The Stream Cipher Core of the 3GPP Encryption Standard 128-EEA3: Timing Attacks and Countermeasures*

Gautham Sekar

Indian Statistical Institute, Chennai Centre,
SETS Campus, MGR Knowledge City, CIT Campus, Taramani,
Chennai 600113, India.
sgautham@isichennai.res.in

Abstract. The core of the 3^{rd} Generation Partnership Project (3GPP) encryption standard 128-EEA3 is a stream cipher called ZUC. It was designed by the Chinese Academy of Sciences and proposed for inclusion in the cellular wireless standards called "Long Term Evolution" or "4G". The LFSR-based cipher uses a 128-bit key. In this paper, we first show timing attacks on ZUC that can recover, with about 71.43% success rate, *(i)* one bit of the secret key immediately, and *(ii)* information involving 6 other key bits. The time, memory and data requirements of the attacks are negligible. While we see potential improvements to the attacks, we also suggest countermeasures.

Keywords: Stream cipher, cache timing attack, key recovery.

1 Introduction

ZUC [8] is a stream cipher designed by the Data Assurance and Communication Security Research Center (DACAS) of the Chinese Academy of Sciences. The cipher forms the core of the 3GPP mobile standards 128-EEA3 (for encryption) and 128-EIA3 (for message integrity) [7]. It was proposed for inclusion in the Long Term Evolution (LTE) or the 4^{th} generation of cellular wireless standards (4G).[1] ZUC is LFSR-based and uses a 128-bit key and a 128-bit initialization vector (IV). Some key points in the evolution of ZUC are listed in the following timeline.

Timeline:

* A shorter and slightly older version of this paper is due to appear in the proceedings of Inscrypt 2012. It was written when the author was with the National University of Singapore.

1 Strictly speaking, LTE is not 4G as it does not fully comply with the International Mobile Telecommunications Advanced (IMT-Advanced) requirements for 4G. Put differently, LTE is beyond 3G but pre-4G.

- 18^{th} *June 2010:* The Security Algorithms Group of Experts (SAGE) of the European Telecommunications Standards Institute (ETSI) published a document providing the specifications of the first version of ZUC. The document was indexed "Version 1.0".
- 26^{th}–30^{th} *July 2010:* Improvements and minor corrections were made successively to the C implementation of the ZUC algorithm of Version 1.0. These resulted in versions 1.2 and 1.3 of the ETSI/SAGE document. The preface to Version 1.3 was corrected and the resulting document released as Version 1.4.
- 02^{nd}–03^{rd} *December 2010 (First International Workshop on ZUC Algorithm):* A few observations on the algorithm of Version 1.4 were reported (see [6]) but none of these posed any immediate threat to its security.
- 05^{th}–09^{th} *December 2010 (ASIACRYPT):* The algorithm of Version 1.4 was cryptanalysed by Wu *et al.* [24] and the results were presented at the rump session of ASIACRYPT 2010.

  The attack reduces the effective key size of ZUC to about 66 bits by exploiting the fact that a difference set between a pair of *IV*s may result in identical keystreams.
- 08^{th} *December 2010:* Gilbert *et al.* reported an existential forgery attack on the 128-EIA3 MAC algorithm.

  The attack allows, given any message and its MAC value under an unknown integrity key and an initialization vector, to predict the MAC value of a related message under the same key and the same initialization vector with a success probability of 1/2.

  Gilbert *et al.* also gave a modified version of the 128-EIA3 algorithm (*cf.* [12, Algorithm 2]).

  In the original 128-EIA3 construction, some 32-bit keystream words are used in computing the universal hash function, and then the next whole word of keystream is used as a mask. But in [12, Algorithm 2], the first keystream word is used as the mask. The latter algorithm better fits the standard Carter-Wegman construction [5].
- 04^{th} *January 2011:* In response to Wu *et al.*'s key recovery attack, the initialization of ZUC was modified. Version 1.5 contains the new algorithm [8]. This algorithm is the one we analyse in this paper; we have been and shall henceforth be simply calling it "ZUC" (i.e., without any accompanying version numbers).
- 05^{th}–06^{th} *June 2011 (The 2nd International Workshop on ZUC Algorithm and Related Topics):* Gilbert *et al.* presented an updated version (*cf.* [12]) of their paper. In this they argue that [12, Algorithm 2] might have slightly greater resistance against nonce reuse.
- 07^{th} *June 2011 – $26^{th} August 2011$:* Changing the ZUC integrity algorithm of 128-EIA3 to [12, Algorithm 2] was being considered by the ETSI/SAGE in June 2011. Although [12, Algorithm 2] offers some advantages, they appear to be marginal.

  In this paper, we present two timing attacks on ZUC, each of can (in the best case) recover with (nearly) 0.7143 success probability, *(i)* one bit of the key immediately, and *(ii)* information involving 6 other bits of the key. Before describing how this paper is organised, we shall discuss timing attacks briefly.

Timing attack: This is a side-channel attack in which the attacker exploits timing measurements of (parts of) the cryptographic algorithm's implementation. For example, in the case of unprotected AES implementations based on lookup tables, the dependence of the lookup time on the table index can be exploited to speed up key recovery [4]. A *cache timing attack* is a type of timing attack which is based on the idea that the adversary can observe the cache accesses of a legitimate party. The cache is an intermediate memory between the CPU and the RAM and is used to store frequently used data fetched from the RAM. The problem with the cache memory is that, unlike the RAM, it is shared among users sharing a CPU.[2] Hence, if Bob and Eve are sharing a CPU and Eve is aware that Bob is about to encrypt, Eve may initiate her cache timing attack as follows. She first fills the cache memory with values of her choice and waits for Bob to run the encryption algorithm. She then measures the time taken to load the earlier cache elements into the CPU; loading is quick if the element is still in cache (such an event is called a *cache hit*; its complement is a *cache miss*) and not overwritten by one of Bob's values. This technique is known as Prime+Probe [18]. Cache timing attacks have been successfully mounted on several ciphers, notably the AES [4, 18, 26, 14].

In [18], two types of cache timing attacks are introduced – *synchronous* and *asynchronous*. In a synchronous attack, the adversary can make cache measurements only after certain operations of the cipher (e.g., a full update of a stream cipher's internal state) have been performed. In this attack scenario, the plaintext or the ciphertext is assumed to be available to the adversary. An asynchronous cache adversary, on the other hand, is able to make cache measurements in parallel to the execution of the routine. She is able to obtain a list of all cache accesses made *in chronological order* [26]. Here, there are different viewpoints on the resources available to the adversary. According to Osvik *et al.*, the adversary has only the distribution of the plaintext/ciphertext and not sample values [18]. Zenner differs in [26] where he argues that the adversary can (partially) control input/output data and observe cache behaviour. Asynchronous attacks are particularly effective on processors with simultaneous multithreading. One of the timing attacks in this paper is an asynchronous cache timing attack, and the other is a straightforward timing attack that does not involve the cache.

Organisation: Section 2 provides the specifications of ZUC along with some notation and convention. The preliminary observations that lead us to timing attacks are listed in Sect. 3 and the attacks are detailed in Sect. 4. We follow this with an analysis of some design/implementation modifications that resist the attacks, in Sect. 5. In Sect. 6, we see possible improvements to the timing

[2] Actually, in most modern CPUs the cache is simply the static RAM (SRAM) and the dynamic RAM (DRAM) is the other, predominant type of computer memory that we simply call "the RAM".

attacks and find that the proposed design modifications resist these improved attacks too. In addition, we see several highlights of our attacks such as the novelty of an employed technique. The paper concludes with a suggestion for future work, in the same section.

2 Specifications of ZUC

In this paper, we use several of the notation and convention followed in [8] in addition to that provided in Table 1.

Table 1. Notation and convention

Notation	Meaning
LSB	Least significant bit
MSB	Most significant bit
$\odot$	Multiplication modulo $(2^{31}-1)$
$t_{i(j)}$	The jth bit ($j = 0$ denoting the LSB) of t_i
$[\beta_1\beta_2\ldots\beta_n]$	$\beta_1\|\|\beta_2\|\|\ldots\|\|\beta_n$
Y_H	$[Y_{(30)}Y_{(29)}\ldots Y_{(15)}]$, when $\|Y\| = 31$ bits $[Y_{(31)}Y_{(29)}\ldots Y_{(16)}]$, when $\|Y\| = 32$ bits
Y_L	$[Y_{(15)}Y_{(14)}\ldots Y_{(0)}]$

As previously mentioned, the inputs to the ZUC cipher are a 128-bit key and a 128-bit IV. The algorithm has three parts or "layers" – a linear feedback shift register (LFSR) layer, a bit-reorganisation ("BR") layer and a nonlinear function F. – that are shown in Figure 1. The execution of the algorithm proceeds in two stages – an initialization stage and a "working" stage. Each iteration of the algorithm in the working stage generates 32 bits of keystream output. We shall now detail the layers and stages to the level that is required for the understanding of the results to follow. For the complete specifications, the interested reader is referred to [8, Sect. 3].

The LFSR layer: ZUC uses one LFSR that contains sixteen 32-bit cells containing 31-bit values $s_0, s_1, \ldots, s_{15}$. However, none of the 31-bit elements can assume the value 0; the remaining $2^{31}-1$ values are allowed. The steps of the LFSR layer in the initialization mode comprise Algorithm 1.
The steps of the LFSR layer in the working mode comprise Algorithm 2.

The BR layer: In this layer, 128 bits are extracted from the cells of the LFSR and four 32-bit words are formed. Three of these words (X_0, X_1, X_2) are used by the nonlinear function F, and the fourth word (X_3) is used in producing the keystream.

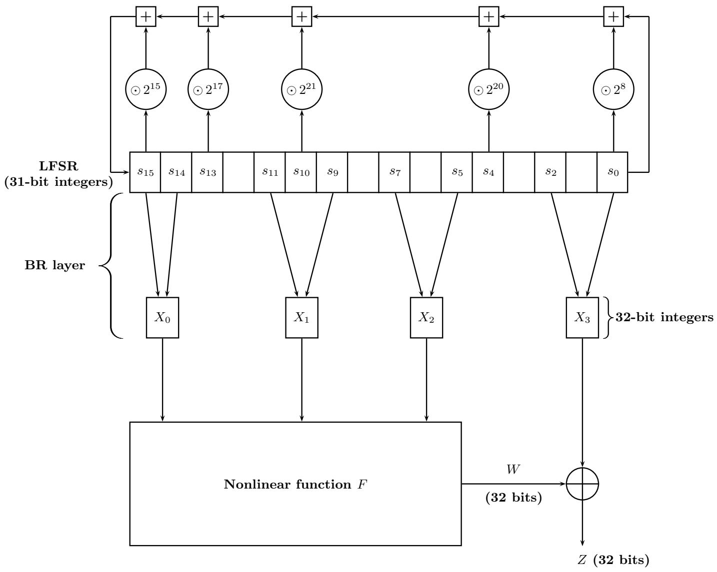

Fig. 1. The three layers of the ZUC algorithm

Algorithm 1 The LFSR layer in the initialization mode

$v \coloneqq 2^{15} \odot s_{15} + 2^{17} \odot s_{13} + 2^{21} \odot s_{10} + 2^{20} \odot s_4 + 2^8 \odot s_0 + s_0 \bmod (2^{31} - 1);$
$s_{16} \coloneqq (v + u) \bmod (2^{31} - 1);$ /* u is derived from the output of F */
if $s_{16} = 0$ **then**
    $s_{16} \leftarrow 2^{31} - 1;$
$(s_1, s_2, \ldots, s_{15}, s_{16}) \rightarrow (s_0, s_1, \ldots, s_{14}, s_{15});$

The nonlinear function F: This function involves two 32-bit values in memory cells (R_1, R_2), one 32×32 S-box (S), two linear transforms (L_1, L_2) and the aforementioned three 32-bit words produced by the BR layer. The output of the function F is a 32-bit word W. The 32-bit keystream word Z, that is produced in every iteration of the working mode of the ZUC algorithm, is simply $W \oplus X_3$. The F function is defined as follows:

$F(X_0, X_1, X_2)\{$
$W = (X_0 \oplus R_1) + R_2 \bmod 2^{32};$
$W_1 \coloneqq R_1 \oplus X_1;$
$W_2 \coloneqq R_2 \oplus X_2;$
$R_1 = S(L_1(W_{1L} || W_{2H}));$
$R_2 = S(L_2(W_{2L} || W_{1H}));\}$

Algorithm 2 The LFSR layer in the working mode

1: $s_{16} = 2^{15} \odot s_{15} + 2^{17} \odot s_{13} + 2^{21} \odot s_{10} + 2^{20} \odot s_4 + 2^8 \odot s_0 + s_0 \bmod (2^{31} - 1)$;
2: **if** $s_{16} = 0$ **then**
3: $\quad s_{16} \leftarrow 2^{31} - 1$;
4: $(s_1, s_2, \ldots, s_{15}, s_{16}) \rightarrow (s_0, s_1, \ldots, s_{14}, s_{15})$;

Key loading: The key loading procedure expands the 128-bit secret key and the 128-bit IV to form the initial state of the LFSR. In [8], this key is denoted as k ($= k_0||k_1|| \ldots ||k_{15}$, where each k_i is a byte) and the IV as iv ($= iv_0||iv_1|| \ldots ||iv_{15}$, where each iv_i is a byte). In addition to k and iv, a 240-bit constant D ($= d_0||d_1|| \ldots ||d_{15}$) is used in the key loading procedure. We shall now provide the binary representations of the d_i's first (in Table 2), followed by the key loading procedure.

Table 2. The constants d_i, $i \in \{0, 1, \ldots, 15\}$, used in the key loading procedure

d_0	100010011010111	d_8	100110101111000
d_1	010011010111100	d_9	010111100010011
d_2	110001001101011	d_{10}	110101111000100
d_3	001001101011110	d_{11}	001101011110001
d_4	101011110001001	d_{12}	101111000100110
d_5	011010111100010	d_{13}	011110001001101
d_6	111000100110101	d_{14}	111100010011010
d_7	000100110101111	d_{15}	100011110101100

Given this, the key loading is a set of very simple and straightforward steps given by:

$$s_i = k_i||d_i||iv_i\,, \text{ for } i \in \{0, 1, \ldots, 15\}\,. \tag{1}$$

The execution of ZUC: As mentioned earlier, the execution of the ZUC algorithm proceeds in two stages. We shall now describe these stages.

The initialization stage: This stage is given by Algorithm 3.
The working stage: This stage, in turn, has two sub-stages that are given by Algorithms 4 and 5.

3 Motivational Observations

We start with the following two trivial observations.

Algorithm 3 The initialization stage of ZUC execution

$ctr = 0$;
repeat
  Execute the BR layer;
  Compute the nonlinear function F taking as inputs the outputs X_0, X_1 and X_2 of the BR layer;
  Run Algorithm 1;
  $ctr \leftarrow ctr + 1$;
until ctr = 32

Algorithm 4 First sub-stage of the working stage of ZUC execution

Execute the BR layer;
Compute the nonlinear function F taking as inputs the outputs X_0, X_1 and X_2 of the BR layer;
Discard the output W of F;
Run Algorithm 2;

Algorithm 5 Keystream generating sub-stage of the working stage of ZUC execution

repeat
  Execute the BR layer;
  Compute the nonlinear function F taking as inputs the outputs X_0, X_1 and X_2 of the BR layer;
  Compute the keystream as $Z = W \oplus X_3$;
  Run Algorithm 2;
until one 32-bit keystream word more than the required number of words is generated

Observation 1 *The ZUC key is initially loaded directly into the 16 LFSR cells.*

Observation 2 *Multiplication and addition in the initialization mode and working mode of the LFSR layer are modulo* $(2^{31}-1)$*. Other additions and multiplications are modulo* 2^{32}*.*

Addition modulo $(2^{31}-1)$ of two 31-bit integers x and y is performed in [8] as follows. First, they are stored in 32-bit cells and $z = x + y \bmod 2^{32}$ is computed. If the *end carry*, meaning the carry-in at the MSB position of a 32-bit word/register/memory cell, is b, the MSB of the 32-bit z is first discarded and then this 31-bit word is incremented by b. This is implemented in C in [8] as:

```
u32 Add(u32 x, u32 y) {
 u32 z = x + y;
 if (z & 0x80000000)
  z = (z & 0x7FFFFFFF) + 1;
 return z;
}
```

It is to be noted that the increment step in *Add*() cannot regenerate end carry[3] because $x, y \in \{1, 2, \ldots, 2^{31}-1\}$ implies that $u32\,z$ has at least one zero in its 31 LSBs.

An end carry of 1 brings in one extra 32-bit *AND* operation and one 32-bit addition in the software implementation (in hardware implementation, we have 32 bitwise *AND* operations and one 32-bit ripple carry addition). Let T_{carry} denote the total time taken by the processor to perform these additional operations and T denote the time taken to run the *Add*() subroutine without the step where z is incremented. We now have the following simple observation that forms the base of our timing analysis.

Observation 3 *If the attacker observes that the time taken to run the Add() subroutine is* $T + T_{carry}$*, then she necessarily concludes that the end carry is 1, and can use this to retrieve some information on the summands* x *and* y *in general and their MSBs in particular.*

In Sect. 4, we shall show how we exploit Observations 1–3 to mount (partial) key recovery attacks on ZUC.

4 The Timing Attacks

In this section, we shall examine the first invocation of the LFSR layer in the initialization mode. Recall that the first step of Algorithm 1 is:

$$v := 2^{15} \odot s_{15} + 2^{17} \odot s_{13} + 2^{21} \odot s_{10} + 2^{20} \odot s_4 + 2^8 \odot s_0 + s_0 \bmod (2^{31}-1)\,. \quad (2)$$

[3] Throughout this paper, a 'generated' or 'produced' end carry is always 1 unless otherwise stated.

Given a 32-bit cell containing a 31-bit integer δ, the product $2^n \odot \delta$ is implemented in C in [8] as $((\delta \ll n)|(\delta \gg (31-n)))\,\&\,0x7FFFFFFF$. Given this and the manner in which the key bits are loaded into the cells initially (see [8, Sect. 3]), we see that the 31-bit summands on the RHS of (2) in the first round of the initialization mode are:

$$\begin{aligned}
z_1 &:= [k_{0(7)}k_{0(6)}\ldots k_{0(0)}d_{0(14)}d_{0(13)}\ldots d_{0(0)}iv_{0(7)}iv_{0(6)}\ldots iv_{0(0)}]\,,\\
z_2 &:= [d_{0(14)}\ldots d_{0(0)}iv_{0(7)}\ldots iv_{0(0)}k_{0(7)}\ldots k_{0(0)}]\,,\\
z_3 &:= [d_{4(2)}d_{4(1)}d_{4(0)}iv_{4(7)}\ldots iv_{4(0)}k_{4(7)}\ldots k_{4(0)}d_{4(14)}\ldots d_{4(3)}]\,,\\
z_4 &:= [d_{10(1)}d_{10(0)}iv_{10(7)}\ldots iv_{10(0)}k_{10(7)}\ldots k_{10(0)}d_{10(14)}\ldots d_{10(2)}]\,,\\
z_5 &:= [d_{13(5)}\ldots d_{13(0)}iv_{13(7)}\ldots iv_{13(0)}k_{13(7)}\ldots k_{13(0)}d_{13(14)}\ldots d_{13(6)}]\,,\\
z_6 &:= [d_{15(7)}\ldots d_{15(0)}iv_{15(7)}\ldots iv_{15(0)}k_{15(7)}\ldots k_{15(0)}d_{15(14)}\ldots d_{15(8)}]\,.
\end{aligned}$$

In the C implementation of ZUC in [8], the z_i's are added modulo $(2^{31}-1)$ as $((((z_1+z_2)+z_3)+z_4)+z_5)+z_6$,[4] using the *Add*() subroutine. Recall that the $d_i(j)$'s are known (see Table 2). There is no vector $[z_{1(30)}z_{2(30)}\ldots z_{6(30)}]$ such that an end carry is not produced. This is because $d_{0(14)}=1$ and $d_{15(7)}=1$. Let c_1 denote the carry bit produced by the addition of $z_{1(29)}$, $z_{2(29)}$ and the carry coming in from bit position 28 (bit position 0 denotes the LSB), in the first step of the *Add*() subroutine. The sum bit in this addition is added with $z_{3(29)}$ and the corresponding carry coming in from bit position 28.[5] Let c_2 denote the carry bit produced therefrom. Similarly c_3, c_4 and c_5 are defined. The only binary vectors $\Gamma := [c_1c_2\ldots c_5z_{1(30)}]$ that are capable of producing end carry exactly once are:

$$\begin{aligned}
\Gamma_1 &:= [0\,0\,0\,0\,0\,0]\,,\\
[\Gamma_2\Gamma_3\Gamma_4\Gamma_5\Gamma_6\Gamma_7]^T &:= \mathbf{I}_6\,,
\end{aligned}$$

where $\mathbf{I}_6$ is the identity matrix of size 6.

Clarification: Among the MSBs of the 31-bit z_i's, all but the MSB of z_1 are known to us. Let us, for example, suppose that this unknown bit is 1. Then, we are bound to have a carry-out (in other words, carry-in at the bit position 31 or 'end carry'). Since the z_i's are added progressively modulo $(2^{31}-1)$, we can have end carry produced many times (λ, say, in total). If the MSBs of the z_i's are all variables, λ is bounded from above by 5, the number of additions modulo

[4] Evidently there are other orders in performing the modular additions; e.g., $((((z_1+z_3)+z_2)+z_4)+z_5)+z_6$. However, a similar analysis as that in this paper can be performed for each of these orders.

[5] Strictly speaking, the sum bit may be flipped before it is added with $z_{3(29)}$ and the carry-in from bit position 28. This is because of the increment-by-1 step in *Add*(). However, the sum bit is flipped only *(i)* when there is an end carry and *(ii)* if all the 29 LSBs in the sum are 1's. The probability for such an event is intuitively negligible, even considering that many bits of the z_i's are constants. We therefore ignore such bit flips.

2^{32}. (For the case at hand, though, this upper bound is conjectured to be 3 by means of a simulation.)

Now, what must be the carry-in's at the bit position 30, for each of these additions, such that we have only one carry-out? It is rather straightforward to see that the answer is [00000] for the 5 additions. If one of these bits is 1 instead of 0, then we would certainly have one more carry-out. Thus, when the MSB of z_1 is 1, the only *favourable* carry vector is [00000]. This is what Γ_7 means. We similarly have $\Gamma_1, \Gamma_2, \ldots, \Gamma_6$ as the favourable binary vectors for the case when the MSB of z_1 is 0. □

Reverting back to the Γ_i's, one can see that in 5 out of 7 cases, $z_{1(30)} = 0$ and $c_1 = 0$. In each of $z_1, z_2, \ldots, z_6$, we have the unknown key bits, (un)known IV bits and known d-bits. If all the 31 z-bits are unknown variables, one could assume that they are uniformly distributed at random[6] and evaluate the likelihood of the occurrence of each of $\Gamma_1, \Gamma_2, \ldots, \Gamma_7$.[7] Because at least 15 bits of each of $z_1, z_2, \ldots, z_6$ are constants, the assumption of uniform distribution cannot be right away made anymore. If the IV is a known constant, one can assume that the 40 key bits $k_0||k_4||k_{10}||k_{13}||k_{15}$ are uniformly distributed at random and compute $Pr(\Gamma_i)$, for $i \in \{1, 2, \ldots, 7\}$, by running a simulation. Otherwise, the 40 IV bits $iv_0||iv_4||iv_{10}||iv_{13}||iv_{15}$ may also be assumed to be uniformly distributed at random, and the probabilities $Pr(\Gamma_i)$ estimated theoretically. However, the latter approach appears to be highly involved, so we instead performed Experiment 1.

Experiment 1 *The key/*IV *bytes k_0 and iv_0 are exhaustively varied, setting every other key/*IV *byte to $0x00$, and the cases where end carry is produced exactly once, when the $z_1, z_2, \ldots, z_6$ are added modulo $(2^{31} - 1)$, are examined.*

We found 6995 such cases (out of a total of $256 \times 256 = 65536$ cases). In 3444 of the cases, the vector was Γ_6; in 3030 cases, Γ_5; and in the remaining 521 cases, the vector was Γ_3. (A few of these cases are listed in Appendix A.) Firstly, this affirms that there are binary vectors that occur in practice. Next, if these are the only such vectors that occur in practice, then we have recovered $z_{1(30)}$, or the MSB of k_0, with probability 1 when the time taken to execute (2) is at its minimum. This minimum time period would naturally be $T_{const} + T_{carry}$, with T_{const} being the *constant time* component (i.e., the sum total of the execution times of the steps, of the *Add*()'s invoked for (2), that are independent of the respective x's and y's). With this, let us proceed to the second step of the initialization mode, viz.,

$$s_{16} = v + u \bmod (2^{31} - 1)\,, \tag{3}$$

[6] The probability distribution here is *a priori*.

[7] Here, one may choose to ignore negligible biases in the carry probabilities. For example, when two 32-bit words are added modulo 2^{32}, the carry-in at the MSB position is likely to be 0 with a very small bias probability of 2^{-32}. Bias probabilities of the carries generated in modular sums have been examined in several works [23, 17, 21, 20].

where $u = W \gg 1$ (see Sect. 2). We shall now argue that there are significantly many cases where (3) does not involve an end carry generation.

We performed Experiment 1 again, this time counting the frequency at which the MSB of the 31-bit v took the value 0. The total number of such cases was 32840, translating to a probability of 0.5011. Therefore, $v_{(30)}$ appears to be uniformly distributed at random. The first value that u takes after it is initialised is $W = (X_0 \oplus R_{1(ini)}) + R_{2(ini)} \bmod 2^{32}$, where $R_{1(ini)}$ and $R_{2(ini)}$ are the initial values of R_1 and R_2, respectively. From [8, Sect. 3.6.1], we infer that $R_{1(ini)} = 0$ and $R_{2(ini)} = 0$. Hence, $W = X_0$ and

$$\begin{aligned} u = W \gg 1 &= X_0 \gg 1 \\ &= s_{15H} || s_{14L} \gg 1 \\ &= [s_{15(30)} s_{15(29)} \ldots s_{15(15)}] || [s_{14(15)} s_{14(14)} \ldots s_{14(1)}] \\ &= k_{15(7)} || \{0,1\}^{30}; \end{aligned} \tag{4}$$

and this is value of u that goes into step 2 of the first invocation of Algorithm 1. Since $k_{15(7)}$ is an unknown key bit, $u_{(30)}$ can be reasonably assumed to be uniformly distributed at random. Given this, even if the carry-in at the bit position 30 were to be heavily biased towards 1, with 0.25 probability we would still have the carry-out to be 0. In summary, the minimum execution time of Algorithm 1 can reasonably be expected to be $T'_{const} + T_{carry}$, T'_{const} being the constant time component, for at least 25% of the key-IV pairs. We shall now show two ways to measure the execution time of Algorithm 1 and, using it, recover key-dependent information.

1. Through cache measurements: In [26], Zenner makes a mention of a side-channel oracle `ACT_KEYSETUP()` that provides an asynchronous cache adversary a list of all cache accesses made by `KEYSETUP()`, the key setup algorithm of HC-256, in chronological order. Similarly, we introduce an oracle `ACT_Algorithm-3()` that provides the adversary with a chronologically ordered list of all cache accesses made by Algorithm 3. Zenner does not mention in [26] whether or not such an ordered list normally contains the time instants of the cache accesses as well. We assume that the instants are contained in the list. This is a rather strong assumption because in the absence of the oracle, the adversary has to have considerable control over the CPU of the legitimate party, in order to obtain the cache access times.

Given this assumption, the adversary scans through the list and calculates the time difference between the third and the fourth accesses of the S-box S. The first access to S is when it is initialised. Before Algorithm 1 is invoked for the first time, the nonlinear function F is computed (see Algorithm 3). During this computation, S is accessed twice (see the definition of F in Sect. 2). The next (i.e., the fourth) access of S happens after a few constant-time operations (e.g., executing the BR layer, computing W) that follow the first invocation of Algorithm 1. Let the time taken to perform these operations be denoted by T''_{const}. Then, the aforesaid time difference between the third and the fourth cache accesses of S provides the adversary with $T'_{const} + \lambda T_{carry} + T''_{const}$, $\lambda \in \{1, 2, 3\}$.

The adversary can easily measure T_{carry}, T'_{const} and T''_{const} by simulating with an arbitrarily chosen key-IV pair (in practice, quite a few pairs will be required for precision). Thereby, the adversary obtains the value of λ. When $\lambda = 1$, the adversary is able to recover the MSB of k_0 immediately with probability 1.

Now, since Experiment 1 cannot be performed over all key-IV pairs, we reasonably assume that $\Gamma_1, \Gamma_2, \ldots, \Gamma_7$ are equally likely to occur in practice. Under this assumption, $Pr(k_{0(7)} = 0)$ falls to $6/7 = 0.8571$. This probability is further reduced to $5/7 = 0.7143$ if we are to additionally have $c_1 = 0$.

The timing analysis above assumes that S is in cache. This is a very realistic assumption for the following reason. In [8, Appendix A], the S-box S is implemented using two 8×8 lookup tables, viz., $S0$ and $S1$. Encryption performed many times on a single CPU would ideally result in the elements of these tables to be frequently accessed. And, every element of $S0$ and $S1$ could be expected to be accessed frequently if each encryption, in turn, invokes Algorithm 5 multiple times (i.e., long keystream is generated). This would ideally place the lookup tables in the cache.

2. Using statistical methods that do not involve any cache measurement: The execution time of Algorithm 1 can also be estimated without performing cache measurements. Let us recall that Algorithm 1 is run 32 times during the initialization process (see Algorithm 3). Following this, Algorithm 2 is run once (along with constant time steps of Algorithm 4 and Algorithm 5) before the first 32-bit keystream word is output (see Algorithms 4 and 5). Now, the first step of Algorithm 2 is identical to the first step of Algorithm 1. The subsequent steps of Algorithm 2 are constant time operations.[8] Thereby, the total execution time till the first keystream word is generated is

$$T'''_{const} + T_{carry} \cdot (\sum_{j=0}^{31} \lambda_j) + T^{(w)}_{const} + \lambda^{(w)}_1 \cdot T_{carry} \, , \tag{5}$$

where

1. T'''_{const} is the sum total of $32 \cdot T'_{const}$ and the constant-time steps of Algorithm 3;
2. λ_j, $j \in \{0, 1, \ldots, 31\}$, is the number of times end carry is generated in the $(j+1)$th iteration of Algorithm 1;
3. $T^{(w)}_{const}$ is the sum total of the execution times of the constant time steps of Algorithm 2, plus the time to compute steps 1–3 of Algorithm 4 and steps 2–4 of Algorithm 5;

[8] Throughout this paper, we ignore steps 3 and 4 of Algorithm 1 (and, naturally, steps 2 and 3 of Algorithm 2) because the event $s_{16} = 0$ occurs randomly with probability 2^{-31} which is negligible when compared to the probability that end carry is generated exactly once. Besides, the step 4 of Algorithm 1 is just an assignment operation and consumes only a small fraction of the time it takes to perform one 32-bit AND and one 32-bit addition. Therefore, we can safely assume that steps 3 and 4 of Algorithm 1 have negligible influence on the timing analysis.

4. $\lambda_1^{(w)}$ is the λ of the first run of Algorithm 2;
5. $\lambda_j, \lambda_1^{(w)} \in \{0, 1, \ldots, 5\}$, $\forall j \in \{0, 1, \ldots, 31\}$.

Let us now try to estimate the mean of the λ's assuming the z-terms are uniformly distributed from iteration 17 of Algorithm 1 onwards. This assumption is very reasonable at iteration 17, when every LFSR element has been updated once, and the subsequent iterations. We performed Experiment 2 to determine the mean.

Experiment 2 *The new* $[z_{1(30)} z_{2(30)} \ldots z_{6(30)}]$ *is exhaustively varied, and so is* $[c_1 c_2 \ldots c_5]$*. The* λ *for each* $[z_{1(30)} z_{2(30)} \ldots z_{6(30)} c_1 c_2 \ldots c_5]$ *is counted.*

We obtained the frequencies 12, 220, 792, 792, 220, and 12 for $\lambda = 0, 1, 2, 3, 4, 5,$ and 6 respectively.

From these frequencies we obtain that the mean λ,

$$\bar{\lambda} = \frac{0 \cdot 12 + 1 \cdot 220 + 2 \cdot 792 + 3 \cdot 792 + 4 \cdot 220 + 5 \cdot 12}{2^{11}} = 2.5\,. \tag{6}$$

For the iterations 17–32 of Algorithm 1 and iteration 1 of Algorithm 2, the expected cumulative λ is $17 \cdot \bar{\lambda} = 42.5$. The cumulative λ (expected) can be computed for iterations 2–16, but in these computations one needs to make certain assumptions. This is because, in any iteration before the 17th, at least one of the z-vectors is composed of bits loaded directly from the key, IV and the d-constants. Assuming that the incoming carries at the bit position 30 are uniformly distributed can make the λ calculations erroneous. One may instead resort to simulations, but even then would have to perform extrapolations. For example, if the IV is unknown, then in iteration 2, to determine

- $Pr(c_1 = 0)$, the simulation takes $O(2^{15})$ time (15 unknown key and IV bits);
- $Pr(c_2 = 0)$, the simulation takes $O(2^{15} \cdot 2^{16}) = O(2^{31})$ time (16 unknown key and IV bits and 2^{15} possible outputs of the previous simulation);
- $Pr(c_3 = 0)$ or $Pr(c_4 = 0)$, the simulation takes $O(2^{31} \cdot 2^{16}) = O(2^{47})$ time (similar reasoning as the above);
- $Pr(c_5 = 0)$, the simulation takes $O((2^{31} - 1) \cdot 2^{31}) = O(2^{62})$ time (because s_{15} has been changed at the end of iteration 1 and the new s_{15} can assume any value in the set $\{1, 2, \ldots, 2^{31} - 1\}$).

From these probabilities, it is rather easy to compute the average λ by building a truth table of the λ-values and the corresponding vectors $[z_{1(30)}\ z_{2(30)} \cdots z_{6(30)}\ c_1\ c_2\ \ldots\ c_5]$. Such a table would consist of 2^7 rows because $z_{2(30)}$, $z_{3(30)}$, $z_{4(30)}$ and $z_{5(30)}$ are known constants. Looking at the $O(2^{62})$ time complexity, however, one can at the best perform a partial simulation and extrapolate the result. This means that there is always an error in computing the expected λ for each of the iterations 2–16. Hence, we can instead assume that the expected λ is $\bar{\lambda}$ for each of these iterations. This is also error-prone, but we can construct an appropriate credible interval to mitigate the error. This is done as follows. First, upon performing Experiment 2 with more z- (and hence c-) bits and observing the resultant frequencies (i.e., similar to those corresponding to (6)), we will observe that λ is near-normally distributed. Given this, we first choose a confidence level[9] (say, α) and construct a credible interval around $\bar{\lambda}$. To reduce the error in assuming that the λ's of iterations 2–16 are also near-normally distributed, we widen the credible interval corresponding to α while maintaining that the confidence level is α.

Let λ_{min} and λ_{max} denote the lower and upper limits of the resulting credible interval around $\bar{\lambda}$. Now, let us suppose that the attacker clocks the encryption up to the generation of the first keystream word. If this duration falls within the interval (see (5)):

$$\begin{aligned}
&[T'''_{const} + 31 \ \cdot \ T_{carry} \cdot (\bar{\lambda} - \lambda_{min}) + T^{(w)}_{const} + T_{carry} \cdot (\bar{\lambda} - \lambda_{min}) + T_{carry}\,, \\
&\ T'''_{const} + 31 \ \cdot \ T_{carry} \cdot (\bar{\lambda} - \lambda_{min}) + T^{(w)}_{const} + T_{carry} \cdot (\bar{\lambda} - \lambda_{min}) + 2 \cdot T_{carry}) \\
&\qquad\qquad\qquad = [T'''_{const} + T^{(w)}_{const} + 81 \cdot T_{carry} - 32 \cdot \lambda_{min} \cdot T_{carry}\,, \\
&\qquad\qquad\qquad\quad T'''_{const} + T^{(w)}_{const} + 82 \cdot T_{carry} - 32 \cdot \lambda_{min} \cdot T_{carry})\,, \qquad (7)
\end{aligned}$$

then the attacker concludes that the λ for iteration 1 of Algorithm 1 is 1 (just like T'_{const} and T_{carry}, the attacker can measure $T^{(w)}_{const}$). When this is the case, the attacker concludes that $k_{0(7)} = 0$ and $c_1 = 0$ with probability 5/7. □

Given that $k_{0(7)}$ and c_1 are recovered, using $[d_{0(14)} d_{0(13)} \ldots d_{0(7)}] = [10001001]$, we arrive at Theorem 1.

Theorem 1. *When* $c_1 = 0$ *and* $k_{0(7)} = 0$*, we have:*

$$(k_{0(1)} \cdot k_{0(2)} \cdot \bar{k}_{0(3)} + k_{0(3)}) \cdot k_{0(4)} \cdot k_{0(5)} \cdot k_{0(6)} = 0\,, \qquad (8)$$

with the '+' symbol denoting standard integer addition.

Proof. We begin by examining the addition of $[k_{0(7)} k_{0(6)} \ldots k_{0(0)}]$ and $[d_{0(14)} d_{0(13)} \ldots d_{0(7)}]$ while performing the first step of $Add(u32\, z_1, u32\, z_2)$. We know that the incoming carry at the MSB position (of the 31-bit z_1 or z_2) is c_1. Let $c_{1[-1]}, c_{1[-2]}, \ldots, c_{1[-7]}$ denote the incoming carries at the bit positions of $k_{0(6)}, k_{0(5)}, \ldots, k_{0(0)}$, respectively. For the sake of simplicity and clarity, we denote c_1 by $c_{1[0]}$. Now, we know that

$$c_{1[i+1]} = k_{0(i+7)} \cdot d_{0(i+14)} + c_{1[i]} \cdot (k_{0(i+7)} \oplus d_{0(i+14)})\,, i = -1, -2, \ldots, -7\,, \quad (9)$$

where the '+' denotes standard integer addition. Solving the recurrence equation (9), we arrive at (8). □

4.1 Complexities and Success Probabilities

The cache attack requires a few cache timing measurements for precision. If the S-boxes $S0$ and $S1$ are not in the cache, then Eve performs a few encryptions,

[9] The term 'confidence level' is accepted in Bayesian inference also.

using key-IV pairs of her choice, until the instant when Bob starts encrypting. We recall from Sect. 2 that the S-boxes are accessed twice in every iteration of Algorithm 5. From [8, Appendix A], we infer that 4 elements of $S0$ and $S1$ are used in every iteration of Algorithm 5. In the initialization mode, we have 32 similar iterations where F is computed and, hence, $S0$ and $S1$ accessed. Let η denote the number of iterations of Algorithm 5. Then, the total number of iterations per key-IV pair is $32 + 1 + \eta = 33 + \eta$ (includes one iteration of Algorithm 4). This translates to a total of $2 \cdot (33 + \eta)$ ($= \eta'$, say) draws of elements from each of $S0$ and $S1$. Assuming that the draws are uniform and independent, the probability that every 8-bit S-box element appears at least θ times in the list of draws is given by:

$$\frac{1}{256^{\eta'}} \cdot \left(\sum_{\substack{\omega_0, \omega_1, \ldots, \omega_{255} \in \mathbb{N}, \\ \omega_0 + \omega_1 + \ldots + \omega_{255} = \eta', \\ \omega_0 \geq \theta, \omega_1 \geq \theta, \ldots, \omega_{255} \geq \theta}} \binom{\eta'}{\omega_0, \omega_1, \ldots, \omega_{255}} \right), \tag{10}$$

where θ is the number of quickly successive RAM-fetches after which the concerned memory element is placed in the cache. The problem now is to find the smallest η such that the probability given by (10) is reasonably close to 1. We are not aware of any simple method to solve this problem. However, when $\eta' = 256 \cdot \theta$, one *expects* that every element appears θ times in the list of *uniform and independent* draws. Given this, $\eta = 128 \cdot \theta - 33$. Therefore, the attack requires $128 \cdot \theta - 33$ keystream words to be generated with one key-IV pair. The time cost is $(128 \cdot \theta - 33) \cdot T_{KGA} + T_{ini}$, where T_{KGA} is the execution time of one iteration of the keystream generating algorithm (i.e., Algorithm 5) and T_{ini} is the initialization time. Alternatively, the attack can be performed with many key-IV pairs with each generating fewer keystream words. The time complexity in this case will obviously be higher than $(128 \cdot \theta - 33) \cdot T_{KGA} + T_{ini}$. But since the attacker does not require the keystream words for the attack (so it is an asynchronous attack even in the stricter viewpoint of Osvik *et al.* [18]), the data complexity is irrelevant here. Hence, we choose one key-IV pair and mount the attack in order to minimise its time complexity.

As an example, when $\theta = 100$, the pre-computation phase of the single-(key, IV) attack is expected to require $2^{13.64} \cdot T_{KGA} + T_{ini}$ time. In practice, θ is such that the time complexity is not significantly larger than that for $\theta = 100$, we believe. Besides, if the S-boxes are already in the cache, key recovery is almost immediate.

For the statistical timing attack, when the IV is unknown, the attack requires one 32-bit keystream word and the time needed to generate it. The success probability is less than 5/7 because of the errors caused by the approximations involved in the attack. While it seems extremely tedious to accurately compute the error, its magnitude can intuitively be made negligible by choosing a wide credible interval as stated earlier.

4.2 Implications of the Attacks to 128-EEA3

The 3GPP encryption algorithm 128-EEA3 is also a stream cipher that is built around ZUC [7]. It uses a 128-bit "confidentiality key" (denoted in [7] as CK) and encrypts data in blocks of size ranging from 1 bit to 20 kbits. Aside from the ZUC algorithm, the 128-EEA3 contains the following main steps.

Key initialization: The confidentiality key CK initialises the ZUC key in a straightforward manner as follows [7].

Let $CK = CK_0||CK_1||\ldots||CK_{15}$, where each CK_i is a byte. Then,

$$k_i = CK_i\,, \text{ for } i \in \{0, 1, \ldots, 15\}\,. \tag{11}$$

***IV* initialization:** The IV of ZUC is initialised using three parameters of ZUC, viz., $COUNT$, $BEARER$ and $DIRECTION$. The parameter $COUNT$ ($= COUNT_0||COUNT_1||\ldots||COUNT_4$, where each $COUNT_i$ is a byte) is a counter, $BEARER$ is a 5-bit "bearer identity" token and $DIRECTION$ is a single bit that indicates the direction of message transmission [7]. Given these, the IV of ZUC is initialised as:

$$\begin{aligned}
iv_i &= COUNT_i, \text{ for } i \in \{0, 1, 2, 3\}\,,\\
iv_4 &= BEARER||DIRECTION||00_2\,,\\
iv_5 &= iv_6 = iv_7 = 00000000_2\,,\\
iv_j &= iv_{j-8}, \text{ for } j \in \{8, 9, \ldots, 15\}\,.
\end{aligned}$$

From (11), it trivially follows that the timing attacks on ZUC are also attacks on the 128-EEA3, with the corresponding bits of the confidentiality key CK being (partially) recovered. In other words, if bit $k_{i(j)}$ of the ZUC key is recovered then the bit $CK_{i(j)}$ of the 128-EEA3's confidentiality key is recovered as well.

5 Countermeasures

In the previous sections, we described timing weaknesses that are mainly attributable to the design/implementation flaws listed in Observations 1 and 2. Consequently, we see the following countermeasures for the attacks that stem from these weaknesses:

1. A constant-time implementation of the modulo $(2^{31} - 1)$ addition in software and hardware.
2. A more involved key loading procedure.

Table 3 compares and contrasts the two countermeasures.

Of course, a conservative approach would be to complicate the key loading procedure as well as implement the modulo $(2^{31} - 1)$ addition as a constant-time operation.

For the key loading procedure, we suggest the following alternatives:

Table 3. A comparison of the suggested countermeasures

Involved key loading	Constant-time implementation
May open doors to new attacks	At times, like in the case of ZUC, it may be easier to find a safe implementation – this point will become evident from the discussion to follow in this section
Affects the performance only if the key is changed frequently	Can affect the performance even if rekeying is rare
The timing weaknesses of this paper still remain but cannot be exploited to recover the key or key-dependent information	The timing weaknesses of this paper are removed; any other similar timing analysis, however, poses a risk of a straightforward (partial) key recovery

1. *Applying a secure hash function to the s_i's of (1):* A preimage and timing attack resistant hash function would solve our problem, ideally, if applied to the s_i's of (1). The size of the string $[s_{15}s_{14}\ldots s_0]$ is 496 bits. For the countermeasure, this string is fed (after padding) into the compression function of a secure hash function, such as SHA-512 [16], on which there is no known preimage or timing attack despite years of scrutiny.[10] The 512-bit output is truncated to 496 bits, replacing $[s_{15}s_{14}\ldots s_0]$.
2. *Employing 16 carefully chosen, secret $N \times 31$ ($N \geq 31$) S-boxes:* The inputs to the S-boxes (call them B_i, $i \in \{0, 1, \ldots, 15\}$) are the s_i's of (1). When the S-boxes are all secret, $N = 31$ can suffice even though at least 15 input bits are known constants. This is because *(i)* S-boxes are secret, and *(ii)* S-boxes with outputs larger than inputs can still accomplish Shannon's *confusion* [22] (note that Shannon's *diffusion*, as interpreted by Massey in [15], does not apply to stream ciphers) [1].

   Recall that the timing attacks of Sect. 4 can recover only one bit of $B_0(s_0)$ and some information on 6 other bits. While these may be improved in the future (directions for this are provided in Sect. 6) to possibly recover more key bits, recovering an entire 31-bit block seems far-fetched. Actually, with the use of secret S-boxes it is no longer possible, in the first place, to perform the exact same analysis as in Sect. 4. This is because we will have unknown bits in place of the $d_{i(j)}$'s that constitute the MSBs of $z_1, z_2, \ldots, z_6$ (see Sect. 4). Therefore, even upon making precise timing measurements, the attacker will very likely have to guess the bits in place of the $d_{i(j)}$'s before trying to determine the bits in the LFSR. The attacker can, given precise timing measurements, find the number of 0's in $[z_{1(30)}z_{2(30)}\cdots z_{6(30)}]$, but is unlikely to be able to ascertain which bits are 0's. For example, the six z-bits being uniformly distributed (given that the S-boxes are secret) and

[10] There are, however, preimage attacks on step-reduced SHA-512 (see e.g. [2, 13]). The best of these, due to Aoki *et al.* [2], works on 46 steps (out of the total 80), has a time complexity of $2^{511.5}$ and requires a memory of about 2^6 words.

the carries into the bit position 30 being distributed close to uniformly (see footnote 4), there is about $2^{-7.42}$ probability that there is no end carry.[11] Given that there is no end carry, the attacker deduces that there are fewer than two 1's in $[z_{1(30)}z_{2(30)} \dots z_{6(30)}]$. Consequently, the attacker is able to recover at least 4 of the 6 z-bits but she cannot immediately ascertain if a particular bit guess is correct.

Despite the seeming infeasibility, even if an entire 31-bit block $B_i(s_i)$ is recovered somehow, the input key bits cannot be recovered because B_i is secret.

Caveat: As mentioned earlier, an S-box implemented as a lookup table is stored in the cache if its elements are used frequently. One should therefore ensure that the B_i, $i \in \{0, 1, \dots, 15\}$, are placed directly in the processor registers so that memory accesses are avoided. We borrow this idea from [18] where the authors also state that some architectures like the x86-64 and PowerPC AltiVec have register files sufficiently large to store large lookup tables.

Secret S-boxes have previously been used in ciphers (see e.g. GOST [25]). However, *security through obscurity* is in direct violation of the Shannon's maxim [22]. Using a hash function like SHA-512 may be practical provided that the ZUC key is not changed very often.

For the constant-time implementation in software, our suggestion is to change the *Add*() subroutine to the following (we call it "*AddC*()", with the 'C' denoting 'constant-time'):

```
u32 AddC(u32 x, u32 y) {
 u32 z = x + y;
 z = (z & 0x7FFFFFFF) + ((z & 0x80000000) >> 31);
 return z;
}
```

Osvik *et al.* provide some generic countermeasures against cache timing attacks in [18, Sect. 5]. We have already stated one of them, i.e., avoiding memory accesses by placing lookup tables in CPU registers wherever the architecture permits to do so. Some of the other suggestions of [18, Sect. 5] that are relevant to our cache timing analysis are:

1. disabling cache sharing,
2. disabling the cache mechanism per se,
3. adding noise to the cache access pattern (only mitigates the cache timing attack), and
4. adding noise to the timing information (again, only mitigates the attack).

[11] This probability is simply the ratio of the frequency corresponding to $\lambda = 0$ to the total of the frequencies corresponding to (6). The probability was 0 in the timing attacks of Sect. 4 because $d_{0(14)} = 1$ and $d_{15(7)} = 1$.

Placing entire lookup tables in the cache, by the legitimate party, prior to encryption – a process known as *cache warming* – is suggested as a countermeasure in [19]. Let us suppose that our lookup tables fit completely into the cache. Further let us assume that the adversary's instructions or system processes do not evict the contents of these tables. Then, by placing the tables in the cache, one ensures that all S-box accesses are cache hits. Cache attacks that exploit the difference in the register loading time between a cache hit and a miss are precluded by this process. As we do not mount such cache attacks in this paper, cache warming is not a useful countermeasure here.
Nomenclature: To facilitate future reference, we label some of the above, secure modifications of ZUC in Appendix B.

6 Conclusions

In this paper, we have presented timing attacks on the stream cipher ZUC that recover, under certain practical circumstances, one key bit along with some key-dependent information with about 0.7 success probability and negligible time, memory and data. To the best of our knowledge, these are the first attacks on the ZUC cipher of Version 1.5. The following are other highlights of this paper.

- This is one of the very few and early papers analysing the cache timing resistance of stream ciphers. As noted in [14], block ciphers (mainly the AES) have been the prominent targets of cache timing attacks. Besides, cache timing analyses of stream ciphers are recent additions to the cryptanalysis literature, with the first paper (viz., [26]) being published as late as 2008 [14].
- The statistical timing attack is novel, to the best of our knowledge.
- The timing attacks of this paper warn that algorithms must be designed or implemented to resist single-round/iteration timing weaknesses. This single round can even belong to the key/IV setup of the cipher.

The weaknesses we have found that lead us to the attacks may be certificational. Nonetheless, we see a possibility for improving the attacks to recover a few other key bits by, for example, examining the cases where end carry is generated twice.

We have also proposed modifications to ZUC that resist not only the initiatory timing attacks but, evidently, also their potential improvements suggested above. Analysis of these new schemes comes across to us as an interesting problem for future research.

Acknowledgements. The author would like to thank Steve Babbage, Hongjun Wu, Erik Zenner and the anonymous referees of Inscrypt 2011 for their useful comments and suggestions.

Update. The timing analyses and countermeasures presented above were privately communicated to the ETSI/SAGE before the *2nd International Workshop on ZUC Algorithm and Related Topics*, and consequently the software implementation of ZUC was modified [10, Sect. 12.9]. The latest C code (see Version 1.6 [9, Appendix A]) implements the *AddC*() subroutine that we have suggested in Sect. 5), in place of the variable-time *Add*(). The ZUC specification with this revised C code has been approved by the 3GPP for inclusion in the LTE standards (see [10, Sect. 12.9] and [11]).

References

1. C. M. Adams, "Constructing Symmetric Ciphers Using the CAST Design Procedure", *Designs, Codes and Cryptography*, vol. 12, pp. 283–316, 1997.
2. K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, L. Wang, "Preimages for Step-Reduced SHA-2", *ASIACRYPT 2009* (M. Matsui, ed.), vol. 5912 of *LNCS*, pp. 578–597, Springer, 2009.
3. M. Bellare, T. Kohno, "Hash Function Balance and Its Impact on Birthday Attacks", *EUROCRYPT 2004* (C. Cachin, J. Camenisch, eds.), vol. 3027 of *LNCS*, pp. 401–418, Springer, 2004.
4. D. J. Bernstein, "Cache-timing attacks on AES", Preprint, 14^{th} April, 2005. Available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
5. J. L. Carter, M. N. Wegman, "Universal Classes of Hash Functions", *Journal of Computer and System Sciences*, vol. 18(2), pp. 143–154, April 1979.
6. *Data Assurance and Communication Security Research Center*, "Workshop Presentations", *First International Workshop on ZUC Algorithm*, 02^{nd}–03^{rd} December, 2010. Available at `http://www.dacas.cn/zuc10/`.
7. *ETSI/SAGE*, "Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specification", *ETSI/SAGE Specification*, Version 1.5, 04^{th} January, 2011.
8. *ETSI/SAGE*, "Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification", Version 1.5, 04^{th} January, 2011.
9. *ETSI/SAGE*, "Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification", Version 1.6, 28^{th} June, 2011 (Published in 2012). Available at `http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/eea3eia3zucv16.pdf`.
10. *ETSI/SAGE*, "Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report", Version 2.0, 09^{th} September, 2011 (Published in 2012). Available at `http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/EEA3_EIA3_Design_Evaluation_v2_0.pdf`.
11. *GSM Association*, "3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3", *Security Algorithms*, 27^{th} July 2012 (last accessed). Available at `http://www.gsma.com/technicalprojects/fraud-security/security-algorithms/`.
12. T. Fuhr, H. Gilbert, J.-R. Reinhard, M. Videau, "A Forgery Attack on the Candidate LTE Integrity Algorithm 128-EIA3", *Cryptology ePrint Archive*, Report 2010/618, 08^{th} December, 2010. Available at `http://eprint.iacr.org/2010/618.pdf`.

13. T. Isobe, K. Shibutani, “Preimage Attacks on Reduced Tiger and SHA-2”, *Fast Software Encryption 2009* (O. Dunkelman, ed.), vol. 5665 of *LNCS*, pp. 139–155, Springer, 2009.
14. G. Leander, E. Zenner, P. Hawkes, “Cache Timing Analysis of LFSR-Based Stream Ciphers”, *IMA International Conference, Cryptography and Coding 2009* (M. G. Parker, ed.), vol. 5921 of *LNCS*, pp. 433–445, Springer, 2009.
15. J. L. Massey, “An Introduction to Contemporary Cryptology”, *Proceedings of the IEEE*, vol. 76(5), pp. 533–549, May 1988.
16. *National Institute of Standards and Technology*, US Department of Commerce, “Secure Hash Standard (SHS)”, *Federal Information Processing Standards Publication*, FIPS PUB 180-3, October 2008. Available at `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.
17. K. Nyberg, J. Wallén, “Improved Linear Distinguishers for SNOW 2.0”, *Fast Software Encryption 2006* (M. J. B. Robshaw, ed.), vol. 4047 of *LNCS*, pp. 144–162, Springer, 2006.
18. D. A. Osvik, A. Shamir, E. Tromer, “Cache Attacks and Countermeasures: the Case of AES (Extended Version)”, revised 20^{th} November, 2005. Available at `http://www.osvik.no/pub/cache.pdf`. Original version: Proceedings of *The Cryptographers' Track at the RSA Conference 2006* (D. Pointcheval, ed.), vol. 3860 of *LNCS*, pp. 1–20, Springer, 2006.
19. D. Page, “Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel”, *Cryptology ePrint Archive*, Report 2002/169, 11^{th} November, 2002. Available at `http://eprint.iacr.org/2002/169.pdf`.
20. P. Sarkar, “On Approximating Addition by Exclusive OR”, *Cryptology ePrint Archive*, Report 2009/047, 03^{rd} February, 2009. Available at `http://eprint.iacr.org/2009/047.pdf`.
21. G. Sekar, S. Paul, B. Preneel, “New Weaknesses in the Keystream Generation Algorithms of the Stream Ciphers TPy and Py”, *Information Security Conference 2007* (J. A. Garay, A. K. Lenstra, M. Mambo, R. Peralta, eds.), vol. 4779 of *LNCS*, pp. 249–262, Springer, 2007.
22. C. E. Shannon, “Communication Theory of Secrecy Systems”, *Bell Systems Technical Journal*, vol. 28(4), pp. 656–715, 1949.
23. O. Staffelbach, W. Meier, “Cryptographic Significance of the Carry for Ciphers Based on Integer Addition”, *CRYPTO 1990* (A. Menezes, S. A. Vanstone, eds.), vol. 537 of *LNCS*, pp. 601–614, Springer, 1991.
24. H. Wu, P. H. Nguyen, H. Wang, S. Ling, “Cryptanalysis of the Stream Cipher ZUC in the 3GPP Confidentiality & Integrity Algorithms 128-EEA3 & 128-EIA3”, *Presentation at the Rump Session of ASIACRYPT 2010*, 07^{th} December, 2010. Available at `http://www.spms.ntu.edu.sg/Asiacrypt2010/Rump%20Session-%207%20Dec%202010/wu_rump_zuc.pdf`.
25. *Gosudarstvennyi Standard*, “Cryptographic Protection for Data Processing Systems,” *Government Committee of the USSR for Standards*, GOST 28147-89, 1989.
26. E. Zenner, “A Cache Timing Analysis of HC-256”, *Selected Areas in Cryptography 2008* (R. M. Avanzi, L. Keliher, F. Sica, eds.), vol. 5381 of *LNCS*, pp. 199–213, Springer, 2009.

A Practical Occurrences of Γ_3, Γ_5 and Γ_6

Table 4 provides some example key-IV values that produce the favourable Γ-vectors (i.e., the vectors that generate end carry exactly once).

Table 4. Some practical occurrences of the vectors Γ_3, Γ_5 and Γ_6 (all entries except those in the last column are in hexadecimal); in each of these examples, $k_i, iv_i = 0 \,\forall\, i \neq 0$

k_0	iv_0	z_1	z_2	z_3	z_4	z_5	z_6	$[c_1 c_2 \ldots c_5 z_{1(30)}]$
0	0	44D700	44D70000	10000AF1	1AF1	1A0000F1	56000047	[000010] (Γ_6)
2	1	144D701	44D70102	10000AF1	1AF1	1A0000F1	56000047	[000010] (Γ_6)
4	0	244D700	44D70004	10000AF1	1AF1	1A0000F1	56000047	[000010] (Γ_6)
5	5	2C4D705	44D70505	10000AF1	1AF1	1A0000F1	56000047	[000010] (Γ_6)
2F	2B	17C4D72B	44D72B2F	10000AF1	1AF1	1A0000F1	56000047	[000100] (Γ_5)
30	7	1844D707	44D70730	10000AF1	1AF1	1A0000F1	56000047	[000100] (Γ_5)
30	28	1844D728	44D72830	10000AF1	1AF1	1A0000F1	56000047	[000100] (Γ_5)
31	2E	18C4D72E	44D72E31	10000AF1	1AF1	1A0000F1	56000047	[000100] (Γ_5)
6E	5C	3744D75C	44D75C6E	10000AF1	1AF1	1A0000F1	56000047	[010000] (Γ_3)
6F	B1	37C4D7B1	44D7B16F	10000AF1	1AF1	1A0000F1	56000047	[010000] (Γ_3)
72	A1	3944D7A1	44D7A172	10000AF1	1AF1	1A0000F1	56000047	[010000] (Γ_3)
75	E	3AC4D70E	44D70E75	10000AF1	1AF1	1A0000F1	56000047	[010000] (Γ_3)

B ZUC Modifications

We list our proposed algorithm/implementation modifications in Table 5.

Table 5. ZUC modifications: To each label we suffix a '+' if one or more of the generic countermeasures suggested by Osvik *et al.* in [18] are applied

Label	**Reference**
ZUC-1.5C	Constant-time software implementation of modulo $(2^{31}-1)$ addition (i.e., implementation of Version 1.5 with $AddC()$ replacing the variable-time $Add()$)
ZUC-1.5H	Involved key loading: hash function
ZUC-1.5S	Involved key loading: S-boxes
ZUC-1.5CH	Constant-time implementation of modulo $(2^{31}-1)$ addition along with involved key loading using a hash function
ZUC-1.5CS	Constant-time implementation of modulo $(2^{31}-1)$ addition along with involved key loading using S-boxes