint, i.e. $\Gamma_s \cap \Gamma_{s'} = \emptyset$ as long as $s \neq s'$, where $\Gamma_s = \{x \in \mathbb{L} | x = C(s, y);$ $y \in \mathbb{L}_s\}$ and $\bigcup_s \mathbb{L}_s = \mathbb{L}$.

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 $x \in \Gamma_s$, then it cannot occur in any other $\Gamma_{s'}$, otherwise the condition $\Gamma_s \cap \Gamma_{s'} = \emptyset$ is violated.

Encoding – given a state $x \in \Gamma_s$ 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 $x \in \Gamma_s$ 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 $k_s = k_s(x) = R - \lfloor \log_2 x \rfloor$. The k_s-bit string is read from $\mathcal{B}$ or $b_s = MSB(\mathcal{B})_{k_s}$, 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.

B. ANS Algorithms

The tANS 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. However, if symbol statistics changes, the algorithm may need to be rerun. $\mathbf{C}$ 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 $C(s, y)$ and $k_s(x)$ and
- the decoding functions $D(x)$ and $k_s(x)$.

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 $\bar{s} : \mathbb{L} \to \mathbb{S}$, such that $|\{x \in \mathbb{L} : \bar{s}(x) = s\}| = L_s$;
- 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 $C(D(x)) = x$ and $D(C(s, x)) = (s, x)$.
- calculate the function $k_s(x) = R - \lfloor \lg(x) \rfloor$, 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*.

Symbol Frame Encoding C

Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_n)$ and an initial state $x_n \in \mathbb{L}$; 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_F is the final state.

```
Steps: For $i = n, n-1, \ldots, 2, 1$ do    (encoding has to be in reverse direction)
  {
  $s := s_i$;
  $k_s(x) = \lfloor \lg(x/L_s) \rfloor$;    (compute the number of bits to be extracted)
  $b_i = x \mod 2^k$;    (send $k_s$ 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 $x_F$;
```

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

Binary Frame Decoding D

```
Input: A binary frame $\mathcal{B}$ and the final state $x = x_F \in \mathbb{L}$ 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 $y$)
    $k_s(x) = R - \lfloor \lg(x) \rfloor$;   (compute the number of bits to be read)
    $b_s = MSB(\mathcal{B})_{k_s}$;   (extract k MSB from $\mathcal{B}$)
    $\mathcal{B} := LSB(\mathcal{B})_{|\mathcal{B}|-k_s}$;   (update the stream of bits to be processed)
    $x := 2^k y + b_s$;   (update the state $x_{i-1} \to x_i$)
    }
  Check $x \overset{?}{=} x_n$   (integrity check)
```

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

C. Example

Design compression and decompression algorithms for a source with $m = 3$ symbols, where $\mathbb{S} = \{s_0, s_1, s_2\}$, $p_0 = \frac{3}{16}$, $p_1 = \frac{8}{16}$, $p_2 = \frac{5}{16}$ and a free parameter $R = 4$. Note that R determines quality of compression. The higher R the better compression but the algorithm is less efficient. The number of states $L = 2^R = 16$ and the state set $\mathbb{L} = \{16, 17, \ldots, 31\}$.

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 $L_2 = |\{19, 20, 23, 26, 28\}| = 5$. Note that sets Γ_s need to be arranged in increasing order.

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
s_0	$\binom{22}{00}$	$\binom{22}{01}$	$\binom{22}{10}$	$\binom{22}{11}$	$\binom{25}{00}$	$\binom{25}{01}$	$\binom{25}{10}$	$\binom{25}{11}$
s_1	$\binom{16}{0}$	$\binom{16}{1}$	$\binom{17}{0}$	$\binom{17}{1}$	$\binom{21}{0}$	$\binom{21}{1}$	$\binom{24}{0}$	$\binom{24}{1}$
s_2	$\binom{26}{0}$	$\binom{26}{1}$	$\binom{28}{0}$	$\binom{28}{1}$	$\binom{19}{00}$	$\binom{19}{01}$	$\binom{19}{10}$	$\binom{19}{11}$

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

To illustrate calculations, 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 $k_{s_0} = \lfloor \lg(x_i/L_0) \rfloor = \lfloor \lg(25/3) \rfloor = 3$, compute $x_{i+1} = C(s_0, \lfloor \frac{x_i}{2^k} \rfloor) = C(s_0, 3) = 18$ and $b_i = x_i \bmod 2^k = 25 \bmod 8 = 1 \longrightarrow 001$. Given an initial state $x_0 = 19$ (normally chosen at random), 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

$$\begin{tikzcd} (19) \arrow[r] & \binom{19}{s_1} \arrow[d] \arrow[r] & \binom{17}{s_1} \arrow[d] \arrow[r] & \binom{16}{s_2} \arrow[d] \arrow[r] & \binom{26}{s_1} \arrow[d] \arrow[r] & \binom{29}{s_2} \arrow[d] \arrow[r] & {} \\ & 1 & 1 & 0 & 0 & 01 & \end{tikzcd}$$

$$\begin{tikzcd} {} \arrow[r] & \binom{23}{s_1} \arrow[d] \arrow[r] & \binom{24}{s_1} \arrow[d] \arrow[r] & \binom{27}{s_0} \arrow[d] \arrow[r] & \binom{18}{s_2} \arrow[d] \arrow[r] & (28) \\ & 1 & 0 & 011 & 0 & \end{tikzcd}$$

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

Build the decoding table. The decoding function $D(x) = (s, y)$ can be obtained from $C(s, y) = x$ and is

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
s_i	s_1	s_1	s_0	s_2	s_2	s_1	s_0	s_2
k	1	1	3	2	2	1	2	2
x_{i+1}	$16{+}b_i$	$18{+}b_i$	$24{+}b_i$	$20{+}b_i$	$24{+}b_i$	$20{+}b_i$	$16{+}b_i$	$28{+}b_i$

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

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.

$$\begin{tikzcd} (28) \arrow[r] & \binom{28}{0} \arrow[d] \arrow[r] & \binom{18}{011} \arrow[d] \arrow[r] & \binom{27}{0} \arrow[d] \arrow[r] & \binom{24}{1} \arrow[d] \arrow[r] & \binom{23}{01} \arrow[d] \arrow[r] & {} \\ & s_2 & s_0 & s_1 & s_1 & s_2 & \end{tikzcd}$$

$$\begin{tikzcd} {} \arrow[r] & \binom{29}{0} \arrow[d] \arrow[r] & \binom{26}{0} \arrow[d] \arrow[r] & \binom{16}{1} \arrow[d] \arrow[r] & \binom{17}{1} \arrow[d] \arrow[r] & (19) \\ & s_1 & s_2 & s_1 & s_1 & \end{tikzcd}$$

III. Pseudorandom Bit Generation

For many IT protocols and simulations, there is a need for a source of random bits. A typical solution applies a pseudorandom bit generator (PRBG). Unfortunately, generated bits are no longer truly random. Many applications can be run successfully as long as pseudorandom sequences "look" random. Looking random can be equated to passing some statistical tests (such as the ones recommended by NIST [19]). However, a choice of statistical tests is highly arbitrary and having many tests to choose from, one can ask which test is really important and which ones can be ignored. Yao in his work [24] argues that there is an universal test, which, if passed, assures that all other statistical tests hold. This is the well known *next-bit test*. Given an adversary with polynomially-bounded computing resources who can observe a polynomial-size output sequence generated by PRBG. Then PRBG passes the next-bit test if the adversary is able to predict the next bit with probability no better than $1/2 + \varepsilon$, where ε is negligible. A *distinguisher* is an algorithm that implements the next-bit test. PRBG is called cryptographically strong (or CSPRBG) if it passes the next-bit test or alternatively, there is no distinguisher that can tell apart it from a truly random source.

There are two classes of CSPRBG: one whose security is anchored to a heuristic argument and the other – to an intractability assumption. The first class includes numerous designs based on nonlinear feedback shift registers (NFSR). For example, Trivium, Snow and Sober (see eStream portfolio *https://www.ecrypt.eu.org/stream/*). The second class includes a RSA-based PRBG that assumes intractability of integer factorisation [1] and a Bum-Blum-Shub PRBG, whose security rests on intractability of quadratic residuosity [2]. Needless to say, CSPRBG based on an intractability assumption tends to be inherently slow and, thus, not appropriate for devices (such as IoT) with limited computing resources.

In the context of this work, we target PRBG, whose security is heuristic as they are very efficient and can be easily implemented in both software and hardware. More secure candidates include winners of the eStream competition. Less secure PRBGs should not be discounted as they can be a viable option for lower-end security applications especially if one requires a security against a ciphertext-only adversary. In this circumstances, PRBG solutions based on linear feedback registers and linear congruences [12] can be used.

IV. 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 the 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 $C(s, y)$. $\mathcal{A}$ 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 $\mathcal{W}$ (this is also called a synchronisation problem). 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.

TABLE I
Plain ANS Adversarial Models

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 a binary frame $\mathcal{B}$ and x_F	– guessing a window frame $\mathcal{W}$ – finding full/partial symbol frame
known-plaintext	– As above + – observes a symbol frame $\mathcal{S} = (s_i)_1^n$	– 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

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.

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. This is true if an IoT device uses broadcast communication (such as Bluetooth or Wi-Fi) to interact with a server. The known-plaintext attack can be launched if an adversary has additionally an 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.

A. 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 it has guessed it, it 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.

1) Symbol Versus Window 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. In general, a symbol s with probability p_s gets either $\lfloor \log 1/p_s \rfloor$ or $\lceil \log 1/p_s \rceil$ bits. Consequently, the statistics of windows of different lengths is different from the source statistics. This is illustrated below.

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

$P(W = i) = P_i$ gives probability that ANS produces a window of the length i, where $i = 1, \ldots, \alpha$ 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; $i = 1, \ldots, \alpha$ 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.

2) 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 [12] 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 $k = 0, \ldots, n$.

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. It 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; $i = 1, \ldots, \alpha$. It is easy to determine a space of solutions $(n_1, \ldots, n_\alpha)$ for which Equation (1) holds. For a given $(n_1, \ldots, n_\alpha)$, the adversary needs to look through

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

equally probable events (window frames). Equation (2) represents a multinomial coefficient [12].

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. It 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.* [8] 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.

B. 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

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

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

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

□

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

- If $p_s = (1/2)^i$, then ANS assigns a i-bit window.
- 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 probabilities, 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 $\beta_{s_0}=1/2$.

1) 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
$$k_1+k_2+\ldots+k_n=N, \qquad (3)$$
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 patterns is
$$\binom{n}{N-\sum_{i=1}^n c_i.}$$
To maximise chances, the adversary tries from most probable. This can be done as it 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 of 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.

2) 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 $C(s,y)$, the adversary can design (adaptively) her own 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 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 ANS$_\mathcal{A}$ produces a bitstream that can be translated to bitstream generated by the original ANS. In this sense, both ANS and ANS$_\mathcal{A}$ are isomorphic. In other words, we are looking for a function F such that

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

The adaptive attack proceeds along the following steps:

1) The adversary $\mathcal{A}$ designs her 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), it finds b_1' from the encoding table of ANS$_\mathcal{A}$. It 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 is fully determined for all symbols and states. If the original ANS or ANS$_\mathcal{A}$ contain cycles then the algorithm fails. If a cycle occurs in the original ANS, $\mathcal{A}$ needs to "re-design" ANS$_\mathcal{A}$ by introducing the cycle of an appropriate length. On the other hand, if ANS$_\mathcal{A}$ hits a cycle, it needs re-design to remove the cycle.

C. 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:

$$(19)\to\underset{1}{\overset{\binom{28}{s_2}}{\downarrow}}\to\underbrace{\underset{00}{\overset{\binom{23}{s_2}}{\downarrow}}\to\underset{11}{\overset{\binom{19}{s_2}}{\downarrow}}\to\underset{1}{\overset{\binom{28}{s_2}}{\downarrow}}}_{cycle}\to\underset{00}{\overset{\binom{23}{s_2}}{\downarrow}}\to\underset{11}{\overset{\binom{19}{s_2}}{\downarrow}}\to\cdots$$

An adversary can inject/delete/replace the cycle and a receiver fails to detect it as the other bits are correctly decompressed and ANS reaches the correct final state. The deletion is illustrated below.

$$(19) \rightarrow \underset{1}{\overset{\binom{28}{s_2}}{\downarrow}} \rightarrow \underset{00}{\overset{\binom{23}{s_2}}{\downarrow}} \rightarrow \underset{11}{\overset{\binom{19}{s_2}}{\downarrow}} \rightarrow \cdots$$

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 $\mathcal{A}$ 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.
- $\mathcal{A}$ can increase its chances of guessing lengths of possible cycles by experimenting with random instances of ANS for a known R and a symbol statistics. $\mathcal{A}$ hopes that cycles of a target ANS follow the statistics gathered from random instances.

V. Lightweight Encryption With ANS

The analysis given in Section IV 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). In general, IoT devices have very restricted CPU and storage resources and adding extra encryption algorithm may not be practical. As IoT devices communicate with their servers via broadcasting (Bluetooth and Wi-Fi), the data can be subject to both eavesdropping and tampering with its contents. 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 IV, 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 IV, probability of a successful guess is negligible even for short sequences of symbols and decreases exponentially with the number of compressed symbols.

A. Compcrypt With State Jumps

This solution follows close the original ANS. The only change is pseudorandom selection of the next state. Consequently, it preserves the efficiency of ANS and enhances resistance against confidentiality and integrity attacks. The main cryptographic tool used here is a pseudorandom bit generator (PRBG), whose seed K is a secret cryptographic key that is shared between encoder and decoder. PRBG is used to produce sequence of integers *state_cor*, where $0 \leq$ *state_cor* $\leq 2^R$. The integer *state_cor* determines a jump from the current state $x \in \mathbb{L}$ to a new one

$$x := (x + \mathit{state\_cor}) \bmod 2^R + 2^R. \tag{4}$$

The integer $\mathit{state\_cor} := PRBG(i, K)$ 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.

A simple illustration of compcrypt with state jumps is given below.

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

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.

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 $PRBG$.
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)
    state_cor := $PRBG(no\_jumps, K)$;   (state correction)
    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
            $x := (x + state\_cor) \mod 2^R + 2^R$;   (jump)
            $offset := length$;   (reset offset)
            $no\_jumps --$;   (decrease the variable by 1)
            $state\_cor := PRBG(no\_jumps, K)$;   (next state correction)
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;   (the number of output bits)
        $b_i := x \mod 2^k$;   (output k LSB of current state $x = x_i$)
        $x := C(s, \lfloor x/2^k \rfloor)$;   (update the state $x_i \to x_{i-1}$)
    Store the final state;
```

- Consider a state jump. Note that a binary encoding b_i has to be computed for the state after jump, i.e. $x_i + state\_cor$. Otherwise, decoding fails.
- The only cryptographic component used is *PRBG*. 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 *PRBG* based on nonlinear feedback shift register (NFSR) or block cipher or hashing.
- Generation of integers $PRBG(i, K)$ for state correction should be easy in both directions: backward (for encoding where i decreases) and forward (for decoding where i increases).

B. Compcrypt With Double ANS

This solution is more expensive than the first one, offers better security especially against a known-plaintext adversary and preserves compression quality. 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:

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

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

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

```
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 $PRBG$.
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)
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;   (the number of output bits)
        $b_i := x \mod 2^k$;   (output k LSB of current state $x = x_i$)
        if $PRBG(i, K) = 0$ then
            $x := C_1(s, \lfloor x/2^k \rfloor)$;   (update the state $x_i \to x_{i-1}$)
        else
            $x := C_2(s, \lfloor x/2^k \rfloor)$;   (update the state $x_i \to x_{i-1}$)
    Store the final state;
```

We assume that *PRBG* generates a single bit for each call $PRBG(i, K)$. 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.
- An adversary who detects a cycle in the bit stream is unlikely to succeed in injecting it into the stream without detection.

C. 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 *PRBG*. As already noted that may be perceived as a waste of resources. An option could be to modify an 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.

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

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

$$x_i' = x_i + PRBG(i, K) \mod 2^R + 2^R,$$

where $0 \le PRBG(i, K) \le 2^R$. 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 $PRBG$.
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)
        $k := k_s(x) = \lfloor \lg(x/L_s) \rfloor$;    (the number of output bits)
        $b_i := x \mod 2^k$;    (output k LSB of current state $x = x_i$ )
        $x_{new} := x + PRBG(i, K)$;    (new pseudorandom state)
        $C(s, y)$ 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.

VI. 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 PRBG (for possible solutions see [4], [15]). 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 applied in PRBG.

A. Security of Compcrypt With State Jumps

We start from a lemma that sets up the background for our security discussion.

Lemma 1: Given a plain ANS as described in Section II. Then for a symbol $s \in \mathbb{S}$, ANS generates

- 1-bit encodings and for the symbol s, the encoding table row contains equal number of zeros and ones if $L_s = 2^{R-1}$,
- either empty-bit or 1-bit encodings and for the symbol s, the encoding table row contains equal number of zeros and ones if $L_s > 2^{R-1}$,
- either k_s-bit or (k_s+1)-bit encodings and for the symbol s, the encoding table row includes multiples of 2^{k_s} and 2^{k_s+1} if $L_s < 2^{R-1}$, where all 2^{k_s} and 2^{k_s+1} entries run through all possible k_s-bit or (k_s+1)-bit strings.

Proof: According to the frame coding algorithm (see Section II), for a state x, the algorithm extracts $k_s(x) = \lfloor \lg(x/L_s) \rfloor$ bits. As the state $x \in \{2^R, \ldots, 2^{R+1}-1\}$, we can write that

$$\lfloor \log_2 \frac{2^R}{L_s} \rfloor \le k_s \le \lfloor \log_2 \frac{2^{R+1}-1}{L_s} \rfloor \tag{5}$$

Case 1 If $L_s = 2^{R-1}$, then Equation (5) becomes

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

There is a single value $k_s = 1$, for which the above relation holds. As states x are chosen from the range $\{2^R, \ldots, 2^{R+1}-1\}$, it is easy to see that encodings are equal 1 if x is odd or 0, otherwise. The numbers of zeros and ones are the same (x runs through all consecutive integers from the interval).

Case 2 If $L_s > 2^{R-1}$, then the left side of Equation (5) gives $k_s = 0$, while the right side equals to $k_s = 1$. We can find the smallest x, for which $k_s(x) = 1$. It is easy to see that $x = 2L_s$. ANS produces empty encodings for $x \in \{2^R, \ldots, 2L_s - 1\}$. The other states output 1-bit encodings. As the number of states in the set $\{2L_s, \ldots, 2^{R+1}-1\}$ is even, the encodings contains equal number of zeros and ones.

Case 3 if $L_s < 2^{R-1}$, then $k_s = \lfloor \lg(2^R/L_s) \rfloor$. The smallest x that yields (k_s+1)-bit encoding is $x = 2^{k_s+1}L_s$. All states $x \in \{2^R, \ldots, 2^{k_s+1}L_s - 1\}$ generate k_s-bit encodings, while $x \in \{2^{k_s+1}L_s, \ldots, 2^{R+1}-1\}$ produce (k_s+1)-bit encodings. The number of states in the set

$\{2^{i+1}L_s, \ldots, 2^{R+1}-1\}$ equals to
$$2^{R+1} - 2^{k_s+1}L_s = 2^{k_s+1}(2^{R-k_s} - L_s),$$
where $(2^{R-k_s} - L_s) \geq 1$ is a multiplier and it has to be positive as the expression is positive. The encoding table row contains a multiple of 2^{k_s+1} entries. Any 2^{k_s+1} consecutive entries cover all possible k_s+1-bit strings as they correspond to consecutive states in the interval. Similarly, the number of states in the set $\{2^R, \ldots, 2^{i+1}L_s - 1\}$ can be calculated as $2^{k_s+1}L_s - 2^R = 2^{k_s}(2L_s - 2^{R-k_s})$. Using similar arguments, we argue that the entries cover all possible k_s-bit strings and they are repeated $(2L_s - 2^{R-k_s})$ times. □

We are ready to prove security of the compcrypt algorithm. Assume that we deal with a chosen-plaintext adversary $\mathcal{A}$, which can be defined as a known-plaintext adversary with extra ability to choose a symbol frame at will. $\mathcal{A}$ goal is to recover a pseudorandom sequence. This is a necessary prerequisite step for a possible attack against PRBG aiming at the secret key K recovery.

Theorem 1: Given ANS with state jumps described by Algorithm 1 and a chosen-plaintext adversary $\mathcal{A}$, who wants to recover a pseudorandom sequence. Assume further that $\mathcal{A}$ inputs n symbol frame and observes N-bit binary frame. Then $\mathcal{A}$ is able to guess $state\_cor$ integers with probability

- $2^{-n(R-1)}$ if $p_s = 1/2$;
- $\binom{N}{N-n}^{-1} L_s^{-(N-n)}(2^{R-1} - L_s/2)^{-n}$ if $p_s > 1/2$;
- $\binom{n}{\alpha}^{-1}\left(2^{-(R-1)}L_s - 2^{-k_s}\right)^{\alpha}\left(2^{-k_s} - L_s 2^{-R}\right)^{n-\alpha}$ if $p_s < 1/2$, where α is the number of k_s-bit encodings.

Proof: As $\mathcal{A}$ controls the input, it can apply different strategies for choosing symbols. Assume that $\mathcal{A}$ sends a symbol frame that repeats n times the same symbol s. There are the following three possible cases:

- If $p_s = 1/2$, $\mathcal{A}$ knows that each b_i is either 0 or 1 for $i = 1, \ldots, n$. Assume that PRBG generates random jumps $state\_cor$ uniformly at random. $\mathcal{A}$ needs to identify a $state\_cor$ from the values of two consecutive output bits b_i, b_{i+1}. $\mathcal{A}$ knows an encoding table and according to Lemma 1, $\mathcal{A}$ can guess $state\_cor$ with probability $2^{-(R-1)}$ as precisely half of states produce the correct b_i, b_{i+1}. For n symbols, $\mathcal{A}$ succeeds with probability $2^{-n(R-1)}$.
- If $p_s > 1/2$, $\mathcal{A}$ observes output of $N < n$ bits with $N-n$ bits empty encodings $\varnothing$. It can compute all possible $\binom{N}{N-n}$ binary frames, where each frame includes n bits and $(N-n)$ empty encodings $\varnothing$. Only one of them is correct. It is easy to verify that we have L_s distinct $state\_cor$ values when moving from b_i to b_{i+1}. They are $0 \to \varnothing$, $1 \to \varnothing$ and $\varnothing \to \varnothing$. Each of the other six options (i.e. $0 \to 0$, $0 \to 1$, $1 \to 0$, $1 \to 1$, $\varnothing \to 0$ and $\varnothing \to 1$) involves $2^{R-1} - L_s/2$ possible $state\_cor$ values. The probability of guessing correct values of $state\_cor$ is therefore equal to
$$\binom{N}{N-n}^{-1} L_s^{-(N-n)}(2^{R-1} - L_s/2)^{-n}.$$
- If $p_s < 1/2$, $\mathcal{A}$ observes output of $N > n$ bits, where each encoding b_i can be either k_s or (k_s+1)-bit long. Let α and β be the numbers of encodings with the length k_s and (k_s+1), respectively. $\mathcal{A}$ can compute α and β by solving the following two equations: (1) $\alpha+\beta = n$ and (2) $k_s\alpha + (k_s+1)\beta = N$. $\mathcal{A}$ does not know partition of output bits into encodings or in other words, it does not know the correct window frame. Clearly, there are $\binom{n}{\alpha} = \binom{n}{\beta}$ possibilities and only one correct. For each guess of a window frame, we analyse possible $state\_cor$ values that lead to correct transition of b_i to b_{i+1}. If b_{i+1} is k_s-bit long, then there are $(2L_s - 2^{R-k_s})$ possibilities (out of 2^R) that are consistent with the observation (Lemma 1). If b_{i+1} is (k_s+1)-bit long, then there are $(2^{R-k_s} - L_s)$ possibilities aligned with the observation (Lemma 1). Wrapping up, the probability of a successful guess of $\mathcal{A}$ is
$$\binom{n}{\alpha}^{-1}\left(2^{-(R-1)}L_s - 2^{-k_s}\right)^{\alpha}\left(2^{-k_s} - L_s 2^{-R}\right)^{n-\alpha}.$$
□

Few points are relevant here.

- Theorem 1 proves security when $state\_cor$ have the same length as the ANS states. It means that each jump is chosen independently at random from the full range of 2^R states. Probabilities of guessing pseudorandom bits $state\_cor$ are the smallest and they give the upper bound on security. Should $state\_cor$ be shorter, guessing probabilities are growing. In the case when $state\_cor$ is a single bit, guessing probabilities are equal to 1.
- In Lemma 1, $\mathcal{A}$ is free to choose an arbitrary strategy of symbol selection. However, it is expected that for a given instance of the algorithm, $\mathcal{A}$ first evaluates its chances by computing the relevant success probabilities from Lemma 1 and then $\mathcal{A}$ chooses the one that maximises its success probability.
- State jumps have a negative impact on compression. The reason for this is a flat probability distribution of states forced by PRBG. Note that probability distribution of a plain ANS follows $\approx 1/x$, where x is a state. So a plain ANS favours states with shorter encodings.
- Let us compare the compcrypt with state jumps with a plain ANS, whose output is XOR-ed with a PRBG keystream (ANS⊕PRBG). From the efficiency point of view, both solutions are more or less equivalent. A major difference lays in security. Note that for ANS⊕PRBG, a chosen-plaintext adversary $\mathcal{A}$ can extract whole keystream generated by PRBG. This may have grave implications for integrity as $\mathcal{A}$ can create valid but fake binary frames. For the compcrypt in hand, this is still possible but with a probability that quickly becomes negligible (see Lemma 1). Additionally, because $\mathcal{A}$ is forced to make guesses about $state\_cor$ generated by PRBG, it is possible to use PRBG with a lower security level that is more efficient.

B. Security of Compcrypt With Double ANS

Consider a chosen-plaintext adversary $\mathcal{A}$, who can input a symbol frame of its choice and can observe a corresponding binary frame. Again we assume that the compcrypt is public and the only unknown part is a secret key (or seed of PRBG). $\mathcal{A}$ knows the two ANS encoding tables and can use the least probable symbol to create a symbol frame that repeats the symbol. It can identify a (hopefully) unique states for both ANS copies. Now it 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 PRBG bit. This is to say that the algorithm leaks information of PRBG bits. Nevertheless, the leak is probabilistic and for long symbol frames, a chosen-plaintext adversary loses the guessing game most of the time.

C. Security of Compcrypt With Encoding Function Evolution

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. Our chosen-plaintext adversary $\mathcal{A}$ starts from the initial and known encoding table and identify the state just before the first swap. After the first state swap, it guesses the second state (or equivalently PRBG bits). For each guess, it recalculates the encoding table and checks if it is consistent with sequence of observed symbols and their encodings. This costs it 2^{R-1} guesses on the average. The probability of success depends on the number of symbols encoded between two consecutive swaps. This attack fails if the state swaps are done too frequently.

D. Other Cryptographic Attacks

Because of its internal structure, ANS is difficult to analyse using standard tools such man-in-the-middle, differential and linear attacks. The main difficulty seems to be irregular lengths of binary encodings that are assigned to symbols. The encodings are glued together with a single binary frame. To do any meaningful analysis, an adversary needs to split the frame into separate encodings. This unavoidably leads to guessing. There is, however, an exception – algebraic cryptanalysis. The heart of ANS is its symbol spread function. If the function can be represented by short polynomials or short Boolean expressions, then there is a hope that this analysis can work.

E. 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 installed on a Dell Precision T3610 desktop PC with 32 GB of RAM and an Intel Xeon E5-1650 with 6 physical cores running at 3.5 GHz and hyper-threading enabled, which makes 12 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 compcrypt algorithms with:

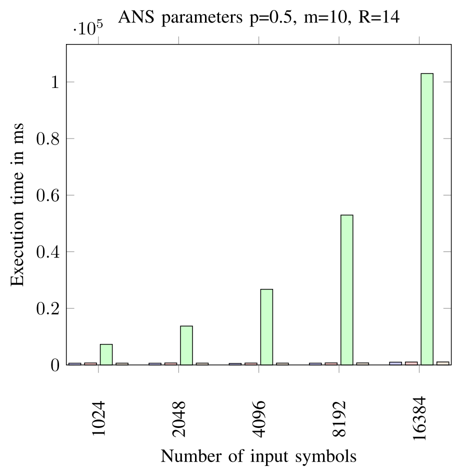
- state jumps – our experiments assume that state jumps are performed for each input symbol. The initial encoding/decoding tables are created precisely as in the plain ANS.
- double ANS – there are two plain ANS algorithms. The switch between the two is done by PRBG for each input symbol. The execution 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 geometric probability distributions with $p = 0.5$. The number of states in ANS is 2^R, where $R = 14$ and the parameter $m = 10$, which indicates the number of symbols in the source alphabet. We have processed 1024, 2048, 4096, 8192, 16394 randomly generated symbols and counted the output bits and execution times. Each experiment is executed 1000 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 (observe the multiplier 10^5 for the vertical axis reflecting the execution time). 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 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.

F. Comparison of ANS Algorithms

Table II summarizes our security and efficiency discussion. A plain ANS with a secret symbol spread function is a good option, when ciphertext-only security is required and integrity is not important. The ANS-AES compcrypt provides strong



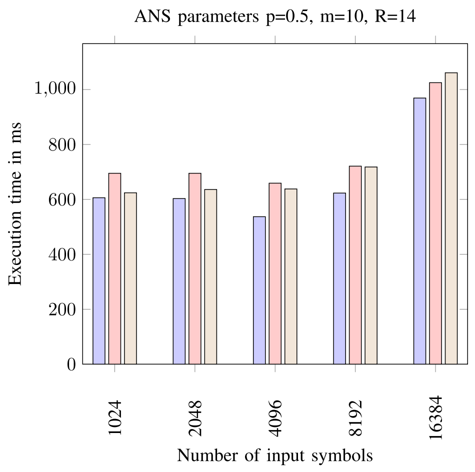


Fig. 1. Comparison of execution times of plain ANS (blue ▯) and compcrypt algorithms with: double tables (red ▯), encoding function evolution (green ▯) and state jumps (brown ▯).

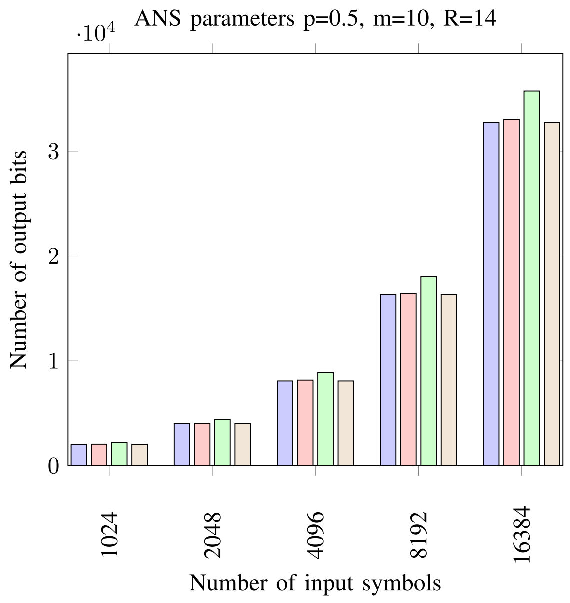


Fig. 2. Quality of the compression (measured by the number of output bits) for plain ANS (blue ▯) and compcrypt algorithms with: double tables (red ▯), encoding function evolution (green ▯), state jumps (brown ▯).

security and if implemented as authenticated encryption can support strong integrity. Its Achilles heel is poor efficiency that precludes it from IoT applications. ANS⊕PRBG is defenceless against a chosen-plaintext adversary, which can recover a PRBG sequence and reuse it to launch integrity attacks such as sending fake binary frames. ANS with state jumps offers chosen-plaintext security. It allows to check binary frame integrity that finds fakes with probability $(1 - 2^{-R})$. State jumps, however, interfere with state probability distribution causing a slight deterioration of compression measured by entropy. This weakness can be mitigated by restricting swaps to states that differ on least significant bits. Needless to say, this has security implications. The double ANS tends to leak pseudorandom bits to a chosen-plaintext adversary at the beginning of a session. It maintains compression rate as for a plain ANS. It consumes twice as memory as a plain ANS. The ANS with state evolution provides chosen-plaintext security and integrity. Its compression rate is as good as for a plain ANS. However, its main weakness is low efficiency due to the need of redesigning encoding table after each state evolution step.

Table III shows results of IoT implementations for compcrypt algorithms. Experiments have been done for frames of 1000 symbols generated by a source of 256 symbols according to the geometric probability distribution with $p = 0.5$. Instead of random selection of states, evolution light

TABLE II
COMPARISON OF COMPCRYPT ALGORITHMS

Algorithm	Security	Integrity	Entropy	Efficiency	IoT
Plain ANS [7]	CTO	✗	7.99 bits/byte	325MB/s	✓
ANS-AES	CPT	128 bits	7.99 bits/byte	321MB/s	✗
ANS⊕PRBG	CTO	✗	7.99 bits/byte	320MB/s	✓
ANS Sec. V-A	CPT	R bits	7.97 bits/byte	312MB/s	✓
ANS Sec. V-B	CPT	R bits	7.99 bits/byte	310MB/s	✓
ANS Sec. V-C	CPT	R bits	7.99 bits/byte	2MB/s	✓

Notation: CTO – ciphertext-only; CPT – chosen-plaintext; R-bit integrity means that tampering with binary frame is detected with probability $(1-2^{-R})$.

TABLE III
IOT IMPLEMENTATION OF COMPCRYPT ALGORITHMS

Platform	Algorithm	Efficiency
Raspberry Pi3	Plain ANS	35MB/s
	ANS-AES	11MB/s
	ANS⊕PRBG	34MB/s
	ANS with State Jumps	34MB/s
	ANS with Double Table	33MB/s
	ANS with Evolution	0.2MB/s
	ANS with Evolution Light	8MB/s
Raspberry Pi4	Plain ANS	56MB/sec
	ANS-AES	43MB/s
	ANS⊕PRBG	54MB/s
	ANS with State Jumps	54MB/s
	ANS with Double Table	51MB/s
	ANS with Evolution	0.5MB/s
	ANS with Evolution Light	18MB/s

swaps randomly chosen neighbouring states so an encoding table does not need to be re-built.

VII. Conclusion 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 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 [4]) 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.

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.

References

[1] W. B. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr, "RSA and Rabin functions: Certain parts are as hard as the whole," *SIAM J. Comput.*, vol. 17, pp. 194–208, Apr. 1988.

[2] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM J. Comput.*, vol. 15, no. 2, pp. 364–383, May 1986.

[3] W. J. Buchanan, S. Li, and R. Asif, "Lightweight cryptography methods," *J. Cyber Secur. Technol.*, vol. 1, nos. 3–4, 2017, pp. 197–201.

[4] C. D. Cannière, "Trivium: A stream cipher construction inspired by block cipher design principles," in *Information Security*. Berlin, Germany: Springer, 2006, pp. 171–186.

[5] J. Duda, "Asymmetric numeral systems as close to capacity low state entropy coders," *CoRR*, vol. abs/1311.2540, pp. 1–24, Oct. 2013.

[6] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *Proc. Picture Coding Symp. (PCS)*, Cairns, QLD, Australia, 2015, pp. 65–69.

[7] J. Duda and M. Niemiec, "Lightweight compression with encryption based on asymmetric numeral systems," 2016, *arXiv:1612.04662*. [Online]. Available: http://arxiv.org/abs/1612.04662

[8] D. W. Gillman, M. Mohtashemi, and R. L. Rivest, "On breaking a Huffman code," *IEEE Trans. Inf. Theory*, vol. 42, no. 3, pp. 972–976, May 1996.

[9] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[10] J. Kelley and R. Tamassia, "Secure compression: Theory & practice," Cryptol. ePrint Arch., IACR, Santa Barbara, CA, USA, Tech. Rep. 2014/113, 2014.

[11] M. O. Kulekci, "On scrambling the Burrows–Wheeler transform to provide privacy in lossless compression," *Comput. Secur.*, vol. 31, no. 1, pp. 26–32, 2012.

[12] D. Knuth, *The Art of Computer Programming*, vol. 2. Reading, MA, USA: Addison-Wesley, 1973.

[13] X. Liu, Y. Yang, K.-K. R. R. Choo, and H. Wang, "Security and privacy challenges for Internet-of-Things and fog computing," in *Proc. Wireless Commun. Mobile Comput.* London, U.K.: Hindawi, 2018, pp. 1–3.

[14] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 620–636, Jul. 2003.

[15] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, 2001.

[16] S. Mihov and K. U. Schulz, "Finite-state techniques: Automata, transducers and bimachines," *Cambridge Tracts Theor. Comput. Sci.*, 1st ed., vol. 60. Cambridge, U.K.: Cambridge Univ. Press, 2019.

[17] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inf. Syst.*, vol. 16, no. 3, pp. 256–294, Jul. 1998.

[18] A. Moffat and M. Petri, "Large-alphabet semi-static entropy coding via asymmetric numeral systems," *ACM Trans. Inf. Syst.*, vol. 38, no. 4, pp. 1–33, 2020.

[19] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST Special Publication 800-22, Gaithersburg, MD, USA, May 2001, vol. 800, p. 163.

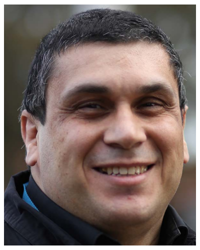
[20] V. Pudi, A. Chattopadhyay, and K. Lam, "Secure and lightweight compressive sensing using stream cipher," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 3, pp. 371–375, Mar. 2018.

[21] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM J. Res. Develop.*, vol. 20, no. 3, pp. 198–203, May 1976.

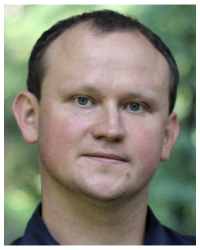
[22] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Oct. 1948.

[23] D. Xie and C.-C. Kuo, "Secure Lempel-Ziv compression with embedded encryption," *Electron. Imag., Int. Soc. Opt. Photon.*, vol. 5681, Mar. 2005, pp. 318–327.

[24] A. C. Yao, "Theory and application of trapdoor functions," in *Proc. 23rd Annu. Symp. Found. Comput. Sci. (SFCS)*, Chicago, IL, USA, Nov. 1982, pp. 80–91.



Seyit Camtepe (Senior Member, IEEE) received the Ph.D. degree from Rensselaer Polytechnic Institute in 2007. He is currently a Principal Research Scientist and the Team Leader with CSIRO Data61. He was with Technische Universitaet Berlin as a Senior Researcher and QUT as a Lecturer. He was among the first to investigate the security of android smartphones and inform society for the rising malware threat. His research interests include autonomous security, malware detection and prevention, smartphone security, applied and malicious cryptography, and CII security.



Jarek Duda received the M.Sc. degree in mathematics, the Ph.D. degree in computer science, and the Ph.D. degree in physics. He is currently an Assistant Professor with Jagiellonian University. He is mainly focused on information theory and statistical analysis, and is known from introduction of asymmetric numeral systems.

Arash Mahboubi received the B.E. degree(Hons.) in computer science specializing in computer security from Staffordshire University, Kuala Lumpur, Malaysia, in 2012, the master's degree in information security from the University of Technology Malaysia, Johor Bahru, Malaysia, in 2013, and the Ph.D. degree in computer science from the Queensland University of Technology (QUT), Brisbane, Australia, in 2018. From 2016 to 2019, he was a Sessional Academic with the School of Electrical Engineering and Computer Science, QUT. Since 2019, he has been a Lecturer with the School of Computing and Mathematics, Charles Sturt University, NSW, Australia. His research interests include computer/mobile malware, ransomware, malware analysis, modeling, and malware epidemic.

Paweł Morawiecki is currently an Associate Professor with the Institute of Computer Science, Polish Academy of Sciences, where he has been the Head of the Cryptography Group since 2017. His field of expertise includes cryptanalysis and cryptographic algorithms design. Recently he is involved in research at the intersection of security and artificial intelligence, particularly deep neural networks.

Surya Nepal is currently a Senior Principal Research Scientist with CSIRO's Data61. He also leads the distributed systems security group comprising more than 30 research staff and more than 50 postgraduate students. He is also the Team Leader of the Cybersecurity Cooperative Research Centre (CRC), a national initiative in Australia. His main research focus is in the development and implementation of technologies in the area of cybersecurity and privacy, and AI and cybersecurity. He has more than 300 peer-reviewed publications to his credit. He is a member of the Editorial Board of IEEE TRANSACTIONS ON SERVICES COMPUTING, *ACM Transactions on Internet Technology*, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and *Frontiers of Big Data Security Privacy, and Trust*.

Marcin Pawłowski received the Ph.D. degree from the Warsaw University of Technology. His Ph.D. thesis was on the security of Internet of Things. He currently holds a post-doctoral position at the Institute of Computer Science, Polish Academy of Sciences. He is also a consultant working for startups from blockchain and the Internet of Things industries, where he crosses scientific theories with real-world applications. His main research interests lie around design and analysis of security, network, and cryptographic protocols, especially with applications to distributed systems.

Josef Pieprzyk is currently a Senior Principal Research Scientist with Data61, CSIRO, and a Professor with the Institute of Computer Science, Polish Academy of Sciences. He has published 5 books, and edited 10 books (conference proceedings), 6 book chapters, and more than 300 articles in refereed journals and refereed international conferences. His main research interests are cryptology and information security, including design and analysis of cryptographic algorithms (such as encryption, hashing, and digital signatures), secure multiparty computations, cryptographic protocols, copyright protection, e-commerce, web security, and cybercrime prevention.