International Journal of Information Security (2022) 21:1051–1067
https://doi.org/10.1007/s10207-022-00597-4

REGULAR CONTRIBUTION

ANS-based compression and encryption with 128-bit security

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

Published online: 13 July 2022

Abstract
The bulk of Internet interactions is highly redundant and also security sensitive. To reduce communication bandwidth and provide a desired level of security, a data stream is first compressed to squeeze out redundant bits and then encrypted using authenticated encryption. This generic solution is very flexible and works well for any pair of (compression, encryption) algorithms. Its downside, however, is the fact that the two algorithms are designed independently. One would expect that designing a single algorithm that compresses and encrypts (called compcrypt) should produce benefits in terms of efficiency and security. The work investigates how to design a compcrypt algorithm using the ANS entropy coding. First, we examine basic properties of ANS and show that a plain ANS with a hidden encoding table can be broken by statistical attacks. Next, we study ANS behavior when its states are chosen at random. Our compcrypt algorithm is built using ANS with randomized state jumps and a sponge MonkeyDuplex encryption. Its security and efficiency are discussed. The design provides 128-bit security for both confidentiality and integrity/authentication. Our implementation experiments show that our compcrypt algorithm processes symbols with a rate up to 269 MB/s (with a slight loss of compression rate) 178 MB/s.

Keywords Encryption · Authentication · Integrity · Statistical attacks · Entropy encoding · Compression · Asymmetric numeral systems · ANS · Keccak

1 Introduction

Shannon in his seminal paper [1] investigates a problem of data transmission via a noisy and unreliable communication channel. He has shown that errors during transmission can be corrected if data are encoded with enough redundancy. Error correcting codes are developed to provide a carefully designed redundancy so the original data can be recovered even if some bits have been corrupted during transmission. The reverse problem and main focus of theory of entropy coding is how to remove redundancy from transmitted data. This is extremely important for growing Internet applications, where almost all data transmitted are highly redundant (for instance, photographs, music and video streaming). Compression of data simply saves time and bandwidth. Original data can be easily recovered by running a decompression algorithm.

The first entropy coding algorithm applies the well-known Huffman coding [2]. It offers optimal compression for symbols that follow a probability distribution that are integer powers of $1/2$. It is suboptimal, however, for any probability distribution that deviates from it. An interesting analysis of the code can be found in [3]. The arithmetic/range coding [4–6] offers a significant improvement as it allows to compress symbols for an arbitrary probability distribution. Its main weakness, however, is a heavy computation overhead. Asymmetric numeral systems (ANS) are relatively a newcomer that provides both efficient and close to optimal entropy coding (see [7–9]). Since its invention, ANS has been adopted widely in the IT industry. It is being used in the following compressors: Facebook Zstandard, Apple LZFSE, Google Draco 3D, PIK image, CRAM DNA and Dropbox DivANS, to name a few most prominent ones. It is recommended by RFC 8478 for MIME and HTTP. ANS is also

✉ Josef Pieprzyk
josef.pieprzyk@csiro.au

1 Data61, CSIRO, Sydney, Australia

2 Jagiellonian University, Cracow, Poland

3 School of Computing and Mathematics, Charles Sturt University Port Macquarie, Port Macquarie, Australia

4 Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

used by the JPEG XL next generation image compression standard.

The recent Covid-19 pandemic is forcing people to social-distance themselves by working/interacting via Internet. This further boosts already accelerating trend for people to turn to the Internet for entertainment (games and video streaming) and work teleconferencing (via Zoom, Skype, Webex or Microsoft Teams). Clearly, these events put data compression and security (confidentiality, authentication and integrity) at the forefront [10]. A natural solution is to use compression followed by encryption. A better option is to design a joint compression and encryption (further called compcrypt). It is expected that it is possible to exploit the natural properties of ANS to simplify encryption without compromising security. A research problem we address in this work is the design of a compcrypt algorithm such that

$$\begin{aligned}&\text{cost(compcrypt)} \ll \text{cost(compression)} + \text{cost(encryption)};\\ &\text{security(compcrypt)} = \text{security(encryption)};\\ &\text{comp\_rate(compcrypt)} \approx \text{comp\_rate(compression)},\end{aligned}$$

where comp_rate stands for compression rate.

Duda and Niemiec consider a plain ANS as a compcrypt algorithm in their work [11]. In their solution, a sender designs ANS by selecting a symbol spread function using pseudorandom bit generator (PRBG) initialized with a cryptographic key. The symbol spread determines encoding and decoding tables. The receiver also gets an encrypted final ANS state that serves as an authentication/integrity tag. The receiver checks if after decompression, ANS arrives in the correct state. The compcrypt algorithm holds up well against ciphertext-only adversaries. It is completely defenceless against integrity attacks as shown in the work [12]. The work also proposes three versions of compcrypt with a better security level. The versions target low-security devices/sensors with limited computing resources. They apply as little cryptography as possible. In fact, the only cryptographic tool is PRBG. An integrity tag is implemented by encrypted final state that provides 11-bit security for ANS with 2048 states.

Motivation All major teleconferencing vendors (Zoom, Webex and Microsoft) have already implemented end-to-end encryption. Likewise, video streaming services (Netflix, Stan, Foxtel, etc.) routinely apply strong encryption. Encryption is done for already compressed video/voice bitstreams using a full AES. This is expensive as it does not exploit potential benefits flowing from designing a single algorithm that compresses and encrypts. Compcrypt algorithms published so far are appropriate for lightweight applications only. This paper fills the gap and proposes a 128-bit secure compcrypt that can be easily upgraded to a higher security level if needed. This is due to the fact that encryption is done using a sponge structure.

Contribution The work starts from an analysis of a plain ANS. By plain ANS, we mean ANS without encryption. The analysis guides our design effort. More precisely, we claim the following contributions:

- Analysis of a plain ANS compression rate. We exploit Markov chains to determine precisely probability distribution of ANS states. As a result, we can calculate probability distribution of the lengths of binary encodings.
- Statistical attacks on a plain ANS. We show that a adaptive-statistics adversary is able to apply a divide-and-conquer algorithm to determine an encoding table (or equivalently a symbol spread function) much faster than the exhaustive search.
- Development of ANS variants, where ANS are either fully randomized (as in [12]) or partially randomized. We prove that the variants leak no information about the internal ANS structure. They are used as a basic building block for our compcrypt algorithm.
- Design of a 128-bit secure compcrypt algorithm. The algorithm uses a sponge MonkeyDuplex with the KECCAK-f round permutation P. We evaluate its security and efficiency. We also propose an extension of the compcrypt algorithm for infinite streams.

The rest of the paper is structured as follows. Section 2 introduces asymmetric numeral systems. Section 3 investigates compression rates of ANS and compression losses. The section also presents a statistical attack against ANS. Section 4 discusses design principles for a joint compression and encryption. It examines behavior of a plain ANS when its states are chosen at random. The section also introduces the MonkeyDuplex structure, which is used for authenticated encryption. Section 5 describes our compcrypt algorithm. Section 6 analyses security of the algorithm. Section 7 compares efficiency of our algorithm with other algorithms for joint compression and encryption. Section 8 shows how to adapt our compcrypt algorithm for streaming. Section 9 concludes the work.

In the work, we use notations that are presented in Table 1.

2 Asymmetric numeral systems

The main idea behind ANS is an observation that for any integer $x \in \mathbb{N}$, it is possible to append an encoding of a symbol $s \in \mathbb{S}$ that occurs with probability p_s, hence carrying $\lg(1/p_s)$ bits of information ($\lg \equiv \log_2$). Assuming that we need $\lg x$ bits to represent the integer x, the new integer $x' \in \mathbb{N}$ that includes the encoding of x and s should contain $\lg(x') \approx \lg x + \lg 1/p_s$ bits. In other words, we need to

Table 1 Notations

$\mathcal{A}$	Adversary
$\mathbf{b}$	Bitstream frame
$\mathcal{CH}$	Challenger
$C(x, s)$	ANS encoding function
$\mathcal{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_N)$	Ciphertext frame
$D(x')$	ANS decoding function
$\mathbb{E}(x_i, s_i) = (x_{i+1}, b_i)$	ANS encoding table
$H(\mathbb{S})$	Symbol source entropy
$\mathbb{I} = \{L, \ldots, 2L-1\}$	Set of ANS states
$k(x) = R - \lfloor \lg(x) \rfloor$	Number of bits read from $\mathbf{b}$
$k_s(x) = \lfloor \log_2 \frac{x}{L_s} \rfloor$	Number of bits of encoding of s when ANS is in state x
K	Cryptographic key
$L = \lvert\mathbb{I}\rvert = 2^R$	Number of ANS states
$L_s = Lp_s$	Number of states assigned to $s \in \mathbb{S}$
$\mathbb{L}_i = \{x \lvert \bar{s}(x) = s_i\}$	ANS states assigned to s_i
$\mathbb{N}$	Set of natural numbers
p_s	Probability of symbol s
P	Round of Keccak permutation
R	Parameter of ANS
$\langle \mathcal{R}_x \lvert x \in \mathbb{I} \rangle$	Markov chain equations
$\mathbb{S}$	Set of symbols
$\lvert\mathbb{S}\rvert = n$	Cardinality of $\mathbb{S}$
$\mathcal{S} = (s_1, \ldots, s_\ell)$	Symbol frame
$\bar{s} : \mathbb{I} \to \mathbb{S}$	Symbol spread function
T	128-bit authentication tag

Table 2 Encoding table for binary symbols with probabilities $p_s = 1/2$

s/x'	0	1	2	3	4	5	6	7	8	9	⋯
$s=0$	0		1		2		3		4		⋯
$s=1$		0		1		2		3		4	⋯

find a suitable reversible encoding function $C(x, s)$ such that $x' = C(x, s) \approx x/p_s$.

Consider a binary case, when $s \in \{0, 1\}$ and occurs with the probability p_s. ANS uses an encoding function $C(x, s) = x' = 2x + s$. A decoding function $D(x')$ allows to recover both x and s as $D(x') = (x, s) = (\lfloor x'/2 \rfloor, x' \pmod 2)$. A standard binary coding $C(s, x)$ for uniform probability distribution $p_s = 1/2$ is represented by Table 2.

The coding function generates a set of all integers. It splits into two disjoint subsets $\mathbb{I}_0 = \{x' | x' = C(x, s = 0); x \in \mathbb{N}\}$ all even integers and $\mathbb{I}_1 = \{x' | x' = C(x, s = 1); x \in \mathbb{N}\}$ all odd integers. For example, to encode a sequence of symbols 0111 starting from $x = 0$, we have the following sequence $x = 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{1} 7$. Decoding is done in reverse. This works as for each x', there is a unique pair (x, s) or $D(x') = (x, s)$. If starting with $x = 1$ instead, decoder could easily determine when to stop.

Table 3 Encoding table for binary symbols with probabilities $p_0 = 1/4$ and $p_1 = 3/4$

s/x'	0	1	2	3	4	5	6	7	8	9	⋯
$s=0$	0				1				2		⋯
$s=1$		0	1	2		3	4	5		6	⋯

The binary case can be generalized for an arbitrary probability distribution $p_s \neq 1/2$. This time we need to modify sets $\mathbb{I}_s; s \in \{0, 1\}$, such that the cardinality of the set $\mathbb{I}_s \cap [0, x)$ follows closely $p_s \cdot x$ or $|\mathbb{I}_s \cap [0, x)| \approx p_s \cdot x$, where $[0, x) := \{0, 1, \ldots, x-1\}$ denotes a set of all integers between 0 and x (including 0). Let us take an example, when $s \in \{0, 1\}$ with probabilities $p_0 = 1/4$ and $p_1 = 3/4$. Our encoding function $x' = C(x, s)$ is given by Table 3.

Take the same sequence of symbols 0111 and start from $x = 0$. The encoding is as follows: $x = 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3$. We can see that this encoding is shorter than for the encoding considered in the first example.

The approach described above can be extended for an arbitrary number of symbols, where $s \in \mathbb{S}$ and $|\mathbb{S}| \geq 2$. Sequence of natural numbers $\mathbb{N}$ is divided into intervals, each containing 2^R integers, where R is an integer parameter. Each interval includes $L_s \approx 2^R p_s$ integers/states, where $\sum_s L_s = 2^R$. Given a symbol s and an interval with 2^R integers, whose locations are indexed by integers from $0, \ldots, 2^R - 1$. Then integers/states assigned to s are put in L_s consecutive locations from $[c_s, c_{s+1})$, where c_s is the first location, $c_{s+1} - 1$ is the last location and $c_s = \sum_{i=0}^{s-1} L_i$. For example consider blocks of 4 columns (intervals) from Table 3. For $s = 1$, each block contains states at locations $[1, 4)$, where $L_0 = 1$ and $L_1 = 3$. We can construct an appropriate encoding table that has $n = |\mathbb{S}|$ rows and enough column so you can process a long enough sequence of symbols. Due to an elegant mathematical structure, encoding can be done using simple calculations. Given a state $x \in \mathbb{N}$ and a symbol $s \in \mathbb{S}$, then we can calculate $C(x, s)$ as follows:

- Identify a block/interval that contains x. The integer $2^R \lfloor x/L_s \rfloor$ points to first x' of the block.
- Compute an offset (within L_s block locations), which is $(x \bmod L_s)$.
- Find c_s, which gives the location of first state associated with s in the block.
- Determine $C(x, s) = 2^R \lfloor x/L_s \rfloor + (x \bmod L_s) + c_s$

A decoding function $D(x') = (x, s)$ can be calculated as $D(x') = \left(L_s \lfloor x'/2^R \rfloor + x' \bmod 2^R - c_s, s\right)$, where s is iden-

tified by checking if $c_s \leq x \bmod 2^R \leq c_{s+1}$. For instance, consider Table 3 and its $C(x, s)$. Finding state $x' = C(6, 1)$ can done directly from the table but also computed as $x' = 2^2 \cdot \lfloor 6/3 \rfloor + (6 \mod 3) + 1 = 9$. To decode $x' = 9$, we first determine s by computing $x' \bmod 2^R = 1$. This is a first location for $s = 1$. Knowing that $s = 1$, we can find $x = 3 \cdot 2 + 1 - 1 = 6$.

We have shown that compression operations can be simplified by defining an appropriate interval/block of the length 2^R. However, while encoding a sequence of ℓ symbols $s \in \mathbb{S}$, the final state x' grows very quickly and $\lg(x') \approx \ell H(\mathbb{S})$, where $H(\mathbb{S})$ is an entropy of the symbol source. Clearly, handling very large integers (thousands or millions of bits) becomes a major efficiency bottleneck. To deal with this, ANS uses the so-called *re-normalization* operation. The idea is to keep a state x within an interval of a fixed length, for instance $x \in \mathbb{I} = [2^\alpha, 2^{2\alpha})$. If ANS is in the state x and gets a sequence of symbols so $x' \geq 2^{2\alpha}$, then it outputs bits $x' \pmod{2^\alpha}$ (as partially compressed bits) and reduces the state $x \longleftarrow \lfloor x'/2^\alpha \rfloor$. Note that re-normalization is reversible. Knowing the pair $(x' \pmod{2^{16}}, \lfloor x'/2^{16} \rfloor)$, it is easy to reconstruct x'. In practice, ANS applies $\mathbb{I} = [2048, 4096)$ for 256 symbols. Using re-normalization allows ANS to achieve efficient entropy coding and also it can be conveniently represented as an encoding table - we focus here on this fully tabled case (called tANS).

The ANS entropy coding includes the following algorithms: initialization, symbol frame coding and binary frame decoding. They are given below.

Initialization

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

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

Steps: Initialization proceeds as follows:

- calculate the number of states $L = 2^R$;
- determine the set of states $\mathbb{I} = \{L, \ldots, 2L - 1\}$;
- 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 $\bar{s} : \mathbb{I} \to \mathbb{S}$, such that $|\{x \in \mathbb{I} : \bar{s}(x) = s\}| = L_s$;
- establish the coding function $C(s, y) = x$ for the integer $y \in \{L_s, \ldots, 2L_s - 1\}$, which assigns states $x \in \mathbb{I}$ according to the symbol spread function;
- compute the function $k_s(x) = \lfloor \lg(x/L_s) \rfloor$ for $x \in \mathbb{I}$ 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 $x \in \mathbb{I}$ assigns its unique symbol (given by the symbol spread function) and the integer y, where $L_s \leq y \leq 2L_s - 1$. 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.

Symbol Frame Coding

Input: A sequence of symbols (frame) $\mathcal{S} = (s_1, s_2, \ldots, s_\ell)$ and an initial state $x = x_\ell \in \mathbb{I}$.
Output: An output bit stream $\mathbf{b} = (b_1|b_2|\ldots|b_\ell)$, where $|b_i| = k_{s_i}(x_i)$ and x_i is state in the i-th step.
Steps: For $i = \ell, \ell - 1, \ldots, 2, 1$ do
{
$s := s_i$;
$k = k_s(x) = \lfloor \lg(x/L_s) \rfloor$;
$b_i = x \mod 2^k$;
$x := C(s, \lfloor x/2^k \rfloor)$;
};
Store the final state $x_0 = x$;

Binary Frame Decoding

Input: A bitstream frame $\mathbf{b}$ and the final state $x = x_0 \in \mathbb{I}$ of the encoder.
Output: Symbol frame $\mathcal{S}$
Steps: while $\mathbf{b} \neq \emptyset$:
{
$(s, y) = D(x)$;
$k = k(x) = R - \lfloor \lg(x) \rfloor$;
$b = MSB(\mathbf{b})_k$;
$\mathbf{b} := LSB(\mathbf{b})_{|\mathbf{b}|-k}$;
$x := 2^k y + b$;
}

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

The following instance of ANS is used throughout the work. Given 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 the parameter $R = 4$. The number of states is $L = 2^R = 16$ and the state set equals $\mathbb{I} = \{16, 17, \ldots, 31\}$. A symbol spread function $\bar{s} : \mathbb{I} \to \mathbb{S}$ is chosen as follows:

$$\bar{s}(x) = \begin{cases} s_0 & \text{if } x \in \{18, 22, 25\} = \mathbb{L}_0 \\ s_1 & \text{if } x \in \{16, 17, 21, 24, 27, 29, 30, 31\} = \mathbb{L}_1 \\ s_2 & \text{if } x \in \{19, 20, 23, 26, 28\} = \mathbb{L}_2 \end{cases}$$

where $L_0 = |\{18, 22, 25\}| = 3$, $L_1 = |\{16, 17, 21, 24, 27, 29, 30, 31\}| = 8$ and $L_2 = |\{19, 20, 23, 26, 28\}| = 5$. The frame encoding table $\mathbb{E}(x_i, s_i) = (x_{i+1}, b_i) \stackrel{def}{\equiv} \binom{x_{i+1}}{b_i}$ is illustrated in Table 4.

3 Properties of ANS

We investigate properties of a plain ANS. The goal is to identify potential cryptographic weaknesses but also strengths. The findings guide a design process of our compcrypt algorithm. In particular,

- we show that compression rate can be precisely calculated, when ANS states are modeled as a Markov chain. So we can determine equilibrium statistics of ANS states,

Table 4 ANS for 16 states ($p_{s_0} = 3/16$, $p_{s_1} = 8/16$, $p_{s_2} = 5/16$)

s_i/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/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	$\binom{20}{00}$	$\binom{20}{01}$	$\binom{20}{10}$	$\binom{20}{11}$	$\binom{23}{00}$	$\binom{23}{01}$	$\binom{23}{10}$	$\binom{23}{11}$

- the ability to compute probability distribution of ANS states is important as this allows us to determine quality of compression for a compcrypt algorithm, where ANS state statistics follows for instance, uniform distribution,
- there are also security implications, when an adversary is allowed to manipulate statistics of symbols and try to determine an ANS encoding table.

3.1 Compression rate of ANS

There is no proof that ANS achieves optimal entropy coding (symbols are encoded into binary strings with no redundancy). However, experiments show that the ANS entropy coding is very close to optimal. This sorry state is due to the complexity of the internal structure of ANS. In particular, the ANS symbol spread function can be chosen in many different ways. The main roadblock for computing the entropy of output/compressed bits is the difficulty of finding the state probabilities (we treat ANS as FSM). Typically, given a state x, it is argued that it occurs with the probability $\approx 1/x$ (see [13]). The approximation does not work well for a small numbers of states and even for larger numbers of states, it introduces a slight bias. The algorithm given below allows us to determine the probability distribution of the ANS states $x \in \{2^R, \ldots, 2^{R+1} - 1\}$ precisely. Recall that we use a shorthand $\mathbb{I} = \{2^R, \ldots, 2^{R+1} - 1\}$.
Let us make few observations about Algorithm 1, namely, it allows

- to evaluate the compression rate of ANS without experiments. This seems to be very crucial for ANS with a very large number of states (thousands),
- to choose a variant of ANS that provides the best compression rate. If the compression rate is equal to the symbol source entropy, it is possible to claim the optimal solution.

Example 1 Consider ANS given by Table 4. After running Algorithm 1, we have got the probabilities as shown below

Algorithm 1: Calculating probabilities of ANS states

Data: ANS represented by its encoding table $\mathbb{E}$ for symbols $s \in \mathbb{S}$ (rows) and states $x \in \mathbb{I}$ (columns)
Result: Probabilities $P(x)$, where $x \in \mathbb{I}$.
begin
  Assume that $\mathbb{E}$ describes a stationary Markov chain, where the probabilities before and after a single compression step are the same.
  for $x = 2^R, \ldots, 2^{R+1} - 1$ **do**
    create a linear relation $\mathcal{R}_x \equiv P(x) = \sum_{y \in \mathbb{I}} P(y) p_{s_y}$, where $x = \mathbb{E}(s_y, y)$;
  Ensemble a system of linear equations $\langle \mathcal{R}_x | x \in \mathbb{I} \rangle$ (note that $\sum_{x \in \mathbb{I}} P(x) = 1$).
  Solve the system using the Gaussian elimination algorithm.

x	16	17	18	19	20	21	22	23
$P(x)$	0.08	0.08	0.079	0.081	0.067	0.067	0.06	0.064

x	24	25	26	27	28	29	30	31
$P(x)$	0.062	0.048	0.05	0.055	0.05	0.052	0.051	0.051

It is easy to compute the average output bit length per symbol, which is 1.478. The symbol entropy is $H(\mathbb{S}) = 1.477$. Indeed, ANS is very close to the optimal one.

3.2 ANS compression losses

Ideally, ANS is expected to remove all redundancy and produce an uncorrelated random sequence of bits. In practice, however, there are circumstances, in which redundancy is unavoidable. Let us consider a few, which are relevant to our work.

- A source generates a short burst of symbols that may not follow the source statistics. A possible solution is to customize ANS for the current observed statistics. However, as a symbol frame is short and gain from compression low, a good option could be to avoid compression altogether.
- A symbol source statistics differs from the statistics used to design ANS. This increases redundancy of output bits.
- Any tampering with the internal structure of ANS (such as state jumps) may introduce extra redundancy.

Let us discuss the last two points. Feeding symbols with probability distribution $\mathcal{P} = \{p(s) | s \in \mathbb{S}\}$ to ANS designed for probability distribution $\mathcal{Q} = \{q(s) | s \in \mathbb{S}\}$ causes a compression loss. The Kullback-Leibler relative entropy can be used to approximate the loss as

$$\Delta H = \sum_{s\in\mathbb{S}} p_s \lg \frac{p_s}{q_s} \approx \frac{1}{\ln(4)} \sum_{s\in\mathbb{S}} \frac{(p_s - q_s)^2}{p_s}$$

Unfortunately, this approximation is very rough. The main reason is that the symbol probability distribution $\mathcal{Q}$ changes the probability distribution of ANS states as the corresponding Markov chain attains a new equilibrium. This fact is ignored in the approximation given above. To get a precise evaluation of compression loss, it is necessary first to compute state probabilities for a corresponding Markov chain and then calculate the average number of output bits per symbol.

Example 2 Let us take ANS given by Table 4 and assume that symbol source probability is $\mathcal{P} = \{1/4, 1/2, 1/4\}$ instead of expected $\mathcal{Q} = \{3/16, 1/2, 5/16\}$. The equilibrium probabilities for ANS states are given below.

x	16	17	18	19	20	21	22	23
$P(x)$	0.083	0.083	0.105	0.061	0.058	0.058	0.083	0.047

x	24	25	26	27	28	29	30	31
$P(x)$	0.065	0.061	0.042	0.063	0.042	0.052	0.047	0.047

The average output bit length per symbol is 1.52. The compression loss is $\Delta H \approx 0.04$ bits per symbol. Note that calculation for the Kullback-Leibler divergence gives $\Delta H = 0.02$. This illustrates the point that a change of symbol statistics causes a compression loss that is attributed to both a "wrong" symbol statistics and a change of ANS state probability distribution induced by it.

The work [12] proposes an ANS variant, where ANS states are chosen randomly using a cryptographically strong pseudorandom bit generator. This means that ANS state probability distribution is uniform. For ANS from Table 4 with state jumps, compression loss is $\Delta H \approx 0.03$ bits per symbol. The reader is referred to the papers [12,13] for a detailed discussion of ANS properties. An important property of ANS is that compressed bits are concatenated together in a single continuous stream. The information about how many bits need to concatenated are kept in the current ANS state. This forces a potential adversary to make guesses about how to partition a long stream into varying length substrings corresponding to the processing of individual symbols. As ANS is, in fact, a finite state machine (FSM) with a finite number of states, it has a cyclic nature that allows an adversary to inject/remove output bits without detection (see [12] for details).

3.3 Statistical attacks against ANS

An interesting ANS property, not investigated in [12], is its vulnerability to statistical attacks. The property informs design options for our compcrypt algorithm. We show that there are no security benefits by making encoding table secret and controlled by a cryptographic key. In particular, we demonstrate that ANS with secret encoding table leaks information about symbol spread function or alternatively, about encoding table entries.

Assume that an adversary $\mathcal{A}$ has an access to a communication channel so it can see the binary stream. Additionally, suppose $\mathcal{A}$ knows a symbol source statistics and an algorithm used to construct ANS. This is a typical ciphertext-only adversary. ANS seems to be immune to it. However, in some circumstances, $\mathcal{A}$ may be more powerful and can interfere with symbol source statistics. This is to say that we deal with a adaptive-statistics adversary, who

- Sees an output binary stream and can calculate its length,
- can force symbol source statistics to follow an arbitrary probability distribution including probabilities, where a single symbol occurs with probability 1,
- Knows the number of processed symbols,
- is familiar with a design algorithm for ANS (but does not know its encoding table).

The goal of $\mathcal{A}$ is to recover the ANS encoding table or equivalently its symbol spread function. The idea behind the attack is an observation that we can use Markov chains (see Algorithm 1) to calculate state probabilities and consequently, compression rate (measured by bits per symbol). The calculated compression rate is used as a litmus test. A trivial statistical attack is an exhaustive search through all possible symbol spread functions. Our adversary $\mathcal{A}$ is able to identify the correct instance of ANS by comparing an observed compression rate with a calculated using an appropriate Markov chain. The complexity of the exhaustive attack is determined by the number of all ANS instances that need to be tested, which is

$$\begin{aligned}\frac{L!}{\prod_{s\in\mathbb{S}} L_s!} &= \binom{L}{L_{s_1}}\binom{L-L_{s_1}}{L_{s_2}}\cdots\binom{L-\sum_{i=1}^{n-2} L_i}{L_{s_{n-1}}} \\ &\approx 2^{LH(\{L_s/L\})} \text{ for } H(\{p_s\}) \\ &= -\sum_s p_s \lg(p_s) \end{aligned} \quad (1)$$

where $n = |\mathbb{S}|$ and L_s is the number of states assigned by the symbol spread function for the symbol s [14].

The exhaustive search attack can be substantially improved by applying the divide-and-conquer principle. Note that $\mathcal{A}$ can select a symbol source distribution in arbi-

trary way. The most promising strategy for $\mathcal{A}$ seems to be to choose a pair of symbols with nonzero probabilities making the others equal to zero. This excludes states associated with missing symbols from symbol processing. To illustrate the point, consider ANS given by Table 4. If $\mathcal{A}$ chooses $p_{s_0} = \alpha$, $p_{s_1} = 0$ and $p_{s_2} = 1 - \alpha$, then states $\{16, 17, 21, 24, 26, 27, 29, 30, 31\}$ occur with zero probability. Table 4 becomes

s_i/x_i	18	19	20	22	23	25	26	28
s_0	$\binom{22}{10}$	$\binom{22}{11}$	$\binom{25}{00}$	$\binom{25}{10}$	$\binom{25}{11}$	$\binom{18}{001}$	$\binom{18}{010}$	$\binom{18}{100}$
s_2	$\binom{28}{0}$	$\binom{28}{1}$	$\binom{19}{00}$	$\binom{19}{10}$	$\binom{19}{11}$	$\binom{20}{01}$	$\binom{20}{10}$	$\binom{23}{00}$

In other words, the encoding table is "reduced" to two rows only and other rows play no role in compression (also columns of inactive states can be removed). We can start from a pair of symbols with the smallest probabilities as they involve the smallest number of ANS states. The flowchart of the attack is given by Algorithm 2. Note that we denote $\mathbb{L}_i = \{x|\bar{s}(x) = s_i\}$ a subset of states assigned to the symbol $s_i \in \mathbb{S}$ by the symbol spread function $\bar{s}(x)$; $i = 1, \ldots, n$.

Algorithm 2: Divide-and-Conquer Statistical Attack against ANS

Data: An instance of ANS with a adaptive-statistics adversary and the symbol source probability distribution $\mathcal{S}$.
Result: Encoding table or symbol spread function used by ANS.
begin
  Choose two rows s_1 and s_2 of encoding table whose symbol probabilities are the smallest;
  Create a reduced encoding table for the pair of symbols;
  Run exhaustive search attack for the reduced encoding table;
  Store $\mathbb{L}_1 = \{x|\bar{s}(x) = s_1\}$ and $\mathbb{L}_2 = \{x|\bar{s}(x) = s_2\}$;
  for $i = 3, \ldots, n-1$ **do**
    Run exhaustive search attack for reduced encoding table for s_1 and s_i;
    Store $\mathbb{L}_i = \{x|\bar{s}(x) = s_i\}$;
  Return the symbol spread function $\bar{s}(x)$;

The workload needed to calculate $\mathbb{L}_1$ and $\mathbb{L}_2$ takes generation of $\binom{L}{L_{s_1}}\binom{L-L_{s_1}}{L_{s_2}}$ variants of (reduced) ANS. The j-th step in the "for" loop of Algorithm 2 costs $\binom{L-\sum_{i=1}^{j-1} L_i}{L_{s_j}}$. The complexity of the algorithm is dominated by

$$\binom{L}{L_{s_1}}\binom{L-L_{s_1}}{L_{s_2}} + \sum_{j=3}^{n-1}\binom{L-\sum_{i=1}^{j-1} L_i}{L_{s_j}}. \tag{2}$$

Note a significant complexity reduction of the exhaustive attack given by Eq. (1) compared to the divide-and-conquer one described by Eq. (2).

So far we have not discussed the case, where an adversary $\mathcal{A}$ is forcing a single symbol to occur with probability one. The Markov chain for state probabilities degenerates to few disjoint cycles that happen with probability 1. If $\mathcal{A}$ can switch for a moment to a different symbol, then it can reset the internal state and by repeating the experiment with the same symbol, it can discover the next cycle. By continuing this procedure, $\mathcal{A}$ can identify all cycles. This knowledge can be used to

- speed up the divide-and-conquer attack by considering instances of reduced ANS that contain the identified cycles only and
- significantly improve probability of guessing the output bitstream (if $\mathcal{A}$ does not have access to it).

4 Compression and encryption

Security-sensitive applications (such as video streaming) can protect compressed stream by using standard encryption algorithms such AES [15]. In this case, a simple solution is to first compress symbols and then encrypt. It, however, ignores the fact that compression itself provides some degree of security. To launch an attack, an adversary needs to guess a split of output binary stream into symbol encodings (with different lengths). Thus, one would expect that using full-blown encryption is overkill. In other words, the research problem in hand is how to weaken encryption (by reducing a number of rounds for example) so the concatenation of ANS and a weaken encryption maintains the same security level while improving efficiency.

A different approach for compression and encryption is taken by Duda and Niemiec in [11]. Encryption is incorporated by a secret selection of symbol spread function. The selection is then communicated to a receiver as an encrypted decoding table (using a standard AES). For this solution, compression efficiency is as good as in the original ANS. Also, security against a ciphertext-only adversary is maintained. However, as shown in [12], bitstream is vulnerable to integrity attacks. Also encryption of decoding tables needs to be authenticated and time-stamped to protect it against replay attacks.

4.1 Design principles

The currently published solutions for ANS-based compcrypt (see [11,12]) suffer from the following weaknesses:

- Integrity of encryption/compression is supported by sending an encrypted final ANS state. The bitstream is considered to be authentic if the encrypted final state equals a final state of the decoder. Even for ANS

with $2048 = 2^{11}$ states, this guarantees 11-bit security only, which is much too low for transmission of highly sensitive data. This also means that an adversary can inject/delete arbitrary segments of the bitstream and this action escapes detection with probability 2^{-11}.
- Related weakness of ANS is its cyclic nature (as shown in [12]). Cycles can be detected by scanning bitstream for repetitions of the same pattern. Once such a cycle is correctly identified, an adversary can remove or inject it with an arbitrary number of times without detection.
- As shown in Sect. 3.3, ANS is inherently susceptible to statistical attacks. As the designer of ANS has freedom to choose a symbol spread function in many ways, this allows her to end up with a design, whose internal states occur with a unique probability distribution. As a result, ANS produces binary encoding with an (almost) unique probability distribution. The probability distribution can be seen as a unique fingerprint that allows to identify the symbol spread function and consequently encoding table (see Algorithm 2) after observing a long enough bitstream.

Let us formulate a collection of principles that are going to guide us during the design of our compcrypt algorithm. Our compcrypt algorithm should

- be (almost) as efficient as a plain ANS (without encryption). A slight efficiency loss, however, is inevitable;
- preserve compression rate of a plain ANS. Again, to guarantee authenticity and integrity of communication, additional data needs to be appended to bitstream;
- guarantee at least 128-bit security. This security level applies to chosen-plaintext attacks;
- detect any interference with encrypted bitstream with probability $(1–2^{-128})$. This is done by appending a 128-bit authentication tag.
- be immune against statistical attacks.

4.2 ANS with randomized states

ANS is inherently susceptible to statistical attacks due to uneven probability distribution of states as $P(x) \approx 1/x$. An adaptive-statistics adversary $\mathcal{A}$ can further tamper with the distribution by modification of symbol probability distribution. Consequently, $\mathcal{A}$ can switch off some states completely making other states more probable. To mitigate this weakness, we may force states to occur randomly and uniformly. So instead of a normal state flow

$$\cdots \xrightarrow{s_{i-1}} \boxed{x_i - 1} \xrightarrow{s_i} \boxed{x_i} \xrightarrow{s_{i+1}} \boxed{x_i + 1} \xrightarrow{s_{i+2}} \cdots$$

we select the next state randomly and uniformly using a random bit generator (RBG) as shown below

$$\cdots \xrightarrow{s_{i-1}} \boxed{x_i - 1} \xrightarrow{s_i} \boxed{x_i \longleftarrow x_i \oplus \mathrm{RBG}} \xrightarrow{s_{i+1}} \boxed{x_i + 1} \xrightarrow{s_{i+2}} \cdots$$

where RBG strings need be replicated at the receiver side. Let us see what is an impact of such state evolution on probability of guessing an output bit stream when we know its length. In our discussion, we need the following lemma.

Lemma 1 *Given a plain ANS as described in Sect. 2. Then for a symbol $s \in \mathbb{S}$, ANS generates*

- *either empty-bit or 1-bit encodings and again the encoding table row for the symbol s contains equal number of zeros and ones if $L_s > 2^{R-1}$,*
- *k_s-bit encodings and the encoding table row for the symbol s contains equal number of zeros and ones if $L_s = 2^{R-k_s}$, i.e., the symbol probability $p_s = (1/2)^{k_s}$, where k_s is a positive integer (natural number),*
- *either k_s-bit or $(k_s + 1)$-bit encodings and the encoding table row for the symbol s 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.*

The lemma is a slight generalization of the lemma from [12] and covers all symbols that occur with probabilities that are natural powers of 1/2. The reader is referred to [12] for a proof.

Example 3 Consider the ANS instance given by Table 4. The row for s_1 illustrates the case when $L_s = 2^{R-1}$. All encodings are 1-bit long and zeros and ones occur equal number of times. The row for s_0 represent a case when $L_s < 2^{R-1}$. Encodings are either 2 or 3-bits long. 3-bit encodings run through all 3-bit string once as the multiplier $(2^{R-k_s} - L_s) = 2^{4-2} - 3 = 1$. 2-bit encodings are repeated twice as their multiplier $(2L_s - 2^{R-k_s}) = 6 - 4 = 2$.

4.2.1 ANS with fully randomized states

Corollary 1 *Given a plain ANS as described in Sect. 2. Assume that the ANS algorithm chooses next states uniformly at random (i.e., $x_i \leftarrow x_i \oplus RBG$). Suppose further that an adversary $\mathcal{A}$ inputs a sequence of ℓ symbols s, i.e., $\underbrace{(s, s, \ldots, s)}_{\ell}$ and $\mathcal{A}$ can observe the length k of the resulting bitstream (in bits), then probability of guessing the bitstream by $\mathcal{A}$ is 2^{-k}.*

Proof Knowing the length k, s and parameters of ANS, $\mathcal{A}$ may first try to guess lengths of encodings (or sizes of encoding windows). For each guess, any single window of the length k_s can take 2^{k_s} possible strings as Lemma 1 asserts.

This scenario occurs consistently for any encoding length guessing and the conclusion follows. □

Example 4 Take ANS from Table 4. Our adversary inputs a two symbols (s_0, s_0) and knows that ANS has produced a 5-bit string. $\mathcal{A}$ deals with two options, when ANS has generated first 2-bit encoding followed by 3-bit one or vice versa. For the first option, $\mathcal{A}$ has 2^2 possibilities for 2-bit encoding and 2^3 possibilities for 3-bit encoding. For the second option, the number is the same. Both cases generate all 2^5 possible binary strings.

4.2.2 ANS with partially randomized states

An undesired side effect of uniformity of ANS states is a loss of compression rate. The reader can easily note that smaller states are producing shorter encodings, while bigger states – longer ones (see Table 4 as an example). As probability of an ANS state x can be approximated by $\approx 1/x$, a plain ANS favors smaller states over bigger ones giving a better compression rate. Let us explore an option, where states are XOR-ed with a shorter PRBG sequence. Assume that PRBG produces a sequence of u bits, where $u < R$. A state x can be equivalently represented by a pair $(\lfloor x/2^u \rfloor, x \bmod 2^u)$. During compression, ANS modifies the state according to the relation given below

$$x \leftarrow \left(\lfloor x/2^u \rfloor, \left(x \mod 2^u\right) \oplus PRBG\right)$$

The new state keeps the same most significant bits, while the u least significant bits cover all 2^u binary strings. This is illustrated as

$$x \xleftarrow{PRBG} \begin{cases} (\lfloor x/2^u \rfloor, & 0) \\ \vdots & \vdots \\ (\lfloor x/2^u \rfloor, & 2^u - 1) \end{cases} \tag{3}$$

Given a symbol s that is being processed. According to Lemma 1, states produce either k_s or (k_s+1)-bit long encodings. An encoding is extracted from a state x chosen by PRBG and contains either k_s or $(k_s + 1)$ least significant bits. Consider the following two cases.

1. $u = k_s+1$, an encoding is chosen at random from the full range of $2^u = 2^{k_s+1}$ possibilities. More precisely, if $x \in \{2^R, \ldots, 2^{k_s+1}L_s - 1\}$, then encodings are k_s-bit long and are repeated twice in the collection of states in Eq. (3). All k_s-bit long encodings happen uniformly at random. For $x \in \{2^{k_s+1}L_s, \ldots, 2^{R+1} - 1\}$, their encodings are $(k_s + 1)$-bit long and occur ones only in Eq. (3). This also means that all encodings are equally probable.
2. $u = k_s$, when $x \in \{2^R, \ldots, 2^{k_s+1}L_s - 1\}$, then all encodings are possible as they happen exactly once in the list given in Eq. (3). This no longer true for $x \in \{2^{k_s+1}L_s, \ldots, 2^{R+1} - 1\}$. To see this, note that the number of all candidate states is $2^u = 2^{k_s}$ or in other words k_s bits are random but the (k_s+1)-th bit is inherited from the old state x. Consequently, the bit is fixed for all possible encodings.

Example 5 Consider our toy ANS from Table 4. Assume that ANS has reached the state $x = 25$ and PRBG generate 2-bit strings ($u = 2$). Then ANS chooses at random a new state from the set {24, 25, 26, 27} or

$$x \xleftarrow{PRBG} \begin{cases} (110,\ 00) \\ (110,\ 01) \\ (110,\ 10) \\ (110,\ 11) \end{cases}$$

For a symbol s_0, there are four equally probable encodings (000, 001, 010, 011). The most significant bit is fixed. Other two are random.

Corollary 2 *Assume that an ANS algorithm chooses next states uniformly at random* $x \leftarrow (\lfloor x/2^u \rfloor, (x \bmod 2^u) \oplus PRBG)$ *that randomly chooses u least significant bits of the new state. Then*

- *all encodings are random (chosen by PRBG) if* $u = \max_{s\in\mathbb{S}}(k_s + 1)$*;*
- *encodings are random for symbols s, for which* $u \geq (k_s + 1)$*;*
- *encodings include u random bits for symbols s, for which* $u < (k_s + 1)$*;*

The above discussion leads to the following design hints.

- ANS with uniformly random states does not leak any information about input symbols. From the $\mathcal{A}$ point of view, ANS generates random sequence of bits (assuming $\mathcal{A}$ cannot break RBG). This means that ANS with RBG provides a high level of confidentiality. Thus encryption algorithm does not need to be very strong (a single permutation layer could be enough).
- Compression loss caused by flattening state probabilities can be mitigated by using shorter PRBG strings. Using PRBG with u bits, ANS randomly updates a state by selecting one from 2^u consecutive states.
- There is, however, "a sting in the tail". ANS with RBG is defenceless against integrity attacks such as deletion or injection. In other words, we need strong protection against such attacks.
- Implementation of RBG is crucial. It seems that a simple linear feedback shift register (LFSR) controlled by a secret cryptographic key may be enough. In case this is

not enough, we can apply nonlinear feedback shift register (NFSR) or cryptographically strong PRBG.

4.3 MonkeyDuplex

The sponge structure has been applied by Bertoni et al. [16] in the KECCAK cryptographic hashing. The sponge structure is illustrated in Fig. 1 and consists of concatenation of a permutation P. The name comes from the fact that hashing works in two phases: absorption and squeezing. In the first phase, message blocks are injected block by block into the bitrate blocks. In the second one, a message digest is squeezed out from bitrate blocks. Note that the capacity blocks are left secret throughout hashing process. The round permutation KECCAK–$f[b]$ includes seven members, where b determines the number of bits in sponge state and $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. There are two sponge parameters: bitrate r and capacity c, where $b = r + c$.

Bertoni et al. [17] have introduced a duplex construction, which provides a framework for authenticated encryption. As shown in Fig. 1, we use a duplex variant called MonkeyDuplex. Consecutive bitrate blocks XOR bitstream blocks $\mathbf{b}_i$ producing ciphertext blocks $\mathbf{c}_i$. A permutation P is the most crucial building block of sponge. It is shown in [16] that security level can be proven under the assumption that the underlying permutation P does not have any "exploitable" properties. In our design, we use the well-established and thoroughly analyzed KECCAK–$f[1600]$ permutation. The permutation is used in the SHA-3 hashing standard and in authenticated ciphers such as Keyak and Ketje.

Given a fixed state size, a relation between bitrate and capacity gives a tradeoff between speed and security. A higher bitrate results in a faster construction but at the expense of its security and vice versa. MonkeyDuplex allows efficient encryption, especially, when dealing with a very long stream of plaintext blocks. The permutation P operates on the large states of 1600 bits, which boosts efficiency. Note that only initialization requires a full number of permutation iterations. A number of iterations between two consecutive plaintext blocks injections can be reduced. As data compression handles very long bitstreams, MonkeyDuplex seems to be a very attractive option.

Encryption security is usually evaluated using different attacks. Two main ones are chosen-plaintext and integrity attacks. In a chosen-plaintext attack (CPA), an adversary $\mathcal{A}$ has access to an encryption oracle as a black box. $\mathcal{A}$ is able to choose arbitrary plaintext blocks and can observe ciphertext blocks generated by the oracle. The goal of $\mathcal{A}$ is recovery of a secret cryptographic key. In an integrity attack, $\mathcal{A}$ has the same access to the oracle but its goal is different. $\mathcal{A}$ wishes to modify observed ciphertext blocks in such a way that they are accepted on the receiver side.

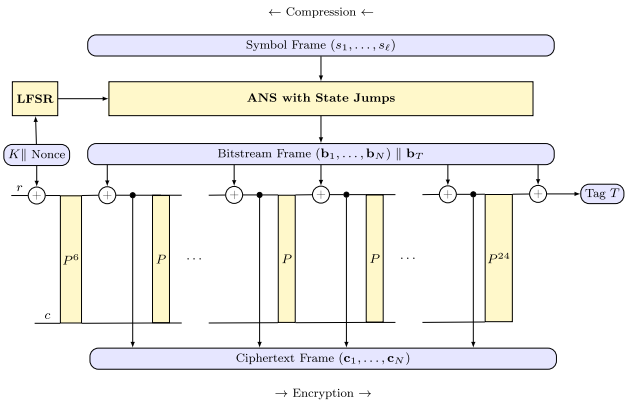

Fig. 1 Compcrypt based on ANS and sponge, where $r = 512$ bits, $c = 1088$ bits and size of sponge state is $b = 1600$ bits

5 Compcrypt algorithm

The overall data flow for our proposed compcrypt algorithm is shown in Fig. 1. The two main components are ANS with state jumps and a MonkeyDuplex sponge.

5.1 Description of compcrypt

A data flow during encryption is illustrated in Fig. 1. The algorithm steps are shown in Algorithm 3. Let us discuss its steps in more detail.

Algorithm 3: Compcrypt Encryption

(h) **Data**: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_\ell)$, a 128-bit secret key K and a 128-bit random nonce α.
Result: A ciphertext frame $\mathcal{C} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$ of compressed bit stream together with a 128-bit tag T, where $\mathbf{c}_i$ is a 512-bit long block; $i = 1, \ldots, N$.
begin
(1) Upload the symbol frame and compute symbol probabilities $\{p_s | s \in \mathbb{S}\}$;
(2) Initialize a sponge for $K \| \alpha$ by running $f_{start} = P^6$, where the number of iterations $n_{start} \geq \lceil \frac{R \cdot \ell}{512} \rceil$;
(3) Design ANS instance for the symbol statistics;
(4) Compress $\mathcal{S}$ in the reverse order. This creates a bitstream frame;
(5) Split the bitstream frame into 512-bit blocks $(\mathbf{b}_1, \ldots, \mathbf{b}_N)$, where the last block is padded to the full length;
(6) Encrypt the decoding table and the bitstream frame $(\mathbf{b}_1, \ldots, \mathbf{b}_N)$ into $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$ using the sponge;
(7) Generate the 128-bit tag T. First, extract 128-bit string $\mathbf{b}_T$ from ANS states when their jumps are controlled by LFSR only (no symbol compression) and then, create the tag $T = \mathbf{b}_T \oplus P(\mathbf{c}_N)|_{128}$, where $P(\mathbf{c}_N)|_{128}$ denotes the top 128 bits of the sponge state;
(8) Send the ciphertext frame and the tag T to the receiver.

Step 1 A symbol frame is uploaded into a (LIFO) stack and its statistics is computed. The probabilities $\{p_s | s \in \mathbb{S}\}$ are needed for constructing ANS encoding and decoding tables.
Step 2 The 1600-bit sponge is initiated by the secret key K and the nonce α and can include associated data (such an initial vector, time, sequence number, etc.). The initial sponge state is translated by f_{start}. Note that for short symbol frames, $n_{start} \geq 6$.
Step 3 We assume that ANS is designed for $R = 11$ so it has 2048 states. A spread symbol function can be randomly chosen or deterministic. Both encoding and decoding tables are created.
Step 4 Compression starts from reading out the first symbol s_ℓ. It continues until the last symbol s_1 is taken from the stack. After each symbol, an ANS forces a state jump that is controlled by pseudorandom bits (see Sect. 4.2). Note that the bits should be used in the reverse order, i.e., last generated bits must be used first. The compressed bitstream includes a final ANS state x_F together with the numbers ℓ, n, where n is the total length of bitstream frame.
Step 5 The bitstream frame is split into 512-bit blocks $(\mathbf{b}_1, \ldots, \mathbf{b}_N)$. The last block N can be padded by a string of constants to the full length.
Step 6 The bitstream frame $(\mathbf{b}_1, \ldots, \mathbf{b}_N)$ is encrypted into its ciphertext $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$ using the sponge. Note that the sponge applies $f_{step} = P$ between two consecutive blocks.
Step 7 A 128-bit authentication tag T is obtained by XOR-ing 128 bits of the final state of sponge with 128 bits least significant bits extracted from ANS states, where jumps are forced by LFSR.
Step 8 Finally, the ciphertext frame and the tag T are sent to the receiver side.

At the receiver side, decryption starts from the initialization of the sponge and LFSR. Next, the sponge decrypts ciphertext frame and verifies validity of the received tag T.

6 Compcrypt security analysis

According to the well-known Kerckhoff's principle, an adversary $\mathcal{A}$ knows details of the compcrypt algorithm. The only unknown part is a cryptographic key K. In our adversarial model, we assume that $\mathcal{A}$ is a chosen-plaintext adversary. Table 5 takes a closer look at the adversary ability and its goals. To avoid trivial attacks that intend to simplify ANS or even bypass it, we make the following caveat.

Important Caveat: From now on, we assume that our compcrypt algorithm terminates if symbol frame statistics does not cover the full range of 256 bytes. In other words, the algorithm runs only if ANS is able to build a full-size encoding table.

Table 6 describes the CPA security game. The challenger $\mathcal{CH}$ selects its secret key K and a public *Nonce*. The adversary $\mathcal{A}$ is free to select a symbol frame $\mathcal{S}$ provided the caveat holds. Upon receiving $\mathcal{S}$, the challenger compresses and encrypts it into $\mathcal{C}_0$. Next, it prepares a fake $\mathcal{C}_1$ by randomly selecting all bits apart from padding bits, which are preserved as in $\mathcal{C}_0$. Note that the lengths of $\mathcal{C}_0$ and $\mathcal{C}_1$ are the same. The challenger selects at random b and sends $\mathcal{C}_b$ together with *Nonce* to the adversary. The adversary wins the game if it can identify whether the cryptogram $\mathcal{C}_b$ corresponds to $\mathcal{S}$ or is random. The theorem given below shows that the adversary wins the game with a negligible advantage.

Theorem 1 (CPA Security) *Given a compcrypt as described by Algorithm 3 and Fig. 1. Assume that ANS states are controlled by LFSR, whose probability distribution is uniform and whose cycle is maximum, i.e.,* $2^L - 1$, *where* L *is the number of bits of the LFSR state. We assume that* $L \geq 256$. *Suppose also that* $P : \{0,1\}^{r+c} \rightarrow \{0,1\}^{r+c}$ *is a sponge random permutation that is used between two consecutive bitstream blocks. Then an advantage of the adversary playing the game* G^{CPA} *is negligible, i.e.,*

$$\begin{aligned} Adv_{\mathcal{A}}^{CPA} &= P(\mathcal{A}\ wins | b = 0) - P(\mathcal{A}\ wins | b = 1) \\ &\leq \max\left(\frac{\varepsilon}{2}, \frac{\gamma}{2 \cdot 2^{128}}\right), \end{aligned}$$

where $\varepsilon = \frac{\gamma(2d + \ell R)}{2^L - 1}$, γ *is the number of key guesses by* $\mathcal{A}$, d *is the number bits generated by LFSR per a key guess,* ℓ *is the number of symbols in the frame* $\mathcal{S}$ *and* R *is the ANS parameter.*

Proof According to our assumptions, $\mathcal{A}$ has a full control over a symbol frame (assuming the caveat holds). It can prepare a long frame, which contains bursts of the same symbols but also can mix them as it wishes. No matter, what is its strategy, Lemma 1 and Corollary 1 assert us that each block $\mathbf{b_i}$; $i = 1, \ldots, N$ is random as long as LFSR produces bits with uniform probability distribution. In other words, the sponge absorbs $\mathbf{b_i}$ and squeezes out $\mathbf{c_i}$, where

$$\mathbf{c}_i = \mathbf{b}_i \oplus \mathbf{SP}_{i-1}^{(r)},$$

where $\mathbf{SP}_{i-1}^{(r)}$ are the bitrate bits of the sponge state before absorption of $\mathbf{b_i}$. The block $\mathbf{b_i}$ masks perfectly the r bits coming from the sponge (identical to OTP). This is to say that $\mathcal{A}$ has no chance to tell apart $\mathbf{c}_i$ from a truly random sequence. However, $\mathcal{A}$ knows that $\mathcal{CH}$ has applied ℓR bits

Table 5 Compcrypt adversarial model

Attack	$\mathcal{A}$'s *Knowledge/Ability*	$\mathcal{A}$'s *Goals*
Chosen-Plaintext	Symbol statistics $\{p_s : s \in \mathbb{S}\}$ ANS encoding table: ℓ, n and state x_F Sponge construction: P, f_{start}, f_{step}, f_F $\mathcal{A}$ can run compcrypt for its chosen symbol frame $(s_1, s_2, \ldots, s_\ell)$ and observe their cryptogram blocks $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$	Winning the game G^{CPA}
Integrity	As above + $\mathcal{A}$ can tamper with ciphertext blocks and tag T by injecting/deleting/modifying their parts	Winning the game G^{INT}

Table 6 Game G^{CPA}

Challenger $\mathcal{CH}$		Adversary $\mathcal{A}$
Selects $K, Nonce \xleftarrow{\$} \{0,1\}^{128}$;		
		Chooses a symbol frame
Computes statistics p_s;	$\longleftarrow$	$\mathcal{S} = (s_1, s_2, \ldots, s_\ell)$;
Designs ANS for $\mathcal{S}$;		
Initializes LFSR and sponge for $K \| Nonce$;		
For $\mathcal{S}$, computes $\mathcal{C}_0 = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N, T)$;		
Creates a random string $\mathcal{C}_1 = (\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, \ldots, \tilde{\mathbf{c}}_N, \tilde{T})$;		
Tosses a coin $b \xleftarrow{\$} \{0,1\}$		
Sends off $(\mathcal{C}_b, Nonce)$;	$\longrightarrow$	Determines b';
		Wins if $b' = b$;

of LFSR, where the initial state is $(K \| Nonce)$. It may try to guess the key. Let a $\mathcal{A}$ guess is K', then after loading $K' \| Nonce$ into LFSR, it can generate d bits. It succeeds if d bits intersect with the ℓR bits generated by $\mathcal{CH}$ and the probability of success is $\frac{2d+\ell R}{2^L-1}$. $\mathcal{A}$ can repeat its guesses γ times and its success probability grows up to $\varepsilon = \frac{\gamma(2d+\ell R)}{2^L-1}$, when bits generated by LFSR for the $\mathcal{A}$ key guesses do not intersect. It is reasonable to assume that if $\mathcal{A}$ succeeds in its guessing, then it will be able to identify the bit b and win the game G^{CPA}.

Let us consider $P(\mathcal{A}\text{ wins}|b = 1)$. In this case, cryptograms and a tag are random. This means that it does not matter what strategy $\mathcal{A}$ chooses, it wins with probability $1/2$. To compute $P(\mathcal{A}\text{ wins}|b = 0)$, observe that $\mathcal{A}$ fails with the probability $(1-\varepsilon)$ and succeeds with the probability ε. As for the previous case, when $\mathcal{A}$ fails with its guessing, it wins the game with the probability $1/2$. Otherwise, it wins the game. This means that $P(\mathcal{A}\text{ wins}|b = 0) = 1/2(1 - \varepsilon) + 1 \cdot \varepsilon = 1/2 + \varepsilon/2$. The advantage $Adv_{\mathcal{A}}^{CPA} = \varepsilon/2$.

As $\mathcal{A}$ knows parameters of the compcrypt, it can evaluate its chances or more precisely, it can calculate ε. If $\varepsilon < \frac{1}{2^{128}}$, then a better strategy is to guess the key K. This concludes our proof. □

The following comments are relevant to the above proof.

- The first part of the proof applies a variant of the well-known square-root attack (see [18]), whose complexity depends on the length L of a LFSR state. It is easy to see that if $L = 128$ bits (and there is no *Nonce*), then $\mathcal{A}$ can find a string of the length $d = 2^{64}$ that intersects with ℓR bits after $\gamma = 2^{64}$ key guesses. However in our case $L = 256$ and the attack has a complexity $O(2^{128})$, which is comparable to the exhaustive search of the key space. The attack becomes much less effective than the exhaustive search of the key space if $L > 256$. The best strategy for $\mathcal{A}$ in this case is guessing the key.
- Note that even after finding a string that intersects with ℓR bits generated by $\mathcal{CH}$, $\mathcal{A}$ faces a difficult task to identify it. The main obstacle is the sponge, which is likely to

Table 7 Game G^{INT}

Challenger $\mathcal{CH}$		Adversary $\mathcal{A}$
Selects $K, Nonce \xleftarrow{\$} \{0,1\}^{128}$;		
		Chooses a symbol frame
Computes statistics p_s;	$\longleftarrow$	$\mathcal{S} = (s_1, s_2, \ldots, s_\ell)$;
Designs ANS for $\mathcal{S}$;		
Initializes LFSR and sponge for $K \| Nonce$;		
For $\mathcal{S}$, computes $\mathcal{C} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N, T)$;		
Sends $\mathcal{C}$	$\longrightarrow$	
		Creates $\tilde{\mathcal{C}} = (\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, \ldots, \tilde{\mathbf{c}}_N, \tilde{T})$
Recoverssymbol frame $\tilde{\mathcal{S}}$ and tag T';	$\longleftarrow$	Sends $\tilde{\mathcal{C}}$
Accepts $\tilde{\mathcal{C}}$ if $T' = \tilde{T}$ otherwise rejects;		

significantly reduce correlation between bits generated by LFSR and bits of cryptograms.

Table 7 shows our integrity game G^{INT}. $\mathcal{A}$ is allowed to select a symbol frame $\mathcal{S}$ provided the caveat holds. The challenger calculates a cryptogram $\mathcal{C} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N, T)$, where T is an authentication tag. Next $\mathcal{CH}$ sends $\mathcal{C}$ to the adversary. $\mathcal{A}$ modifies it somehow and gets $\tilde{\mathcal{C}} = (\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, \ldots, \tilde{\mathbf{c}}_N, \tilde{T})$. It wins the game if $\mathcal{CH}$ accepts $\tilde{\mathcal{C}}$. This happens if $\mathcal{CH}$ computes a tag T' that matches $\tilde{T}$.

Theorem 2 *(*Integrity*) Given a compcrypt as described by Algorithm 3 and Fig. 1 and assumptions as for Theorem 1. Then the adversary $\mathcal{A}$ wins the game G^{INT} with a negligible probability or more precisely*

$$Adv_{\mathcal{A}}^{\mathrm{INT}} = P(\mathcal{A}\, wins) \leq 2^{-r},$$

where r is the length of bitrate bits of the KECCAK-*f sponge.*

Proof There are three possible cases.

(1) $\mathcal{A}$ adds one or more fake cryptogram blocks. To make $\mathcal{CH}$ accept, $\mathcal{A}$ has to determine bits generated by LFSR. As we have argued in Theorem 1 this task is more difficult than guessing the key K if $L \geq 256$. This means that $\mathcal{A}$ wins with probability 2^{-128}.
(2) $\mathcal{A}$ removes one or more cryptogram blocks. This time $\mathcal{A}$ has to recalculate tag using 24 iterations of KECCAK -f permutation P. Note that the capacity $c = 1088$ bits so $\mathcal{A}$ needs to guess 1088 bits. Much better strategy is to guess the key or the tag. In both cases, $\mathcal{A}$ wins with probability 2^{-128}.
(3) $\mathcal{A}$ keeps the number of cryptogram blocks intact and slightly modifies bits in the last block $\mathbf{c}_N = (c_{1,N}, \ldots, c_{512,N})$. To fix our attention, we assume that $\mathcal{A}$ flips the bit $c_{1,N}$ and the other bits are left unchanged. Denote $\mathbf{y} = (y_1, \ldots, y_{128})$ as the output of P^{24} that is used to derive the tag. Then the tag $T = \mathbf{b}_T \oplus \mathbf{y}$. $\mathcal{A}$ does not know $\mathbf{b}_T$ but is sure that it is fixed. To get a correct tag $\tilde{T}$, $\mathcal{A}$ has to modify a bit t_i of the tag T if and only if

$$\delta = y_i(c_{1,N}, \ldots, c_{512,N}, \mathbf{x}) \oplus y_i(\bar{c}_{1,N}, c_{2,N}, \ldots, c_{512,N}, \mathbf{x}) = 1$$

for $i = 1, \ldots 128$. There are two possible approaches. First, $\mathcal{A}$ may try to obtain a short expression for y_i. A close look at the KECCAK-f permutation [16] reveals that the degree of P^{24} is 2^{24} and it involves 1088 binary unknowns. Even if $\mathcal{A}$ finds a short expression for δ, the design of the KECCAK-f permutation P indicates that is very likely that it produces $\delta = 1$ for half of the possible 2^{1088} cases. In the second approach, $\mathcal{A}$ runs experiments for subset of $\mathbf{x}$ vectors to evaluate how many times $\delta = 1$. Both approaches are much worse than random selection of the key or the tag.

□

We make the following remarks.

- For the game G^{INT}, we assume that the challenger knows the decoding table. This is a realistic as it has produced the cryptogram. In real applications, however, a receiver needs to get it from a sender. A typical solution is to inject it into the sponge just after initialization. This allows the receiver to obtain an authenticated decoding table.
- The sponge may also include the number ℓ of symbols in the frame and the number n of compressed bits. This

makes attacks against integrity more difficult as $\mathcal{A}$ cannot change the length of the cryptogram.
- As recommended by the KECCAK designers, we apply 24 iterations of P. It is likely that the number of 24 iterations of P for tag generation can be reduced without reducing security. However, this needs a further investigation that is beyond the scope of this paper.

6.1 Compcrypt resistance against classical attacks

There is a wide range of attacks against encryption algorithms. It includes the linear [19], differential [20], algebraic [21], cube [22] and rotational [23] cryptanalyses. A distinct feature of ANS is its variable-length encoding. Before applying any standard cryptanalytic tool, an adversary $\mathcal{A}$ needs to identify internal variables of compcrypt. According to Lemma 1, all symbols s are assigned either k_s or $(k_s + 1)$-bit encoding. Thus, for each symbol, $\mathcal{A}$ needs to guess the lengths of their encodings. Note that an ANS state is chosen uniformly at (pseudo)random after each symbol.

Consider an example, when $\mathcal{A}$ wishes to guess a block of 512-bits for ANS described by Table 4. Let a symbol frame consists of a long sequence of s_2. Then $\mathcal{A}$ knows that the best probability of a correct guess is $(3/4)^{256} \approx 2^{-106}$, where all encodings are 2-bit long.

Note that a correct guess is just a beginning and to apply any meaningful cryptanalysis, the adversary needs to make many such guesses. This is to say that to be successful, $\mathcal{A}$ needs to avoid guessing in its cryptanalysis.

The following remarks seem to be relevant.

- Compcrypt is CPA secure and provides integrity assurance provided ANS states occur with uniform probability and sponge rounds employ the KECCAK-f permutation. In fact, ANS encodings perfectly mask the sponge state.
- According to Corollary 2, we can trade security with compression rate. This can be achieved by selecting a subset of ANS state bits that are XOR-ed with pseudorandom bits. However, a compression rate gain seems to be too small for short symbol frames. But it could be of a substantial benefit when compcrypt is used for streaming. In this case, a careful security analysis needs to be done.
- Compcrypt can be also used to authenticate plaintext like in authenticated encryption with associated data (AEAD). Our adversarial model makes it very clear that decoding table can be transmitted un-encrypted. For a sake of argument, assume that decoding table is communicated in plain but a compressed bitstream is encrypted (with a tag T). The receiver decrypts, recovers the bitstream frame and verifies the tag T. Now if an adversary has replaced an original decoding table with a fake one, then the receiver reconstructs a wrong symbol frame. To authenticate decoding table, it must be absorbed into sponge so any attempt to change it will be detected.

6.2 Selection of sponge parameters

Let us take a closer look at the rationale behind the choice of a number of iterations n_{step} of permutations P in a single round. We follow the heuristics used for the lightweight authenticated cipher Ketje [24]. The heuristics is captured by the following relation:

$$n_{\text{unicity}} = \left\lceil \frac{b-r}{r} \right\rceil n_{\text{step}}, \tag{4}$$

where n_{unicity} is the number of iterations (of permutation P), for which state-recovery attacks fail and b is the size of a sponge state. The number n_{unicity} is estimated from the best results achieved for cryptanalysis of the KECCAK-f permutation. A tradeoff between the size of bitrate r and the number of iterations n_{step} of P used by the f_{step} permutation is given by Eq. (4). In particular, for KECCAK-f with a 1600-bit state, n_{unicity} is estimated to be equal to 6. Assuming $r = 512$, we get $n_{\text{step}} = 6/\left\lceil \frac{1600-512}{512} \right\rceil = 2$.

The heuristics assumes that every ciphertext leaks r bits of a sponge state. However, this is not true in the case of compcrypt. According to Corollaries 1 and 2, ANS binary encoding are random if state jumps are random. Thus we argue that r-bit leakage is heavily limited. Due to this extra randomness introduced by ANS, we can reduce n_{step} from the default 2 to 1. This makes the algorithm roughly two times faster, particularly, for long data streams, where initialization and authentication overheads are negligible.

7 Efficiency analysis

We have implemented our compcrypt algorithm on different platforms, namely PC, Raspberry Pi 3 and 4. Apart from KECCAK , we also take into account two popular ciphers AES [25] and ChaCha/Poly1305 [26]. Our experiments have been done for a source that contains 256 symbols, whose probabilities follow a geometric distribution with the parameter $p = 0.5$. Symbol frames are 32kB long.

The results of our experiments are shown in Table 8. We take efficiency of plain ANS as our baseline for comparison. ANS with state jumps (denoted as ANS*) suffers a significant efficiency penalty. But uniform state distribution is essential for our compcrypt security claims. Our compcrypt ANS*+KECCAK has almost the same efficiency as ANS*, which means that encryption comes at a very low cost. As a benefit the compcrypt claims 128-bit security (for both CPA and integrity). The price to pay is slightly reduced compression rate. We have also implemented a version with a full

Table 8 Comparison of Algorithms

Algorithm	Efficiency MB/s PC	RPI3	RPI4	Efficiency Drop PC	RPI3	RPI4	128-bit Security	Comp. Rate Loss
Plain ANS	220	17	60				✗	0
ANS*	**182**	**15**	**49**	**17%**	**12%**	**18%**	✗	≈0.5%
ANS* + *Keccak*	**178**	**13**	**48**	**19%**	**24%**	**20%**	✓	≈0.5%
ANS+Full_KECCAK	135	9	27	39%	47%	55%	✓	0
ANS+AES-NI	152	8	29	31%	53%	52%	✓	0
ANS+AES	100	8	29	55%	53%	52%	✓	0
ANS+ChaCha20	95	6	16	57%	65%	73%	✓	0

The row in bold relates to the solution from Section 5
Notation: ANS* – ANS with state jumps

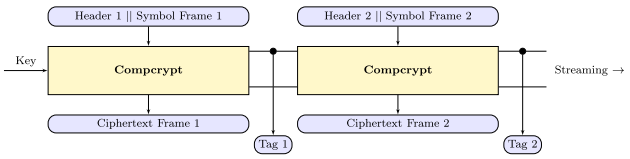

Fig. 2 Streaming with the compcrypt algorithm

KECCAK with 12 iterations of P per round. This reduces efficiency $\approx$ 20% compared to ANS*+KECCAK for PC and more for Raspberries. Experiments with AES are quite optimistic for encryption done with hardware support (AES-NI). Efficiency of ANS+AES drops dramatically when AES is implemented fully in software (for PC). The last row of Table 8 shows efficiency of ANS when it is concatenated by a low complexity cipher such as ChaCha/Poly1305.

The last column describes a compression quality loss. On the average, a binary stream with 1000 bits (for plain ANS) becomes 5 bits longer (for ANS*). A precise loss can be calculated when the state equilibrium probability distribution is known. State equilibrium probabilities depend on ANS symbol spread function and can found using Algorithm 1. It is an interesting research question about how to design ANS symbol spread function so the compression loss is the smallest for its ANS* with uniform state distribution.

8 Streaming with compcrypt

Many Internet applications need to compress and encrypt data streams. They include teleconferencing (such as Zoom, Webex or Skype) and other As ANS compression and decompression must be processed in reverse orders, it is impossible to use a single frame. A solution applied in the Zstandard (zstd) suite of compression algorithms (see https://en.wikipedia.org/wiki/Zstandard) splits a symbol stream into a sequence of frames. Clearly, it introduces a delay but it can be made negligible by choosing short frames (tens of kilobytes).

An input to compcrypt consists of a header and symbol frame. The header determines parameters of the algorithm and in particular, includes a decoding table, a starting ANS state, the number of symbols in the frame, the number of bits in the bitstream frame, the consecutive number of the frame, an initial vector and perhaps a seed/salt. Figure 2 shows a data flow when the compcrypt is applied for streaming. Note that a stream is split into symbol frames and processed in the natural order. The output of the i-th compcrypt is an input of the $(i+1)$-th compcrypt. Tags allow a receiver to authenticate frames. In case of a corrupted tag, the receiver aborts and resets the connection. In case when a connection reset is not possible, frames are generated independently and corrupted ciphertext frames are ignored.

Designing compcrypt for streaming is a delicate balancing act. A selection of short frames reduces transmission delay but also compromises compression rate as the size of a header is typically fixed and dominated by the size of decoding table. The list given below describes a few possibilities for improving streaming with compcrypt.

- Context adaptive entropy coding distributes encoding tables for typical probability distributions well ahead of time [27]. When a frame is transmitted, a short pointer to an appropriate statistics (and decoding table) is included. Alternatively, for non-stationary statistics, decoding tables can be periodically updated so a receiver needs a short correction only.
- An attractive option, for slowly changing symbol statistics, is to remove decoding tables completely from headers. Parties start from a pre-agreed statistics and then adjust it depending on statistics of previously transmitted frames. Note that as a receiver is always one frame behind a sender in its knowledge of symbol statistics, there is a loss of compression quality. There is an interesting research problem of how to approximate frame statistics from the past ones.
- Pseudorandomness required for state jump control does not need to be generated at the initialization stage of com-

pcrypt. To speed up frame processing, it may be produced in parallel by a separate sponge, which is created during the execution of compcrypt for the first frame.

Currently, ANS is used in the standardized JPEG XL image format and it is intended to be a long-term replacement for JPEG. The above considerations point at a great ANS potential as a highly competitive candidate for a future video streaming compression with a low-cost authenticated encryption. However, to make any definitive statements, a further in-depth investigation is necessary.

9 Conclusions and future works

Joint ANS-based compression and encryption with 128-bit security is the focus of the work. After presenting the ANS algorithm, we investigate ANS compression rate and present a simple algorithm that allows the calculation of its compression rate precisely. One of many interesting features of ANS is its slight variations of compression rate depending on its symbol spread function (or encoding table). As we have shown, this gives rise to a statistical attack. The attack is devastating for ANS up to few hundred states and still permits to extract a part of symbol spread function for ANS with a thousand states.

We have taken a closer look at ANS with state jumps as one of the building block for our compcrypt algorithm. If state jumps are forced by LFSR, then the output bit sequence is random. The randomness is used by us to reduce the number of the KECCAK-f permutations to a single P. The second building block is KECCAK based the MonkeyDuplex sponge. It encrypts ANS bitstream frames. Note that decoding table may also be encrypted for authentication as its confidentiality does not matter. We claim that our compcrypt provides 128-bit security in terms of confidentiality of bitstream frame and authentication/integrity of ciphertext. We have implemented the algorithm and compared it with other ones. Our compcrypt is almost as fast as ANS with state jumps. This also means that the KECCAK encryption degrades efficiency slightly only.

Our compcrypt is quite flexible and can be easily adjusted to the current needs. It can be extended to secure data streaming as shown in the work. If there is a need for a very fast compcrypt, then a designer can trade state-jump properties (length of LFSR string or/and frequency of jumps) with the number of KECCAK-f permutations P per round. If a higher than 128-bit security is required, then our compcrypt can be re-designed after a careful security consideration. It is worth mentioning that it is possible to design compcrypt using any round-reduced cipher (such as a round-reduced AES).

Acknowledgements 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.

Author Contributions The authors confirm their compliance with ethical standards. In particular, we would like to make clear that the last author is a member of the journal EB. Apart from this, we are not aware of any conflict of interest. The work does not involve human participants and/or animals. Additionally, we confirm that the paper is not currently under submission at any conference/journal and that the paper covers our original work that has not been published/presented at any journal/conference. The funding for the research is acknowledged at the back of the paper.

References

1. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Fast Software Encryption 2005. Lecture Notes in Computer Science, vol. 3557, pp. 32–49
2. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
3. Gillman, David W., Mohtashemi, Mojdeh, Rivest, Ronald L.: On breaking a Huffman code. IEEE Trans. Inf. Theory **42**(3), 972–976 (1996)
4. Martin, G.: Range Encoding: an Algorithm for Removing Redundancy from a Digitised Message. (1979)
5. Moffat, A., Neal, R.M., Witten, I.H.: Arithmetic coding revisited. ACM Trans. Inf. Syst. **16**(3), 256–294 (1998)
6. Rissanen, J.J.: Generalized kraft inequality and arithmetic coding. IBM J. Res. Dev **20**(3), 198–203 (1976)
7. Duda, J.: Asymmetric Numeral Systems as Close to Capacity Low State Entropy Coders. CoRR, arxiv:1311.2540 (2013)
8. Duda, J., Tahboub, K., Gadgil, N.J., Delp, E.J.: The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In: Picture Coding Symposium (PCS), Cairns, QLD, Australia, 2015, pp. 65–69
9. Moffat, A., Petri, M.: Large-alphabet semi-static entropy coding via asymmetric numeral systems. ACM Trans. Inf. Syst. **38**(4), 1–33 (2020)
10. Kelley, J., Tamassia, R.: Secure compression: Theory & Practice. Cryptology ePrint Archive, Report 2014/113. (2014)
11. Duda, J., Niemiec, M.: Lightweight Compression with Encryption Based on Asymmetric Numeral Systems. arxiv:1612.04662 (2016)
12. Camtepe, S., Duda, J., Mahboubi, A., Morawiecki, P., Nepal, S., Pawłowski, M., Pieprzyk, J.: Compcrypt - lightweight ANS-based compression and encryption. IEEE Trans. Inf. Forensics Secur. **16**, 3859–3873 (2021)

13. Duda, J.: Asymmetric Numeral Systems. Internet Archive. arxiv:0902.0271 (2009)
14. Donald, K.: The Art of Computer Programming. Addison-Wesley, Boston (1973)
15. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Florida (2001)
16. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic Sponges. http://sponge.noekeon.org/CSF-0.1.pdf
17. Guido, B., Joan, D., Michaël, P., Gilles Van, A.: Duplexing the Sponge: Single-pass Authenticated Encryption and other Applications. Cryptology ePrint Archive, Report 2011/499, 2011. http://eprint.iacr.org/
18. Alex, B., Adi, S.: Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers Lecture Notes in Computer Science, vol 1976, pp. 1–13
19. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. Advances in Cryptology — EUROCRYPT '93, pp. 386–39. Springer, New York (1994)
20. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. **4**, 3–72 (1991)
21. Courtois, N., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. Advances in Cryptology - ASIACRYPT 2002. Lecture Notes in Computer Science, vol. 2501, pp. 267–287
22. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg (2009)
23. Khovratovich, D., Nikolić, I.: Rotational cryptanalysis of ARX. In: Fast Software Encryption, pp. 333–346. Springer, New York (2010)
24. Guido, B., Joan, D., Michaël, P., Gilles Van, A., Ronny Van, K.: CAESAR Submission: Ketje v2 https://K.team/files/Ketjev2-doc2.0.pdf
25. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, New York (2002)
26. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Fast Software Encryption 2005. Lecture Notes in Computer Science, vol. 3557, pp. 32–49
27. Marpe, D., Schwarz, H., Wiegand, T.: Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. IEEE Trans. Circuits Syst. Video Technol. **13**(7), 620–636 (2003)