Extracting the RC4 secret key of the Open Smart Grid Protocol

Linus Feiten and Matthias Sauer

Chair of Computer Architecture, University of Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg, Germany
{feiten,sauerm}@informatik.uni-freiburg.de

Abstract. The Open Smart Grid Protocol (OSGP) is a widely used industry standard for exchanging sensitive data between devices inside of smart grids. For message confidentiality, OSGP implements a customised form of the RC4 stream cipher. In this work, we show how already known weaknesses of RC4 can be exploited to successfully attack the OSGP implementation as well. The attack modification is able to effectively derive the secret OSGP encryption and decryption key, given that an attacker can accumulate the cipher streams of approximately 90,000 messages. The possession of this key allows the attacker to decrypt all data intercepted on the OSGP smart grid and thereby obtain privacy critical information of its participants.

Keywords: Cryptography, Security, RC4, Smart Grid, Protocol

1 Introduction

A *smart grid* is an electricity grid that does not just transport electricity from suppliers to consumers, but does incorporate real-time information sharing between the participants in an intelligently automated (hence *smart*) way [6,8]. Such information about electricity supply, demand and consumption can be used for dynamic billing or load balancing. Especially as regenerative energy sources like solar panels or wind engines are becoming more popular, an automated continuous communication between suppliers and consumers is vital for dealing with load peaks and drops [12]. While the short-term availability of electricity from regenerative sources is hard to predict, the challenge of load balancing is aggravated when private households are not just consumers of electricity but also producers, feeding excess energy from their solar panels or combined heat and power plants back into the grid.

The metering or load regulating devices at the consumer's site are called *smart meters* [23]. Surveys found the number of smart meters installed in Europe in 2012 to be 52 million, and forecast an amount of 170 million until 2019 [20]. Similarly, 28 million smart meters were reported for North America in 2010 with a prognosis of 87 million until 2016 [19].

It is imperative that the data transmission between the grid operator and the smart meters guarantees data confidentiality and integrity [22]. It must thus

not be possible for an attacker to learn the content of an eavesdropped message containing sensitive data like the energy consumption of another household [17] or to change the content of intercepted messages unnoticed. The latter would for instance allow him to report a false energy consumption, thereby stealing electricity or hampering the load balancing efforts of the grid operator. Data confidentiality is usually achieved through encryption using a cryptographic secret key, whereas data integrity is usually achieved by calculating a check value from a combination of the data to be verified and another secret key.

In this work, we show a new method to compromise the data confidentiality of a widely used smart grid standard, the *Open Smart Grid Protocol* (OSGP) [2]. For data confidentiality, OSGP implements a customised form of the RC4 stream cipher [9]. Our new method comprises the modification of a known attack exploiting biases in the RC4 cipher stream output to effectively calculate the secret encryption key. Once this secret key is obtained, it can be used to decrypt all intercepted data sent in an OSGP smart grid.

The remainder of this paper is organised as follows: Section 2 introduces the OSGP standard. While a description of the basic RC4 algorithm is given in Appendix A, Sections 3.1, 3.2, and 3.3 sketch known RC4 biases and the attack, of which the method presented here is a modification. The new method itself is detailed in Section 4. Experimental results demonstrating the attack's efficiency are presented in Section 5, followed by Section 6 in which makeshift measures to hamper the attack are suggested. Section 7 concludes the paper.

2 Open Smart Grid Protocol (OSGP)

The Open Smart Grid Protocol (OSGP) was inaugurated in 2010 by the Energy Services Network Association (ESNA, now OSGP Alliance); a non-profit consortium of leading companies in the energy industry [4]. OSGP's aim as an open protocol standard is to enable different vendors to develop and sell mutually compatible smart grid devices like smart meters, power quality sensors, load control modules or controllers of solar panels, combined heat-and-power plants or electric car charging stations. According to the OSGP Alliance's website (www.osgp.org), there are currently over 4 million OSGP based devices deployed world wide. In 2012, the OSGP specification was formally published by the European Telecommunications Standards Institute (ETSI) [5] as group specification GS OSG 001 [2]. If not otherwise stated, all statements about OSGP in this work are based on this publicly available document. In the following, we give a brief overview of the OSGP structure and aspects particularly relevant for this work.

Figure 1 shows the structure of an OSGP smart grid with its different communication layers. The OSGP protocol specified in ETSI GS OSG 001 [2] only defines the topmost application layer. Below, there is the networking layer which is defined by the ISO/IEC 14908-1 standard [3] (a.k.a. LonWorks or LonTalk). Each OSGP application layer message is embedded in an ISO/IEC 14908-1 message before it is sent via the respective physical media channel. Currently, the

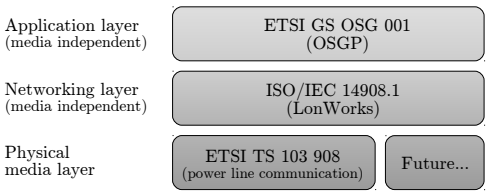

Fig. 1. The OSGP layer structure.

power line communication standard ETSI TS 103 908 [1] is the designated physical layer for OSGP smart grids, but as OSGP and ISO/IEC 14908-1 are themselves media independent, OSGP could be operated on any current or future physical media.

Data confidentiality and integrity mechanisms could be implemented on each of the three communication layers. The OSGP application layer specification, though, implements its own security mechanisms and does thereby not rely on security of the networking or physical media layer. The attack described in this paper targets this application layer. It compromises data confidentiality while data integrity remains intact.

The communication within an OSGP smart grid follows a master/slave architecture, where each OSGP device operates as a slave and does therefore never initiate a message exchange by itself. This is only done by the central *data concentrator* (DC) which is commissioning and controlling all OSGP devices under its domain. Thus, whenever the DC needs to read from or write to a device's data table, it sends a request to be answered by the device.

To avoid replay attacks, each OSGP message includes a 4-byte sequence number. A device only accepts request messages from the DC, whose sequence number matches the device's internally expected number. After successfully processing a valid request, the device's expected number is incremented, such that a message with an already used sequence number is not accepted any more. The number is only incremented, when the request has been valid.

Data integrity is achieved by appending an 8-byte *digest* value to each message. The digest is created by a hash function (also defined in ETSI GS OSG 001 [2]) whose input is the message itself and a secret key (*open media access key*: OMAK) known to the DC and *all* OSGP devices. (There is really only one key for the whole network! Although this fact is not explicitly stated in ETSI GS OSG 001, it is implied by OSGP providing a message broadcast mode in which a single message can be sent to several devices at once.) The digest is sent unencrypted with the otherwise encrypted message. The receiver decrypts the message and validates it by creating the digest again and comparing it with the received digest.

Data confidentiality is achieved by encrypting the message with a customised variant of the RC4 stream cipher. RC4 in general is explained in Section A while the OSGP specific RC4 variant is explained in Section 4.1. The secret key used for both RC4 encryption and decryption is called *base encryption key* (BEK).

The BEK (16 bytes) is actually derived[1] from the OMAK (12 bytes) and is – like the OMAK – the same for *all* devices. It is assumed that both OMAK and BEK are stored securely on the devices, such that they cannot be extracted through a user interface or an invasive hardware-tampering attack. The cryptanalytic attack presented here derives the BEK by analysing a multitude of intercepted messages; or more precisely: by analysing the used RC4 cipher streams. The scenarios in which this succeeds are quite conceivable and discussed in Section 4. When the BEK is known, it can be used to decrypt all data intercepted on the OSGP smart grid; i.e. data confidentiality is broken. As the OMAK, however, cannot be derived from the BEK, data integrity is still intact.

There has been another recent security analysis of OSGP in [11]. The authors focus on the OSGP digest algorithm and find that even the OMAK can be extracted by sending a number of messages with forged digests to the DC and analysing the responses. As obtaining the OMAK allows an attacker to also derive the BEK, their attack breaks both confidentiality and integrity. However, as that attack and the attack presented here target completely independent components of OSGP (here: RC4 encryption, [11]: digest algorithm), it is imperative to bring both attacks to public attention. Another analysis of OSGP has also been given in [14], demonstrating attacks on the digest algorithm. Vulnerabilities of RC4 are hinted at but no attack is described.

3 Preliminary Works

3.1 Roos correlation

In 1995 Roos [18] published his findings about a correlation between the secret key bytes $k[0], \ldots, k[15]$ and the first internal state bytes $S[0], \ldots, S[15]$ after KSA. (In the following we are always assuming 16 key bytes.) His most important result for this work is that the values of $S[r]$, $r < 15$ after KSA are much more likely to hold a certain value $X(r)$ than any other value: $P(S[r] = X(r)) \approx 0.37$, whereas the ideal probability for any value would be $1/256 \approx 0.0039$. The $X(r)$ values are defined as:

$$X(r) = \sum_{i=0}^{r} (k[i] + i) \mod 256$$

[1] This deriving is done using the authentication encryption algorithm of the ISO/IEC 14908-1 standard [3] and can only be obtained with charge.

When $X(r)$ and all $k[i]$, $0 \leq i < r$ are known, $k[r]$ can be calculated:

$$\begin{aligned} & X(r) - \sum_{i=0}^{r-1}(k[i]+i) - r && \mod 256 \\ &= \sum_{i=0}^{r}(k[i]+i) - \sum_{i=0}^{r-1}(k[i]+i) - r && \mod 256 \\ &= k[r] + r - r && \mod 256 \\ &= k[r] \end{aligned}$$

Thus, if $X(0), X(1), \ldots, X(15)$ are known, it is possible to derive all 16 secret key bytes $k[0], k[1], \ldots, k[15]$.

3.2 Jenkins correlation

In 1996, Jenkins [10] reported a correlation in the RC4 cipher stream bytes. The relevant result for this work is that a cipher stream byte $z[r]$ has the probability of $2/256$ to hold the value $r + 1 - S_{r-1}[r + 1]$, which is twice as high as the probability for any other value:

$$P(z[r] = r + 1 - S_{r-1}[r + 1] \mod 256) = \frac{2}{256}$$

whereas

$$\begin{aligned} & P(z[r] = x) = \frac{1}{256} - \left(\frac{2}{256} \cdot \frac{1}{255}\right) \approx \frac{1}{256} \\ & \forall x \in \{0, \ldots, 255\}, x \neq r + 1 - S_{r-1}[r + 1] \mod 256 \end{aligned}$$

The value $S_{r-1}[r + 1]$ is the internal state at index $r + 1$ after PRGA round $r - 1$, preceding round r in which $z[r]$ is created. If the $z[r]$ value with increased probability is obtained, $S_{r-1}[r + 1]$ can be calculated from it: $S_{r-1}[r + 1] = r + 1 - z[r] \mod 256$. Thus, each cipher stream byte $z[r]$ can be used to reveal the content of the internal state byte $S_{r-1}[r + 1]$ with an increased probability. Notice that it is hence not possible to derive $S[0]$ in this way.

While Jenkins just published his observation based on empirical analyses, Mantin [15] provided a proof for the correlation in 2001. Another proof – apparently unaware of [15] – was given by Klein [13] in 2006.

3.3 Klein attack

In 2006, Klein [13] combined Roos's and Jenkins's correlations to present an attack on variants of RC4 in which the used RC4 key $(k[0], k[1], \ldots, k[15])$ is the concatenation of the secret key sk and a publicly known message-specific initialisation vector (IV). A prominent example of such an RC4 variant had been

the Wired Equivalent Privacy (WEP) standard for Wi-Fi networks, in which a 3-byte IV is prepended to a 13-byte secret key. To explain Klein's attack, however, it is simpler to assume the case in which the IV is appended: $k = sk|IV$.

As Klein does neither refer to Roos nor to Jenkins or Mantin when describing the correlations, it seems that he discovered them independently, noteworthily providing a novel proof. His contribution most relevant to this work is the combination of the aforementioned correlations; thus being able to derive the $X(r+1)$ value from cipher stream byte $z[r]$. He calculates the probability of a cipher stream byte to hold the desired value as:

$$\begin{aligned} &P(z[r] = r + 1 - X(r + 1) \mod 256) \\ &\approx 0.37 \cdot \frac{2}{256} + 0.63 \cdot \frac{1}{256} \approx 0.0053 \end{aligned}$$

The first summand $0.37 \cdot 2/256$ stems from the cases in which $S_{r-1}[r + 1] = X(r+1)$ (due to Roos's correlation; probability 0.37) and $z[r] = r+1-S_{r-1}[r+1]$ (due to Jenkins's correlation; probability $2/256$). The second summand $0.63 \cdot 1/256$ stems from the remaining cases in which $S_{r-1}[r+1] \neq X(r+1)$ (probability $1.0 - 0.37$), but still $z[r] = r + 1 - X(r + 1)$ (probability $1/256$ as any other value).

Notice, that this formula is actually only correct for cipher stream byte $z[0]$. To make it more realistic, it would have to be changed into:

$$\begin{aligned} &P(z[r] = r + 1 - X(r + 1) \mod 256) \\ &\approx 0.37 \cdot \frac{2}{256} \cdot \left(\frac{255}{256}\right)^r + 0.63 \cdot \frac{1}{256} \end{aligned}$$

The additional factor $(255/256)^r$ expresses that in each of the r PRGA rounds preceding round r (round numbering starts with 0), there is a chance of $1/256$ for $S[r + 1] = X(r + 1)$ to be swapped with $S[i]$, such that the desired $X(r + 1)$ value would no longer be in $S[r + 1]$. However, even with this more realistic formula, the probability of e.g. $z[14]$ to reveal $X(15)$ is almost 0.0052, which is still notably larger than the probability of $1/256 \approx 0.0039$ for any other value.

As each message is calculated with a different IV appended to the secret key, the used key $k = sk|IV$ is also different for each message, such that a different cipher stream z is produced. This allows an attacker to analyse the value occurrences of the cipher stream bytes. Assuming a 16 byte secret key sk, the attacker records the cipher stream bytes $z[0], z[1], \ldots, z[14]$ for several messages and calculates $S_{r-1}[r + 1] = r + 1 - z[r] \mod 256$, as explained in Section 3.2.

After a sufficiently large number of messages, a certain value with most occurrences will emerge. This is the desired $X(r+1)$ value. From the cipher stream bytes $z[0], z[1], \ldots, z[14]$, the attacker thus gets $X(1), X(2), \ldots, X(15)$. The first key byte $k[0]$ has to be guessed, which is a negligible effort. Then, key bytes $k[1], \ldots, k[15]$ can be calculated from the X values as described in Section 3.1.

A difficulty in undertaking this attack is that the attacker needs to record the cipher stream z, which is not transmitted with the encrypted message e. Should

the attacker, however, be able to correctly guess a byte $m[r]$ of the plain text message m, he would be able to derive the corresponding cipher stream byte by calculating $z[r] = e[r] \oplus m[r]$. It is quite common that certain bytes in messages – particularly at the beginning where header data is transmitted – are always the same or at least predictable.

4 Attack on RC4 in OSGP

The RC4 variant used in OSGP is similar but slightly different from the variants targeted by Klein's attack. Section 4.1 describes the RC4 implementation of OSGP and Section 4.2 explains how to modify Klein's attack in order to exploit the respective RC4 weaknesses there.

But first we discuss the conditions under which this attack on OSGP is practical. Being a known-cipher-stream attack, the attacker must know the first 15 cipher stream bytes of enough messages in order to derive the 16 BEK bytes. This can be difficult because the cipher stream z is never transmitted. However, there are plausible scenarios in which it can be obtained never the less.

The first possibility, already mentioned in Section 3.3, is that certain bytes of the plain text message can be guessed. This is indeed conceivable in an OSGP scenario: e.g. among the first 15 bytes of the request to read out the clock of an OSGP device (cf. Section 5), only four bytes comprising the sequence number are not predictable whereas the other 11 bytes are always the same. Thus, if an attacker knows at what times the DC reads out the clock of a device, he could obtain the values of 11 cipher stream bytes. The four unobtainable bytes prevent him from directly deriving the value of four BEK bytes. These have to be brute forced just as the first BEK byte. Trying all 2^{40} possible values of five bytes would take about a month on a single 3.40 GHz Intel i7-4770 Core. However, in the average case, the correct key is already found after trying half of all values, reducing the time to two weeks. Using a cluster consisting of 256 CPU cores, the time can be reduced to only 1.5 hours.

Another possibility is that even though the OMAK and BEK cannot be accessed directly on an OSGP device, the cipher streams might not be protected in an equally secure manner. If the attacker has access to one or several OSGP devices and is able to access the memory in which the cipher streams are processed, the attack can be executed.

If the cipher stream z of a processed message m is not directly accessible, it would be sufficient to know the plain text of m. It is assumable that this can be achieved through an interface of the device. The attacker would XOR m with the encrypted message e, which is eavesdropped at the device's output, and thereby get z for each message.

4.1 RC4 in OSGP

As further described in Appendix A.1, two different messages must never be encrypted with the same RC4 key. To avoid this, the secret key – known only to

the communicating parties – must be altered for each message. This is usually done by combining it with a random message-specific initialisation vector (IV) that is sent as plain text with the encrypted message.

The RC4 variants targeted by Klein's attack simply concatenate the message-specific IV with the secret key. OSGP takes a different approach, in which the first 8 bytes of the BEK are XOR'ed with an 8-byte IV; in fact OSGP uses the message digest as IV. Attacking such RC4 implementations has already been discussed in [7]. The attack presented there could therefore also be applied to RC4 in OSGP. Our modification of Klein's approach, however, is much more effective. The attack of [7] depends on weak IVs. If the attacker cannot control the IV, the expected number of messages to derive one key byte is estimated to be 1,000,000. As the OSGP key consists of 16 key bytes, the required number of messages is thus two orders of magnitude larger than what the attack described here needs. There is a multitude of other attacks on RC4, but to best of our knowledge the modification of Klein's attack is best suited for the OSGP implementation.

For the description of the attack in Section 4.2, we set the following definitions. Let $m_1, m_2, \ldots, m_\tau$ be the set of τ plain text messages, whose cipher streams are intercepted during the attack. Message m_t consists of n_t many bytes:

$$m_t = (m_t[0], m_t[1], \ldots, m_t[n_t - 1])$$

Let furthermore d_t be the 8-byte digest created for message m_t.

$$d_t = (d_t[0], d_t[1], \ldots, d_t[7])$$

With BEK $= (bek[0], bek[1], \ldots, bek[15])$, the RC4 key for message m_t is:

$$k_t = (k_t[0], k_t[1], \ldots, k_t[15])$$

with

$$\begin{aligned} k_t[r] &= bek[r] \oplus d_t[r], && 0 \le r \le 7 \\ k_t[r] &= bek[r], && 8 \le r \le 15 \end{aligned}$$

Let $z_t = (z_t[0], z_t[1], \ldots)$ be the sequence of cipher stream bytes generated by RC4 from k_t. The encrypted message e_t corresponding to m_t then is

$$e_t = (e_t[0], e_t[1], \ldots, e_t[n_t - 1])$$

with

$$e_t[r] = m_t[r] \oplus z_t[r] \qquad 0 \le r \le n_t - 1.$$

To be able to talk about $X(r)$ for a specific message m_t, the formula introduced in Section 3.1 is extended to:

$$X_t(r) = \sum_{i=0}^{r} (k_t[i] + i) \mod 256$$

4.2 OSGP RC4 attack

We now come to the main contribution of this work, which is how to modify the attack presented by Klein [13] to work when the IV is XOR'ed with the secret key. In OSGP the secret key is the BEK and the IV is the digest of a message. The goal of the attack is to obtain BEK = ($bek[0]$, $bek[1], \ldots, bek[15]$) from τ recorded cipher streams $z_1, z_2, \ldots, z_\tau$ used for the encryption of τ different messages.

As no information about $bek[0]$ can be obtained from the cipher stream bytes, $bek[0]$ has to be guessed. In the following, we assume that the value of $bek[0]$ has already been guessed correctly. In the real implementation, the attack process is performed once for each of the 256 possible $bek[0]$ values, which is absolutely no obstacle for the attack.

Thanks to Roos, Jenkins and Klein, we know that for each cipher stream byte $z_t[r]$, there is one value with the increased probability of 0.0053 whereas all other values have the lower probability of 0.0039. This most occurring value is $r + 1 - X_t(r + 1)$.

As k_t is different for each message m_t, however, we cannot simply count the occurrences of $z_t[0]$ to $z_t[14]$ and thereby get $X_t(1)$ to $X_t(15)$, as it is the case for the attack shown in Klein's original work. We must start by treating each $z_t[0]$ as if it actually holds $X_t(1)$. From this putative $X_t(1) = k_t[0] + k_t[1] + 1$ value we calculate a putative $k_t[1]$, using the guessed $bek[0]$ for $k_t[0] = bek[0] \oplus d_t[0]$. From this putative $k_t[1]$ we can calculate a *candidate* $bek[1] = k_t[1] \oplus d_t[1]$. We count the occurrences of these candidate $bek[1]$ values over all messages. After a sufficient amount of messages, the value with most occurrences is the correct $bek[1]$ value. We do the same treating each $z_t[1]$ as a putative $X_t(2)$. Using the guessed $bek[0]$ and the currently most occurring $bek[1]$ candidate, we calculate the most occurring candidate for $bek[1]$, and so forth for the remaining BEK bytes.

We shall go through this procedure in detail by means of the pseudo code given in Figure 2. Line 1 starts the loop, each of whose iterations tries another $bek[0]$ value. At the end of each iteration (Line 10) we check, if the correct BEK has been found. This is practically done by using it to decrypt the intercepted encrypted messages and check if the result is plausible. If all values for $bek[0]$ have been tried and the correct BEK was not derived, the number of messages τ was not enough for the most probable value to emerge (Line 12) and more messages have to be accumulated.

Line 2 starts another loop, which goes through the first 15 cipher stream bytes $z[0], \ldots, z[14]$. In each iteration of this loop, one single BEK byte is derived. This is the major difference between this attack and the attack by Klein for which an analysis of all cipher stream bytes is conducted and all key bytes are derived at once. In the r'th iteration, we examine $z[r]$ of all messages to derive $bek[r + 1]$.

For each of the 256 possible values of $bek[r + 1]$ we define a variable to count its occurrences. In Line 3, these counters are reset to 0. Line 4 starts the loop going through the cipher streams of all τ messages.

```
    // The first BEK byte bek[0] must be brute-forced.
1:  for (bek[0] = 0x00; bek[0] ≤ 0xFF; bek[0]++) {
      // Each bek[r + 1] is derived separately by
      // examining the values of cipher stream byte z[r].
2:    for (r = 0; r ≤ 14; r++) {
        // Reset counters for bek[r + 1] candidate occurrences.
3:      for (k = 0; k ≤ 255; k++) counter[k] = 0;
        // Go through the cipher streams of all τ messages.
4:      for (t = 1; t ≤ τ; t++) {
          // Calculate a candidate for bek[r + 1] from z_t[r] and
          // bek[0], ..., bek[r] (calculated in previous iterations).
          // k_t[i] = bek[i] ⊕ d_t[i], if 0≤i≤7, else: k_t[i] = bek[i]
5:        bek[r + 1] = −z_t[r] − Σ_{i=0}^{r} (k_t[i] + i)%256;
6:        if (0 ≤ r ≤ 6) bek[r + 1] = bek[r + 1] ⊕ d_t[r + 1];
          // Count the occurrence of bek[r + 1] candidate value.
6:        counter[bek[r + 1]]++;
7:      }
        // Store most occurring bek[r + 1] candidate value
        // as the derived bek[r + 1] for coming iterations.
8:      bek[r + 1] = arg max_k counter[k];
9:    }
      // After all BEK bytes have been derived,
      // test if this BEK can decrypt all messages.
10:   if (BEK test is successful) return true;
11: }
    // If the BEK was not derived, more messages are needed.
12: return false;
```

Fig. 2. The OSGP RC4 attack.

In Line 5, we calculate a *candidate* value for $bek[r+1]$; *candidate*, because only with a probability of 0.0053 will we get the correct $bek[r+1]$ value from $z_t[r]$. With a probability of 0.9947 we will get an incorrect value, but each of the 255 incorrect values has only a probability of 0.0039, such that the correct $bek[r+1]$ value will eventually emerge as the most frequently occurring value after enough messages.

The formula for calculating the candidate for $bek[r+1]$ is derived as follows (all arithmetic operations are mod 256):

$$\begin{aligned}
& z_t[r] = r + 1 - X_t(r+1) \\
\Rightarrow & z_t[r] = r + 1 - \sum_{i=0}^{r+1}(k_t[i] + i) \\
\Rightarrow & z_t[r] = r + 1 - \left(\sum_{i=0}^{r}(k_t[i] + i) + k_t[r+1] + r + 1\right) \\
\Rightarrow & z_t[r] = -\sum_{i=0}^{r}(k_t[i] + i) - k_t[r+1] \\
\Rightarrow & k_t[r+1] = -z_t[r] - \sum_{i=0}^{r}(k_t[i] + i)
\end{aligned}$$

For $0 \leq r \leq 6$ we get $bek[k+1] = k_t[r+1] \oplus d_t[r+1]$. For $7 \leq r \leq 14$ we get $bek[k+1] = k_t[r+1]$ (Lines 5 and 6).

It is important when calculating $bek[r+1]$ with the above formula, that the sum $\sum_{i=0}^{r}(k_t[i] + i) = X_t(r)$ is not tried to be extracted directly from the previous cipher stream byte $z_t[r-1]$. This could occur to someone, as the most probable value of $z_t[r-1]$ is $r - X_t(r)$. However, this would only work, if both $z_t[r-1]$ and $z_t[r]$ would hold their most probable values for one and the same message, which is utmost unlikely ($P = 0.0053^2$). Instead we have to calculate $\sum_{i=0}^{r}(k_t[i] + i)$ from the BEK bytes $bek[0], \ldots, bek[r]$, we have derived in the previous iterations.

In Line 6, we increment the counter variable for the derived $bek[r+1]$ value by one, before proceeding to the next message. After all messages have been processed (Line 8), we regard the $bek[r+1]$ value with most occurrences as the correct value and store it permanently in $bek[r+1]$, before proceeding to derive the next BEK byte $bek[r+2]$ by examining $z[r+1]$ of all messages' cipher streams. If the number of messages was enough, we have all BEK bytes by the end of the algorithm.

As the BEK is the same for all devices in an OSGP smart grid, it can – once obtained – be used to decrypt all future intercepted messages, regardless of who is the sender or recipient. As the message digest does not rely on the BEK but on the OMAK, the attacker would at least not be able to forge messages. But being able to monitor all data traffic poses a severe threat to data confidentiality and privacy. The attacker could for example monitor the electricity consumption of a household, granting him detailed insights into the inhabitants' ways of living [16,17].

Treatment of special keys For about 15% of all possible BEKs, the attack needs to be further modified. These are BEKs of the kind that a j_ω at KSA

round ω (with $\omega \geq 8$) is the same as a previous j_α at round α (with $\omega > \alpha \geq 7$). In this case the desired $X(\alpha)$ value is swapped away from $S[\alpha]$ regardless of the used IV that only affects the first 8 key bytes. This BEK property is formalised as: there exists a sequence of BEK bytes

$$bek[x+1], bek[x+2], \ldots, bek[y-1], bek[y]$$

such that

$$\sum_{i=x+1}^{y} (i + bek[i]) = 0 \mod 256$$

Because in the i'th KSA round, j is incremented by $S[i] + bek[i]$, with $S[i] = i$ in most cases.

This is best brought to intuition by means of an example. For brevity, all numbers are written in hexadecimal format and all arithmetic operations are mod 256. Consider the following BEK with $bek[0]$ being the leftmost byte:

$$E5, A2, 8D, 25, DD, 99, A1, 03, F8, F7, 92, 5A, 97, 74, 8D, 4F$$

The first 8 bytes will be XOR'ed with the digest of each message, so they will always be different. The last 8 bytes – here $F8, F7, 92, \ldots, 4F$ – however, will be the same for every message. For simplicity let us neglect the digest and see what happens during KSA.

S is initialised as:

$$S_{-1} = (00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, \ldots)$$

In KSA round 0, we have $i_0 = 00$ and $j_0 = j_{-1} + S_{-1}[i_0] + bek[0] = 00 + 00 + E5$. Hence, $S[00]$ is swapped with $S[E5]$, leading to:

$$S_0 = (\mathbf{E5}, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, \ldots)$$

In the next round we have $i_1 = 01$ and $j_1 = j_0 + S_0[i_1] + bek[1] = E5 + 01 + A2 = 88$ leading to:

$$S_1 = (\mathbf{E5}, \mathbf{88}, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, \ldots)$$

Let us skip to the end of round 6, at which we have

$$S_6 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, 07, 08, 09, 0A, 0B, \ldots)$$

In round 7 we have $i_7 = 07$ and $j_7 = j_6 + S_6[i_7] + bek[7] = 65 + 07 + 03 = 6F$, such that $S[07]$ and $S[6F]$ are swapped:

$$S_7 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, 08, 09, 0A, 0B, \ldots)$$

Now consider round 8. We have $i_8 = 08$ and $j_8 = j_7 + S_7[i_8] + bek[8] = 6F + 08 + F8 = 6F$. Because of $08 + F8 = 00 \pmod{256}$ we get the same value for j_8 as we got for j_7, such that $S[08]$ and $S[6F]$ are swapped:

$$S_9 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, \underline{\mathbf{07}}, 09, 0A, 0B, \ldots)$$

Instead of $S[r-1] = j_r = X(r-1)$, as it was the case for all previous KSA rounds, we now have $S[08] = 07$. As $bek[8]$ is not subject to the XOR with the IV, we will get $j_8 = j_7$ for *every* message and hence $S[08] = 07$ during the PRGA for almost every message.

This makes it impossible that the actually desired value $X(8) = 6F$ can be extracted thanks to the Jenkins correlation. To overcome this, we have to modify our attack by the following rule: Let v be the putative $X(r+1)$ value with most occurrences derived from cipher stream byte $z[r]$ ($7 \leq r \leq 14$). Whenever $v < r+1$, we use $X(v)$ instead of $X(r+1)$.

In our example, we would see that the most occurring value derived from $z[7]$ is 07, which is smaller than 8. We thus use $X(07) = 6F$, that we have previously obtained from $z[6]$, as value for $X(08)$ as well.

Continuing our example, we see that this method also works if there are more such special key bytes. In round 9 we get $i_9 = 09$ and $j_9 = j_8 + S_8[i_9] + bek[9] = 6F + 09 + F7 = 6F$. Because of $09 + F7 = 00 \pmod{256}$ we get the same value for j_9 as we got for j_8 *and* j_7. Thus, $S_8[09] = 09$ and $S_8[6F] = 08$ are swapped, leading to:

$$S_9 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, \underline{\mathbf{07}}, \underline{\mathbf{08}}, 0A, 0B, \ldots)$$

Following the new rule, we see that the most occurring value derived from $z[8]$ is 08, which is smaller than 9. We thus use $X(8) = 6F$ to derive $bek[9]$.

So far, the sequence of BEK bytes fulfilling the formula above only consisted of two bytes each. Let us look at a variation of our example, in which the sequence consists of more bytes. Let us therefore assume $bek[8] = FF$ and $bek[9] = F0$:

$$E5, A2, 8D, 25, DD, 99, A1, 03, FF, F0, 92, 5A, 97, 74, 8D, 4F$$

Up to KSA round 7, it is the same as before:

$$S_7 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, 08, 09, 0A, 0B, \ldots)$$

Now, in round 8 we get $i_8 = 08$ and $j_8 = j_7 + S_7[i_8] + bek[8] = 6F + 08 + FF = 76$, such that $S[08]$ and $S[76]$ are swapped, leading to:

$$S_8 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, \mathbf{76}, 09, 0A, 0B, \ldots)$$

This is not yet problematic, but when we continue to round 9 we get $i_9 = 09$ and $j_9 = j_8 + S_8[i_9] + bek[9] = 76 + 09 + F0 = 6F$. Because of $08 + FF + 09 + F0 = 00 \pmod{256}$ we get the same value for j_9 as we got for j_7. Thus, $S_8[09] = 09$ and $S_8[6F] = 07$ are swapped, leading to:

$$S_9 = (\mathbf{E5}, \mathbf{88}, \mathbf{17}, \mathbf{3F}, \mathbf{20}, \mathbf{BE}, \mathbf{65}, \mathbf{6F}, \mathbf{76}, \underline{\mathbf{07}}, 0A, 0B, \ldots)$$

But this is recognised by the new rule as well, such that $X(7) = 6F$ is used as $X(9)$ to derive $bek[9]$.

5 Experimental results

We evaluated the attack by simulating messages as they are exchanged in an OSGP smart grid. First, a random 16-byte BEK is created. Then, the simulator starts generating OSGP messages, encrypting them with this BEK and analyses the resulting cipher stream until the BEK is successfully derived. Then, a new random BEK is created and the procedure starts anew.

The simulated OSGP message type was a *partial table read* request and the corresponding response, defined in ETSI GS OSG 001 [2]. As OSGP treats a message response (sent back from the device) as part of the same RC4 cipher stream as the request (sent from the DC), the request and the response are considered as one message from the attacker's perspective. The messages were thus of the following form:

- **Request** message ID (1 byte): 0x3F
  The value 0x3F indicates that this request is a partial table read and is hence the same for all messages.
- Table ID (2 bytes): 0x00, 0x34
  Reading from OSGP basic table 52 (device clock).
- Table offset (3 bytes): 0x00, 0x00, 0x00
  Tables can be read from a certain offset on.
  Here: 0 means no offset.
- Count (2 bytes): 0x00, 0x06
  Here: read the next 6 bytes after offset.
- Sequence number (4 random bytes)
  Each OSGP message has a unique sequence number consisting of 4 bytes. In a real OSGP grid, devices accept only sequence numbers corresponding to their respective internal states. We gave each simulated message a random sequence number, which is hence the only aspect by which the message requests differed.
- **Response** answer (1 byte): 0x00
  0x00 stands for OK.
- Count (2 bytes): 0x00, 0x06
  Always the same as in request.
- Data (6 random bytes)
  The data has the amount of bytes requested by count. In this partial table read, data would contain the device clock. We gave each simulated message 6 random bytes.

Notice, that in order to mount the attack on the 16 byte BEK, only the first 15 cipher stream bytes are needed. Hence, the last six message bytes are not even relevant for the experiment.

We simulated the attacks on 145,000 random BEKs, each of which could be derived with less than 415,000 messages. Figure 3 shows the results in form of a histogram. Each column represents the percentage of simulated BEKs that could be derived with the respective number of messages. The leftmost column

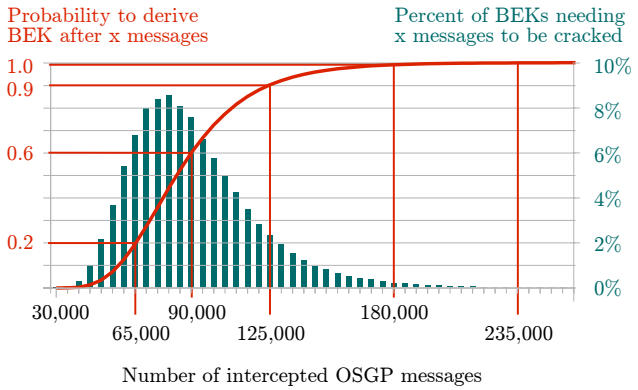

Fig. 3. Probability to derive a random BEK (and the percentage of BEKs derived) with a certain amount of messages.

representing 30,000 messages is almost not visible, as only 24 BEKs could be derived with 30,000 messages. The mean value of required messages was 90,773. Only 5 BEKs needed more than 300,000 messages. Figure 3 also shows the probability for successfully deriving a random BEK with a certain amount of messages. 20% of the 145,000 simulated BEKs could be derived with 65,000 or less messages. Hence, the probability of deriving a random BEK with 65,000 messages is 0.2. For 90,000 messages, the probability is 0.6, for 125,000 messages 0.9, for 180,000 messages 0.99 and for 235,000 messages 0.999.

The results showed that about 15% of the created random BEKs had special key bytes as described in Section 4.2. They were successfully derived at the same rate as BEKs without special key bytes.

The runtime to derive a BEK was proportional to the number of processed messages. On a single 3.40 GHz Intel i7-4770 core, we measured about 0.023 milliseconds CPU-time per message, which included the creation of the message, its digest and the RC4 cipher stream. The processing of 90,000 messages could thus be simulated in 2 seconds. As we did not simulate the brute-forcing of $bek[0]$ and considered it known, we have to multiply the required time by 128 for the average case, which is still less than 5 minutes.

6 Hampering the attack

To hamper the presented attack without altering the specification of ETSI GS OSG 001 [2], the following makeshift measures can be suggested until an update of OSGP will have taken place.

The attack can be hampered by avoiding data traffic. If one cipher stream can be captured per second, it would only take 25 hours to accumulate 90,000 messages. If, on the other hand, only one cipher stream byte per hour can be captured, the accumulation of 90,000 messages would take over 10 years. Notice,

that an attacker cannot actively evoke new message from an OSGP device. Only authenticated messages from the DC lead to a device responding. Replayed messages are only answered with an unencrypted "invalid sequence number" message. The operator of an OSGP smart grid should make an estimate of how many cipher streams an attacker could obtain within a given time. The domain-wide OMAK and with it the BEK should be changed in time intervals shorter than that. It is the operator's decision, how many collected cipher streams are tolerable based on the probabilities shown in Figure 3.

To avoid the guessing of message plain text bytes, the operator could add randomness to the times at which regular requests are transmitted and intersperse them with dummy requests reading from random tables with random offsets. Furthermore, OSGP devices should be designed such that it is not just impossible to extract the OMAK or BEK but also the cipher stream bytes.

7 Conclusion

We presented an approach exploiting known RC4 weaknesses to derive the secret BEK key used in the open smart grid protocol (OSGP), which breaks its data confidentiality and thus privacy of the participants. By obtaining the BEK, which is by design identical for all OSGP devices within a network, the complete communication between the data concentrator and the devices can be decrypted. The attack exploits different known biases of the RC4 variant used in OSGP. A novel method to handle special key swaps during the RC4 key scheduling algorithm was presented. Experiments showed that every possible BEK can be derived requiring on average 90,000 cipher streams with negligible CPU run times. It shows that the security in a widely applied protocol (approx. 4 million deployed devices) has noteworthy security issues.

References

1. ETSI TS 103 908 v1.1.1 (2011-10), PowerLine Telecommunications (PLT). http://www.etsi.org/deliver/etsi_ts/103900_103999/103908/01.01.01_60/ts_103908v010101p.pdf, Oct. 2011.
2. ETSI GS OSG 001 v1.1.1 (2012-01), Open Smart Grid Protocol (OSGP). http://www.etsi.org/deliver/etsi_gs/OSG/001_099/001/01.01.01_60/gs_osg001v010101p.pdf, Jan. 2012.
3. ISO/IEC 14908-1, Information technology – Control network protocol – Part 1: Protocol stack. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=60203, 2012.
4. ESNA press release. Industry leaders bring open smart grid protocol (OSGP) to market. http://esna.org/uploads/2010-09-22%3b%20ESNA%20press%20release.pdf, Sept. 2010.
5. ETSI press release. ETSI approves open smart grid protocol (OSGP) for grid technologies. http://www.etsi.org/news-events/news/382-news-release-18-january-2012, Jan. 2012.

6. X. Fang, S. Misra, G. Xue, and D. Yang. Smart grid – the new and improved power grid: A survey. *Communications Surveys & Tutorials, IEEE*, 14(4):944–980, 2012.
7. S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In S. Vaudenay and A. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2001.
8. J. Gao, Y. Xiao, J. Liu, W. Liang, and C. P. Chen. A survey of communication/networking in smart grids. *Future Gener. Comput. Syst.*, 28(2):391–404, Feb. 2012.
9. S. S. Gupta. *Analysis and Implementation of RC4 Stream Cipher*. PhD thesis, Indian Statistical Institute, 2013. Available at: http://souravsengupta.com/pub/phd_thesis_2013.pdf.
10. R. J. Jenkins Jr. ISAAC and RC4. Published online: http://burtleburtle.net/bob/rand/isaac.html, 1996.
11. P. Jovanovic and S. Neves. Practical cryptanalysis of the open smart grid protocol. In *Fast Software Encryption, FSE*, 2015.
12. I. Khanna. Smart grid application: Peak demand management trial - the western australian experience. In *Innovative Smart Grid Technologies Asia (ISGT), 2011 IEEE PES*, pages 1–6, Nov 2011.
13. A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, Sept. 2008. Published online in Febr. 2006: http://cage.ugent.be/~klein/RC4/RC4-en.ps.
14. K. Kursawe and C. Peters. Structural weaknesses in the open smart grid protocol. *IACR Cryptology ePrint Archive*, 2015:88, 2015.
15. I. Mantin. Analysis of the stream cipher RC4. Master's thesis, The Weizmann Institute of Science, Israel, 2001. Available at: http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Mantin1.zip.
16. F. Mármol, C. Sorge, O. Ugus, and G. Pérez. Do not snoop my habits: preserving privacy in the smart grid. *Communications Magazine, IEEE*, 50(5):166–172, May 2012.
17. P. McDaniel and S. McLaughlin. Security and privacy challenges in the smart grid. *Security Privacy, IEEE*, 7(3):75–77, May 2009.
18. A. Roos. A class of weak keys in the RC4 stream cipher. Posted in sci.crypt newsgroup: http://groups.google.com/groups?selm=43u1eh$1j3@hermes.is.co.za, Sept. 1995. See also: http://marcel.wanda.ch/Archive/WeakKeys.
19. T. Ryberg. Smart metering in North America and Asia-Pacific. Technical report, Berg Insight, 2011. Abstract available at http://www.berginsight.com/ReportPDF/ProductSheet/bi-smnaap2-ps.pdf.
20. T. Ryberg. Smart metering in Europe. Technical report, Berg Insight, 2013. Abstract available at http://www.berginsight.com/ReportPDF/ProductSheet/bi-sm10-ps.pdf.
21. P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Discovery and exploitation of new biases in RC4. In *Proceedings of the 17th International Conference on Selected Areas in Cryptography*, SAC'10, pages 74–91, Berlin, Heidelberg, 2011. Springer-Verlag.
22. W. Wang and Z. Lu. Survey cyber security in the smart grid: Survey and challenges. *Computer Networks*, 57(5):1344–1371, Apr. 2013.
23. J. Zheng, D. Gao, and L. Lin. Smart meters in smart grid: An overview. In *Green Technologies Conference, 2013 IEEE*, pages 57–64, April 2013.

A RC4

RC4 is one of the most widely used software stream ciphers. Because of its structural simplicity and computational efficiency it has been applied in many industrial standards such as Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) or Wi-Fi Protected Access (WPA) and its predecessor Wired Equivalent Privacy (WEP) [9].

Originally, RC4 was invented by Ron Rivest for RSA Data Security in 1987 where it was handled as a trade secret until its source code was anonymously posted to the electronic mailing list *Cypherpunks* in 1994. Since then, it has been thoroughly examined within the cryptography research community and several weaknesses could be identified, e.g. [7,9,21]. In spite of these weaknesses, however, it has so far been possible to compensate them by changing the RC4 implementation in appropriate ways. OSGP uses its own version of RC4 hoping to avoid known weaknesses, but as this work shows there are weaknesses of RC4 to be exploited in the OSGP implementation as well. In the following we describe how RC4 works.

A.1 Stream cipher encryption and decryption

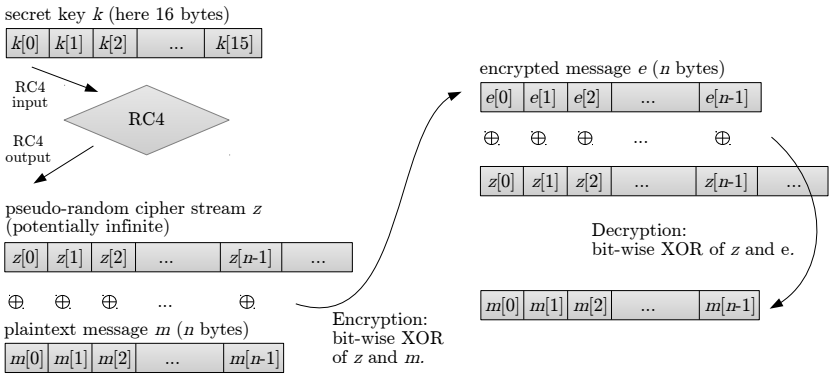

Fig. 4. Encryption and decryption with stream ciphers.

Figure 4 visualises the encryption and decryption flow of stream ciphers in general. Encryption is done by a bit-wise XOR between the plain text message m and the cipher stream z. z is generated depending on the secret key k. The RC4-specific generation of z from k is discussed in Section A.2.

Due to the pseudo-random nature of z, it is not possible to reconstruct k from z except by trying all possible values of k (brute force attack), which is not practical for a sufficiently large k. In most RC4 implementations (and so in OSGP) k has 128 bits (16 bytes) and trying all 2^{128} possible values for 16 bytes would last over $2.5 \cdot 10^{25}$ years on a single 3.40 GHz Intel i7-4770 processor.

We address the i'th byte of k as $k[i]$, just as $z[i]$ and $m[i]$ address the i'th bytes of the cipher stream z and the plain text message m. Thus, the i'th byte of the encrypted message e becomes $e[i] = m[i] \oplus z[i]$. An XOR between two bytes means that a bit-wise XOR is done between their 8 bits; e.g. $00110101 \oplus 10010110 = 10100011$.

If a potential attacker intercepts only e, m cannot be derived without knowing z. The legitimate recipient of e, however, knows k and can thus generate z to decrypt e by performing the XOR operation $z \oplus e = m$. This is possible because of $x \oplus x = 0$ and $0 \oplus x = x$ for any $x \in \{0, 1\}$; hence $z \oplus e = z \oplus z \oplus m = 0 \oplus m = m$.

If two messages m_1 and m_2 were encrypted with the same k and hence the same z, an attacker could perform the XOR operation $e_1 \oplus e_2 = m_1 \oplus z \oplus m_2 \oplus z = m_1 \oplus m_2 \oplus z \oplus z = m_1 \oplus m_2$. From knowing only $m_1 \oplus m_2$ it is generally not possible to derive m_1 or m_2. However, it is possible to know which bits are equal and which are inverse between m_1 and m_2, allowing for deductions of the message content. If one messages contains redundancy or otherwise predictable bits, it is thus also possible to obtain knowledge about the other message. It is therefore imperative to never reuse the same key. In most RC4 implementations this is achieved by combining the secret key with a fresh random bit string (initialisation vector) for each message. This initialisation vector is sent unencrypted with the encrypted message, such that the receiver can again combine it with the secret key to decrypt the message. k and z are thus different for each message.

A.2 RC4 cipher stream generation

In this section we describe how RC4 generates the cipher stream z from k following the notation of [9]. The key length is variable. For sake of simplicity we assume k to consist of 16 bytes, which is also the case for the OSGP RC4 implementation. In addition to the key k, RC4 has three internal data structures. The first is an array S of 256 *state variables*, each consisting of one byte: $S = (S[0], S[1], S[2], \ldots, S[255])$. The other two are the *pointer variables* i and j, also consisting of one byte each. Initially, these variables are set to $i = j = 0$ and $S[x] = x$ for $0 \leq x \leq 255$, before RC4 starts its operation divided into two phases: the *key scheduling algorithm* (KSA) and the *pseudo-random generation algorithm* (PRGA).

Both KSA and PRGA have several *rounds*. The KSA has 256 rounds in which the initial values of S are shuffled depending on k. Taking this shuffled S as input, PRGA then produces one cipher stream byte $z[r]$ per round r and can potentially run ad infinitum. In this paper, it will always be clear by the context whether we are talking about KSA or PRGA. We therefore denote the values of $S[x]$, i and j at round r of either KSA or PRGA as $S_r[x]$, i_r, j_r.

Figure 5 shows the pseudo code of KSA. Lines 1 and 2 initialise the variables as stated above. In Lines 4 and 5, i and j are updated. i is set to r, such that after all rounds each $S[i]$ has been subject to the swap with $S[j]$ (Lines 6 and 7). Figure 6 shows the pseudo code of PRGA. Similar to KSA, i and j are updated in each round (Lines 3 and 4) and a swap of $S[i]$ and $S[j]$ takes place (Lines 5 and 6), before one byte of the cipher stream is generated per round.

```
    // Input: secret key k.
    // Output: Shuffled S depending on k.

    // Initialise variables:
1:  S_{-1} = (0, 1, 2, ..., 255);
2:  i_{-1} = j_{-1} = 0;
    // Shuffle S:
3:  for (r = 0; r ≤ 255; r++) {
4:      i_r = r;
        // j += S[i] + k[i%16] % 256;
5:      j_r = j_{r-1} + S_{r-1}[i_r] + k[i_r%16] % 256;
        // Swap S[i] and S[j].
6:      S_r[j_r] = S_{r-1}[i_r];
7:      S_r[i_r] = S_{r-1}[j_r];
8:  }
```

Fig. 5. The RC4 key scheduling algorithm (KSA).

```
    // Input: S_{-1} generated by KSA.
    // Output: cipher stream z, one byte per round

    // Initialise variables:
1:  i_{-1} = j_{-1} = 0;
    // Generate cipher stream z:
2:  for (r = 0; ; r++) {
        // i++;
3:      i_r = r % 256;
        // j += S[i] % 256;
4:      j_r = j_{r-1} + S_{r-1}[i_r] % 256;
        // Swap S[j] and S[i].
5:      S_r[j_r] = S_{r-1}[i_r];
6:      S_r[i_r] = S_{r-1}[j_r];
        // Output of cipher stream byte.
        // z[r - 1] = S[S[i] + S[j] % 256]
7:      z[r] = S_r[S_r[i_r] + S_r[j_r] % 256];
8:  }
```

Fig. 6. The RC4 pseudo-random generation algorithm (PRGA).