Fast Software Polynomial Multiplication on ARM Processors Using the NEON Engine

Danilo Câmara, Conrado Gouvêa, Julio López, Ricardo Dahab

Fast Software Polynomial Multiplication on ARM Processors using the NEON Engine

Danilo Câmara, Conrado P. L. Gouvêa[*], Julio López[**], and Ricardo Dahab

University of Campinas (Unicamp)
{dfcamara,conradoplg,jlopez,rdahab}@ic.unicamp.br

Abstract. Efficient algorithms for binary field operations are required in several cryptographic operations such as digital signatures over binary elliptic curves and encryption. The main performance-critical operation in these fields is the multiplication, since most processors do not support instructions to carry out a polynomial multiplication. In this paper we describe a novel software multiplier for performing a polynomial multiplication of two 64-bit binary polynomials based on the VMULL instruction included in the NEON engine supported in many ARM processors. This multiplier is then used as a building block to obtain a fast software multiplication in the binary field $\mathbb{F}_{2^m}$, which is up to 45% faster compared to the best known algorithm. We also illustrate the performance improvement in point multiplication on binary elliptic curves using the new multiplier, improving the performance of standard NIST curves at the 128- and 256-bit levels of security. The impact on the GCM authenticated encryption scheme is also studied, with new speed records. We present timing results of our software implementation on the ARM Cortex-A8, A9 and A15 processors.

Keywords: binary field arithmetic, ARM NEON, elliptic curve cryptography, authenticated encryption, software implementation

1 Introduction

Mobile devices such as smartphones and tablets are becoming ubiquitous. While these devices are relatively powerful, they still are constrained in some aspects such as power consumption. Due to the wireless nature of their communication, it is very important to secure all messages in order to prevent eavesdropping and disclosure of personal information. For this reason, the research of efficient software implementation of cryptography in those devices becomes relevant. Both public key and symmetric cryptography are cornerstones of most cryptographic solutions; in particular, the public-key elliptic curve schemes and the symmetric authenticated encryption schemes are often used due to their high efficiency. Elliptic curve schemes include the well known Elliptic Curve Digital Signature

[*] Supported by FAPESP grant 2010/15340-3

[**] Partially supported by a research productivity scholarship from CNPq-Brazil.

Algorithm (ECDSA) and the Elliptic Curve Diffie Hellman (ECDH) key agreement scheme; while the Galois/Counter Mode (GCM) is an important example of authenticated encryption scheme which is included in many standards such as IPSec and TLS.

A significant portion of mobile devices uses processors based on the 32-bit RISC ARM architecture, suitable for low-power applications due to its relatively simple design, making it an appropriate choice of target platform for efficient implementation. Many ARM processors are equipped with a NEON engine, which is a set of instructions and large registers that supports operations in multiple data using a single instruction. Thus, our objective is to provide an efficient software implementation of cryptography for the ARM architecture, taking advantage of the NEON engine. We have aimed for standard protection against basic side-channel attacks (timing and cache-leakage). Our main contributions are: (i) to describe a new technique to carry out polynomial multiplication by taking advantage of the `VMULL` NEON instruction, achieving a binary field multiplication that is up to 45% faster than a state-of-the-art LD [15] multiplication also using NEON; (ii) using the new multiplier, to achieve speed records of elliptic curve schemes on standard NIST curves and of authenticated encryption with GCM; (iii) to offer, for the first time in the literarure, comprehensive timings for four binary NIST elliptic curves and one non-standard curve, on three different ARM Cortex processors. With this contributions, we advance the state of the art of elliptic curve cryptography using binary fields, offering an improved comparison with the (already highly optimized) implementations using prime fields present in the literature. Our code will be available[1] to allow reproduction of results.

Related work. Morozov et al. [20] have implemented ECC for the OMAP 3530 platform, which features a 500 MHz ARM Cortex-A8 core and a DSP core. Taking advantage of the XORMPY instruction of the DSP core, they achieve 2,106 µs in the B-163 elliptic curve and 7,965 µs in the B-283 curve to compute a shared key, which should scale to 1,053 and 3,982 Kcycles respectively.

Bernstein and Schwabe [6] have described an efficient implementation of non-standard cryptographic primitives using the NEON engine on a Cortex-A8 at the 128-bit security level, using Montgomery and Edwards elliptic curves over the prime field $\mathbb{F}_{(2^{255}-19)}$. The primitives offer basic resistance against side-channel attacks. They obtain 527 Kcycles to compute a shared secret key, 368 Kcycles to sign a message and 650 Kcycles to verify a signature.

Hamburg [9] has also efficiently implemented non-standard cryptographic primitives on a Cortex-A9 *without* NEON support at the 128-bit security level, using Montgomery and Edwards Curves over the prime field $\mathbb{F}_{(2^{252}-2^{232}-1)}$, with basic resistance against side-channel attacks. He obtains 616 Kcycles to compute a shared key, 262 Kcycles to sign a message and 605 Kcycles to verify a signature.

Faz-Hernández et al. [7] have targeted the 128-bit security level with a GLV-GLS curve over the prime field $\mathbb{F}_{(2^{127}-5997)^2}$, which supports a four dimensional

[1] http://conradoplg.cryptoland.net/ecc-and-ae-for-arm-neon/

decomposition of the scalar for speeding up point multiplication. The implementation also provides basic resistance against side-channel attacks. They have obtained 417 and 244 Kcycles for random point multiplication on the Cortex-A9 and A15 respectively; 172 and 100 Kcycles for fixed point multiplication and 463 and 266 Kcycles for simultaneous point multiplication.

Krovetz and Rogaway [13] studied the software performance of three authenticated encryption modes (CCM, GCM and OCB3) in many platforms. In particular, they report 50.8 cycles per byte (cpb) for GCM over AES with large messages using the Cortex-A8; an overhead of 25.4 cpb over unauthenticated AES encryption.

Polyakov [22] has contributed a NEON implementation of GHASH, the authentication code used by GCM, to the OpenSSL project. He reports a 15 cpb performance on the Cortex-A8.

Paper structure. This paper is organized as follows. In Section 2 we describe the ARM architecture. In Section 3, the binary field arithmetic is explained, along with our new multiplier based on the the `VMULL` instruction. Section 4 describes the high-level algorithms used and Section 5 presents our results. Finally, concluding remarks are given in Section 6.

2 ARM Architecture

The ARM is a RISC architecture known for enabling the production of low-power processors and is widely spread in mobile devices. It features a fairly usual instruction set with some interesting characteristics such as integrated shifts, conditional execution of most instructions, and optional update of condition codes by arithmetic instructions. There are sixteen 32-bit registers (`R0`–`R15`), thirteen of which are general-purpose. The version 7 of the ARM architecture has added an advanced Single Instruction, Multiple Data (SIMD) extension referred as "NEON engine", which is composed of a collection of SIMD instructions using 64- or 128-bit operands and a bank of sixteen 128-bit registers. These are named `Q0`–`Q15` when viewed as 128-bit, and `D0`–`D31` when viewed as 64-bit. There are many CPU designs based on the ARM architecture such as the ARM7, ARM9, ARM11 and the ARM Cortex series. In this work, we used three ARM Cortex devices, which we now describe.

Cortex-A8. The ARM Cortex-A8 processor is a full implementation of the ARMv7 architecture including the NEON engine. Compared to previous ARM cores the Cortex-A8 is dual-issue superscalar, achieving up to twice the instructions executed per clock cycle. Some pairs of NEON instructions can also be dual-issued, mainly a load/store or permutation instruction together with a data-processing instruction. Its pipeline has 13 stages followed by 10 NEON stages; its L2 cache is internal. The Cortex-A8 is used by devices such as the iPad, iPhone 4, Galaxy Tab, and Nexus S.

Cortex-A9. The ARM Cortex-A9 shares the same instruction set with the Cortex-A8, but it features up to four cores. It no longer supports NEON dual-issue and its L2 cache is external. However, it supports out-of-order execution of regular ARM instructions and register renaming, and has a 9–12 stage pipeline (more for NEON, we were unable to find how many). Devices that feature the Cortex-A9 include the iPad 2, iPhone 4S, Galaxy S II, and Kindle Fire.

Cortex-A15. Implements the ARMv7 architecture, provides dual-issue and out-of-order execution for most NEON instructions and can feature up to four cores. Its pipeline is wider, with 15 to 25 stages. The Cortex-A15 is present in devices such as the Chromebook, Nexus 10, and Galaxy S4.

Instructions. We highlight the NEON instructions which are important in this work, also illustrated in Figure 1. The `VMULL` instruction is able to carry out several multiplications in parallel; the `VMULL.P8` version takes as input two 64-bit input vectors A and B of eight 8-bit binary polynomials and returns a 128-bit output vector C of eight 16-bit binary polynomials, where the i-th element of C is the multiplication of the i-th elements from each input.

The `VEXT` instruction, for two 64-bit registers and an immediate integer i, outputs a 64-bit value which is the concatenation of lower $8i$ bits of the first register and the higher $64 - 8i$ bits of the second register. Note that if the inputs are the same register then the `VEXT` instruction computes right bit rotation by multiples of 8 bits. The instruction also supports 128-bit registers, with similar functionality.

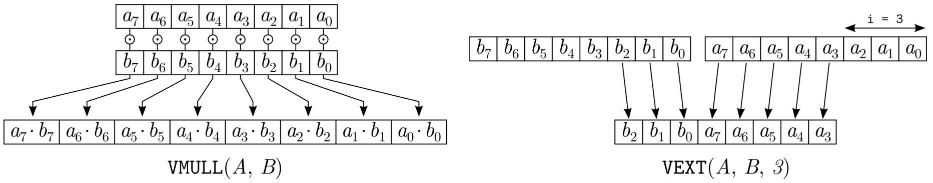
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
$a_7 \cdot b_7$ $a_6 \cdot b_6$ $a_5 \cdot b_5$ $a_4 \cdot b_4$ $a_3 \cdot b_3$ $a_2 \cdot b_2$ $a_1 \cdot b_1$ $a_0 \cdot b_0$
VMULL(A, B)
i = 3
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_2 b_1 b_0 a_7 a_6 a_5 a_4 a_3
VEXT(A, B, 3)

Fig. 1. Main NEON instructions in this work: `VMULL.P8` (shortened as `VMULL)` and `VEXT`. Each square is 8 bits; rectangles are 16 bits.

3 Binary Field Arithmetic

Binary field arithmetic is traditionally implemented in software using polynomial basis representation, where elements of $\mathbb{F}_{2^m}$ are represented by polynomials of degree at most $m - 1$ over $\mathbb{F}_2$. Assuming a platform with a W-bit architecture ($W = 32$ for ARM), a binary field element $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$

may be represented by a binary vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ of length m using $t = \lceil m/W \rceil$ words. Remaining $s = Wt - m$ bits are left unused.

Multiplication in $\mathbb{F}_{2^m}$ (*field multiplication*) is performed modulo $f(z) = z^m + r(z)$, an irreducible binary polynomial of degree m. This multiplication can be carried out in two steps: first, the polynomial multiplication of the operands; second, the polynomial reduction modulo $f(z)$. The basic method for computing the polynomial multiplication of $c(z) = a(z)b(z)$ is to read each i-th bit of $b(z)$ and, if it is 1, xor $a(z) \ll i$ into an accumulator. However, since the left-shifting operations are in general expensive, faster variations of this method have been developed. One of the fastest known methods for software implementation is the López-Dahab (LD) algorithm [15], which processes multiple bits in each iteration. We have implemented it using the NEON engine, taking advantage of the `VEXT` instruction and larger number of registers.

While the LD algorithm is often the fastest in many platforms, the presence of the `VMULL.P8` NEON instruction has the potential to change this landscape. However, it is not obvious how to build a n-bit polynomial multiplier for cryptographic applications ($n \geq 128$) using the eight parallel 8-bit multiplications provided by `VMULL.P8`. Our solution is a combination of the Karatsuba algorithm and a multiplier based on `VMULL.P8` which we have named the Karatsuba/NEON/`VMULL` multiplier (KNV), described below.

3.1 New Karatsuba/NEON/VMULL (KNV) Multiplier

Our new approach was to built a 64-bit polynomial multiplier, which computes the 128-bit product of two 64-bit polynomials. This multiplier was then combined with the Karatsuba algorithm [11] in order to provide the full multiplication.

The 64-bit multiplier was built using the `VMULL.P8` instruction (`VMULL` for short) as follows. Consider two 64-bit polynomials $a(z)$ and $b(z)$ over $\mathbb{F}_2$ represented as vectors of eight 8-bit polynomials:

$$A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0); \quad B = (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0).$$

To compute the polynomial multiplication $c(z) = a(z) \cdot b(z)$ (represented as a vector C), the schoolbook method would require sixty-four 8-bit multiplications with every (a_i, b_j) combination, where each product is xored into an accumulator in the appropriate position. In our proposal, these multiplications can be done with eight executions of `VMULL` by rearranging the inputs. Let $\ggg$ denote a circular right shift; compute $A_1 = A \ggg 8$, $A_2 = A \ggg 16$, $A_3 = A \ggg 24$, $B_1 = B \ggg 8$, $B_2 = B \ggg 16$, $B_3 = B \ggg 24$ and $B_4 = B \ggg 32$ using `VEXT`. This results in:

$$\begin{aligned}
A_1 &= (a_0, a_7, a_6, a_5, a_4, a_3, a_2, a_1); & B_1 &= (b_0, b_7, b_6, b_5, b_4, b_3, b_2, b_1);\\
A_2 &= (a_1, a_0, a_7, a_6, a_5, a_4, a_3, a_2); & B_2 &= (b_1, b_0, b_7, b_6, b_5, b_4, b_3, b_2);\\
A_3 &= (a_2, a_1, a_0, a_7, a_6, a_5, a_4, a_3); & B_3 &= (b_2, b_1, b_0, b_7, b_6, b_5, b_4, b_3);\\
& & B_4 &= (b_3, b_2, b_1, b_0, b_7, b_6, b_5, b_4).
\end{aligned}$$

Now compute these VMULL products:

$$\begin{aligned}
D = \texttt{VMULL}(A, B) &= (a_7b_7, a_6b_6, a_5b_5, a_4b_4, a_3b_3, a_2b_2, a_1b_1, a_0b_0);\\
E = \texttt{VMULL}(A, B_1) &= (a_7b_0, a_6b_7, a_5b_6, a_4b_5, a_3b_4, a_2b_3, a_1b_2, a_0b_1);\\
F = \texttt{VMULL}(A_1, B) &= (a_0b_7, a_7b_6, a_6b_5, a_5b_4, a_4b_3, a_3b_2, a_2b_1, a_1b_0);\\
G = \texttt{VMULL}(A, B_2) &= (a_7b_1, a_6b_0, a_5b_7, a_4b_6, a_3b_5, a_2b_4, a_1b_3, a_0b_2);\\
H = \texttt{VMULL}(A_2, B) &= (a_1b_7, a_0b_6, a_7b_5, a_6b_4, a_5b_3, a_4b_2, a_3b_1, a_2b_0);\\
I = \texttt{VMULL}(A, B_3) &= (a_7b_2, a_6b_1, a_5b_0, a_4b_7, a_3b_6, a_2b_5, a_1b_4, a_0b_3);\\
J = \texttt{VMULL}(A_3, B) &= (a_2b_7, a_1b_6, a_0b_5, a_7b_4, a_6b_3, a_5b_2, a_4b_1, a_3b_0);\\
K = \texttt{VMULL}(A, B_4) &= (a_7b_3, a_6b_2, a_5b_1, a_4b_0, a_3b_7, a_2b_6, a_1b_5, a_0b_4).
\end{aligned}$$

These vectors of eight 16-bit polynomials contain the product of every (a_i, b_j) combination, as required. We now need to xor everything into place. Let $L = E + F$, $M = G + H$ and $N = I + J$. Let k_i be the i-th element of vector K and analogously to L, M and N. Now, compute:

$$\begin{aligned}
P_0 &= (0,\ 0,\ 0,\ 0,\ \ell_7,\ 0,\ 0,\ 0); & P_4 &= (0,\ 0,\ 0,\ 0,\ n_7,\ n_6,\ n_5,\ 0);\\
P_1 &= (0,\ \ell_6,\ \ell_5,\ \ell_4,\ \ell_3,\ \ell_2,\ \ell_1,\ \ell_0); & P_5 &= (0,\ 0,\ 0, n_4,\ n_3,\ n_2,\ n_1, n_0);\\
P_2 &= (0,\ 0,\ 0,\ 0, m_7, m_6,\ 0,\ 0); & P_6 &= (0,\ 0,\ 0,\ 0,\ k_7,\ k_6,\ k_5, k_4);\\
P_3 &= (0,\ 0, m_5, m_4, m_3, m_2, m_1, m_0); & P_7 &= (0,\ 0,\ 0,\ 0,\ k_3,\ k_2,\ k_1, k_0).
\end{aligned}$$

The final result is obtained with:

$$C = A \cdot B = D + (P_0 + P_1) \ll 8 + (P_2 + P_3) \ll 16 + (P_4 + P_5) \ll 24 + (P_6 + P_7) \ll 32.$$

The expansion of the above equation produces the same results of the schoolbook method for multiplication, verifying its correctness. The whole process is illustrated in Figure 2, and Algorithm 6 in the Appendix lists the assembly code for reference. The partial results $(P_0 + P_1) \ll 8$, $(P_2 + P_3) \ll 16$ or $(P_4 + P_5) \ll 24$ can each be computed from L, M or N with four instructions (two xors, one mask operation and one shift). The partial result $(P_6 + P_7) \ll 32$ can be computed from K with three instructions (one xor, one mask operation and one shift). To clarify our approach, we list the assembly code used in the computation of L and $(P_0 + P_1) \ll 8$ from A and B in Algorithm 1 and describe it below.

In Algorithm 1, the 128-bit NEON register `tq` can be viewed as two 64-bit registers such that `tq` = `th`||`tl` where `tl` is the lower part and `th` is the higher part; the same applies to other registers. In line 1, the `VEXT` instruction concatenates the lower 8 bits of A with the higher $(64 - 8) = 56$ bits of A, resulting in the value A_1 being stored in `tl`. Line 2 computes $F = \texttt{VMULL}(A_1, B)$ in the `tq` register. Lines 3 and 4 compute B_1 and then $E = \texttt{VMULL}(A, B_1)$ in the `uq` register, while line 5 computes $L = E + F$ in the `tq` register. Observe that the result we want, $(P_0 + P_1)$, can be viewed as $(0, \ell_6, \ell_5, \ell_4, \ell_3 + \ell_7, \ell_2, \ell_1, \ell_0)$. The straightforward way to compute $(P_0 + P_1)$ from L would be to use a mask operation to isolate ℓ_7, xor it to `tq` in the appropriate position and do another mask operation to clear the highest 16 bits. However, we use another approach

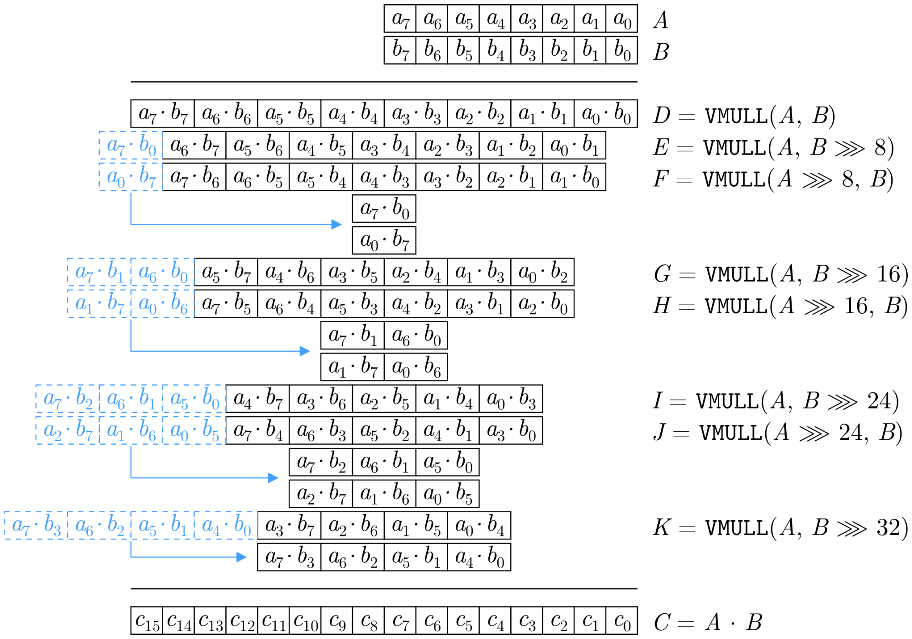
a7 a6 a5 a4 a3 a2 a1 a0 A
b7 b6 b5 b4 b3 b2 b1 b0 B
a7·b7 a6·b6 a5·b5 a4·b4 a3·b3 a2·b2 a1·b1 a0·b0 D = VMULL(A, B)
a7·b0 a6·b7 a5·b6 a4·b5 a3·b4 a2·b3 a1·b2 a0·b1 E = VMULL(A, B ⋙ 8)
a0·b7 a7·b6 a6·b5 a5·b4 a4·b3 a3·b2 a2·b1 a1·b0 F = VMULL(A ⋙ 8, B)
a7·b0
a0·b7
a7·b1 a6·b0 a5·b7 a4·b6 a3·b5 a2·b4 a1·b3 a0·b2 G = VMULL(A, B ⋙ 16)
a1·b7 a0·b6 a7·b5 a6·b4 a5·b3 a4·b2 a3·b1 a2·b0 H = VMULL(A ⋙ 16, B)
a7·b1 a6·b0
a1·b7 a0·b6
a7·b2 a6·b1 a5·b0 a4·b7 a3·b6 a2·b5 a1·b4 a0·b3 I = VMULL(A, B ⋙ 24)
a2·b7 a1·b6 a0·b5 a7·b4 a6·b3 a5·b2 a4·b1 a3·b0 J = VMULL(A ⋙ 24, B)
a7·b2 a6·b1 a5·b0
a2·b7 a1·b6 a0·b5
a7·b3 a6·b2 a5·b1 a4·b0 a3·b7 a2·b6 a1·b5 a0·b4 K = VMULL(A, B ⋙ 32)
a7·b3 a6·b2 a5·b1 a4·b0
c15 c14 c13 c12 c11 c10 c9 c8 c7 c6 c5 c4 c3 c2 c1 c0 C = A · B

Fig. 2. The 64 × 64-bit polynomial multiplier using VMULL

which does not need a temporary register, described as follows. In line 6, we xor the higher part of tq into the lower part, obtaining $(\ell_7, \ell_6, \ell_5, \ell_4, \ell_3 + \ell_7, \ell_2 + \ell_6, \ell_1 + \ell_5, \ell_0 + \ell_4)$. Line 7 uses a mask operation to clear the higher 16 bits of tq, which now holds $(0, \ell_6, \ell_5, \ell_4, \ell_3 + \ell_7, \ell_2 + \ell_6, \ell_1 + \ell_5, \ell_0 + \ell_4)$. In line 8, the higher part of tq is again xored into the lower part, resulting in the expected $(0, \ell_6, \ell_5, \ell_4, \ell_3 + \ell_7, \ell_2, \ell_1, \ell_0)$ which is finally shifted 8 bits to the left with the VEXT instruction in line 9.

3.2 Additional Binary Field Operations

Squaring a binary polynomial corresponds to inserting a 0 bit between every consecutive bits of the input, which often requires precomputed tables. The VMULL instruction can improve squaring since, when using the same 64-bit value as the two operands, it computes the 128-bit polynomial square of that value.

Multiplication and squaring of binary polynomials produce values of degree at most $2m-2$, which must be reduced modulo $f(z) = z^m + r(z)$. Since $z^m \equiv r(z) \pmod{f(z)}$, the usual approach is to multiply the upper part by $r(z)$ using shift and xors. For small polynomials $r(z)$ it is possible to use the VMULL instruction to carry out multiplication by $r(z)$ with a special 8×64-bit multiplier; this was done for $\mathbb{F}_{2^{128}}$ $(r(z) = z^7 + z^2 + z + 1)$ and $\mathbb{F}_{2^{251}}$ $(r(z) = z^7 + z^4 + z^2 + 1)$. Reduction in $\mathbb{F}_{2^{283}}$ takes advantage of the factorization of $r(z) = z^{12} + z^7 + z^5 + 1 =$

Algorithm 1 Computation of L and $(P_0 + P_1) \ll 8$ from A and B

```
Input: 64-bit registers ad (holding $A$), bd (holding $B$) and k48 (holding the constant
    0x0000FFFFFFFFFFFF)
Output: 128-bit register tq (th|tl) (holding $(P_0 + P_1) \ll 8$)
vext.8    tl, ad, ad, $1
vmull.p8  tq, tl, bd
vext.8    ul, bd, bd, $1
vmull.p8  uq, ad, ul
veor      tq, tq, uq
veor      tl, tl, th
vand      th, th, k48
veor      tl, tl, th
vext.8    tq, tq, tq, $15
```

$(z^7 + 1)(z^5 + 1)$ as described in [2]. For $\mathbb{F}_{2^{571}}$, $r(z) = z^{10} + z^5 + z^2 + 1$, and its reduction is computed with the usual shifts and xors.

Field inversion is commonly carried out with the well-known extended Euclidean algorithm, but it does not take constant time and may be vulnerable to side channel attacks. For this reason, we have used the Itoh-Tsujii algorithm [10], which is an optimization of inversion through Fermat's little theorem ($a(x)^{-1} = a(x)^{2^m - 2}$). The algorithm uses a repeated field squaring operation $a(x)^{2^k}$ for some values of k; we have implemented a special function where field squaring is completely done using NEON instruction and registers using the same techniques described for squaring and reduction, but avoiding reads and writes to memory.

4 Algorithms

The KNV multiplier was used as the building block for a implementation of Elliptic Curve Cryptography (ECC) and of authenticated encryption (AE), which we now describe together with our implementation of side-channel resistance.

4.1 Side-Channel Resistance

Side-channel attacks [12] are a serious threat for cryptographic implementations; different attacks require different levels of protection. Here we consider the basic level of resistance which avoids: branching on secret data, algorithms with timings dependent on secret data, and accessing table indexes with secret indices.

The building block of a side-channel resistant (SCR) implementation can be considered the "select" operation $t \leftarrow \textsc{Select}(a, b, v)$, which copies a into t if the bit v is 0 or copies b if v is 1. This operation can be implemented without branching as described in [14] and listed for reference in Algorithm 3 in the Appendix. In ARM assembly, Select can be implemented easily since most instructions can be made conditional to a previous register comparison.

However, a faster approach is to use the NEON instruction `VBIT Qd, Qn, Qm` (bitwise insert if false) — it inserts each bit in `Qn` into `Qd` if the corresponding bit in `Qm` is 1, otherwise it leaves the corresponding bit in `Qd` unchanged. If the m value from Algorithm 3 is stored in `Qm`, then `VBIT` is precisely the Select operation restricted to the case where t and a refer to the same location (which is often the case).

Some of the algorithms we will describe use precomputed tables to improve performance. However, looking up a table entry may leak its index through side-channels, since it affects the contents of the processor cache. For this reason, we employ a side-channel resistant table lookup. We follow the strategy found in the source code of [6], listed for reference in Algorithm 4 in the Appendix, where s can be computed without branches by copying the sign bit of r (e.g. in the C language, convert r to unsigned and right shift the result in order to get the highest bit). We have implemented the SCR table lookup for elliptic curve points entirely in assembly with the `VBIT` instruction. It is possible to hold the entire t value (a point) in NEON registers, without any memory writes except for the final result.

4.2 Elliptic Curve Cryptography

Elliptic Curve Cryptography is composed of public key cryptographic schemes using the arithmetic of points on elliptic curves over finite fields, and it uses shorter keys at the same security level in comparison to alternative public-key systems such as RSA and DSA. Two types of fields are mainly used: prime fields (with p elements, where p is prime) and binary fields (with 2^m elements for some m). While prime fields are used more often (and most literature on ECC for ARM uses them), we decided to study the efficiency of ECC using binary fields with our KNV multiplier.

Four standardized curves for Elliptic Curve Cryptography (ECC) [21] were implemented: the random curves B-283 and B-571 which provide 128 and 256 bits of security respectively; and the Koblitz curves K-283 and K-571 which provide the same bits of security respectively. A non-standard curve over $\mathbb{F}_{2^{251}}$ [5] (“B-251”, roughly 128 bits of security) was also implemented, due to its high efficiency.

The main algorithm in ECC is the point multiplication, which often appears in three different cases: the random point multiplication kP (k terms of the elliptic point P are summed), where the point P is not known in advance; the fixed point multiplication kG, where G is fixed; and the simultaneous point multiplication $kP + \ell G$ where P is random and G is fixed. In the random point case, we chose the Montgomery-LD multiplication [16] which offers high efficiency and basic side-channel resistance (SCR) without precomputed tables. In the fixed point case, the signed multi-table Comb method is employed [9], with side-channel resistant table lookups. It uses t tables with 2^{w-1} points. For simultaneous point multiplication, we have used the interleaving method [8,18] of w-(T)NAF. It employs two window sizes: d for the fixed point (requiring a precomputed table with 2^{d-2} elements) and w for the random point (requiring a on-the-fly table with 2^{w-2} elements). SCR is not required in this case since the algorithm is only used for signature verification, whose inputs are public.

The main advantage of Koblitz curves is the existence of specialized algorithms for point multiplication which take advantage of the efficient endomorphism τ present in those curves [24]. However, we have not used these algorithms since we are not aware of any SCR methods for recoding the scalar k into the representation required by them. Therefore, the only performance gain in those curves were obtained using a special doubling formula with two field multiplications; see Algorithm 2. Montgomery-LD also requires one less multiplication per iteration in Koblitz curves.

Algorithm 2 Our proposed point doubling on the Koblitz curve $E_a : y^2 + xy = x^3 + ax^2 + 1$, $a \in \{0, 1\}$ over $\mathbb{F}_{2^m}$ using LD projective coordinates

```
Input: Point $P = (X_1, Y_1, Z_1) \in E_a(\mathbb{F}_{2^m})$
Output: Point $Q = (X_3, Y_3, Z_3) = 2P$
$S \leftarrow X_1 Z_1$
$T \leftarrow (X_1 + Z_1)^2$
$X_3 \leftarrow T^2$
$Z_3 \leftarrow S^2$
if $a = 0$ then
    $Y_3 \leftarrow ((Y_1 + T)(Y_1 + S) + Z_3)^2$
else
    $Y_3 \leftarrow (Y_1(Y_1 + S + T))^2$
return $(X_3, Y_3, Z_3)$
```

We have selected the three following well known ECC protocols. The Elliptic Curve Digital Signature Algorithm (ECDSA) requires a fixed point multiplication for signing and a simultaneous point multiplication for verification. The Elliptic Curve Diffie-Hellman (ECDH) [4] is a key agreement scheme which requires a random point multiplication, and the Elliptic Curve Schnorr Signature (ECSS) [23] is similar to ECDSA but does not require an inversion modulo the elliptic curve order.

4.3 Inversion modulo the elliptic curve order

When signing, the ECDSA generates a random secret value k which is multiplied by the generator point; this requires side-channel resistance since if k leaks then it is possible to compute the signer's private key. However, an often overlooked point is that ECDSA also requires the inversion of k modulo the elliptic curve order n. This is usually carried out with the extended Euclidean algorithm, whose number of steps are input-dependent and therefore theoretically susceptible to side-channel attacks. While we are not aware of any concrete attacks exploiting this issue, we are also not aware of any arguments for the impossibility of such an attack. Therefore, we believe it is safer to use a SCR inversion.

The obvious approach for SCR inversion would be to use Fermat's little theorem ($a^{-1} \equiv a^{n-2} \pmod{n}$), which would require a very fast multiplier modulo n to be efficient. However, we have found a simple variant of the binary extended Euclidean algorithm by Niels Möller [19] which takes a fixed number of steps. For reference, it is described in Algorithm 5 in the Appendix, where branches are used for clarity and can be avoided with SELECT. The algorithm is built entirely upon four operations over integers with the same size as n: addition, subtraction, negation and right shift by one bit. These can be implemented in assembly for speed; alternatively the whole algorithm can be implemented in assembly in order to avoid reads and writes by keeping operands (a, b, u and v) in NEON registers. We have followed the latter approach for fields at the 128-bit level of security, and the former approach for the 256-bit level, since the operands are then too big to fit in registers.

Interestingly, implementing this algorithm raised a few issues with NEON. The right shift and SELECT can be implemented efficiently using NEON; however, we had to resort to regular ARM instructions for addition and subtraction, since it is difficult to handle carries with NEON. This requires moving data back and forth from NEON to ARM registers; which can be costly. In the Cortex A8, since the NEON pipeline starts after the ARM pipeline, a move from NEON to ARM causes a 15+ cycles stall. The obvious approach to mitigate this would be to move from NEON to ARM beforehand, but this is difficult due to the limited number of ARM registers. Our approach was then to partially revert to storing operands in memory since it becomes faster to read from cached memory than to move data between NEON and ARM. In the Cortex A9 we followed the same approach, but with smaller gains, since the ARM and NEON pipelines are partly parallel and moving from NEON to ARM is not that costly (around 4 cycles of latency). However, the Cortex A15 is much more optimized in this sense and our original approach of keeping operands in registers was faster.

4.4 Authenticated Encryption

An authenticated encryption (AE) symmetric scheme provides both encryption and authentication using a single key, and is often more efficient and easy to employ than using two separate encryption and authentication schemes (e.g. AES-CTR with HMAC). The Galois/Counter Mode (GCM) [17] is an AE scheme which is built upon a block cipher, usually AES. It was standardized by NIST and is used in IPSec, SSH and TLS. For each message block, GCM encrypts it using the underlying block cipher in CTR mode and xors the ciphertext into an accumulator, which is then multiplied in $\mathbb{F}_{2^{128}}$ by a key-dependent constant. After processing the last block, this accumulator is used to generate the authentication tag.

We have implemented the $\mathbb{F}_{2^{128}}$ multiplication using the same techniques described above; modular reduction took advantage of the `VMULL` instruction since $r(z)$ in this field is small. We remark that our implementation does not uses precomputed tables (as it is often required for GCM) and is side-channel resistant (if the underlying block cipher also is). For benchmarking, we have used

Table 1. Our timings in cycles for binary field arithmetic

Algorithm/Processor		$\mathbb{F}_{2^{251}}$	$\mathbb{F}_{2^{283}}$	$\mathbb{F}_{2^{571}}$
Multiplication (LD)	A8	671	1,032	3,071
	A9	774	1,208	3,140
	A15	412	595	1,424
Multiplication (KNV)	A8	385	558	1,506
	A9	491	701	1,889
	A15	317	446	1,103
Squaring (Table)	A8	155	179	349
	A9	168	197	394
	A15	128	151	282
Squaring (VMULL)	A8	57	53	126
	A9	63	59	146
	A15	43	42	99
Inversion (Itoh-Tsujii)	A8	18,190	20,777	90,936
	A9	19,565	22,356	97,913
	A15	13,709	16,803	71,220

an assembly implementation of AES from OpenSSL without SCR; however this is not an issue since we are more interested in the overhead added by GCM to the plain AES encryption.

5 Results

To evaluate our software implementation, we have used a DevKit8000 board with an 600 MHz ARM Cortex-A8 processor, a PandaBoard board with a 1 GHz ARM Cortex-A9 processor and an Arndale board with a 1.7 GHz ARM Cortex-A15 processor. We have used the GCC 4.5.1 compiler. Our optimized code is written in the C and assembly languages using the RELIC library [1]. Each function is benchmarked with two nested loops with n iterations each; inside the outer loop, an input is randomly generated; and the given operation is executed n times in the inner loop using this input. The total time taken by this procedure, given by the `clock_gettime` function in nanoseconds, is divided by n^2 in order to give the final result for the given operation. We chose $n = 1024$ for measuring fast operations such as finite field arithmetic, and $n = 64$ for the slower operations such as point multiplication.

Table 1 presents the timings of field operations used in ECC. Our new Karatsuba/NEON/`VMULL` (KNV) multiplication gives a up to 45% improvement compared to the LD/NEON implementation. For field squaring, we have obtained a significant improvement of up to 70% compared to the conventional table lookup approach. The very fast squaring made the Itoh-Tsujii inversion feasible.

Table 2. Our timings in 10^3 cycles for elliptic curve protocols

Algorithm/Processor		B-251	B-283	K-283	B-571	K-571
ECDH Agreement	A8	657	1,097	934	5,731	4,870
	A9	789	1,350	1,148	7,094	6,018
	A15	511	866	736	4,242	3,603
ECDSA Sign	A8	458	624	606	2,770	2,673
	A9	442	612	602	2,880	2,816
	A15	233	337	330	1,740	1,688
ECSS Sign	A8	270	389	371	1,944	1,846
	A9	285	414	404	2,137	2,073
	A15	186	270	263	1,264	1,212
ECDSA Verify	A8	943	1,397	791	6,673	3,069
	A9	1,100	1,644	887	8,171	3,581
	A15	715	1,064	583	4,882	2,237
ECSS Verify	A8	933	1,337	735	6,338	3,064
	A9	1,086	1,572	827	7,776	3,602
	A15	715	1,022	546	4,623	2,228

Timings for ECC protocols are listed in Table 2, while Figure 3 plots the 128-bit level timings to aid visualization. Compared to the LD/NEON multiplier with table-based squaring, the KNV multiplication with `VMULL`-based squaring improved the point multiplication by up to 50%. ECDSA is 25–70% slower than ECSS due to the SCR modular inversion required.

When limited to standard NIST elliptic curves, our ECDH over K-283 is 70% faster than the results of Morozov et al. [20]. Considering non-standard curves, we now compare our binary B-251 to the prime curves in the state of the art; this is also shown in Table 3. On the A8, compared to Bernstein and Schwabe's [6], our key agreement is 25% slower; our signing is 26% faster; and our verification is 43% slower. On the A9, compared to Faz-Hernández et al. [7], our random point multiplication is 88% slower, our fixed point multiplication is 53% slower; and our simultaneous point multiplication is 132% slower. On the A15, also compared to Faz-Hernández et al. [7], our random point multiplication is 108% slower, our fixed point multiplication is 72% slower; and our simultaneous point multiplication is 162% slower. We remark that this is a comparison of our implementation of binary elliptic curves with the state-of-the-art prime elliptic curve implementations, which are very different. In particular, note that the arithmetic of prime curves can take advantage of native 32×32-bit and 64×64-bit multiply instructions.

For the GCM authenticated encryption scheme, we have obtained 38.6, 41.9 and 31.1 cycles per byte for large messages, for the A8, A9 and A15 respectively; a 13.7, 13.6 and 9.2 cpb overhead to AES-CTR. Our A8 overhead is 46% faster than the timing reported by Krovetz and Rogaway's [13] and 8.6% faster than [22].

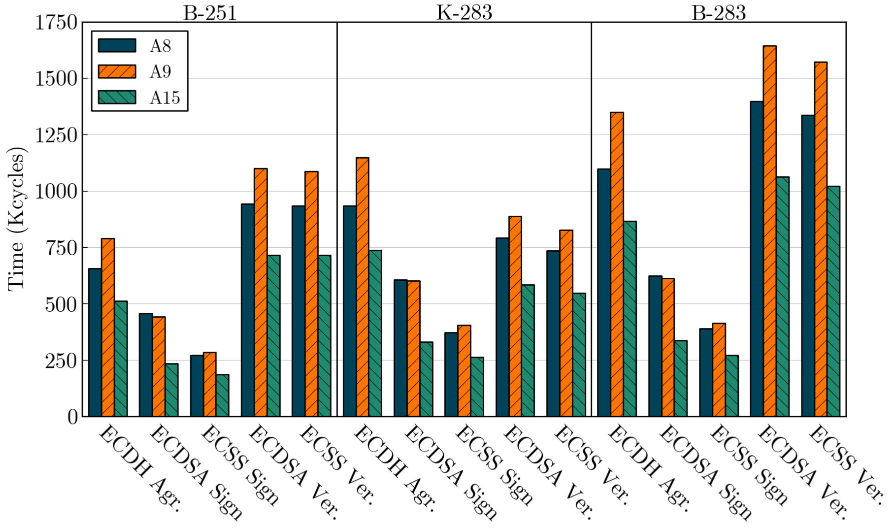

Fig. 3. Our timings for ECC algorithms at the 128-bit level of security

It is interesting to compare the timings across Cortex processors. The A9 results are often slower than the A8 results: while the A9 improved performance of regular ARM code, the lack of partial dual issue in NEON caused a visible drop in performance in NEON-based code, which is our case. (The exception is ECDSA signing where the A8 currently does not have much advantage in the modular inversion, which dilutes any savings in the point multiplication.) On the other hand, the return and expansion of NEON dual issue in A15 caused great performance gains (up to 40%).

6 Conclusions and Future Work

In this paper we have introduced a new multiplier for 64-bit binary polynomial multiplication using the `VMULL` instruction, part of the NEON engine present in many ARM processors in the Cortex-A series. We then explain how to use the new multiplier to improve the performance of finite field multiplication in $\mathbb{F}_{2^m}$. We have also shown the performance gains by the new multiplier in elliptic curve cryptography and authenticated encryption with GCM. We were unable to break speed records for non-standard elliptic curves, but we believe this work offers a useful insight in how binary curves compare to prime curves in ARM processors. For standard curves we were able to improve the state of the art, as well for the GCM authenticated encryption scheme.

An interesting venue for future research is on the implementation of standard prime curves for ARM, which seems to be lacking in the literature. In

Table 3. Our best ECC timings (on the non-standard elliptic curve B-251 over binary field) compared to state-of-the-art timings using non-standard elliptic curves over prime fields, at the 128-bit level of security, in 10^3 cycles

Algorithm/Processor		Ours	[6]	[9]	[7]
Key Agreement	A8	657	527		
	A9	789		616	417
	A15	511			244
Sign	A8	270	368		
	A9	285		262	172
	A15	186			100
Verify	A8	933	650		
	A9	1,086		605	463
	A15	715			266

addition, the arrival of ARMv8 processors in the future (including the Cortex A53 and A57) may provide great speed up to binary ECC, since the architecture will provide two instructions for the full 64-bit binary multiplier (PMULL and PMULL2) and will double the number of NEON registers [3].

References

1. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/
2. 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.) Progress in Cryptology — LATINCRYPT 2012, Lecture Notes in Computer Science, vol. 7533, pp. 177–193. Springer Berlin / Heidelberg (2012)
3. ARM Limited: ARMv8 instruction set overview (2012)
4. Barker, E., Johnson, D., Smid, M.: NIST SP 800-56A: Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (March 2007)
5. Bernstein, D.: Batch binary Edwards. In: Advances in Cryptology — CRYPTO 2009, Lecture Notes in Computer Science, vol. 5677, pp. 317–336. Springer Berlin / Heidelberg (2009)
6. Bernstein, D., Schwabe, P.: NEON crypto. In: Cryptographic Hardware and Embedded Systems — CHES 2012, Lecture Notes in Computer Science, vol. 7428, pp. 320–339. Springer Berlin / Heidelberg (2012)
7. Faz-Hernández, A., Longa, P., Sánchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. Cryptology ePrint Archive, Report 2013/158 (2013), http://eprint.iacr.org/
8. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Advances in Cryptology — CRYPTO

2001. Lecture Notes in Computer Science, vol. 2139, pp. 190–200. Springer Berlin / Heidelberg (2001)

9. Hamburg, M.: Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309 (2012), http://eprint.iacr.org/
10. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. Information and Computation 78(3), 171 – 177 (1988)
11. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Physics Doklady 7, 595 (1963)
12. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology — CRYPTO '96, Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer Berlin / Heidelberg (1996), http://dx.doi.org/10.1007/3-540-68697-5_9
13. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer Berlin / Heidelberg (2011)
14. Käsper, E.: Fast elliptic curve cryptography in OpenSSL. In: Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 7126, pp. 27–39. Springer Berlin / Heidelberg (2012)
15. López, J., Dahab, R.: High-speed software multiplication in $\mathbb{F}_{2^m}$. In: Progress in Cryptology — INDOCRYPT 2000. Lecture Notes in Computer Science, vol. 1977, pp. 93–102. Springer Berlin / Heidelberg (2000)
16. López, J., Dahab, R.: Fast multiplication on elliptic curves over GF(2^m) without precomputation. In: Cryptographic Hardware and Embedded Systems. Lecture Notes in Computer Science, vol. 1717, p. 724. Springer Berlin / Heidelberg (1999)
17. McGrew, D., Viega, J.: The security and performance of the Galois/Counter Mode (GCM) of operation. In: Progress in Cryptology — INDOCRYPT 2004, Lecture Notes in Computer Science, vol. 3348, pp. 377–413. Springer Berlin / Heidelberg (2005)
18. Möller, B.: Algorithms for multi-exponentiation. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 2259, pp. 165–180. Springer Berlin / Heidelberg (2001)
19. Möller, N.: Nettle, low-level cryptographics library. Nettle Git repository (2013), http://git.lysator.liu.se/nettle/nettle/blobs/9422a55130ba65f73a053f063efa6226f945b4f1/sec-modinv.c#line67
20. Morozov, S., Tergino, C., Schaumont, P.: System integration of elliptic curve cryptography on an OMAP platform. In: 2011 IEEE 9th Symposium on Application Specific Processors (SASP). pp. 52–57. IEEE (2011)
21. National Institute of Standards and Technology: FIPS 186-3: Digital signature standard (DSS) (June 2009), http://www.itl.nist.gov
22. Polyakov, A.: The OpenSSL project. OpenSSL Git repository (2013), http://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=crypto/modes/asm/ghash-armv4.pl;h=d91586ee2925bb695899b17bb8a7242aa3bf9150;hb=9575d1a91ad9dd6eb5c964365dfbb72dbd3d1333#l135
23. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology 4(3), 161–174 (1991)
24. Solinas, J.A.: Efficient arithmetic on Koblitz curves. Designs, Codes and Cryptography 19(2), 195–249 (2000)

A Reference Algorithms

Listed below are algorithms for reference and the full code of our multiplier.

Algorithm 3 Branchless select, described by Emilia Käsper in [14]

```
Input: $W$-bit words $a$, $b$, $v$, with $v \in \{0, 1\}$
Output: $b$ if $v$, else $a$
function SELECT$(a, b, v)$
    $m \leftarrow$ TWOSCOMPLEMENT$(-v, W)$    ▷ convert $-v$ to $W$-bit two's complement
    $t \leftarrow (m \,\&\, (a \oplus b)) \oplus a$
    return $t$
```

Algorithm 4 SCR table lookup, contained in the source code of Bernstein and Schwabe's [6]

```
Input: array $a$ with $n$ elements of any fixed type, desired index $k$, $0 \leq k < n$
Output: $a[k]$
function CHOOSE$(a, n, k)$
    $t \leftarrow a[0]$
    for $i \leftarrow 1$ to $n - 1$ do
        $r \leftarrow (i \oplus k) - 1$
        $s \leftarrow (r < 0)$                         ▷ $s$ holds whether $i$ is equal to $k$
        $t \leftarrow$ SELECT$(t, a[i], s)$
    return $t$
```

Algorithm 5 SCR modular inversion algorithm by Niels Möller in the Nettle library [19]

```
Input: integer $x$, odd integer $n$, $x < n$
Output: $x^{-1} \pmod{n}$
function MODINV$(x, n)$
    $(a, b, u, v) \leftarrow (x, n, 1, 1)$
    $\ell \leftarrow \lfloor \log_2 n \rfloor + 1$                 ▷ number of bits in $n$
    for $i \leftarrow 0$ to $2\ell - 1$ do
        odd $\leftarrow a \,\&\, 1$
        if odd and $a \geq b$ then
            $a \leftarrow a - b$
        else if odd and $a < b$ then
            $(a, b, u, v) \leftarrow (b - a, a, v, u)$
        $a \leftarrow a \gg 1$
        if odd then $u \leftarrow u - v$
        if $u < 0$ then $u \leftarrow u + n$
        if $u \,\&\, 1$ then $u \leftarrow u + n$
        $u \leftarrow u \gg 1$
    return $v$
```

Algorithm 6 Our proposed ARM NEON 64-bit binary multiplication $C = A \cdot B$ with 128-bit result

Input: 64-bit registers `ad` (holding A), `bd` (holding B), `k16` (holding the constant `0xFFFF`), `k32` (holding `0xFFFFFFFF`), `k48` (holding the constant `0xFFFFFFFFFFFF`).

Output: 128-bit register `rq (rh|rl)` (holding A).

Uses temporary 128-bit registers `t0q (t0h|t0l)`, `t1q (t1h|t1l)`, `t2q (t2h|t2l)`, `t3q (t3h|t3l)`.

```
vext.8   t0l, ad, ad, $1                          ▷ A1
vmull.p8 t0q, t0l, bd                             ▷ F = A1*B
vext.8   rl, bd, bd, $1                           ▷ B1
vmull.p8 rq, ad, rl                               ▷ E = A*B1
vext.8   t1l, ad, ad, $2                          ▷ A2
vmull.p8 t1q, t1l, bd                             ▷ H = A2*B
vext.8   t3l, bd, bd, $2                          ▷ B2
vmull.p8 t3q, ad, t3l                             ▷ G = A*B2
vext.8   t2l, ad, ad, $3                          ▷ A3
vmull.p8 t2q, t2l, bd                             ▷ J = A3*B
veor     t0q, t0q, rq                             ▷ L = E + F
vext.8   rl, bd, bd, $3                           ▷ B3
vmull.p8 rq, ad, rl                               ▷ I = A*B3
veor     t1q, t1q, t3q                            ▷ M = G + H
vext.8   t3l, bd, bd, $4                          ▷ B4
vmull.p8 t3q, ad, t3l                             ▷ K = A*B4
veor     t0l, t0l, t0h              ▷ t0 = (L) (P0 + P1) << 8
vand     t0h, t0h, k48
veor     t1l, t1l, t1h              ▷ t1 = (M) (P2 + P3) << 16
vand     t1h, t1h, k32
veor     t2q, t2q, rq                             ▷ N = I + J
veor     t0l, t0l, t0h
veor     t1l, t1l, t1h
veor     t2l, t2l, t2h              ▷ t2 = (N) (P4 + P5) << 24
vand     t2h, t2h, k16
veor     t3l, t3l, t3h              ▷ t3 = (K) (P6 + P7) << 32
vmov.i64 t3h, $0
vext.8   t0q, t0q, t0q, $15
veor     t2l, t2l, t2h
vext.8   t1q, t1q, t1q, $14
vmull.p8 rq, ad, bd                               ▷ D = A*B
vext.8   t2q, t2q, t2q, $13
vext.8   t3q, t3q, t3q, $12
veor     t0q, t0q, t1q
veor     t2q, t2q, t3q
veor     rq, rq, t0q
veor     rq, rq, t2q
```