ANS-based Compression and Encryption with 128-bit Security

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

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

Abstract. 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 compression. 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 behaviour when its states are chosen at random. Our compcrypt algorithm is built using ANS with randomised 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).

1 Introduction

Shannon in his seminal paper [17] 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 compression 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, photos, 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 compression algorithm applies the well-known Huffman coding [9]. 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 [8]. The arithmetic/range coding [12, 14, 16] 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 compression (see [5, 6, 15]). 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 used by the JPEG XL next generation image compression standard.

The recent Covid19 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

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

where comp_rate stands for compression rate.

Duda and Niemiec consider a plain ANS as a compcrypt algorithm in their work [7]. 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 [3]. 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 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, whereby 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.
- Variant of ANS with a uniformly random probability distribution of ANS states. This is implemented using a random bit generator (RBG). We prove that the variant leaks no information about the internal ANS structure. It is used as a basic building block for our compcrypt algorithm.
- Design of 128-bit secure compcrypt algorithm. The algorithm uses a sponge structure with the Keccak round permutation P.
- Evaluation of security and efficiency of the compcrypt algorithm.
- Extension of the compcrypt algorithm for infinite streams.

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 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 1. The coding function generates a

$s \setminus x'$	0	1	2	3	4	5	6	7	8	9	$\cdots$
$s = 0$	0		1		2		3		4		$\cdots$
$s = 1$		0		1		2		3		4	$\cdots$

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

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.

The binary case can be generalised 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 2. Take the same

$s \setminus x'$	0	1	2	3	4	5	6	7	8	9	$\cdots$
$s = 0$	0				1				2		$\cdots$
$s = 1$		0	1	2		3	4	5		6	$\cdots$

Table 2. Encoding table for binary symbols with probabilities $p_0 = 1/4$ and $p_1 = 3/4$

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 2. 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 $D(x') = \big(L_s\lfloor x'/2^R \rfloor + x' \bmod 2^R - c_s, s\big)$, where s is identified by checking if $c_s \leq x \bmod 2^R \leq c_{s+1}$. For instance, consider Table 2 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 $x' \mod 2^R = 1$. 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-normalisation* 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-normalisation 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-normalisation allows ANS to achieve efficient compression and also it can be conveniently represented as an encoding table - we focus here on this fully tabled case (called tANS).

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

Initialisation I

Input: A set 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: Initialisation 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 $\overline{s} : \mathbb{I} \to \mathbb{S}$, such that $|\{x \in \mathbb{I} : \overline{s}(x) = s\}| = L_s$;
- establish the coding function $C(s, y) = x$ for the integer $y \in \{L_s, \ldots, 2L_s - 1\}$, which assigns states $x \in \mathbb{L}_s$ 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.

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

Frame Coding C

Input: A sequence of symbols (frame) $\mathbf{s} = (s_1, s_2, \ldots, s_\ell) \in \mathbb{S}^*$ and an initial state $x = x_\ell \in I$; where $\ell = |\mathbf{s}|$.
Output: An output bit stream $\mathbf{b} = (b_1|b_2|\ldots|b_\ell) \in \mathbb{B}^*$, where $|b_i| = k_{s_i}(x_i)$ and x_i is state in 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 $x_0 = x$;

The next algorithm takes a stream of output bits and the final state and produces symbols of the corresponding frame in reverse.

Stream Decoding D

Input: A stream of bits $\mathbf{b} \in \mathbb{B}^*$ and the final state $x = x_0 \in I$ of the encoder.
Output: Sequence of symbols $\mathbf{s} \in \mathbb{S}^*$
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 $\overline{s} : \mathbb{I} \to \mathbb{S}$ is chosen as follows:

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

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 shown below.

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

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

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.

3.1 Compression Rate of ANS

There is no proof that ANS achieves optimal compression (symbols are encoded into binary strings with no redundancy). However, experiments show that the ANS compression 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 [4]). 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. Denote that we use a shorthand $X = \{2^R, \ldots, 2^{R+1} - 1\}$.

Algorithm 1: Calculating probabilities of ANS states

Input: ANS represented by its encoding table $\mathbb{E}$ for symbols $s \in \mathbb{S}$ (rows) and states $x \in X$ (columns)
Output: Probabilities $P(x)$, where $x \in X$.
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 $\mathcal{R}_x \equiv P(x) = \sum_{y \in X} P(y) p_{s_y}$, where $x = \mathbb{E}(s_y, y)$;
  Ensemble a system of linear equations $\langle \mathcal{R}_x | x \in X \rangle$ (note that $\sum_{x \in X} P(x) = 1$).
  Solve the system using the Gaussian elimination algorithm.

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 3. After running Algorithm 1, we have got the probabilities as shown below

x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(x)$	0.08	0.08	0.079	0.081	0.067	0.067	0.06	0.064	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 customise 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 3 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	24	25	26	27	28	29	30	31
$P(x)$	0.083	0.083	0.105	0.061	0.058	0.058	0.083	0.047	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 [3] proposes a variant of ANS, where ANS states are chosen randomly using a cryptographically strong pseudorandom bit generator (PRBG). This means that ANS state probability distribution is uniform. For ANS from Table 3 with state jumps, compression loss is $\Delta H \approx 0.03$ bits per symbol. The reader is referred to the papers [3, 4] 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, FSM with a finite number of states, it has a cyclic nature that allows an adversary to inject/remove output bits without detection (see [3] for details).

3.3 Statistical Attacks against ANS

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. The very first version of our attack is given below. The algorithm exhaustively enumerates all possible spread functions. The

Algorithm 2: Exhaustive Statistical Attack against ANS

Input: An instance of ANS with a adaptive-statistics adversary and the symbol source probability distribution $\mathcal{S}$.
Output: Encoding table or symbol spread function used by ANS.
begin
  Collect n output bits from the target ANS and calculate its compression rate $\beta = \frac{n}{\ell}$, where ℓ is the number of symbols processed by ANS;
  for $ANS_i \in \{all\ instances\ of\ ANS\}$ **do**
    use Algorithm 1 to compute state probabilities for ANS_i;
    compute its compression rate β_i for the distribution $\mathcal{S}$;
    if $\beta \approx \beta_i$ **then**
      return ANS_i;

important point here is that $\mathcal{A}$ is able to identify the correct instance of ANS by checking its compression rate. The check can fail if the observed output stream is not long enough so β is far away from its real value. To fix this, $\mathcal{A}$ may need to collect a very long sequence. In case that there are few other ANS instances, whose compression rates are close to β, $\mathcal{A}$ may test other symbol statistics. The complexity of Algorithm 2 is determined by the number of all ANS instances that need to be tested, which is

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

where $n = |\mathbb{S}|$ and L_s is the number of states assigned by the symbol spread function for the symbol s [11].

Algorithm 2 can be substantially improved by applying the divide-and-conquer principle. Note that $\mathcal{A}$ can select in arbitrary way the symbol source distribution. 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 3. 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 3 becomes

$s_i \backslash x_i$	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. We run Algorithm 2 to determine states in the two rows of the encoding table (encoding table reduced to two rows). Now we repeat the procedure but this time we select a pair of rows (symbols): one with states already found and the other with states that need to be determined. The flowchart

of the attack is given by Algorithm 3. Note that we denote $\Gamma_i = \{x|\bar{s}(x) = s_i\}$ 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 3: Divide-and-Conquer Statistical Attack against ANS

Input: An instance of ANS with a adaptive-statistics adversary and the symbol source probability distribution $\mathcal{S}$.
Output: 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 Algorithm 2 for the reduced encoding table;
  Store $\Gamma_1 = \{x|\bar{s}(x) = s_1\}$ and $\Gamma_2 = \{x|\bar{s}(x) = s_2\}$;
  for $i = 3, \ldots, n-1$ **do**
    Run Algorithm 2 for reduced encoding table for s_1 and s_i;
    Store $\Gamma_i = \{x|\bar{s}(x) = s_i\}$;
  Return the symbol spread function $\bar{s}(x)$;

The work load needed to calculate Γ_1 and Γ_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 3 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 Equation (1) compared to the divide-and-conquer one described by Equation (2).

Example 3. Consider this attack for realistic parameters with the number of states $L = 1024$ and 15 symbols that follow a geometric probability distribution $P(S = k) = (1-p)^{k-1}p$, where $p = 0.3$. The results are displayed in the table below, where WF stands for work factor.

s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L_s	307	215	150	105	73	52	36	26	19	13	9	7	5	4	3
WF	0	$\binom{522}{215}$ $\downarrow$ 2^{350}	$\binom{672}{150}$ $\downarrow$ 2^{353}	$\binom{777}{105}$ $\downarrow$ 2^{439}	$\binom{850}{73}$ $\downarrow$ 2^{354}	$\binom{902}{52}$ $\downarrow$ 2^{282}	$\binom{938}{36}$ $\downarrow$ 2^{149}	$\binom{964}{26}$ $\downarrow$ 2^{117}	$\binom{983}{19}$ $\downarrow$ 2^{91}	$\binom{996}{13}$ $\downarrow$ 2^{96}	$\binom{1005}{9}$ $\downarrow$ 2^{71}	$\binom{1012}{7}$ $\downarrow$ 2^{57}	$\binom{1017}{5}$ $\downarrow$ 2^{43}	$\binom{1021}{4}$ $\downarrow$ 2^{35}	$\binom{1024}{3}$ $\downarrow$ 2^{27}

The attack starts from testing our target ANS by collecting a long enough bitstream for the two least probable symbols $s \in \{15, 14\}$ with some chosen probability distribution. Note that the target ANS encoding table can be reduced by removing all states that do not belong to the symbols $s \in \{15, 14\}$. Next, we enumerate all possibilities for symbol spread function for $s \in \{15, 14\}$. This takes 2^{62} steps. In each step, we create a partial encoding table for the two symbols (i.e. table has two rows only). We calculate state equilibrium probabilities and the corresponding probabilities of different lengths of encodings. We calculate the mean and standard deviation. If the observed length for the target ANS falls within $3 \times \sigma$ from the mean we accept otherwise reject. We hope to end up with only a few instances of symbol spread function. We can further test using different symbol probabilities. Once we have a unique solution, we try the next symbol $s = 13$ and repeat the attack. At this point, we can experiment with two rows or with three to identify the correct symbol spread function for $s = 13$.

Example 4. In this example, we show how to distinguish the ANS given by Table 3 from its variant, where states are chosen uniformly at random. After simple calculations, we can determine the probability distribution of binary encodings with the different lengths for both ANSes as given below.

Probability\ANS	Plain ANS	ANS with state jumps
$P(1)$	0.59994550	0.57812500
$P(2)$	0.3210917750	0.32812500
$P(3)$	0.07896232500	0.093750000

After observing 10,000 symbols, the plain ANS generates bitstreams of the average length $14,790$ bits with $\sigma = 63.8$, while the other ANS gives bitstream of of the average length $15,156$ bits with $\sigma = 66$. If we run few experiments, we can distinguish them with non-negligible probability.

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 [13]. 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 be expected 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 [7]. 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 [3], 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 [3, 7]) 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 [3]). 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 Section 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 the 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 3) 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;
– preserves compression rate of the 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 Randomised 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 Section 2. Then for a symbol $s \in \mathbb{S}$, ANS generates*

– *1-bit encodings and the encoding table row for the symbol s contains equal number of zeros and ones if $L_s = 2^{R-1}$,*
– *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 $L_s > 2^{R-1}$,*

– *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.*

Proof. According to the frame coding algorithm, 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

$$\lfloor \log_2 \frac{2^R}{L_s} \rfloor \leq k_s \leq \lfloor \log_2 \frac{2^{R+1}-1}{L_s} \rfloor \tag{3}$$

Case 1 If $L_s = 2^{R-1}$, then Equation (3) becomes

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

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 (3) 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 has to be positive as the expression is positive. The encoding table row for s 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 conclude the proof.

Example 5. Consider the ANS instance given by Table 3. 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 $(2L_s - 2^{R-k_s}) = 6 - 4 = 2$.

ANS with Fully Randomised States

Corollary 1. *Given a plain ANS as described in Section 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 6. Take ANS from Table 3. 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.

ANS with Partially Randomised 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 3 as an example). As probability of an ANS state x can be approximated by $\approx 1/x$, a plain ANS favours 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

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

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

$$x \xleftarrow{PRBG} \begin{cases} (\lfloor x/2^u \rfloor, & 0) \\ \vdots & \vdots \\ (\lfloor x/2^u \rfloor, & 2^u - 1) \end{cases} \tag{4}$$

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 Equation (4). 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 Equation (4). 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 Equation (4). 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 7. Consider our toy ANS from Table 3. 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/deletion/injection attacks. 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 should be enough. In case this is not enough, we can apply nonlinear feedback shift register (NFSR) or cryptographically strong PRBG. Note that for a sponge construction, a long enough pseudorandom sequence can be generated internally by the first collection of iterations.

4.3 MonkeyDuplex Structure

To make the lightweight ANS secure against more powerful attacks (such as a chosen-plaintext attack), we encrypt the bitstream blocks (obtained from the ANS) with an encryption scheme called MonkeyDuplex. In 2010, Bertoni et al. introduced a duplex construction [2], which provides a framework for authenticated encryption. The duplex construction can be seen as a variant of the well-known sponge construction [1]. There are two main sponge parameters: bitrate r and capacity c. Their sum determines the size of a state. 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. A permutation P is the most crucial building block of sponge. In [1] it is shown 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 permutation. The permutation is used in the SHA-3 hashing standard and in authenticated ciphers such as Keyak and Ketje. MonkeyDuplex with the Keccak-f permutation allows efficient encryption, especially, when dealing with a very long stream of plaintext blocks. The permutation P can operate on large states (up to 1600 bits), which boosts the efficiency. The other factor is 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.

5 Compcrypt Algorithm

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

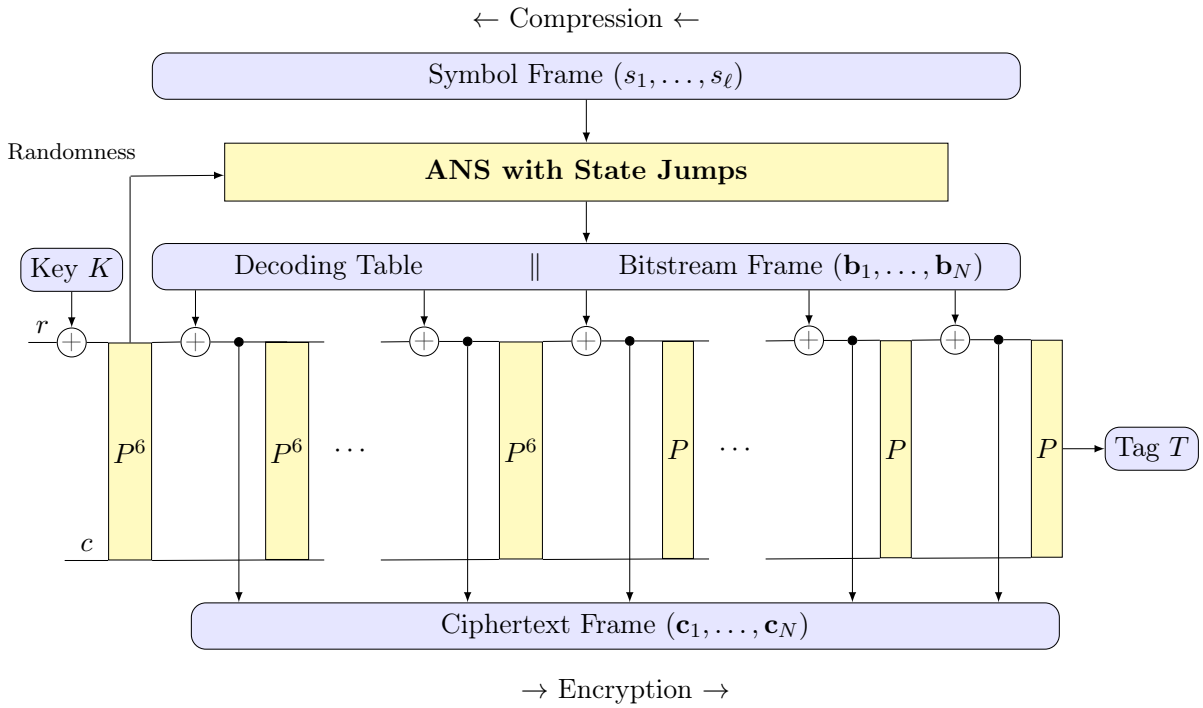

Fig. 1. Compcrypt based on ANS and sponge, where bitrate $r = 512$ bits, capacity $c = 1088$ bits and size of sponge state is $b = 1600$

Let us take a closer look at the rationale behind the choice of the sponge parameters. We follow heuristics developed by the designers of the duplex construction [2]. Let $n_{unicity}$ be a number of iterations (of permutation P), for which state-recovery attacks fail. 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 the following expression:

$$n_{unicity} = \left\lceil \frac{b-r}{r} \right\rceil n_{step}, \tag{5}$$

where b is the size of a sponge state. For Keccak-f with a 1600-bit state, $n_{unicity}$ is estimated to be equal to 6. Assuming $r = 512$, we get $n_{step} = 6/\left\lceil \frac{1600-512}{512} \right\rceil = 2$. The heuristics assumes that every ciphertext leaks r bits of a sponge state. This is not true in the case of compcrypt. According Corollaries 1 and 2, ANS binary encoding are random if state jumps are random. We can modify Equation (5) to take advantage of the extra randomness

$$\lceil n_{unicity} \cdot (1-H) \rceil = \left\lceil \frac{b-r}{r} \right\rceil n_{step}, \tag{6}$$

where H is a normalized entropy (ranging between 0 and 1) of an ANS bitstream injected to the sponge. The higher H the smaller $n_{unicity}$ and consequently, a smaller n_{step} can be chosen. Thus, n_{step} is reduced to 1 or $f_{step} = P$. The choice makes the algorithm roughly two times faster, particularly, for long data streams, where initialisation and authentication overheads are negligible.

5.1 Description of Compcrypt

A general data flow during encryption is illustrated in Figure 1. The algorithm steps are shown in Algorithm 4. Let us discuss its steps in more detail.

Algorithm 4: Compcrypt Encryption

Input: A symbol frame $\mathcal{S} = (s_1, s_2, \ldots, s_\ell)$ and a 128-bit secret key K.

Output: A ciphertext frame $\mathcal{C} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$ of decoding table and 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) Initialise a sponge for K 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 and store ANS decoding table;
(4) Compress $\mathcal{S}$ in the reverse order and prepend the obtained bitstream with decoding table. 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 sponge;
(7) Generate 128-bit tag T;
(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/decoding tables.

Step 2 The 1600-bit sponge is initiated by the secret key K and can include associated data (such an initial vector, time, sequence number, etc.). The initial sponge state is translated by f_{start}. The number of iterations n_{start} has to be big enough so the pseudorandom bits collected from 512-bit bitrates are sufficient to randomise ANS state jumps for all symbols. 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. The decoding table is stored.

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 Section 4.2). Pseudorandom bits are taken from the bit collection generated by f_{start} during sponge initialisation. Note that the bits should be used in the reverse order, i.e. last generated bits must be used first. The compressed bitstream is prepended by a binary representation of the decoding table. It 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 decoding table and the bitstream frame $(\mathbf{b}_1, \ldots, \mathbf{b}_N)$ are encrypted into their ciphertext $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$ using the sponge. Note that the sponge encrypts the first part of the frame (that contains decoding table information) using six iterations of P between two consecutive blocks. For the second part with compressed bits, the sponge applies $f_{step} = P$ between two consecutive blocks.

Step 7 A 128-bit authentication tag T is extracted from the final state of sponge. Note that a permutation f_F consists of 6 iterations of P.

Step 8 Finally, the ciphertext frame and the tag T are sent to the receiver side.

At the receiver side, decryption starts from the initialisation of the sponge, where a big enough collection of pseudorandom bits is stored for decompression. Next, the sponge decrypts ciphertext frame and verifies validity of the received tag T. If T is valid, ANS recovers its decoding table and a starting state x_F. ANS decompresses bitstream and recovers a symbol frame. Note that pseudorandomness for ANS state jumps is used in the natural order.

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 4 takes a closer look at the adversary

Table 4. 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 frames $(s_1, s_2, \ldots, s_\ell)$ and observe their ciphertext blocks $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N)$	– finding cryptographic key K – obtaining sponge state
Integrity	– As above + – $\mathcal{A}$ can tamper with ciphertext blocks and tag T by injecting/deleting/modifying their parts	– acceptance of tampered blocks as genuine

ability and its goals. To avoid trivial attacks that intend to simplify ANS or even bypass it, we make the following caveat.

Important: 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.

6.1 Guessing Lengths of ANS Encodings

One of the distinct features of ANS is its variable-length encoding. Before applying any standard cryptanalytic tool (such as algebraic, linear or differential), $\mathcal{A}$ needs to identify internal variables of a compcrypt. According to Lemma 1, all symbols s, where $p_s \neq 1/2$, 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 3. 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.

6.2 Analysis with Fixed Length Encodings

Fortunately for $\mathcal{A}$, ANS allocates 1-bit encodings for a symbol that occurs with probability $p_s = 1/2$. Although the event is quite unlikely, our chosen-plaintext adversary may prepare a symbol frame that contains a long sequence of such symbols. For simplicity, we assume that $\mathcal{A}$

puts the sequence at the beginning of a symbol frame. For any single bit c_i of a cryptogram, $\mathcal{A}$ is able to write the following relation

$$c_i = z_i \oplus LSB(x \oplus \alpha), \tag{7}$$

where z_i is the bit coming from the sponge, $LSB(x \oplus \alpha)$ is the least significant bit extracted from the ANS state after a jump and α is a corresponding pseudorandom string of R bits that controls the state jump. Corollary 1 assures us that the encoding of s is random if α is random. This is obviously not the case but using the heuristics from [2], we can argue that Equation (7) provides no help to our adversary if α and c_i are separated by a permutation that consists of more than 6 iterations of P.

Note that the first part of ciphertext contains encryption of decoding table and other parameters. As this information can be public, our adversary may try to break this part. We have assumed that a permutation between two consecutive blocks includes six iterations of P so this part is immune against our adversary attacks.

6.3 Integrity of Compcrypt

One can claim that decoding table can be transmitted un-encrypted. Indeed, our adversarial model makes it very clear that it is public. 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 encrypted so any attempt to change it will be detected.

7 Efficiency Analysis

We have implemented our compcrypt algorithm on different platforms, namely PC, Raspberry Pi 3 and 4. Table 5 consists of five rows. Each row describes efficiency of its algorithm. The first three columns give actual efficiency of our implementations. As they are not optimised, our estimation

Table 5. Comparison of Algorithms

Algorithm	Measured MB/s			Estimated MB/s			Drop		
	PC	RPI3	RPI4	PC	RPI3	RPI4	PC	RPI3	RPI4
Plain ANS	220	17	60	325	25	89			
ANS*	182	15	49	269	22	72	17%	12%	18%
ANS*+Keccak	178	13	48	263	19	71	19%	24%	20%
ANS+AES-NI	152	8	29	225	12	43	31%	53%	52%
ANS+AES	100	8	29	148	12	43	55%	53%	52%

Notation: ANS* – ANS with state jumps.

about the efficiency of optimised software implementations are given in the next three columns. Last three columns shows efficiency drops of algorithms, where a plain ANS is a benchmark. 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. Instead of a single state jump per symbol, we force state jump every fourth symbol. Efficiency of an ANS with state jumps (ANS*) drops 17% as compared to a plain ANS. Note that our compcrypt or ANS*+Keccak is almost as fast as ANS*. We have experimented first with encryption done by AES with hardware support (AES-NI) and second with AES implemented fully in software.

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

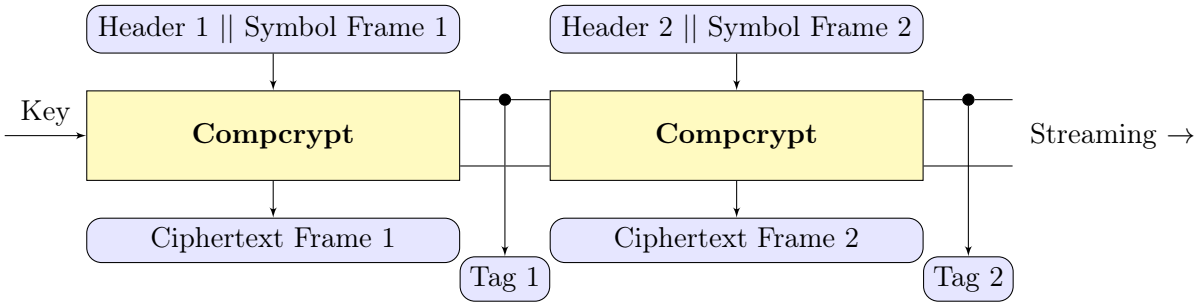

Fig. 2. Streaming with the compcrypt algorithm

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.

- Instead of a full decoding table, a header may include the parts of the table that have changed (due to variations of symbol statistics). A further improvement can be achieved by re-using tables from previous sessions. Instead of a table, a sender can forward an identifier of a previously applied table that is appropriate for a current frame.
- An alternative is to allow both parties to construct their tables independently. Both sending and receiving parties need to agree on an algorithm for ANS design for symbol statistics. There is a technical obstacle as a receiver is always one frame behind the sender in its knowledge of symbol statistics. A way out for the sender could be to communicate a decoding table in the first frame and allow the receiver to update it on the base of gathered symbol statistics. This option can be used for streams with slowly changing symbol statistics.
- Pseudorandomness required for state jump control does not need to be generated at the initialisation stage of compcrypt. 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.

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 a (pseudo)random bit generator, then the output bit sequence is highly random. This randomness is used by us to reduce the number of the Keccak permutations to a single P. The second building block is Keccak based the MonkeyDuplex encryption. MonkeyDuplex supplies pseudorandom bits for ANS state jumps. It encrypts decoding table and ANS bitstream frame. Note that decoding table is 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. There is no security proof as the security algorithm is based on heuristic arguments. However, we give a security justification. 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 PRBG string or/and frequency of jumps) with the number of Keccak 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 a round-reduced AES and an appropriate mode of operation. This is an attractive option as in many computing platforms, encryption is supported by an AES coprocessor.

Acknowledgments

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

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic Sponges. `http://sponge.noekeon.org/CSF-0.1.pdf`.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-pass Authenticated Encryption and other Applications. Cryptology ePrint Archive, Report 2011/499, 2011. `http://eprint.iacr.org/`.

[3] Seyit Camtepe, Jarek Duda, Arash Mahboubi, Paweł Morawiecki, Surya Nepal, Marcin Pawłowski, and Josef Pieprzyk. Compcrypt – Lightweight ANS-based Compression and Encryption. *IEEE Transactions on Information Forensics and Security*, Accepted May 2021

[4] Jarek Duda. Asymmetric Numeral Systems. *Internet Archive*, arxiv-0902.0271, 2009.

[5] Jarek Duda. Asymmetric Numeral Systems as Close to Capacity Low State Entropy Coders. *CoRR*, abs/1311.2540, 2013.

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

[7] Jarek Duda and Marcin Niemiec. Lightweight Compression with Encryption based on Asymmetric Numeral Systems. arXiv, eprint 1612.04662, 2016.
[8] David W. Gillman, Mojdeh Mohtashemi, and Ronald L. Rivest. On breaking a Huffman code. *IEEE Transaction on Information Theory*, 42(3):972–976, 1996.
[9] D.A. Huffman. A method for the Construction of Minimum-redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[10] J. Kelley and R. Tamassia. Secure compression: Theory & Practice. *Cryptology ePrint Archive*, Report 2014/113, 2014.
[11] Donald Knuth. *The art of computer programming, Vol. 2.* Addison-Wesley, 1973.
[12] G. Martin. Range Encoding: an Algorithm for Removing Redundancy from a Digitised Message. 1979.
[13] A. J. Menezes and P.C. van Oorschot and S.A. Vanstone. Handbook of Applied Cryptography. *CRC Press*, 2001
[14] A. Moffat and R.M. Neal and I.H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems*, Vol. 16, No. 3, July 1998, pages 256–294.
[15] A. Moffat and M. Petri. Large-alphabet semi-static entropy coding via asymmetric numeral systems. *ACM Transactions on Information Systems*, 38(4):1–33, 2020.
[16] J.J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.
[17] Claude E. Shannon. *The Bell System Technical Journal.* 27:379–423, 623–656, July, October 1948.