Practical Cryptanalysis of the Open Smart Grid Protocol

Philipp Jovanovic[1] and Samuel Neves[2]

[1] University of Passau, Germany
jovanovic@fim.uni-passau.de
[2] University of Coimbra, Portugal
sneves@dei.uc.pt

Abstract. This paper analyses the cryptography used in the Open Smart Grid Protocol (OSGP). The authenticated encryption (AE) scheme deployed by OSGP is a non-standard composition of RC4 and a home-brewed MAC, the "OMA digest".
We present several practical key-recovery attacks against the OMA digest. The first and basic variant can achieve this with a mere 13 queries to an OMA digest oracle and negligible time complexity. A more sophisticated version breaks the OMA digest with only 4 queries and a time complexity of about 2^{25} simple operations. A different approach only requires one arbitrary valid plaintext-tag pair, and recovers the key in an average of 144 *message verification* queries, or one ciphertext-tag pair and 168 *ciphertext verification* queries.
Since the encryption key is derived from the key used by the OMA digest, our attacks break both confidentiality and authenticity of OSGP.

1 Introduction

Authenticated encryption [7] (AE) is the standard technology to protect data that needs to be sent over unsecured communication channels and is deployed in countless applications and protocols, such as (D)TLS, SSH and IPSec. In comparison to regular symmetric encryption schemes, AE not only ensures *privacy* of the data but also guarantees *integrity* and *authenticity*. Unfortunately, failures in the design and implementation of authenticated encryption schemes are a common sight and there are numerous examples. To name just a few (see also [9]):

- Vaudenay's 2002 CBC *padding oracle attack* on MAC-then-encrypt AE modes allows an active adversary to decrypt messages without access to the secret key [30]. This attack stemmed from the authenticity verification leaking whether the decrypted message was adequately padded. Over the years, this strategy has been used quite successfully against TLS [4,10,12,26].
- In 2007, an attack [29] on the Wired Equivalent Privacy (WEP) standard, used in many 802.11 Wi-Fi networks, allowed to recover the secret key within minutes from a few thousand intercepted messages. The attack exploited weaknesses in RC4.

– In 2009, Albrecht, Paterson, and Watson [2] exploited a flaw in the SSH protocol and its OpenSSH implementation, when coupled with a block cipher in CBC mode. The attack allowed an adversary to recover 14 plaintext bits with probability 2^{-14} or 32 plaintext bits with probability 2^{-18}.
– In 2012, a flaw was uncovered in EAXprime [5], an AE block cipher mode derived from EAX [8], standardized as ANSI C12.22-2008 for Smart Grid applications, and also subject of a forthcoming NIST standard. The flaw facilitates forgery, distinguishing, and message-recovery attacks [25].

In this paper, we investigate another flawed authenticated encryption scheme, which is deployed in the Open Smart Grid Protocol (OSGP) [15]. The latter is an application layer communication protocol for smart grids built on top of the ISO/IEC 14908-1 protocol stack [21], has been developed by the Energy Service Network Association (ESNA), and is a standard of the European Telecommunications Standards Institute (ETSI) since 2012 [1]. According to estimations, OSGP-based smart meters and devices are deployed in over 4 million devices worldwide as of 2015, making OSGP one of the most widely used network protocols for smart grid applications.

Our Results. Table 1 summarises the results of the different attacks on the authenticated encryption scheme of OSGP and also lists the corresponding sections where the attacks are described. While the attacks have various tradeoffs between the number of oracle queries and the computational complexity, each constitutes a complete break of the OSGP AE scheme. We also want to highlight the fact that the attacks from Section 3.4 are particularly powerful in the context of the protocol: verification oracles are easy to come across and the attack in its XOR variant does not need to know plaintext at all, since differences can be injected directly into the ciphertext. In other words, this is a practical attack on the AE scheme of OSGP and completely compromises its security.

Related Work. In late 2013, Kursawe and Peters independently analysed OSGP and identified several security flaws, some of which overlap with our own findings [22]. Their work gives a good overview on the various security flaws and shows how they can be exploited to mount some basic attacks on OSGP's cryptographic infrastructure. We, on the other hand, focus on the digest function in more detail and, as a consequence, are able to further move the attacks into practicality. We note that our analysis has been performed solely against the OSGP specification [15] and not against any deployed devices.

Outline. The paper is organised as follows. Section 2 introduces notation and the cryptographic infrastructure used in the Open Smart Grid Protocol. In Section 3, we give a detailed analysis of the said AE scheme. We start with some basic attacks that already allow recovery of the entire secret key but are not feasible within the scope of the protocol. Based on that we describe further improvements which eventually allow us to mount fast forgery attacks on the OSGP AE scheme

Table 1. Required number of queries and expected complexity for the attacks of Section 3, with varying time-query tradeoff parameter B. The abbreviation KP+ means *known-plaintext with common prefix*, CP denotes *chosen-plaintext*, CC stands for *chosen-ciphertext*, and TG and TV denote *tag-generation* and *tag-verification* oracles, respectively.

Attack	B	Queries	Complexity	Type	Oracle
§3.1	1	13	$2^{3.58}$	CP	TG
	2	7	$2^{10.58}$		
	3	5	$2^{18.00}$		
	4	4	$2^{25.58}$		
	5	4	$2^{33.58}$		
	6	3	$2^{41.00}$		
§3.2	1	24 / 13	$2^{10.58}$	KP+ / CP	TG
	2	12 / 7	$2^{17.58}$		
	3	8 / 5	$2^{25.00}$		
	4	6 / 4	$2^{32.58}$		
	5	6 / 4	$2^{40.32}$		
	6	4 / 3	$2^{48.58}$		
§3.4 (XOR)	—	≈ 168	≈ 168	CP / CC	TV
§3.4 (Additive)	—	≈ 144	≈ 144	CP	

and furthermore enable recovery of the complete secret key and in this case all within the context of the protocol. Finally, Section 4 concludes the paper.

2 Preliminaries

2.1 Notation

An n-bit string x is an element of $\{0,1\}^n$. For $n = 8$ we call x a byte. The size of x in bits is denoted by $|x|$. Concatenation of bit strings is denoted by $\|$. Given a vector of bit strings $(x_0, \ldots, x_{n-1})$, we denote by $x_{i,j}$ the jth bit of the ith word where $0 \leq i \leq n-1$. When interpreting bit strings as integers we always use little-endian format and denote them in hexadecimal format using `typewriter`. A bit string consisting of n zeros is denoted by 0^n. A cyclic rotation of a bit string x by m bits to the left and right is denoted by $x \lll m$ and $x \ggg m$, respectively. The difference of two bit strings x and x' with respect to XOR is denoted by Δx, whereas a difference with respect to addition modulo 2^n is denoted by $\Delta^{\boxminus} x$.

2.2 The Cryptographic Infrastructure of OSGP

In this paper, we focus solely on OSGP's cryptographic infrastructure, and not on the protocol itself. The high-level structure of OSGP's authenticated encryption (AE) scheme is depicted in Figure 1.

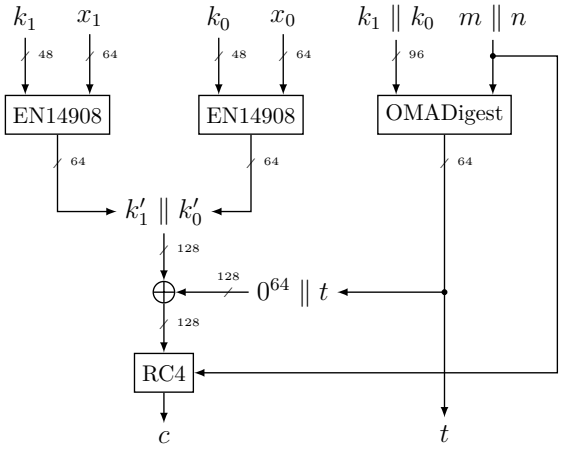

Fig. 1. The OSGP AE scheme. Notation: $x_0 = \{\texttt{81}, \texttt{3F}, \texttt{52}, \texttt{9A}, \texttt{7B}, \texttt{E3}, \texttt{89}, \texttt{BA}\}$, $x_1 = \{\texttt{72}, \texttt{B0}, \texttt{91}, \texttt{8D}, \texttt{44}, \texttt{05}, \texttt{AA}, \texttt{57}\}$, $k = k_1 \parallel k_0$: Open Media Access Key (OMAK), m : message, n : sequence number, t : authentication tag, $k' = k'_1 \parallel k'_0$: Base Encryption Key (BEK), c : ciphertext.

The OSGP AE scheme is based on three algorithms: the EN 14908 algorithm[3], the stream cipher RC4 and the so-called OMA digest, a message authentication code (MAC). These three algorithms are combined in a mixture of the generic composition [7] approaches *MAC-and-encrypt* and *MAC-then-encrypt* to form an authenticated encryption scheme, see again Figure 1. We note that, while the OMA digest is described in the OSGP specification [15], public information on the EN 14908 algorithm, specified in ISO/IEC 14908-1 [21], is hard to come by. All information on the latter was retrieved from the OSGP specification [15] and the related standard ISO/IEC CD 14543-6-1 [20, p.232] which, like ISO/IEC 14908-1 and a few other standards [6,19,28], is also a direct descendant of LonTalk [13].

The security of OSGP's AE scheme depends on the 96-bit *Open Media Access Key* (OMAK) $k = k_1 \parallel k_0$ from which all other key material is derived. The OMAK is usually unique to a device but not hardcoded and can be changed, often to be shared with other devices under the same concentrator [15, §7.1]. Two things are derived from the OMAK: firstly, a so-called *Base Encryption Key* (BEK) $k' = k'_1 \parallel k'_0$ is computed [15, §7.3] which is a 128-bit key forming the basis for the RC4 encryption key. The BEK is constructed[4] using the EN 14908 algorithm

[3] The OSGP specification describes EN 14908 as an encryption algorithm, but it is clearly nothing of the sort. We therefore only talk about the *EN 14908 algorithm* in this work.

[4] The OSGP specification is rather unclear on how the BEK is derived. The presented description is based on our investigations also involving other standards [20, p.232]. The key observation here is that the BEK derived from the OMAK. The concrete realisation is not too important, though, and is only described for the sake of completeness.

which appears to have been the basis for the OMA digest but uses smaller 48-bit keys and processes message bytes in reversed order. The EN 14908 algorithm is applied to each of the halves k_0 and k_1 of the OMAK and the two constants $x_0 = \{81, 3F, 52, 9A, 7B, E3, 89, BA\}$ and $x_1 = \{72, B0, 91, 8D, 44, 05, AA, 57\}$. The two 64-bit results are then concatenated to form k', see Figure 1. Note that the BEK only depends on the OMAK and is thus fixed as long as k remains unchanged.

Secondly, an authentication tag t is produced using the OMA digest on the message m concatenated with a sequence number n and the OMAK k. Let l denote the size of $m \parallel n$ in bytes. The OMA digest starts with its 8-byte internal state $a = (a_0, \ldots, a_7)$ set to zero. First, $m \parallel n$ is zero-padded to a multiple of 144 bytes, meaning

$$m' = m \parallel n \parallel 0^{-l \bmod 144} .$$

Let $m' = m'_0 \parallel \cdots \parallel m'_{143}$ denote the first, and possibly only, 144-byte block of the message. The internal state is updated continuously using a nonlinear function $f_{b,c}$ where $b = k_{i \bmod 12,7-j}$ is a key bit and $c = j$ is the current position in the state. Its specification is as follows:

$$f_{b,c}(x, y, z) = \begin{cases} y + z + (\neg(x + c)) \lll 1 & \text{if } b = 1 \\ y + z - (\neg(x + c)) \ggg 1 & \text{otherwise.} \end{cases}$$

In order to update state element a_j, the function f takes, for $0 \le i \le 17$ and $7 \ge j \ge 0$, two adjacent state elements a_j and $a_{j+1 \bmod 8}$ and a message-byte m'_{8i+7-j} as input, i.e., $a_j = f_{k_{i \bmod 12,7-j},j}(a_j, a_{j+1 \bmod 8}, m_{8i+7-j})$, and depending on the value of the key bit $k_{i \bmod 12,7-j}$ one of the two branches depicted above is evaluated. The next 144-byte message block is processed similarly, with the initial internal state carried over from the previous block. The complete pseudocode of the OMA digest is shown in Algorithm 1 and a visualisation of its innermost loop, where the message bytes are processed, is given in Figure 2. For the reference implementation we refer to [15, Annex E].

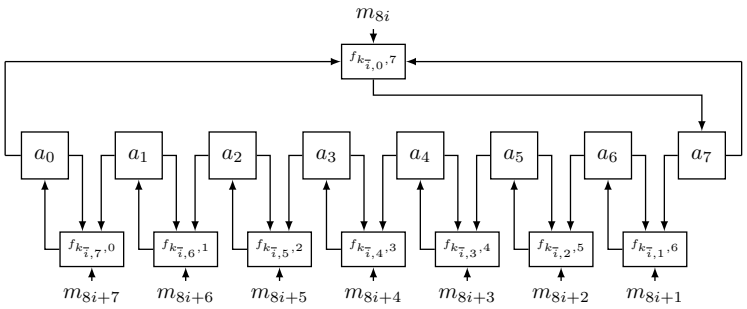

Fig. 2. Data processing (right-to-left) in the OMA digest, with $\bar{i} = i \bmod 12$.

After the tag generation, t is XORed into the lower half of the BEK k' which then produces the final 128-bit RC4 encryption key $k'' = k'_1 \parallel (k'_0 \oplus t)$, see

```
Function OMADigest(m,k)
    a ← (0,0,0,0,0,0,0,0)
    m ← m ‖ 0^{−|m| mod 144}
    foreach 144-byte block b of m do
        for i ← 0 to 17 do
            for j ← 7 to 0 do
                if k_{i mod 12,7−j} = 1 then
                a_j ← a_{(j+1) mod 8} + b_{8i+(7−j)} + (¬(a_j + j)) ⋘ 1
                else  a_j ← a_{(j+1) mod 8} + b_{8i+(7−j)} − (¬(a_j + j)) ⋙ 1
            end
        end
    end
    return a
```

Algorithm 1: The OSGP OMA digest.

again Figure 1. This measure is intended to provide RC4 with ever-changing key material, thus producing a fresh keystream with every new message, since, according to the OSGP specification, the sequence number n, which is appended to m, is continuously increased.

Sequence numbers are shared between sender and receiver in OSGP. The receiver of a message verifies that the correct sequence number was appended to the latter. Messages with sequence numbers in the range $\{n, \ldots, n+8\}$ are accepted as valid requests. If a message with sequence number $n-1$ is received, then the recipient does not execute the request but instead re-sends the answer of the (previously executed) request of number $n-1$. Sequence numbers outside of this range trigger an error and the OSGP device replies with a failure code and the correct sequence number. More details on the handling of sequence numbers can be found in [15, §9.7].

After the setup phase is finished, k'' is used to encrypt $m \parallel n$ via RC4 to obtain the ciphertext c. Finally, $c \parallel t$ is transmitted. Messages $m \parallel n$ processed in OSGP are allowed to have a maximum size of 114 bytes [15, §9.2]. This complicates some attacks that require up to 136-byte messages. Nevertheless, we will also describe scenarios that respect this message size limit.

3 Analysis

OSGP uses RC4 for encryption without discarding any initial bytes. RC4 has known statistical key- and plaintext-recovery attacks, and these have been shown to be practically feasible [3,16,17,18,27,29,31]. However, in this work we do not focus on RC4, but instead on the OMA digest, see Algorithm 1.

The OMA digest algorithm presents multiple flaws. Firstly, it uses a simple zero byte message padding, which results in messages with any number of trailing zeroes sharing the same tag. Secondly, given a tuple (a, m, k) where a is the OMA digest's state or authentication tag, m a message and k the OMAK, the

function is fully reversible (see Algorithm 2) which is a very useful property for the attacks presented in Sections 3.1 and 3.2. Likewise, it is also possible to take an arbitrary internal state, and continue to process it as if to resume a partially digested message. This is depicted in Algorithm 3.

Function `OMABackward`(a,m,k,n)
  `// Assumes` $|m| \leq 144$.
  $m \leftarrow m \parallel 0^{-|m| \bmod 144}$
  for $l \leftarrow 0$ **to** $n-1$ **do**
    $i, j \leftarrow \lfloor l/8 \rfloor, l \bmod 8$
    if $k_{(17-i) \bmod 12, 7-j} = 1$ **then** $x \leftarrow (a_j - a_{(j+1) \bmod 8} - m_{143-8i-j}) \ggg 1$
    else $x \leftarrow (a_{(j+1) \bmod 8} + m_{143-8i-j} - a_j) \lll 1$
    $a_j \leftarrow \neg x - j$
  end
  return a

Algorithm 2: The "backward" OSGP OMA digest, reverting the internal state back by n message bytes.

Function `OMAForward`(a,m,k,n)
  `/* Essentially Algorithm 1, but start at byte` m_n `with a known state` a`, and assume` $|m| \leq 144$. `*/`
  $m \leftarrow m \parallel 0^{-|m| \bmod 144}$
  for $l \leftarrow n$ **to** 143 **do**
    $i, j \leftarrow \lfloor l/8 \rfloor, 7 - l \bmod 8$
    if $k_{i \bmod 12, 7-j} = 1$ **then** $a_j \leftarrow a_{(j+1) \bmod 8} + m_{8i+7-j} + (\neg(a_j + j)) \lll 1$
    else $a_j \leftarrow a_{(j+1) \bmod 8} + m_{8i+7-j} - (\neg(a_j + j)) \ggg 1$
  end
  return a

Algorithm 3: The "forward" OSGP OMA digest, starting with a known initial state and processing message bytes starting at position n.

3.1 Chosen-Plaintext Key Recovery Attacks

Let $a = (a_0, \ldots, a_7)$ denote the 8-byte internal state of the OMA digest. The attacks discussed below use chosen 144-byte messages $m = m_0 \parallel \cdots \parallel m_{143}$[5], and exploit differential weaknesses in the OMA digest.

Bitwise Key Recovery. The first attack recovers the key one bit at a time by differential cryptanalysis. Specifically, we exploit the XOR-differential $(\Delta m_i, \Delta a_j) =$

[5] For simplicity, we use 144-byte messages throughout this section. Note, however, that the presented attacks use messages which are never longer than 136 bytes.

$(\texttt{80}, \texttt{80})$, where Δm_i and Δa_j denote input and output differences, respectively, for $j = 7 - i \bmod 8$. The output difference is obtained immediately after processing message byte m_i (see Algorithm 1) and can be written as

$$\begin{aligned} & f_{k,j}(a_j, a_{j+1 \bmod 8}, m_i \oplus \texttt{80}) \\ = \; & a_{j+1 \bmod 8} + (m_i \oplus \texttt{80}) \pm (\texttt{FF} \oplus (a_j + j) \lll r) \\ = \; & (a_{j+1 \bmod 8} + m_i \pm (\texttt{FF} \oplus (a_j + j) \lll r)) \oplus \texttt{80} \\ = \; & f_{k,j}(a_j, a_{j+1 \bmod 8}, m_i) \oplus \texttt{80} \end{aligned}$$

where the rotation offset $r \in \{1, 7\}$ and the $\pm$ operation depend on the value of the key bit $k \in \{0, 1\}$. This differential has probability 1, by well-known differential properties of addition modulo 2^n [23], and propagates cleanly through the state a for the next 8 iterations, resulting in the following difference over the state:

$$\Delta a = (\texttt{80}, \texttt{80}, \texttt{80}, \texttt{80}, \texttt{80}, \texttt{80}, \texttt{80}, \texttt{80}) \; .$$

The next iteration reveals one key bit. By XOR-linearising the state update function f, the new output difference $\Delta a'_j$ is of the form

$$\begin{aligned} \Delta a'_j = \; & ((a_{j+1 \bmod 8} \oplus \texttt{80}) \oplus m_i \oplus (\texttt{FF} \oplus ((a_j \oplus \texttt{80}) \oplus j) \lll r)) \oplus \\ & (a_{j+1 \bmod 8} \oplus m_i \oplus (\texttt{FF} \oplus (a_j \oplus j) \lll r)) \end{aligned}$$

where $r \in \{1, 7\}$. As a consequence, we have $\Delta a'_j = \texttt{81}$, if bit $7 - i \bmod 8$ of $k_{\lfloor i/8 \rfloor \bmod 12}$ is 1, and $\Delta a'_j = \texttt{C0}$, if the same key bit is 0. While integer addition and XOR behave differently with respect to the propagation of XOR-differences, the *least significant bit* of integer addition and XOR behave identically in this case and can be used to recover the key bit with probability 1.

The above leak, combined with Algorithm 2, can be turned into a chosen-plaintext key-recovery attack retrieving the OMAK k bitwise in at most $96 + 1$ queries. Algorithm 4 describes this attack in full detail. Looking at Figure 1, we see immediately that the reconstruction of k breaks the complete OSGP AE scheme. In the following, we will explore how the attack can be further improved.

Bytewise Key Recovery. Analysing the above attack more thoroughly, we noticed that we can recover one key byte at a time by injecting the input difference $\texttt{80}$ into the message a couple of steps earlier. This reduces the number of queries and the work load of the attack drastically. In other words, we will show how to reconstruct the entire OMAK with only $12 + 1$ chosen-plaintext queries.

Let $k_{i \bmod 12, j}$ denote the jth bit of key byte $i \bmod 12$, for $i = 17, 16, \ldots, 6$ and $j = 0, \ldots, 7$. When injecting the message difference $\Delta m_{8i-8} = \texttt{80}$ and thereupon processing 16 message bytes, we obtain an XOR-difference of the internal state of the form $\Delta a = (\Delta a_0, \ldots, \Delta a_7) = (\Delta x_0, \ldots, \Delta x_7)$ where Δx_l are arbitrary values for $l = 0, \ldots, 7$. The evolution of the difference propagation in the internal state can be visualised as follows:

Function `RecoverKey(`$\mathcal{O}$`)`

```
// O is an oracle returning a message's OMADigest under key k.
k ← {0}^12
m ←$ {0..255}^144
a ← O(m)
for i ← 0 to 11 do
    for j ← 0 to 7 do
        m' ← m
        m'_{136−8i−1−j} ← m'_{136−8i−1−j} ⊕ 80
        a' ← O(m')
        b ← OMABackward(a, m, k, 8i)            // Algorithm 2
        b' ← OMABackward(a', m', k, 8i)         // Algorithm 2
        k_{(17−i) mod 12,7−j} ← (b_{j,0} ⊕ b'_{j,0})
    end
end
return k
```

Algorithm 4: Bit-by-bit chosen-plaintext key-recovery attack.

$i = 17, \ldots, 6$	Δa_0	Δa_1	Δa_2	Δa_3	Δa_4	Δa_5	Δa_6	Δa_7
...	...	...	...	...	...	...	...	...
m_{8i-9}	00	00	00	00	00	00	00	00
m_{8i-8}	00	00	00	00	00	00	00	80
...	...	...	...	...	...	...	...	...
m_{8i-1}	80	80	80	80	80	80	80	80
m_{8i}	80	80	80	80	80	80	80	Δx_7
m_{8i+1}	80	80	80	80	80	80	Δx_6	Δx_7
...	...	...	...	...	...	...	...	...
m_{8i+7}	Δx_0	Δx_1	Δx_2	Δx_3	Δx_4	Δx_5	Δx_6	Δx_7

By analysing again the XOR-linearisation of the state update function f, one realises that a key byte can be recovered in its entirety by exploiting, as in the case of the bitwise key recovery attack, the information on the key bits stored in the least significant bit of the output differences $\Delta x_0, \ldots, \Delta x_7$. More precisely, key byte $k_{i \bmod 12}$ can be reconstructed as follows:

1. $k_{i \bmod 12,0} = \mathtt{lsb}(\Delta x_7) \oplus \mathtt{lsb}(80)$
2. $k_{i \bmod 12,1} = \mathtt{lsb}(\Delta x_6) \oplus \mathtt{lsb}(\Delta x_7)$
3. $k_{i \bmod 12,2} = \mathtt{lsb}(\Delta x_5) \oplus \mathtt{lsb}(\Delta x_6)$
4. $k_{i \bmod 12,3} = \mathtt{lsb}(\Delta x_4) \oplus \mathtt{lsb}(\Delta x_5)$
5. $k_{i \bmod 12,4} = \mathtt{lsb}(\Delta x_3) \oplus \mathtt{lsb}(\Delta x_4)$
6. $k_{i \bmod 12,5} = \mathtt{lsb}(\Delta x_2) \oplus \mathtt{lsb}(\Delta x_3)$
7. $k_{i \bmod 12,6} = \mathtt{lsb}(\Delta x_1) \oplus \mathtt{lsb}(\Delta x_2)$
8. $k_{i \bmod 12,7} = \mathtt{lsb}(\Delta x_0) \oplus \mathtt{lsb}(\Delta x_1)$

In order to verify that the above key recovery indeed works, consider the following steps. As we have already seen in the bitwise key recovery attack, the value of $k_{i \bmod 12,0}$ can be read off right away from Δx_7, see step 1 above. The remaining key bits $k_{i \bmod 12,j+1}$, for $j = 0, \ldots, 6$, can be recovered from the XOR-linearisation of f which gives us the relation

$$\Delta x_{6-j} = \Delta x_{7-j} \oplus (\Delta x'_{6-j} \lll r) = \Delta x_{7-j} \oplus (80 \lll r)$$

where Δx_{7-j} and Δx_{6-j} denote output differences and $\Delta x'_{6-j}$ corresponds to the difference before a_{6-j} is updated in the jth step. The latter simply has the value 80 as can be seen in the table on the difference propagation. The above equation can be re-written as

$$\texttt{lsb}(80 \lll r) = \texttt{lsb}(\Delta x_{6-j}) \oplus \texttt{lsb}(\Delta x_{7-j})$$

and since the rotation offset $r \in \{1, 7\}$ depends on $k_{i \bmod 12, j+1}$, the formula above gives us the value of the latter key bit.

Function RecoverKey($\mathcal{O}$)
  // $\mathcal{O}$ is an oracle returning a message's OMADigest under key k.
  $k \leftarrow \{0\}^{12}$
  $m \xleftarrow{\$} \{0..255\}^{144}$
  $a \leftarrow \mathcal{O}(m)$
  for $i \leftarrow 0$ **to** 11 **do**
    $m' \leftarrow m$
    $m'_{136-8i-8} \leftarrow m'_{136-8i-8} \oplus 80$
    $a' \leftarrow \mathcal{O}(m')$
    $b \leftarrow \texttt{OMABackward}(a, m, k, 8i)$ // Algorithm 2
    $b' \leftarrow \texttt{OMABackward}(a', m', k, 8i)$ // Algorithm 2
    $k_{(17-i) \bmod 12} \leftarrow \texttt{RecoverByte}(b, b')$
  end
  return k
Function RecoverByte(a, a')
  $x \leftarrow 0$
  $x_0 \leftarrow a_{7,0} \oplus a'_{7,0}$
  for $i \leftarrow 0$ **to** 6 **do** $x_{i+1} \leftarrow a_{6-i,0} \oplus a'_{6-i,0} \oplus a_{7-i,0} \oplus a'_{7-i,0}$
  return x

Algorithm 5: Byte-by-byte chosen-plaintext key-recovery attack.

3.2 Known-Plaintext Key Recovery Attack

The second attack is not differential in nature and requires a weaker attacker. We only assume in the following that the attacker is able to capture plaintexts with a *common prefix* of various lengths. This may be feasible by, e.g., capturing repeated messages with different sequence numbers.

This attack relies uniquely on the OMA digest's invertibility, as seen in Algorithm 2. The basic idea here is to have two messages , m and m' that are equal except in the last r bytes; partially reversing the final state of m by r iterations, then using that state to process the final bytes of m' should only happen when the (guessed) key bits used in those iterations are correct. This does not always happen, but it reduces the keyspace to virtually one or two

guesses per key byte. The concrete realisation of the attack is also described in Algorithm 6.

However, due to the slow diffusion of differences already described in Section 3.1, to recover r bits of the key one needs more than r iterations back; this is not a problem, though, as long as the key bits corresponding to the common prefix bytes of the message are the same for the forwards and backwards processing of the message. In practice, we have found that $r + 8$ iterations suffice to recover the key with overwhelming probability.

Function `RecoverKey(`$\mathcal{O}$`)`
  `//` $\mathcal{O}$ `is an oracle returning a message's OMADigest under key` k`.`
  $k \leftarrow \{0\}^{12}$
  $m \xleftarrow{\$} \{0..255\}^{144}$
  $a \leftarrow \mathcal{O}(m)$
  for $i \leftarrow 0$ **to** 11 **do**
    $m' \leftarrow m$
    $m'_{128-8i..|m'|-1} \xleftarrow{\$} \{0..255\}^{|m|-128-8i}$
    $a' \leftarrow \mathcal{O}(m')$
    for $x \leftarrow 0$ **to** 255 **do**
      $k_{(17-i) \bmod 12} \leftarrow x$
      $b \leftarrow$ `OMABackward`$(a, m, k, 8i + 16)$ `// Algorithm 2`
      $b' \leftarrow$ `OMAForward`$(b, m', k, 128 - 8i)$ `// Algorithm 3`
      if $a' = b'$ **then**
        break `// May be a false positive; handling omitted.`
      end
    end
  end
  return k

Algorithm 6: Byte-by-byte known-plaintext key-recovery attack.

3.3 Optimizing the Attacks

The attacks of Section 3.1 and Section 3.2 have an obvious generalization that trades queries for computation time. This is also a consequence of the OMA digest's reversibility.

Let $B \geq 1$ be the number of key bytes to recover per query; the attack from Section 3.2 generalizes trivially to any B, by guessing B adjacent key bytes per query, at an average cost of $\lceil \frac{12}{B} \rceil + 1$ queries and $\lceil \frac{12}{B} \rceil 2^{8B-1}$ operations[6].

The method from Section 3.1 also generalizes well to any B, by guessing the last $B - 1$ bytes and recovering the first one by injecting a difference. Its average

[6] An "operation" here is taken to mean at most the cost of an OMA digest evaluation over a message.

cost is $\lceil \frac{12}{B} \rceil + 1$ queries and $\lceil \frac{12}{B} \rceil 2^{8(B-1)-1}$ operations. We note that for $B \geq 2$ the messages used in either case need not be longer than 113 bytes, bypassing OSGP's restriction on message sizes.

3.4 Forgeries and a Third Key-Recovery Attack

Forgeries in the OMA digest are possible by exploiting the differential properties described in Section 3.1. To this end, we first explore XOR differentials and afterwards describe attacks using additive differentials.

Forgeries using XOR-Differentials. For this attack, we consider input XOR-differences of the shape $(\Delta m_{8i+j}, \Delta m_{8i+j+1}, \Delta m_{8i+j+8}) = (\mathtt{80}, \mathtt{80}, \Delta x)$ for $i = 0, \ldots, 17$ and $j = 0, \ldots, 7$. After processing message bytes $m_{8i+j}, m_{8i+j+1}, \ldots, m_{8i+j+7}$, the XOR-differences in the internal state are, up to a rotation, of the form $\Delta a = (\mathtt{80}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00})$. More precisely, after injecting $\Delta m_{8i+j} = \mathtt{80}$, the difference $\Delta m_{8i+j+1} = \mathtt{80}$ is used to prevent the difference of Δm_{8i+j} from spreading to the rest of the state. Creating this stationary difference can be achieved with probability 1. Finally, the difference $\Delta m_{8i+j+8} = \Delta x$ is used to cancel the stationary difference from above thereby creating a forgery. The success of the forgery hinges on whether the formula

$$(m_{8i+j+8} \oplus \Delta x) \pm (\mathtt{FF} \oplus ((a_j \oplus \mathtt{80}) + j) \lll r) = m_{8i+j+8} \pm (\mathtt{FF} \oplus (a_j + j) \lll r)$$

is satisfied. Note that the above formula again includes both possible cases which depend on the value of the key bit $k \in \{0, 1\}$. Using the formulas of Lipmaa and Moriai [23], we can determine the optimal value for Δx with respect to its probability p and the value of the key bit $k_{i+1 \bmod 12, j}$:

$k_{i+1 \bmod 12,j}$	0		1					
Δx	C0	40	01	03	07	0F	1F	3F	7F	FF
$-\log_2 p$	1	1	1	2	3	4	5	6	7	7

Thus, choosing $\Delta x \in \{\mathtt{C0}, \mathtt{40}, \mathtt{01}\}$ has a probability of about 1/4 of creating a valid forgery, assuming a uniformly random key bit.

Forgeries using Additive Differentials. Injecting additive differences is also useful to get a wider range of possible high-probability differences, since every operation in the OMA digest, with the exception of the cyclic rotation, has additive differential probability 1[7].

Using a similar approach as above, one can inject the additive difference $(\Delta^{\boxminus} x, -\Delta^{\boxminus} x, -\Delta^{\boxminus} y)$ at (m_i, m_{i+1}, m_{i+8}). The success of the forgery here depends on the quality of the approximations

$$\Delta^{\boxminus} y = ((-a_j - j - 1) \lll 1) - ((-a_j - \Delta^{\boxminus} x - j - 1) \lll 1)$$
$$\Delta^{\boxminus} y = -((-a_j - j - 1) \ggg 1) + ((-a_j - \Delta^{\boxminus} x - j - 1) \ggg 1)$$

[7] Note that $\neg x = x \oplus \mathtt{FF} = -x - 1$.

for a_j chosen uniformly at random. Since cyclic rotation is not a deterministic operation with respect to additive differences, one cannot obtain $\Delta^{\boxminus} y$ that works with probability 1. By replacing $((-a_j - \Delta^{\boxminus} x - j - 1) \lll 1)$ by $((-a_j - j - 1) \lll 1) + (-\Delta^{\boxminus} x \lll 1)$, and taking advantage of Daum's results on the interaction of integer addition and rotation [11], we have $\Delta^{\boxminus} y = -((-\Delta^{\boxminus} x \lll 1) - 2\alpha + \beta)$, where (α, β) has, as a function of $\Delta^{\boxminus} x_R = \lfloor (-\Delta^{\boxminus} x)/2 \rfloor$ and $\Delta^{\boxminus} x_L = (-\Delta^{\boxminus} x) \bmod 2^7$, one of the following values of probability p:

(α, β)	p
$(0,0)$	$2^{-8}(2^7 - \Delta^{\boxminus} x_R)(2 + \Delta^{\boxminus} x_L)$
$(0,1)$	$2^{-8}\Delta^{\boxminus} x_R(2 - \Delta^{\boxminus} x_L - 1)$
$(1,0)$	$2^{-8}(2^7 - \Delta^{\boxminus} x_R)\Delta^{\boxminus} x_L$
$(1,1)$	$2^{-8}\Delta^{\boxminus} x_R(\Delta^{\boxminus} x_L + 1)$

Similar remarks apply to the rotation by 7 case. By choosing $\Delta^{\boxminus} x$ carefully, one can maximize the probability of $\Delta^{\boxminus} y$ as well, as also previously exploited by Daum [11]. For instance, choosing the difference $\Delta^{\boxminus} x = \texttt{02}$, one obtains $\Delta^{\boxminus} y \in \{\texttt{01}, \texttt{FC}, \texttt{81}, \texttt{FB}, \texttt{FD}\}$, with respective probabilities $\{127/256, 126/256, 1/256, 1/256, 1/256\}$. Therefore, one can expect 2 queries to be sufficient in over $\approx 98\%$ of the time with this method.

Using Forgeries for Key Recovery. Such a high-probability forgery attack, dependent on the value of key bits, gives us yet another attack vector for key recovery. This attack is much simpler than the previous ones, and unlike those it does not need to work "right to left" on the message bytes: given a known plaintext, inject $(\texttt{02}, -\texttt{02}, -\Delta^{\boxminus} y)$ and query a verification oracle. If the forged message is validated, recover the key bit corresponding to m_{i+8} by looking up which $\Delta^{\boxminus} y$ corresponds to which key bit. This process can be repeated 96 times to recover the entire key.

Additionally, this attack can work even over ciphertext, by using the XOR-differences $(\texttt{80}, \texttt{80}, \Delta x)$ with $\Delta x \in \{\texttt{40}, \texttt{C0}, \texttt{01}\}$. The approach here is the same, albeit requiring a few more queries, but it can be applied over unknown ciphertext encrypted with RC4, as is the case with OSGP. The attack thus completely breaks not only the OMA digest, but also the entire cryptographic security of OSGP.

The average number of queries can be reduced by using the following trick: instead of picking a difference at random from the possible set of differences, pick `C0` and `40` in order. If none of them results in a forgery, the key bit can only be 1; this results in key recovery in an average of 168 queries. Algorithm 7 illustrates the XOR key-recovery attack on OSGP using this trick, only taking as input a valid ciphertext-tag pair and an oracle that verifies *ciphertexts*.

3.5 Extension of the OSGP Analysis to Other Standards

The EN 14908 algorithm, used in OSGP for key derivation and quite similar to the OMA digest, is also used in other LonTalk-derived standards for authentication [6,13,19,20,21,28]. We found evidence that the foundations of the

```
Function RecoverKey(𝒪, c, a)
    // 𝒪 is an oracle that returns 1 if (c, a) is a valid OSGP
       ciphertext-tag pair, 0 otherwise.
    // c, a is a valid OSGP ciphertext-tag pair, i.e., 𝒪(c, a) = 1.
    k ← {0}^12
    for i ← 0 to 95 do
        c' ← c
        c'_i ← c_i ⊕ 80
        c'_{i+1} ← c_{i+1} ⊕ 80
        c'_{i+8} ← c_{i+8} ⊕ C0
        if 𝒪(c', a) = 1 then
            k_{⌊(i+8)/8⌋ mod 12,(i+8) mod 8} ← 0
            continue
        end
        c'_{i+8} ← c_{i+8} ⊕ 40
        k_{⌊(i+8)/8⌋ mod 12,(i+8) mod 8} ← 1 − 𝒪(c', a)
    end
    return k
```

Algorithm 7: Bit-by-bit chosen-ciphertext key-recovery attack, in the context of the OSGP protocol.

technology (presumably also including the EN 14908 algorithm) were laid in 1988 [24, p.3]. LonTalk was estimated to be implemented in over 90 million devices as of 2010 [14]. Given that the EN 14908 algorithm has a 48-bit key, it is already broken by design. That said, the attacks described in the previous sections can be adapted to key recovery attacks on the EN 14908 algorithm—likely present in every other LonTalk-derived standard—in much less than 2^{48} work.

4 Conclusion

We have presented a thorough analysis of the OMA digest specified in OSGP. This function has been found to be extremely weak, and cannot be assumed to provide any authenticity guarantee whatsoever. We described multiple attacks having different levels of applicability in the context of OSGP. The forgery attacks presented in Section 3.4 belong to the most powerful and practical, and allow to retrieve the 96-bit secret key in a mere 144 and 168 chosen-plaintext queries to a tag-verification oracle exploiting the very slow propagation of additive and XOR-differences in the OMA digest. We also described how the latter variant can work as a ciphertext-only attack, making it even more devastating. For easier verifiability, we implemented the attacks of Section 3 in the Python language; the code is listed in Appendix A.

In summary, the work at hand is another entry in the long list of examples of flawed authenticated encryption schemes, and shows once more how easily a determined attacker can break the security of protocols based on weak cryptography.

Acknowledgments. Our results were fully disclosed to the members of OSGP Alliance, who acknowledged our findings on the OSGP standard, in November 2014. We would like to thank Jean-Philippe Aumasson, Tanja Lange and Ilia Polian for helpful discussions during our work.

References

1. Approval of OSGP as an ETSI Standard (2012), http://www.etsi.org/news-events/news/382-news-release-18-january-2012
2. Albrecht, M.R., Paterson, K.G., Watson, G.J.: Plaintext Recovery Attacks Against SSH. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. pp. 16–26. SP '09, IEEE Computer Society (2009)
3. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the Security of RC4 in TLS. In: King, S.T. (ed.) Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14–16, 2013. pp. 305–320. USENIX Association (2013)
4. AlFardan, N.J., Paterson, K.G.: Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. pp. 526–540. IEEE Computer Society (2013), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6547086
5. ANSI: Protocol Specification For Interfacing to Data Communication Networks. ANSI C12.22-2008, American National Standards Institute (January 2009)
6. ANSI: Control Network Protocol Specification. ANSI/CEA-709.1-C, American National Standards Institute (December 2010)
7. Bellare, M., Namprempre, C.: Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In: Okamoto, T. (ed.) Advances in Cryptology – ASIACRYPT 2000. Lecture Notes in Computer Science, vol. 1976, pp. 531–545. Springer (2000)
8. Bellare, M., Rogaway, P., Wagner, D.: The EAX Mode of Operation. In: Roy, B.K., Meier, W. (eds.) Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5–7, 2004, Revised Papers. Lecture Notes in Computer Science, vol. 3017, pp. 389–407. Springer (2004)
9. Bernstein, D.J.: Cryptographic competitions — Disasters. http://competitions.cr.yp.to/disasters.html (2014), accessed on 2014.01.27
10. Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password Interception in a SSL/TLS Channel. In: Boneh, D. (ed.) Advances in Cryptology – CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17–21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 583–599. Springer (2003), http://dx.doi.org/10.1007/978-3-540-45146-4_34
11. Daum, M.: Cryptanalysis of Hash functions of the MD4-family. Ph.D. thesis, Ruhr University Bochum (May 2005), http://www-brs.ub.ruhr-uni-bochum.de/netahtml/HSS/Diss/DaumMagnus/
12. Duong, T., Rizzo, J.: Here Come The ⊕ Ninjas. Unpublished (May 2011)
13. Echelon Corporation: LonTalk Protocol Specification (1994), version 3.0
14. Echelon Corporation: 90 Million Energy-Aware LonWorks Devices Worldwide. http://www.businesswire.com/news/home/20100412005544/en/90-Million-Energy-Aware-LonWorks-Devices-Worldwide (2010)
15. ETSI: Open Smart Grid Protocol (OSGP). Reference DGS/OSG-001, European Telecommunications Standards Institute, Sophia Antipolis Cedex – France (January 2012), http://www.osgp.org/

16. Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers. Lecture Notes in Computer Science, vol. 2259, pp. 1–24. Springer (2001)
17. Fluhrer, S.R., McGrew, D.A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In: Schneier, B. (ed.) Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10–12, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1978, pp. 19–30. Springer (2000)
18. Gupta, S.S., Maitra, S., Paul, G., Sarkar, S.: (Non-)Random Sequences from (Non-)Random Permutations – Analysis of RC4 Stream Cipher. J. Cryptology 27(1), 67–108 (2014)
19. IEEE: Draft Standard for Communications Protocol Aboard Passenger Trains. IEEE P1473/D8 (July 2010), http://ieeexplore.ieee.org/servlet/opac?punumber=5511471
20. ISO: Information Technology — Interconnection of Information Technology Equipment — Home Electronic System (HES) Architecture — Medium-independent Protocol Based on ANSI/CEA-709.1-B. ISO/IEC CD 14543-6-1:2006, International Organization for Standardization (2006), http://hes-standards.org/doc/SC25_WG1_N1229.pdf
21. ISO: Information Technology – Control Network Protocol – Part 1: Protocol Stack. ISO/IEC 14908-1:2012, International Organization for Standardization, Geneva, Switzerland (2012)
22. Kursawe, K., Peters, C.: Structural Weaknesses in the Open Smart Grid Protocol. Cryptology ePrint Archive, Report 2015/088 (2015), https://eprint.iacr.org/2015/088
23. Lipmaa, H., Moriai, S.: Efficient Algorithms for Computing Differential Properties of Addition. In: Matsui, M. (ed.) Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2–4, 2001 Revised Papers. Lecture Notes in Computer Science, vol. 2355, pp. 336–350. Springer (2001)
24. LonMark International: LON and BACnet: History and Approach, http://www.lonmark.org/connection/presentations/2012/Q2/Light-Building/06+LON+and+BACnet-+History+and--Newron+System.pdf
25. Minematsu, K., Lucks, S., Morita, H., Iwata, T.: Attacks and Security Proofs of EAX-Prime. In: Moriai, S. (ed.) Fast Software Encryption, 20th International Workshop, FSE 2013, Singapore, March 11–13, 2013. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8424, pp. 327–347. Springer (2013)
26. Möller, B., Duong, T., Kotowicz, K.: This POODLE Bites: Exploiting The SSL 3.0 Fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf (October 2014)
27. Sepehrdad, P., Vaudenay, S., Vuagnoux, M.: Discovery and Exploitation of New Biases in RC4. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) Selected Areas in Cryptography, 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6544, pp. 74–91. Springer (2010)
28. Standardization Administration of China: Control Network LONWORKS Technology Specification — Part 1: Protocol Specification. GB/Z 20177.1-2006 (2006)
29. Tews, E., Weinmann, R., Pyshkin, A.: Breaking 104 Bit WEP in Less Than 60 Seconds. In: Kim, S., Yung, M., Lee, H. (eds.) Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27–29, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4867, pp. 188–202. Springer (2007)

30. Vaudenay, S.: Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS ... In: Knudsen, L.R. (ed.) Advances in Cryptology — EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 – May 2, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2332, pp. 534–546. Springer (2002)
31. Vaudenay, S., Vuagnoux, M.: Passive-Only Key Recovery Attacks on RC4. In: Adams, C.M., Miri, A., Wiener, M.J. (eds.) Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4876, pp. 344–359. Springer (2007)

A Proof of Concept

```
import os

def ROT8(x, c):
  return ((x%256 << c%8) | (x%256 >> -c%8)) % 256

def OMADigest(m,k):
  a = [0] * 8
  m = m[:] + [0] * (-len(m) % 144)
  for l in range(0, len(m), 144):
    b = m[l:l+144]
    for i in range(18):
      for j in range(7, -1, -1):
        if (k[i%12] >> (7 - j)) & 1:
          a[j] = (a[(j+1)%8] + b[8*i+7-j] + ROT8(~(a[j] + j), 1)) % 256
        else:
          a[j] = (a[(j+1)%8] + b[8*i+7-j] - ROT8(~(a[j] + j), -1)) % 256
  return a

def EN14908(r, m, k):
  mlen, a = len(m) - 1, r[:]
  while True:
    for i in range(6):
      for j in range(7, -1, -1):
        b = 0 if mlen < 0 else m[mlen]
        mlen -= 1
        if k[i] & (1 << (7 - j)):
          a[j] = a[(j+1)%8] + b + ROT8(~(a[j] + j), 1)
        else:
          a[j] = a[(j+1)%8] + b - ROT8(~(a[j] + j), -1)
    if mlen < 0:
      break
  return a

def RC4Encrypt(X,key):
    def RC4(key, b):
      B,S,i,j,l=[],range(256),0,0,len(key)
      while i < 256:
          j = (j + S[i] + key[i%l]) & 0xff
          S[i], S[j] = S[j], S[i]
          i += 1
      i, j = 1, 0
      while b:
          t = S[i]
          j = (j + S[i]) & 0xff
```

```
            S[i], S[j] = S[j], S[i]
            B += [S[(S[i]+S[j]) & 0xff]]
            b -= 1
            i = (i + 1) & 0xff
        return B
    S = RC4(key,len(X))
    for i in xrange(len(X)):
        X[i] ^= S[i]
    return X

def OSGPKeyDerive(k):
  k1 = EN14908([0x81, 0x3f, 0x52, 0x9a, 0x7b, 0xe3, 0x89, 0xba], [], k)
  k2 = EN14908([0x72, 0xb0, 0x91, 0x8d, 0x44, 0x05, 0xaa, 0x57], [], k)
  return k1 + k2

def OSGPEncrypt(m, k):
  k_ = OSGPKeyDerive(k)
  a  = OMADigest(m, k)
  for i in range(8):
    k_[i] ^= a[i]
  return RC4Encrypt(m, k_) + a

def OSGPDecrypt(c, k):
  assert(len(c) >= 8)
  k_ = k_ = OSGPKeyDerive(k)
  a  = c[-8:]
  for i in range(8):
    k_[i] ^= a[i]
  m = RC4Encrypt(c[:-8], k_)
  return OMADigest(m, k) == a, m

# Test vector
m = [0x02,0x02,0x00,0x30,0x00,
     0x03,0x7f,0x30,0xea,0x6d,
     0x00,0x00,0x00,0x0d,0x00,
     0x20,0x98,0x00,0x31,0xc3,
     0x00,0x08,0x00,0x00,0x00,
     0x00,0x00,0x11]
k = [0xDF] * 12
a = [0xdb, 0xe5, 0xcd, 0xe5, 0x07, 0xb1, 0xcb, 0x3d]
assert(OMADigest(m, k) == a)

def OMABackward(a,m,k,n):
  a, m = a[:], m[:] + [0] * (-len(m) % 144)
  for l in range(n):
    i, j = l // 8, l % 8
    if (k[(17-i)%12] >> (7 - j)) & 1:
      x = ROT8(a[j] - a[(j+1)%8] - m[143-8*i-j], -1)
    else:
      x = ROT8(a[(j+1)%8] + m[143-8*i-j] - a[j], 1)
    a[j] = (~x - j) % 256
  return a

def OMAForward(a,m,k,n):
  a, m = a[:], m[:] + [0] * (-len(m) % 144)
  for l in range(n, 144):
    i, j = l // 8, 7 - l % 8
    if (k[i%12] >> (7 - j)) & 1:
      a[j] = (a[(j+1)%8] + m[8*i+7-j] + ROT8(~(a[j] + j), 1)) % 256
    else:
```

```
      a[j] = (a[(j+1)%8] + m[8*i+7-j] - ROT8(~(a[j] + j), -1)) % 256
  return a

m = map(ord, os.urandom(144))
k = map(ord, os.urandom(12))
a = OMADigest(m, k)
assert( OMAForward([0]*8, m, k, 0) == OMADigest(m,k) )
assert( OMAForward(OMABackward(a,m,k,8),m,k,144-8) == a )

def TagGenOracle(m,init=[True]):
    if init[0]:
      print '[ORACLE] k = ' + str(k)
      init[0] = False
    return OMADigest(m,k)

def TagCheckOracle(m,a):
  return TagGenOracle(m) == a

def OSGPEncryptOracle(m, init=[True]):
  return OSGPEncrypt(m, k)

def OSGPCheckOracle(c):
  ok, _ = OSGPDecrypt(c, k);
  return ok

def Algorithm_4():
  m = map(ord, os.urandom(144))
  a = TagGenOracle(m)
  k = [0] * 12
  for i in range(12):
    for j in range(8):
      m_ = m[:]
      m_[136-8*i-j-1] ^= 0x80
      a_ = TagGenOracle(m_)
      b  = OMABackward(a,m,k,8*i)
      b_ = OMABackward(a_,m_,k,8*i)
      k[(17-i)%12] |= ((b[j] ^ b_[j])&1) << (7 - j)
  return k

print 'Algorithm 4: ' + str(Algorithm_4())

def Algorithm_5():
  def RecoverByte(a, b):
    x = (a[7] ^ b[7]) & 1
    for i in xrange(0,7):
        x |= ((a[6-i] ^ b[6-i] ^ a[7-i] ^ b[7-i]) & 1) << (i+1)
    return x
  k = [0] * 12
  m = map(ord, os.urandom(144))
  a = TagGenOracle(m)
  for i in range(12):
    m_ = m[:]
    m_[136-8*i-8] ^= 0x80
    a_ = TagGenOracle(m_)
    b  = OMABackward(a,m,k,8*i)
    b_ = OMABackward(a_,m_,k,8*i)
    k[(17-i)%12] = RecoverByte(b, b_)
  return k

print 'Algorithm 5: ' + str(Algorithm_5())
```

```
def Algorithm_6():
  def recurse(m,a,k,i=0):
    if i >= 12:
      a_ = OMADigest(m,k)
      return a_ == a
    m_ = m[:]
    m_[128-8*i:] = map(ord, os.urandom(144-(128-8*i)))
    a_ = TagGenOracle(m_)
    for x in range(256):
      k[(17-i)%12] = x
      b = OMABackward(a, m, k, 8*i + 16)
      b_ = OMAForward(b, m_, k, 128 - 8*i)
      if a_ == b_ and recurse(m,a,k,i+1):
          return True
    return False
  k = [0] * 12
  m = map(ord, os.urandom(144))
  a = TagGenOracle(m)
  recurse(m,a,k)
  return k

print 'Algorithm 6: ' + str(Algorithm_6())

def Algorithm_7():
  k = [0] * 12
  c = OSGPEncryptOracle(map(ord, os.urandom(96+8)))
  for i in range(96):
    c_ = c[:]
    c_[i+0] ^= 0x80
    c_[i+1] ^= 0x80
    c_[i+8] ^= 0xC0
    if OSGPCheckOracle(c_):
      continue
    c_[i+8] = c[i+8] ^ 0x40
    k[((i+8)//8)%12] |= (0 if OSGPCheckOracle(c_) else 1) << ((i+8)%8)
  return k

print 'Algorithm 7: ' + str(Algorithm_7())

# Key-recovery attack from Section 3.4, using additive differences
def Algorithm_8():
  k = [0] * 12
  m = map(ord, os.urandom(96+8))
  a = TagGenOracle(m)
  for i in range(96):
    m_ = m[:]
    m_[i+0] = (m[i+0] + 0x02) % 256
    m_[i+1] = (m[i+1] - 0x02) % 256
    m_[i+8] = (m[i+8] - 0x01) % 256
    if TagCheckOracle(m_, a): continue
    m_[i+8] = (m[i+8] - 0xfc) % 256
    if TagCheckOracle(m_, a):
      k[((i+8)//8)%12] |= 1 << ((i+8)%8)
      continue
    m_[i+8] = (m[i+8] - 0x81) % 256
    k[((i+8)//8)%12] |= (0 if TagCheckOracle(m_, a) else 1) << ((i+8)%8)
  return k

print 'Algorithm 8: ' + str(Algorithm_8())
```