Faster 128-EEA3 and 128-EIA3 Software

Roberto Avanzi[1] and Billy Bob Brumley[2]

[1] Qualcomm Research, Munich, Germany
mocenigo@qti.qualcomm.com
[2] Qualcomm Research, San Diego, CA, USA
bbb@qti.qualcomm.com

Abstract. The 3GPP Task Force recently supplemented mobile LTE network security with an additional set of confidentiality and integrity algorithms, namely 128-EEA3 and 128-EIA3 built on top of ZUC, a new keystream generator. We propose two novel techniques to improve the software performance of these algorithms. We show how delayed modular reduction increases the efficiency of the LFSR feedback function, yielding performance gains for ZUC and thus both 128-EEA3 and 128-EIA3. We also show how to leverage carryless multiplication to evaluate the universal hash function making up the core of 128-EIA3. Our software implementation results on Qualcomm's Hexagon DSP architecture indicate significant performance gains when employing these techniques: up to roughly a 2-fold and 2.5-fold throughput improvement for 128-EEA3 and 128-EIA3, respectively.

Keywords: Stream ciphers, universal hash functions, ZUC, 128-EEA3, 128-EIA3, carryless multiplication, LTE.

1 Introduction

Existing, widely-deployed, and well-understood confidentiality and integrity algorithms for LTE mobile networks include SNOW 3G-based 128-EEA1 and 128-EIA1 [1] and AES-based 128-EEA2 and 128-EIA2. In 2009, a new requirement arose to augment these with a new set of algorithms. Standardized in 2012, new LTE confidentiality and integrity algorithms 128-EEA3 and 128-EIA3 [2] rely on the new stream cipher ZUC [3] as a keystream generator. Whereas many stream cipher designs use an LFSR over a binary field, ZUC uses a 16-stage LFSR over the field $\mathbb{F}_p$ for $p = 2^{31} - 1$. As a result, ZUC presents a very interesting mix of binary and modular arithmetic. Section 2 discusses these mobile algorithms in more detail.

Existing literature on engineering aspects of ZUC, 128-EEA3, and 128-EIA3 focuses on hardware implementations, including FPGAs [4,5,6,7]. In contrast, this work focuses exclusively on efficient software implementation techniques for these algorithms. Although our techniques are more widely applicable, for concreteness our target architecture is Qualcomm's Hexagon digital signal processor (DSP). Qualcomm's recent MSM8960 and upcoming MSM8974, Snapdragon system on chips (SoCs) aimed at mobile markets, both feature multiple instances of

Hexagon DSPs. Hexagon is the global unit market leading architecture for DSP silicon shipments [8].

The 31-bit finite field in ZUC is an ideal choice to facilitate modular reduction after each addition on 32-bit architectures. However, in the context of curve-based public key cryptography, accumulating the results of several additions before reduction modulo a prime number can be effective: using sufficiently small prime moduli and accumulating without increasing the precision [9,10]. However, the lesson of [11,12] is that increasing the precision of the unreduced accumulator can be acceptable in order to reduce the number of modular reductions, even if this requires extending the modular reduction routine: the latter can take a performance hit but the net effect can still be a significant overall speedup.

Our first contribution is the application of this idea to the ZUC LFSR, and it is the most significant optimization applied to the ZUC keystream generator described in this paper. To our knowledge, this is the first application of the idea to a stream cipher.

Carryless multiplication is a budding trend in commodity microprocessors. In contrast to a typical integer multiplier that multiplies two words in $\mathbb{Z}$, carryless multiplication multiplies two words as polynomials in $\mathbb{F}_2[x]$. Common motivation for integrating this feature in an instruction set architecture (ISA) includes applications to signal processing, finite fields, error correcting codes, and cryptography. In the case of cryptography, previous results show how to leverage carryless multiplication for efficient implementation of many cryptosystems including, but not limited to, GHASH in AES-GCM [13], elliptic curve cryptography (ECC) on curves over binary fields, [14,15], and ECC on Koblitz curves [16].

Our second contribution shows how to leverage carryless multiplication to compute the message authentication code (MAC) or tag for the 128-EIA3 message authentication algorithm (MAA), the core of which is essentially a universal hash function (UHF) similar to the construction by Krawczyk [17, Sec. 3.2]. This allows software to process message bits a word at a time instead of a bit at a time.

Our empirical results in Section 4 demonstrate the effectiveness of these two contributions presented in Section 3. We achieve up to roughly a 2-fold and 2.5-fold throughput improvement for 128-EEA3 and 128-EIA3, respectively, using these techniques. We draw conclusions in Section 5.

2 LTE algorithms

Cipher suites for LTE networks include the following. 128-EEA1 and 128-EIA1 [1] are confidentiality and integrity algorithms, respectively, built upon the stream cipher SNOW 3G [18]. These algorithms are the same as those specified in UEA2 and UIA2 for UMTS networks. 128-EIA1 is a polynomial evaluation (or Galois) scheme. 128-EEA2 and 128-EIA2 are built upon AES, specifically counter mode for confidentiality and CBC-MAC for integrity. The requirement for a third set of algorithms arose in 2009. New algorithms 128-EEA3 and 128-EIA3 [2] are built

upon a new keystream generator ZUC [3]. Evaluation began in 2010 and, after a few minor modifications, this new set of algorithms reached standardization in 2012. This section describes ZUC, 128-EEA3, and 128-EIA3.

2.1 Keystream generator: ZUC

A tried and true design paradigm for secure stream ciphers combines two pivotal ingredients. The first is an LFSR, typically word-based to allow efficient implementation not only in hardware but also in software. Said LFSR stages or cells are typically elements of a binary extension field: $\mathbb{F}_{2^w}$ with $w \in \{8, 32, 64\}$ are common examples. The second is a finite state machine (FSM), the operation of which vaguely resembles a block cipher round function. Said FSM state registers evolve nonlinearly. Clocking the cipher produces a keystream word that is the XOR-sum of the FSM output and LFSR output. In this way, the LFSR output acts as a linear mask on the FSM output, shielding a nonlinear process with a linear process. Stream ciphers that follow this design paradigm include SNOW [19], SNOW 2.0 [20], SOSEMANUK [21], and SNOW 3G [18], all of which utilize an LFSR defined over $\mathbb{F}_{2^{32}}$.

Building upon these previous designs, the ZUC stream cipher instead utilizes a 16-stage LFSR defined over $\mathbb{F}_p$ for $p = 2^{31} - 1$. A description of ZUC follows and Fig. 1 illustrates. Let $\oplus$, $\boxplus$, $\ll$, and $\|$ denote addition in $\mathbb{F}_2^{32}$ (i.e., XOR), addition in $\mathbb{Z}_{2^{32}}$, logical left shift, and concatenation, respectively.

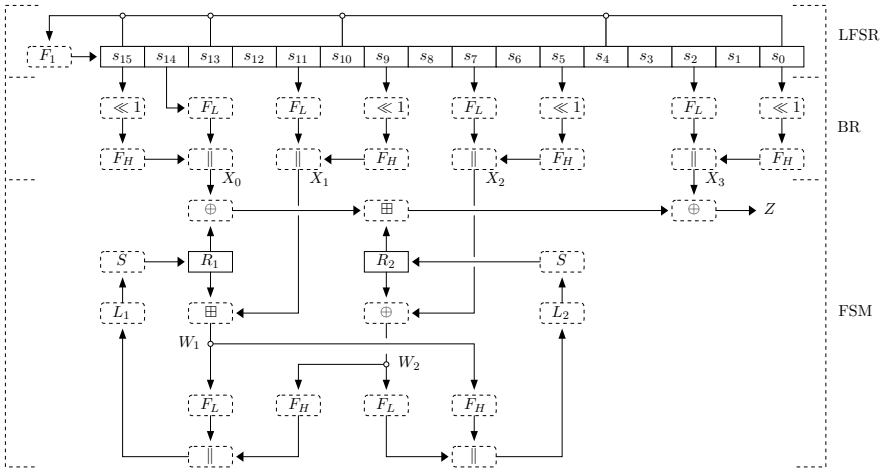

Fig. 1. The ZUC keystream generator. Solid boxes are registers. Dashed boxes are functions. The specification describes the process of extracting and concatenating partial LFSR cells for input to the FSM as the *bit reorganization* (BR) layer.

Let $\mathbb{F}_p = \{1, 2, \ldots, p-1, p\}$, i.e., represent elements using their smallest positive integer representation. This is otherwise the canonical representation

with the exception of representing 0 by p. Define the LFSR feedback function $F_1 : \mathbb{F}_p^5 \to \mathbb{F}_p$ as follows.

$$F_1 : (s_0, s_4, s_{10}, s_{13}, s_{15}) \mapsto (1 + 2^8)s_0 + 2^{20}s_4 + 2^{21}s_{10} + 2^{17}s_{13} + 2^{15}s_{15} \quad (1)$$

Let $F_L, F_H : \mathbb{F}_2^{32} \to \mathbb{F}_2^{16}$ be functions extracting the least and most significant 16-bit word of the input, respectively: $F_L : x \mapsto (x_0, \ldots, x_{15})$ and $F_H : x \mapsto (x_{16}, \ldots, x_{31})$. Let $L_1, L_2 : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ be linear transformations and $S : \mathbb{F}_{2^8}^4 \to \mathbb{F}_{2^8}^4$ be a nonlinear function implemented by four parallel 8 to 8-bit S-boxes. The LFSR and FSM states are $(s_0, \ldots, s_{15})$ and (R_1, R_2), respectively. Clocking ZUC comprises of the following steps.

1. Set $X_0 := F_H(s_{15} \ll 1) \| F_L(s_{14})$, $X_1 := F_L(s_{11}) \| F_H(s_9 \ll 1)$, $X_2 := F_L(s_7) \| F_H(s_5 \ll 1)$, $X_3 := F_L(s_2) \| F_H(s_0 \ll 1)$.
2. Set $Z := (X_0 \oplus R_1) \boxplus R_2) \oplus X_3$, $W_1 := R_1 \boxplus X_1$, $W_2 := R_2 \oplus X_2$.
3. Set $R_1 := S(L_1(F_L(W_1) \| F_H(W_2)))$, $R_2 := S(L_2(F_L(W_2) \| F_H(W_1)))$.
4. Set $s_{16} := F_1(s_0, s_4, s_{10}, s_{13}, s_{15})$.
5. Set $(s_0, \ldots, s_{15}) := (s_1, \ldots, s_{16})$.
6. Return keystream word Z.

In the initialization phase, ZUC packs a 128-bit key, a 128-bit initialization vector (IV), and a number of non-zero constants into the 16 LFSR cells. While the particular assignment is immaterial to this work, it suffices to note that the assignment guarantees all LFSR cell values fall in the range $1 \leq s_i \leq p$. With the LFSR stages set and the FSM initialized to zero, ZUC clocks 32 times, including part of the FSM output as an additional input to the LFSR feedback function in each clock, and discards the resulting keystream words.

2.2 Confidentiality algorithm: 128-EEA3

Built on top of the ZUC keystream generator, 128-EEA3 is a binary additive stream cipher utilizing ZUC keystream words in the obvious way. A 128-bit confidentiality key serves as the ZUC key, while a number of other context-specific parameters (frame counter, etc.) form a 128-bit IV for ZUC. Ciphertext bits are message bits XOR-summed with ZUC keystream word bits.

A noteworthy restriction is that 128-EEA3 limits message lengths from 1 to 65504 bits (i.e., 4 bytes shy of 8kB). The reason is that LTE fixes the MTU (maximum transmission unit) of a PDCP (Packet Data Convergence Protocol) SDU (service data unit) to 8188 bytes [22] to support IPv6 applications (up to 1500 bytes for IPv4).

2.3 Integrity algorithm: 128-EIA3

Built on top of the ZUC keystream generator, 128-EIA3 is an MAA utilizing ZUC keystream words to produce a 32-bit MAC for a message. At a high level, the keystream and message are inputs to an $\mathbb{F}_2$-linear UHF. A dedicated keystream word serves as a one-time pad (OTP), encrypting this 32-bit UHF output. The specification gives the following steps to compute the 128-EIA3 tag. Let ℓ be the bit length of the message m and m_i its individual bits.

1. Generate $L = \lceil \ell/32 \rceil + 2$ ZUC keystream words. Keystream word Z_0 maps to keystream bits $z_0 \| \ldots \| z_{31}$ where z_0 and z_{31} are the MSB and LSB of Z_0, respectively. Word $W_i = z_i \| \ldots \| z_{i+31}$, i.e., 32-bit words formed by sliding a 32-bit window along the keystream bits.
2. Initialize 32-bit word $T := 0$.
3. For i from 0 to $L - 1$: if $m_i = 1$ holds, set $T := T \oplus W_i$.
4. Set $T := T \oplus W_\ell \oplus W_{32(L-1)}$.
5. Return T as the MAC.

We note that the following pre-processing steps, performing message termination, padding, and finalization, can replace Step 4.

1. Append a one followed by 31 zeroes.
2. Append zeroes until the length is a multiple of 32, i.e., from 0 to 31 zeroes.
3. Append a one.

To elaborate, the first 1-bit appended performs message termination. The second 1-bit acts to select a keystream word for inclusion in the sum serving as an OTP, encrypting the output of the UHF. The 0-bit padding between acts to ensure that no part of the OTP is ever used in any other context than encrypting the UHF output (i.e., the OTP is a keystream word never selected fully or partially by post-terminated message bits). For the remainder of this paper, m refers to the post-processed message and m_i its individual bits.

With such pre-processing, the algorithmic steps to compute the MAC can instead be viewed as a function $H : \mathbb{F}_2^n \times \mathbb{F}_2^{n+31} \to \mathbb{F}_2^{32}$ computed as

$$H : (m, z) \mapsto \sum_{i=0}^{n-1} m_i \cdot (z_{i+31}, \ldots, z_i) \tag{2}$$

where here n is the length of the post-processed message. In this light, the function is similar to the formalization by Krawczyk [17, Sec. 3.2].

3 Software optimizations

This section outlines a number of novel techniques used to increase the performance of ZUC, 128-EEA3, and 128-EIA3 software. These techniques can potentially apply to a wide range of architectures: from embedded microprocessors, DSPs, desktops, workstations, on up to server platforms and from 8-bit to 64-bit architectures. The results in Section 4 concentrate on one particular architecture where all of these optimizations apply.

3.1 Delayed modular reduction

We propose delayed modular reduction to optimize the LFSR computation (1). The specification of the cipher exploits the fact that reduction modulo $p = 2^{31} - 1$ is fast by mandating first reduced computation of each summand in (1). Each

multiplication by a power of two is a 31-bit rotation in this field: free in hardware but rather awkward in software. The computation then successively reduces each sum.

The following formula reduces a 32-bit positive value $k \leq 2^{32} - 2$ (such as the sum of two reduced quantities) to the range $1 \leq k' \leq p$. Recall this range is consistent with the definition of $\mathbb{F}_p$ and LFSR initialization described in Sec. 2.

$$k' = (k \;\&\; \texttt{0x7FFFFFFF}) + (k \gg 31) \tag{3}$$

The same formula applies, suitably repeated, to reduce any integer: it is seen at once that any input that is at least 62 bits long shortens by 30 bits (or more). The only 32-bit value that requires (3) to be applied twice is $2^{32} - 1$.

Our approach is to use 64-bit registers (which, on some architectures, are emulated by using pairs of 32-bit registers). We first compute the unreduced sum of *integer values*

$$k = s_0 + 2^8 s_0 + 2^{20} s_4 + 2^{21} s_{10} + 2^{17} s_{13} + 2^{15} s_{15} \;\; .$$

This positive integer is clearly smaller than $2^{31} \cdot 2^{22} = 2^{53}$. Now we proceed to reduce it, first by computing

$$k' = (k \;\&\; \texttt{0x7FFFFFFF}) + (k \gg 31) \leq (2^{31} - 1) + (2^{22} - 1) < 2^{32} - 2$$

and then to further reduce k' to the range $1 \leq k' \leq p$ requires only one additional application of (3) to a 32-bit value. (Stricter bounds can be proved, but these do not improve the analysis.)

Even on a 32-bit architecture, this is faster than the straightforward approach in the specification, as the savings from the fewer reductions vastly offset the more expensive double precision additions.

3.2 Carryless multiplication

Modern commodity microprocessors increasingly feature a carryless multiplication instruction for multiplying two words taken as polynomials in $\mathbb{F}_2[x]$. Similar to integer multiplication, the instruction first computes shifted partial products, yet the final summation in carryless multiplication is an XOR sum, discarding carries. We aim to leverage such an instruction to compute (2) a word at a time rather than a bit at a time.

Define polynomials $a, b, c, d, e \in \mathbb{F}_2[x]$ as follows.

$$a = \sum_{i=0}^{31} z_{31-i} x^i, \quad b = \sum_{i=0}^{31} z_{63-i} x^i, \quad c = ax^{32} + b$$

$$d = \sum_{i=0}^{31} m_i x^i \tag{4}$$

$$e = cd = e_2 x^{64} + e_1 x^{32} + e_0 \tag{5}$$

That is, a is the first 32-bit keystream word as a 31-degree polynomial in $\mathbb{F}_2[x]$, b the next 32-bit keystream word, c the 63-degree polynomial in two 32-bit words. Note the bit ordering of keystream bits to words to polynomials (a, b) is consistent with the standard, yet the ordering in d differs, reversing the message word bits m_i. The three e_i are the 32-bit words of the product of c and d.

Given the above equations, e_1 in (5) is the output of 32 consecutive iterations of the summation in (2). To see why this is so, consider the following matrix.

$$U_0 = \begin{bmatrix} z_{31} & z_{32} & \cdots & z_{62} \\ z_{30} & z_{31} & \cdots & z_{61} \\ \vdots & \vdots & \ddots & \vdots \\ z_0 & z_1 & \cdots & z_{31} \end{bmatrix}$$

Denote $v = (m_0, m_1, \ldots, m_{31})^T$, i.e., d as a column vector, and observe $U_0 v$ computes the first 32 iterations of (2). The low word of the product ad in (5) is $U_1 v$ where U_1 is the following matrix.

$$U_1 = \begin{bmatrix} a_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{31} & a_{30} & \cdots & a_0 \end{bmatrix} = \begin{bmatrix} z_{31} & 0 & \cdots & 0 \\ z_{30} & z_{31} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ z_0 & z_1 & \cdots & z_{31} \end{bmatrix}$$

The high word of the product bd in (5) is $U_2 v$ where U_2 is the following matrix.

$$U_2 = \begin{bmatrix} 0 & b_{31} & \cdots & b_1 \\ 0 & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & b_{31} \\ 0 & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} 0 & z_{32} & \cdots & z_{62} \\ 0 & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & z_{32} \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

The important equality is $U_0 = U_1 + U_2$ hence $U_1 v + U_2 v = U_0 v = e_1$. Repeating these steps for subsequent message words allows us to calculate (2) a message word at a time instead of a message bit at a time, and furthermore using no branch instructions.

Considering software implementation aspects, forming variables a, b, and c have no software implications; they are simply keystream words. In contrast, (4) and (5) must be implemented. Reversing the bits of a message word in (4) requires either a dedicated bit reverse instruction or a sequence of logic implementing a manual bit reverse (bit twiddling, table lookups, etc.). Two instances of a 32 by 32 to 64-bit carryless multiplication instruction (ad, bd) followed by a single 32-bit XOR implement (5) producing e_1 (e_0 and e_2 are discarded). Finally, an additional 32-bit XOR accumulates to the tag.

3.3 Optimizing the S-box

S-boxes S_0 and S_1 (8 to 8 bits) implement the nonlinear function S of ZUC. These must be applied to each byte of a 32 bit word as follows. Let $w = w_0||w_1||w_2||w_3$

be a 32-bit input where each w_i is 8-bit. Its nonlinear transform is $S(w) = S_0(w_0)||S_1(w_1)||S_0(w_2)||S_1(w_3)$, requiring bit shifts and masking for both extracting each w_i and assembling the final result.

Since on the target architecture the ratio of the memory subsystem clock to the CPU clock is relatively small, we reduce the amount of shifts by keeping two 32-bit tables for each S-box where $S'_0 = S_0 \ll 24$, $S''_0 = S_0 \ll 8$, $S'_1 = S_1 \ll 16$, and $S''_1 = S_1$ and computing $S(w)$ as $S(w) = S'_0(w_0) \oplus S'_1(w_1) \oplus S''_0(w_2) \oplus S''_1(w_3)$.

3.4 Optimizing the keystream generator

One important step in the keystream generator is the concatenation of two 16-bit halfwords extracted from 32-bit values into a third 32-bit value, for instance to compute F_L and F_H followed by $\|$ in Fig. 1. There are six such calculations in the keystream generator. The Hexagon architecture includes a `combine` instruction that performs such an operation.

3.5 Classical techniques

We implement the state of the LFSR as a circular buffer, which means that the contents are not shifted completely each time to make room for the new value. A further optimization consists of a *sliding window* on the circular buffer to avoid index wrapping while reading from it [23, Sec. 2.2]: "A buffer up to twice as long as required is used. When a new value is written into the buffer, it is written into two places in the buffer, and all of the intermediate values can be accessed at fixed offsets from the current index or pointer. The pointer starts in the middle of the double length buffer, and when it reaches the end it is reset to the middle again."

Other than this, all optimizations are standard software tuning optimizations, such as, for instance, explicit loop unrolling with overlapping and interleaving of loop body boundaries.

4 Results

Since the architecture of the Hexagon is not widely known, we briefly summarize its salient aspects before presenting and discussing performance results.

4.1 Hexagon architecture

The Hexagon is an unusual type of DSP, because it inherits several features from general purpose CPUs and VLIW machines, enabling it to be programmed with standard development tools and to run generic operating systems with virtualization support.

– Hexagon natively supports a complete RISC instructions set working on integer and floating point types, and vector operations. It features 32 32-bit general purpose registers, which can be paired to form 64-bit registers.
  Furthermore it is *modeless*: most DSPs use sticky register bits to define saturation, scaling or rounding operation modes – as a result, the same routine may work differently depending on the current mode of the DSP. On the Hexagon there are no such sticky bits and saturation, scaling or rounding are encoded in each applicable instruction.
  These design choices make it possible to use standard development tools and languages with the Hexagon. Toolchains based on gcc [24] and clang/llvm [25] are available.
– It is a VLIW architecture. Up to four instructions group together (at compile time) in variable length packets. The packets execute *in order*. The absence of a complex instruction scheduler reduces area and power consumption.
  Furthermore, since some simpler arithmetic operations produce their output early in the pipeline, their output can be provided as an input to other (also simple) operations in the *same* packet – in other words, in some cases both the "old" and the "new" values of a register are available as inputs to other instructions in the same packet.
  Most arithmetic and logic operations can be accumulated: Hexagon does not only offer DSP-typical multiply-and-accumulate operations, but combinations such as and-then-add or add-then-xor as well. Also, various types of operations can be executed conditionally on four different sets of predicates. Special operations support FFT (complex numbers), circular addressing, and zero overhead hardware loops.
  All these features greatly improve code density and throughput.
– Three hardware threads execute in a round robin fashion with single cycle granularity. A 700MHz Hexagon (typical boosted frequency of the DSP inside a recent Qualcomm modem) presents itself as a three-core CPU clocked at 233MHz.
  This feature reduces the visible latency per thread to, usually, a single cycle.
– Whereas traditional DSPs feature a small local memory managed directly by software and rely on DMA to access the data to process, the Hexagon has a unified memory model similar to that of a general purpose CPU: it addresses a linear memory space with memory mapped I/O, integrating an ARM compliant MMU with two stages of translation, has separate L1 data and instruction caches and a large L2 unified cache. Each hardware thread can independently run in user, guest, and monitor execution modes in order to fully support OS virtualization.

Table 1 highlights instructions that are vital to implementing the optimizations described in Section 3 for the Hexagon architecture: The intrinsics can be used as if they were C functions and are directly translated by the compiler into machine instructions.

The ZUC specification mandates big endian byte ordering on keystream words: Hexagon features a dedicated endianness swap instruction. The 128-EIA3

optimizations involve a bit reverse and carryless multiplication: Hexagon features dedicated instructions for both.

Table 1. Noteworthy Hexagon instructions

Mnemonic	Intrinsic	Description
`swiz`	`Q6_R_swiz_R`	4-byte word endianness swap
`brev`	`Q6_R_brev_R`	32-bit word bit reverse
`pmpyw`	`Q6_P_pmpyw_RR`	carryless multiplication of 32-bit operands
`combine`	`Q6_R_combine_R[hl]R[hl]`	combine two halfwords into a word

4.2 Performance

We compare here the performance of two versions of our code.

The first version is a straightforward, clean room implementation of the ZUC specifications, presenting none of the optimizations we described in Section 3. This implementation was the starting point for our work and was also independently checked with the reference code for correctness, showing essentially the same performance. The only significant difference with respect to the reference code is that we used a sliding window over a linear buffer from the start, whereas the reference implementation shifts the whole buffer at each tick.

The second version implements all the optimizations described in Section 3. We achieve these improvements without resorting to manual assembly optimization: all the changes are at the algorithmic level and implemented by inserting compiler intrinsics in the C source code.

For the 128-EIA3 algorithm we also compare the performance of the optimized code but without the special optimization presented in Subsection 3.2, in order to highlight its impact.

Tables 2, 3, and 4 present the performance results as throughput, the unit being *processed bits per hardware thread cycle*.

All the timings include the initialization phase for all three algorithms, whose impact becomes less significant as the buffer size increases. The largest buffer size is 8188 bytes, not 8192, as this is the largest 128-EEA3 packet size, as explained in Subsection 2.2, and for the same reason we also include the length of 1500 bytes.

We include the timings on small buffers to underline the impact of initialization code. However, we note that large buffers are most common, since LTE chiefly carries large amounts of data at high speed.

To build the code we used version 6.2 of Qualcomm's Hexagon development tools. It includes two C compilers based on gcc 4.6.2 [24] and clang 3.2 [25]. We

used the clang compiler since it performs consistently better than gcc on our code base.[3]

Some comments on the results are due:

1. The gain in the keystream generator – *that approaches a 50 % throughput increase* – comes mostly from the improvements in the LFSR – i.e., the delayed modular reduction (Subsection 3.1) and the use of a sliding window circular buffer (Subsection 3.5) – but also, to a lesser extent, the use of the `combine` instruction.
2. Loop unrolling and standard optimizations are the main reason the optimized implementation of EEA3 reduces the gap between the unoptimized implementations of the keystream generator and EEA3: *for large packets throughput roughly doubles.*
3. The impact of routine optimizations is less evident in the case of EIA3 because a straight implementation of the integrity algorithm requires dozens of shifting and masking operations to process just 4 bytes of the message. The mathematical improvements from Subsection 3.2 and the carryless multiplication instruction, however, alone more than double the throughput, bringing it much closer to the performance of the keystream generator alone. *The optimized integrity algorithm performs about two and a half times faster than the standard implementation.*

A single hardware thread on a 700MHz Hexagon is in theory capable to process 168 Mb/sec for integrity and 172 Mb/sec for confidentiality: two threads can meet the LTE CAT 4 150 Mb/sec requirements.

The actual performance is lower since a lightweight operating system is also running on the chip to manage the baseband. However, LTE data streams are split into relatively short segments (of up to 8188 bytes) which can be processed in parallel, so the effective throughput triples with respect of that of a single operation on a single thread. This means that in practice, the Hexagon is comfortably capable of handling LTE CAT 4 150/50 data streams. This performance would not be attainable without the improvements presented here.

4.3 Comparison with Intel CPUs

We also applied the same ideas to an implementation for recent Intel CPUs. The target CPU is an Intel Core i7-2760QM running at 2.40 GHz running OS X 10.8.3 with 16GB 1600MHz DDR3 on board. The chosen compiler is clang, based on LLVM 3.3svn.

The target architecture offers a 64 by 64-bit carryless multiplication that can be used in an obvious way to implement the approach described in Section 3.2 (in fact, a single 64-bit multiplication $(b\|a)$ times d, where d is suitably padded with zeros performs the two 32-bit multiplications needed there).

[3] This also confirms the recent trend of clang quickly catching up with that of gcc, the performance being almost always similar, often slightly better, except on OpenMP multiprocessor code, since clang does not yet support it.

Table 2. ZUC keystream generator performance

Length (bytes)	Unoptimized (bits/cycle)	Optimized (bits/cycle)	Throughput increase
128	0.2792	0.3989	43 %
256	0.3774	0.5460	45 %
512	0.4581	0.6694	46 %
1024	0.5128	0.7547	47 %
1500	0.5332	0.7865	48 %
2048	0.5454	0.8060	48 %
4096	0.5634	0.8344	48 %
8188	0.5728	0.8494	48 %

Table 3. 128-EEA3 confidentiality algorithm performance

Length (bytes)	Unoptimized (bits/cycle)	Optimized (bits/cycle)	Throughput increase
128	0.2225	0.3802	71 %
256	0.2835	0.5136	81 %
512	0.3285	0.6229	90 %
1024	0.3568	0.6971	95 %
1500	0.3670	0.7230	97 %
2048	0.3729	0.7412	99 %
4096	0.3815	0.7654	101 %
8188	0.3860	0.7782	102 %

Table 4. 128-EIA3 integrity algorithm performance

	a	b	c			
Length	Unoptimized	No clmul	Optimized	Throughput increase		
(bytes)	(bits/cycle)	(bits/cycle)	(bits/cycle)	b over a	c over b	c over a
128	0.1665	0.2124	0.3344	28 %	57 %	101 %
256	0.2175	0.2709	0.4502	25 %	66 %	107 %
512	0.2568	0.3143	0.5800	22 %	85 %	126 %
1024	0.2745	0.3336	0.6528	22 %	96 %	138 %
1500	0.2826	0.3423	0.6838	21 %	100 %	142 %
2048	0.2884	0.3484	0.7104	21 %	104 %	146 %
4096	0.2959	0.3563	0.7356	20 %	107 %	149 %
8188	0.2998	0.3603	0.7552	20 %	110 %	152 %

The Intel chip does not offer a bit reversal instruction, but this can be accomplished using a well-know trick with simple shifting and masking: first swap adjacent bits, then adjacent bit pairs, then adjacent nibbles, and so on. This technique easily vectorizes to bit reverse four 32-bit words in parallel.

The optimized performance for EIA3 is 0.6407 bits per cycle for 1500-byte messages, whereas the unoptimized performance is 0.2471 bits per cycle. These values are slightly lower than those of the Hexagon. The relative improvement is around 159%.

A further data point is ZUC keystream generation at 1500 bytes: the (optimized) throughput is 0.8733 bits per cycle, which is about 10% higher than on the Hexagon. This means that the improvements based on the use of carryless multiplication have a higher impact on the Hexagon.

For 8188-byte messages, the ZUC throughput is essentially unchanged at 0.8789 bits per cycle, whereas unoptimized/optimized EIA3 throughputs are 0.2572 and 0.6774 bits per cycle, respectively. These values are significantly lower than the values for the Hexagon.

5 Conclusion

Being not only new algorithms but also standardized and widely-deployed, ZUC, 128-EEA3, and 128-EIA3 are ideal candidates to consider performance optimizations. To this end, the two novel software techniques presented in this work prove highly effective on a particularly relevant platform for these algorithms. Delayed modular reduction for ZUC and carryless multiplication for 128-EIA3 yield up to roughly a 2-fold and 2.5-fold throughput improvement for 128-EEA3 and 128-EIA3, respectively, demonstrated on Qualcomm's Hexagon DSP architecture.

Delayed modular reduction stems from public key cryptography optimization techniques and applies them to a stream cipher. This shows that the linear part of linear masking stream ciphers, traditionally accomplished with an $\mathbb{F}_2$-linear process, can indeed be efficiently realized in other algebraic structures providing similar provable theoretic properties.

Our proposed use of carryless multiplication to evaluate the UHF in 128-EIA3 shows yet another application of this increasingly important microprocessor instruction to standardized symmetric cryptography. In 1999, Nevelsteen and Preneel wrote that Krawczyk's UHF construction "is more suited for hardware, and is not very fast in software" [26, Sec. 3.4]. Unquestionably true at the time, this work exemplifies ways cryptography engineering has evolved to make mutually exclusive design concepts more compatible. On one hand, the throughput improvement shows our proposed technique is dramatically effective. On the other hand, the bit ordering mandated by the specification implies an obtuse bit reversal on message words: fortunately, Hexagon is equipped to handle this natively, but this is not the case for all architectures. Cryptographically speaking, this bit ordering is irrelevant and highlights the importance of careful consideration and close collaboration between cryptologists, standardization bodies, and cryptography engineers.

Acknowledgments We thank Alex Dent for his input on 128-EIA3 performance optimizations.

References

1. ETSI/SAGE: Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2: UEA2 and UIA2 specification. Document 1, Version 2.1 (March 2009)
2. ETSI/SAGE: Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3: 128-EEA3 & 128-EIA3 specification. Document 1, Version 1.7 (December 2011)
3. ETSI/SAGE: Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3: ZUC specification. Document 2, Version 1.6 (June 2011)
4. Kitsos, P., Sklavos, N., Skodras, A.N.: An FPGA implementation of the ZUC stream cipher. In: DSD, IEEE (2011) 814–817
5. Wang, L., Jing, J., Liu, Z., Zhang, L., Pan, W.: Evaluating optimized implementations of stream cipher ZUC algorithm on FPGA. In Qing, S., Susilo, W., Wang, G., Liu, D., eds.: ICICS. Volume 7043 of Lecture Notes in Computer Science., Springer (2011) 202–215 Previous version "Efficient Pipelined Stream Cipher ZUC Algorithm in FPGA" presented at the 1st International Workshop on ZUC Algorithm, Beijing, China, December 2-3, 2010, at `http://www.dacas.cn/zuc10/`. Slides for this version, from the 2st International Workshop on ZUC Algorithm, Beijing, China, June 5-6, 2011, at `http://www.dacas.cn/zuc11/`.
6. Traboulsi, S., Pohl, N., Hausner, J., Bilgic, A., Frascolla, V.: Power analysis and optimization of the ZUC stream cipher for LTE-advanced mobile terminals. In: Proceedings of the 3rd IEEE Latin American Symposium on Circuits and Systems (LASCAS '12), Playa del Carmen, Mexico. (February 2012) 1–4
7. Kitsos, P., Sklavos, N., Provelengios, G., Skodras, A.N.: FPGA-based performance analysis of stream ciphers ZUC, SNOW3G, Grain V1, Mickey V2, Trivium and E0. Microprocessors and Microsystems - Embedded Hardware Design **37**(2) (2013) 235–245
8. Forward Concepts: Qualcomm Leads in Global DSP Silicon Shipments (November 2012) `http://www.fwdconcepts.com/dsp111212.htm`.
9. Lim, C.H., Hwang, H.S.: Fast implementation of elliptic curve arithmetic in GF(p^n). In Imai, H., Zheng, Y., eds.: Public Key Cryptography. Volume 1751 of Lecture Notes in Computer Science., Springer (2000) 405–421
10. Gonda, M., Matsuo, K., Aoki, K., Chao, J., Tsujii, S.: Improvements of addition algorithm on genus 3 hyperelliptic curves and their implementation. IEICE Transactions **88-A**(1) (2005) 89–96
11. Avanzi, R., Mihailescu, P.: Generic efficient arithmetic algorithms for PAFFs (processor adequate finite fields) and related algebraic structures (extended abstract). In Matsui, M., Zuccherato, R.J., eds.: Selected Areas in Cryptography. Volume 3006 of Lecture Notes in Computer Science., Springer (2003) 320–334
12. Avanzi, R.: Aspects of hyperelliptic curves over large prime fields in software implementations. In Joye, M., Quisquater, J.J., eds.: CHES. Volume 3156 of Lecture Notes in Computer Science., Springer (2004) 148–162
13. Gueron, S., Kounavis, M.E.: Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Inf. Process. Lett. **110**(14-15) (2010) 549–553

14. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In Preneel, B., Takagi, T., eds.: CHES. Volume 6917 of Lecture Notes in Computer Science., Springer (2011) 108–123
15. Oliveira, T., Lpez, J., Aranha, D.F., Rodrguez-Henrquez, F.: Lambda coordinates for binary elliptic curves. Cryptology ePrint Archive, Report 2013/131 (2013) `http://eprint.iacr.org//2013/131`.
16. Aranha, D.F., Faz-Hernández, A., López, J., Rodríguez-Henríquez, F.: Faster implementation of scalar multiplication on Koblitz curves. In Hevia, A., Neven, G., eds.: LATINCRYPT. Volume 7533 of Lecture Notes in Computer Science., Springer (2012) 177–193
17. Krawczyk, H.: LFSR-based hashing and authentication. In Desmedt, Y., ed.: CRYPTO. Volume 839 of Lecture Notes in Computer Science., Springer (1994) 129–139
18. ETSI/SAGE: Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2: SNOW 3G specification. Document 2, Version 1.1 (September 2006)
19. Ekdahl, P., Johansson, T.: SNOW – a new stream cipher. In: Proceedings of first NESSIE Workshop, Belgium. (2000)
20. Ekdahl, P., Johansson, T.: A new version of the stream cipher SNOW. In Nyberg, K., Heys, H.M., eds.: Selected Areas in Cryptography. Volume 2595 of Lecture Notes in Computer Science., Springer (2002) 47–61
21. Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., Sibert, H.: SOSEMANUK, a fast software-oriented stream cipher. In Robshaw, M.J.B., Billet, O., eds.: The eSTREAM Finalists. Volume 4986 of Lecture Notes in Computer Science. Springer (2008) 98–118
22. 3GPP: 3rd Generation Partnership Project TS 26.233: Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) Specification (Release 11) (March 2013)
23. Rose, G.G.: A stream cipher based on linear feedback over GF(2^8). In Boyd, C., Dawson, E., eds.: ACISP. Volume 1438 of Lecture Notes in Computer Science., Springer (1998) 135–146
24. GCC: The GNU Compiler Collection `http://gcc.gnu.org`.
25. LLVM: The LLVM Compiler Infrastructure `http://llvm.org`.
26. Nevelsteen, W., Preneel, B.: Software performance of universal hash functions. In Stern, J., ed.: EUROCRYPT. Volume 1592 of Lecture Notes in Computer Science., Springer (1999) 24–41