Implementing GCM on ARMv8

Conrado P. L. Gouvêa[1] and Julio López[2⋆]

[1] KRYPTUS Information Security Solutions
[2] University of Campinas (Unicamp)
conradoplg@kryptus.com, jlopez@ic.unicamp.br

Abstract. The Galois/Counter Mode is an authenticated encryption scheme which is included in protocols such as TLS and IPSec. Its implementation requires multiplication over a binary finite field, an operation which is costly to implement in software. Recent processors have included instructions aimed to speed up binary polynomial multiplication, an operation which can be used to implement binary field multiplication. Some processors of the ARM architecture, which was reported in 2014 to be present in 95% of smartphones, include such instructions. In particular, recent devices such as the iPhone 5s and Galaxy Note 4 have ARMv8 processors, which provide instructions able to multiply two 64-bit binary polynomials and to encrypt using the AES cipher. In this work we present an optimized and timing-resistant implementation of GCM over AES-128 using these instructions. We have obtained timings of 1.71 cycles per byte for GCM authenticated encryption (9 times faster than the timing on ARMv7), 0.51 cycles per byte for GCM authentication only (11 times faster) and 1.21 cycles per byte for AES-128 encryption (8 times faster).

Keywords: GCM, authenticated encryption, ARM, efficient implementation

1 Introduction

Authenticated encryption (AE) schemes provide both confidentiality and authentication in a single algorithm, preventing common errors when combining separate encryption and authentication schemes. The Galois/Counter Mode (GCM) [6] is an authenticated encryption scheme included the TLS and IPSec protocols and in the NIST standard SP 800-38D. It uses an underlying block cipher which is usually AES.

ARM is a RISC processor architecture which ubiquitous in mobile devices due to its relatively low power consumption. The eighth version of the ARM architecture (ARMv8) is the first supporting 64-bit processing and it has become commercially available with the release of the iPhone 5s, a smartphone featuring an ARMv8 processor named Apple A7. Other devices with ARMv8 processors

⋆ The second author was partially supported by a research productivity scholarship from CNPq Brazil.

include the iPhone 6 and 6 Plus (Apple A8 processor), the iPad Air 2 (Apple A8X processor), the Galaxy Note 4 (Cortex A53/A57 processor) and the Nexus 9 (Nvidia Denver processor).

ARMv8 introduced many changes in the architecture, but one specific addition is a 64-bit multiplier capable of multiplying polynomials over $\mathbb{F}_2$, also known as binary polynomials. The GCM employs multiplications over the finite field $\mathbb{F}_{2^{128}}$ which in turn can be computed with the help of binary multiplication, raising the question of how the ARMv8 performs when running GCM. ARMv8 also supports instructions which are able to carry out AES encryption and decryption.

In this work, we present an efficient and timing-resistant implementation of GCM over AES-128 using the new ARMv8 multiplier and AES instructions along with the ARM vector instruction set (named NEON). We compare it to an optimized and timing-resistant implementation for the ARMv7 architecture based on previous works in the literature. We provide benchmarks of our three implementations (ARMv7, ARMv8 on 32-bit mode and ARMv8 on 64-bit mode) for six different processors: the ARMv7-based Cortex A9 and Cortex A15, along with the ARMv8-based Apple A7, Apple A8X and Cortex A53/A57. Our implementation is available online[3] to allow the reproduction of our results.

Related work. An efficient implementation of GCM for ARMv7 using the `VMULL.P8` NEON instruction, which computes eight 8×8-bit polynomial multiplications, is described in [1]. When encrypting and authenticating large messages they have achieved 38.6, 41.9 and 31.1 cycles per byte (cpb), for the Cortex A8, A9 and A15 processors respectively, using a non timing-resistant AES implementation. When using GCM for authentication only their results are 13.7, 13.6 and 9.2 cpb respectively.

Polyakov [7], working for the OpenSSL project, has improved on the [1] implementation by unrolling code, improving modular reduction and reordering instructions, obtaining 8.45 cpb on the Cortex A8, 10.2 cpb on the Cortex A9 and 9.33 cpb on the Snapdragon S4 for GCM authentication.

Recent Intel processors have added the `PCLMULQDQ` instruction, which is able to multiply 64-bit binary polynomials, akin to what is now supported by ARMv8. Gueron and Kounavis [4] describe how to implement GCM with this instruction, also taking advantage of the Intel AES-NI instructions which support AES. Gueron reports [3] 1.79, 1.79 and 0.4 cpb for authenticated encryption on the Sandy Bridge, Ivy Bridge and Haswell processors respectively.

Paper structure. Section 2 describes the ARM architecture, while Section 3 describes the GCM algorithm. Our software implementation is described in Section 4 and results are reported in 5. Concluding remarks are given in Section 6. Appendix A lists the pseudo-assembly code for algorithms described in this work.

[3] `https://github.com/conradoplg/authenc`

2 ARM Architecture

ARM is a well known family of RISC processor architectures introduced in 1985 which now holds 95% of the smartphone segment [8]. Up to version 7, ARM was a 32-bit architecture. Its most recent version, ARMv8, supports both 32-bit and 64-bit processing. The 32-bit ARMv8 architecture is known as AArch32, while the 64-bit is known as AArch64. An ARMv8 processor can support both, allowing the execution of 32-bit and 64-bit applications. ARM processors may also support a single-instruction multiple-data (SIMD) module called the “NEON engine”.

Each architecture is implemented by different core designs, and each core design can be implemented by different chips. For example, Cortex A9 and Apple A6 are two core designs following the ARMv7 architecture, while OMAP 4660 and Exynos 4 are chips implementing the same Cortex A9 core design.

ARMv7 and AArch32 feature sixteen 32-bit registers (`R0`–`R15`) and sixteen 128-bit NEON registers (`Q0`–`Q15`). The NEON registers can also be viewed as pairs of 64-bit registers (`D0`–`D32`) such that, for example, `D0` is the lower part of `Q0` and `D1` is its higher part.

AArch64 features thirty two 64-bit registers (`X0`–`X31`) and thirty two 128-bit NEON registers (`V0`–`V31`). The NEON registers can no longer be viewed as pairs of 64-bit registers, though the lower part of each register can be referenced as `D0`–`D15`.

2.1 Binary Polynomial Multiplication Support

When restricted to ARMv7, the instruction `VMULL.P8` is critical for the efficient implementation of GCM, as shown in [1]. Its inputs are two 64-bit NEON registers, each interpreted as eight 8-bit binary polynomials, and its output is a single 128-bit NEON register interpreted as eight 16-bit binary polynomials containing the eight results of pairwise binary multiplications, as illustrated in Figure 1. In [1] it is shown how to compute a full 64×64-bit multiplication with eight `VMULL.P8` instructions and some additional data processing.

AArch32 provides a new `VMULL.P64` instruction. Its inputs are two 64-bit NEON registers, each interpreted as a single 64-bit binary polynomial, and its output is a single 128-bit NEON register containing the result of the binary multiplication of both operands. Therefore, a single instruction can carry out a 64×64-bit multiplication. It is also shown in Figure 1.

AArch64 provides two instructions, `PMULL` and `PMULL2`, both of which carry out a single 64×64-bit multiplication. In both cases, the inputs are 128-bit registers; their difference is that in `PMULL` the lower 64-bit parts of the inputs are used as operands, while in `PMULL2` the higher 64-bit parts are used. Both are also shown in Figure 1.

2.2 AES Support

ARMv7 does not have any specific AES support, while ARMv8 supports AES in both AArch32 and AArch64 with the `AESE`, `AESD`, `AESMC` and `AESIMC` NEON

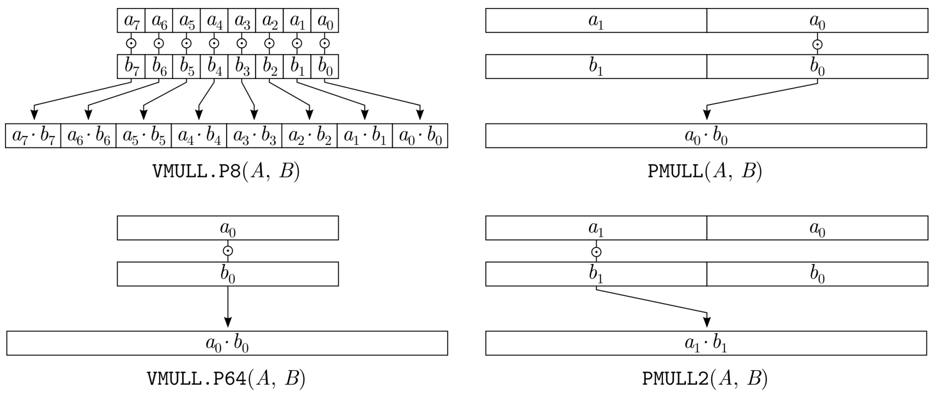
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
$a_7 \cdot b_7$ $a_6 \cdot b_6$ $a_5 \cdot b_5$ $a_4 \cdot b_4$ $a_3 \cdot b_3$ $a_2 \cdot b_2$ $a_1 \cdot b_1$ $a_0 \cdot b_0$
VMULL.P8(A, B)
a_1 a_0
b_1 b_0
$a_0 \cdot b_0$
PMULL(A, B)
a_0
b_0
$a_0 \cdot b_0$
VMULL.P64(A, B)
a_1 a_0
b_1 b_0
$a_1 \cdot b_1$
PMULL2(A, B)

Fig. 1. Binary polynomial multiplication NEON instructions: `VMULL.P8` (ARMv7), `VMULL.P64` (ARMv8 AArch32) and `PMULL/PMULL2` (ARMv8 AArch64).

instructions. `AESE` performs the AddRoundKey, SubBytes and ShiftRows AES steps, while `AESMC` performs MixColumns. The last AddRoundKey step can be carried out with a regular NEON XOR instruction (`VEOR` in AArch32, `EOR` in AArch64).

Instruction support is summarized in Table 1.

Table 1. Instruction support across devices. Assumes NEON support (not every chip supports it).

	ARMv7	ARMv8	
		AArch32	AArch64
`VMULL.P8`	Yes	Yes	No
`VMULL.P64`	No	Yes	No
`PMULL/PMULL2`	No	No	Yes
AES instructions	No	Yes	Yes

3 GCM

The Galois/Counter Mode (GCM) [6] is an authenticated encryption scheme which is built upon a block cipher, usually AES. Its inputs are: a key; a nonce; the plaintext which will be encrypted and authenticated; and additional data which will only be authenticated. It outputs the ciphertext and an 128-bit authentication tag.

Algorithm 1 GHASH function

```
Input: Input $X$ with $n$ 128-bit blocks, 128-bit initial value $Y$, 128-bit constant $H$
Output: Updated $Y$
1: function GHASH($X, Y, H$)
2:     $X_1, \ldots, X_n \leftarrow X$
3:     for $i \leftarrow 1$ to $n$ do
4:         $Y \leftarrow (X_i \oplus Y) \otimes H$                ▷ Multiplication in $\mathbb{F}_{2^{128}}$
5:     return $Y$
```

The GCM encryption is based on the underlying block cipher in CTR mode, while its authentication is based on a function named GHASH, described in Algorithm 1. The GHASH inputs are: the ciphertext; an initial 128-bit value; and a 128-bit constant H derived from the key. It outputs a 128-bit value used to compute the authentication tag. The timing consuming operation in GHASH is the multiplication over the binary field $\mathbb{F}_{2^{128}}$; the fact that one of the operands is always the same (H) can lead into some optimizations.

An often confusing aspect of GCM is its bit order. When interpreting a byte vector as a binary field element, two choices must be made: which byte is the least significant in the vector, and which bit is the least significant in each byte. GCM chooses a little-endian approach for the byte order, but a big-endian approach for the bit order. For example, the polynomial $a(z) = 1$ is represented as the 16-byte string `80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`. However, this means that is not possible to speed up the computation using anything larger than bytes (i.e. words), since left and right shifts will not respect the byte/bit order of GCM. This can be solved by either of two approaches: the first is to reverse the bits inside each byte in the GHASH input before carrying out any computations, and reversing them again when computing the authentication tag. The second approach, proposed by [4], is to reverse the bytes in the vector instead. This leads to a reversed binary field element, which can be correctly multiplied by reversing the modular reduction algorithm in the binary field multiplication. The advantage of the second approach is that in most cases it is simpler to reverse the bytes in a vector than reversing the bits in each byte.

4 Software Implementation

Efficiency is important, but security is even more. A secure implementation offers some degree of protection against side channel attacks. Our aim is to protect against timing attacks by avoiding loops, branches and table lookups which are dependent on secret data.

Given an irreducible m-degree polynomial $f(z)$, the binary field $\mathbb{F}_{2^{128}}$ can be defined as the set of polynomials with degree at most $m-1$ over $\mathbb{F}_2$. In software, each field element can be stored in a vector of W-bit words, where each word contains W coefficients of the polynomial. Field addition is simply the xor of the operands. Field multiplication consists of the polynomial multiplication of the operands, followed by a reduction modulo an irreducible polynomial. GCM uses multiplication in the binary field $\mathbb{F}_{2^{128}}$ with the irreducible polynomial $f(z) = z^{128} + z^7 + z^2 + z + 1$.

4.1 Binary Polynomial Multiplication

The polynomial multiplication used to be carried out with the help of precomputed tables, shifts and xors [5]. However, this has changed with the advent of instructions supporting binary polynomial multiplication. In ARMv7, we simply followed the approach described in [1], which builds a 64×64-bit multiplier using eight invocations of the `VMULL.P8` instruction. This multiplier is then used three times (with the Karatsuba algorithm) to build the full 128×128-bit multiplier.

In AArch32, the whole 64×64-bit multiplication is available in the `VMULL.P64` instruction. Again, we use Karatsuba to build the full 128×128-bit multiplier, which is listed in Algorithm 2, using three `VMULL.P64` calls. In AArch64, however, our approach is a little different. While the `PMULL` and `PMULL2` instructions offer the same operation of `VMULL.P64`, in AArch64 is no longer possible to reference the upper part of a 128-bit register as a separate 64-bit register. In order to get the upper part we use the `EXT` instruction with a register zeroed beforehand. To reduce the use of `EXT` we abandon Karatsuba and call `PMULL(2)` four times. Our AArch64 multiplier is listed in Algorithm 3.

Algorithm 2 128×128-bit binary polynomial multiplier for ARMv8 AArch32 (`VMULL.P64`)

Input: 128-bit registers `aq (ah|al)` (first operand), `bq (bh|bl)` (second operand).
Output: 128-bit registers `r0q (r0h|r0l)` (lower 128 bits of the result), `r1q (r1h|r1l)` (higher 128 bits of the result).
Uses temporary 128-bit register `tq (th|tl)`.

```
1: vmull.p64 r0q, al, bl
2: vmull.p64 r1q, ah, bh
3: veor th, bl, bh
4: veor tl, al, ah
5: vmull.p64 tq, th, tl
6: veor tq, r0q
7: veor tq, r1q
8: veor r0h, tl
9: veor r1l, th
```

4.2 GCM reflection

In ARMv7 there is no straightforward procedure to reflect the bits of each byte in a byte vector using NEON instructions. For this reason, in order to avoid the costly lookup tables required for the reflection, we used the reflection trick described in [4] which requires inverting the bytes of 16-byte vectors. This is carried

Algorithm 3 128×128-bit binary polynomial multiplier for ARMv8 AArch64 (`PMULL`)

```
Input: 128-bit registers a (first operand), b (second operand), z (zeroed register).
Output: 128-bit registers r0 (lower 128 bits of the result), r1 (higher 128 bits of the
    result).
    Uses temporary 128-bit registers t0, t1.
 1: pmull r0.1q, a.1d, b.1d
 2: pmull2 r1.1q, a.2d, b.2d
 3: ext.16b t0, b, b, #8
 4: pmull t1.1q, a.1d, t0.1d
 5: pmull2 t0.1q, a.2d, t0.2d
 6: eor.16b t0, t0, t1
 7: ext.16b t1, z, t0, #8
 8: eor.16b r0, r0, t1
 9: ext.16b t1, t0, z, #8
10: eor.16b r1, r1, t1
```

out in NEON using the `VREV64.8` instruction, which can reverse the bytes inside a pair of 64-bit registers. These registers can then be swapped using the `VSWP` instruction to finish the procedure. In ARMv8 AArch32 the same approach is used. ARMv8 AArch64, however, supports the `RBIT` instruction which reverses the bits of each byte in a byte vector (including 16-byte vectors). This is exactly what is required by GCM and in this case the reflection trick is no longer necessary.

4.3 Modular Reduction

Modular reduction is slightly more complex. Write the GCM modulus as $f(z) = z^{128} + r(z)$, where $r(z) = z^7 + z^2 + z + 1$. The well known approach is to consider that $z^{128} \equiv r(z) \pmod{z^{128} + r(z)}$, allowing us to write the 256-bit operand to be reduced as $a(z) = h(z)z^{128} + \ell(z) \equiv h(z)r(z) + \ell(z)$. That is, we simply multiply the higher part of the operand by $r(z)$ and add it to $\ell(z)$. If the result is still larger than 128 bits, we reduce again.

The old approach to compute the multiplication by $r(z)$ is to use shift and xors. However, it is now possible to simply compute it with the binary multiplication instructions, using the constant $r(z)$ as one of the operands; this is the approach we used on both ARMv8 AArch32 and AArch64 with `VMULL.P64` and `PMULL`. Nevertheless, on ARMv7, the shift and xors approach is slightly faster due to the non straightforward use of the `VMULL.P8` multiplier required to carry out the reduction. Our reduction algorithms are listed in Algorithms 4 and 5. The ARMv7 reduction is from [7] and is listed for reference in Algorithm 8 in the Appendix.

Lazy reduction. On ARMv8 the polynomial multiplication is very fast due to the instruction support for 64-bit polynomial multiplication. For this reason,

Algorithm 4 256-bit to 128-bit GCM reflected polynomial reduction for ARMv8 AArch32 using `VMULL.P64`

Input: 128-bit registers `r0q (r0h|r0l)` (lower 128 bits of the operand), `r1q (r1h|r1l)` (higher 128 bits of the operand), 64-bit register `pd` (holding constant `0xc200000000000000`)
Output: 128-bit register `aq (ah|al)`.
Uses temporary 128-bit register `t0 (t0h|t0l)`. Clobbers inputs.

```
1: vmull.p64 t0q, r0l, pd
2: veor r0h, t0l
3: veor r1l, t0h
4: vmull.p64 t0q, r0h, pd
5: veor r1q, t0q
6: veor aq, r0q, r1q
```

Algorithm 5 256-bit to 128-bit GCM polynomial reduction for ARMv8 AArch64 using `PMULL`

Input: 128-bit registers `r0` (lower 128 bits of the operand), `r1` (higher 128 bits of the operand), `p` (holding constant `0x00000000000000870000000000000087`), `z` (zeroed)
Output: 128-bit register `a`.
Uses temporary 128-bit registers `t0`, `t1`. Clobbers inputs.

```
1: pmull2 t0.1q, r1.2d, p.2d
2: ext t1.16b, t0.16b, z.16b, #8
3: eor.16b r1, r1, t1
4: ext t1.16b, z.16b, t0.16b, #8
5: eor.16b r0, r0, t1
6: pmull t0.1q, a1.1d, p.1d
7: eor.16b a, r0, t0
```

the modular reduction becomes comparatively more expensive and starts to dominate the running time of the field multiplication. Thus we have employed the technique known as "lazy reduction", described in the GCM context in [4]. It requires the unrolling of the GHASH function, as follows. GHASH can be written recursively as $Y_i = (X_i \oplus Y_{i-1}) \otimes H$ which, by decoupling the field multiplication, becomes $Y_i = (X_i \oplus Y_{i-1}) \cdot H \bmod f(z)$. This can be unrolled, for example, as $Y_i = [(X_i \cdot H) \oplus (X_{i-1} \cdot H^2) \oplus (X_{i-2} \cdot H^3) \oplus (X_{i-3} \cdot H^4)] \bmod f(z)$ which requires a single modular reduction for every four polynomial multiplications. This requires 256-bit polynomial addition, which is simple, and the precomputation of powers of the H value, which can be precomputed in advance since H only depends on the key. In our implementation we use a eightfold unrolling of GHASH; the powers of H are entirely kept in NEON registers during the GHASH computation.

4.4 AES

ARMv7 does not have any AES instructions. For this reason, we have used a bitsliced, timing-resistant NEON-based implementation from Bernstein and Schwabe available in SUPERCOP[4].

ARMv8 does support AES with special instructions, as mentioned. The 128-bit key AES with its 10 rounds requires ten AESE instructions, nine AESMC and one xor, totaling twenty instructions. Since these instructions operate on the same 128-bit value they present a great deal of dependency between then, slowing their execution. For this reason, we interleave two AES block encryptions in order to reduce these dependencies (this is possible since GCM uses the counter mode for encryption). For efficiency, we keep the entire AES expanded key in NEON registers throughout the encryption. Algorithm 9 in the Appendix illustrates the use of AES instructions in the encryption of a single block in AES-CTR.

Contrasting with the Intel AES instructions, ARMv8 does not offer instructions for computing the AES key schedule, which therefore must be implemented. The key schedule requires S-box lookups, which are usually implemented with precomputed tables. However, this approach is subject to timing attacks since the indexes are secret. (This may seem to be a non-issue since the key schedule is usually run only once for each key, which makes side channel attacks very difficult. However, we believe it is best to not rely on assumptions on the cipher usage — for example, a simplified API could run the key schedule for every encryption.) In order to offer timing resistance and good performance, we have implemented the key schedule with the help of a function which is able to lookup four bytes in the S-box. It is possible to write this function based on AES instructions, as described for the Intel processor in [2], but we need to adapt the code to the ARM/NEON architecture. We implemented it with the `AESE` instruction by observing that, when used with a zeroed round key, the instruction simply lookups sixteen bytes in the S-box and shuffles them with MixColumns. By unshuffling the bytes it is possible to build our lookup function with `AESE` in a timing-resistant manner. While it could be possible to lookup sixteen bytes with a single instruction, we use only four, since the AES key schedule works in groups of four bytes.

Our four-byte S-box lookup algorithm works by filling a 16-byte register with the constant `0x52` in each byte, then writing the four bytes to look up in the lower four bytes. `AESE` is called with a zero round key, leading to a shuffled 16-byte substituted result. Each `0x52` in the input is substituted by zeroes with `AESE`, and the four substituted bytes we are interested in are now in different columns due to MixColumns. We now simply add the columns (recall that AES works in column-major order) in order to get a four-byte result. Our code is listed in Algorithms 6 and 7.

[4] `http://bench.cr.yp.to/supercop.html`

Algorithm 6 Four-byte AES S-box lookup for ARMv8 AArch32 using `AESE`

```
Input: 32-bit register r0 (four bytes to lookup in the S-box).
Output: 32-bit register r0 (values after lookup).
    Uses temporary 128-bit registers q0 (d1|d0, s3|s2|s1|s0), q1.
1: vmov.i8 q0, #0x52
2: vmov.i8 q1, #0
3: vmov s0, r0
4: aese q0, q1
5: veor d0, d1
6: vpadd.i32 d0, d0, d1
7: vmov r0, s0
```

Algorithm 7 Four-byte AES S-box lookup for ARMv8 AArch64 using `AESE`

```
Input: 32-bit register w0 (four bytes to lookup in the S-box).
Output: 32-bit register w0 (values after lookup).
    Uses temporary 128-bit registers v0, v1.
1: movi.16b v0, #0x52
2: movi.16b v1, #0
3: mov v0.S[0], w0
4: aese.16b v0, v1
5: addv s0, v0.4s
6: mov w0, v0.S[0]
```

4.5 GCM

With the binary multiplier and the AES encryption, the rest of the GCM implementation is straightforward. There are two approaches: encryption can be interleaved with GHASH, or the encryption is completely carried out and then GHASH is executed. We have followed the latter approach, since it allows to keep the all AES round keys in NEON registers throughout the encryption. The former approach has the advantage that each block of ciphertext is written once and is never read again by GCM (since GHASH can read it directly from a NEON register after the encryption); we have not tried this approach but we believe it would be slower.

The first approach does imply that is not possible to use a streaming API that encrypts or decrypts an arbitrary amount of data, since the entire message must be encrypted and the ciphertext must be read from the start in order to compute its GHASH. However, we think this is not a problem since streaming APIs can be dangerous: they release plaintext to the user before authenticating it, burdening the user with the task of destroying any plaintext stored if it is found to be not authentic. It is still possible to build a streaming API by dividing the plaintext in packets and encrypting and authenticating them separately.

Table 2. Results in cycles per byte (except key setup, given in cycles)

Cortex	A9 ARMv7	A15 ARMv7	A53 ARMv8 AArch32		A57 ARMv8 AArch32	
	`.P8`[a]	`.P8`[a]	`.P8`[a]	`.P64`[b]	`.P8`[a]	`.P64`[b]
AES-128-CTR	22.0	15.6	22.3	1.88	15.6	1.84
AES-128 setup	3358	2437	3244	690	2386	647
GCM auth only	10.7	8.3	9.6	1.21	8.1	0.95
GCM encryption	32.8	23.9	32.5	3.08	23.4	2.78
GCM setup	6450	4651	6337	1423	4582	1216

Apple	A7 ARMv8			A8X ARMv8		
	AArch32		AArch64	AArch32		AArch64
	`.P8`[a]	`.P64`[b]	`PMULL`[c]	`.P8`[a]	`.P64`[b]	`PMULL`[c]
AES-128-CTR	9.8	1.21	1.21	9.8	1.19	1.19
AES-128 setup	1420	901	739	1419	875	749
GCM auth only	6.0	0.51	0.50	5.9	0.48	0.51
GCM encryption	15.9	1.71	1.71	15.7	1.68	1.70
GCM setup	2771	1298	1075	2706	1323	1062

[a] Bitsliced software AES; `VMULL.P8`-based binary multiplier
[b] AES instructions; `VMULL.P64`-based binary multiplier
[c] AES instructions; `PMULL`-based binary multiplier

5 Performance Results

Timings were obtained by measuring the time taken by the encryption of a 10,000-byte message inside a loop with 256 iterations. We did not use the popular SUPERCOP benchmark tool since it does not support the iOS and Android operating systems which were required for our experiments. The performance was measured on a PandaBoard board with a 1 GHz ARMv7 Cortex-A9 processor, an Arndale board with a 1.7 GHz ARMv7 Cortex-A15 processor, a Galaxy Note 4 SM-N910C with a hybrid 8-core processor (four 1.3 GHz ARMv8 Cortex-A53 cores and four 1.9 GHz ARMv8 Cortex-A57 cores — all of them supporting only the AArch32 mode), an iPhone 5s with a 1.3 GHz ARMv8 Apple A7 processor and an iPad Air 2 with a 1.5 GHz ARMv8 Apple A8X processor. On the Note 4 and Cortex boards time was measured with the `clock_gettime` function, while on the iPhone and iPad we used the `mach_absolute_time` function. Timings were converted to cycles assuming the clocks listed. On the Note 4 we forced our program to run on a specific core using the `sched_setaffinity` function. Our results are reported in Table 2. Algorithm timings do not include the time required for key setup, which is listed separately.

First, note that simply changing from the Cortex A15 to the Apple A7 processor, using the same implementation, leads to a 33% speedup in GCM. This can be attributed in advancements in the processor design, with a larger number of instructions being issued on the same cycle. However, changing from the Cortex A15 to the Cortex A57 leads to small timing differences, which probably means that the design of these processors are mostly the same, apart from the ARMv8 AArch32 support of the latter. We also found no significant difference between the results for the Apple A7 and A8X processors. The Cortex A53 is slower than the A57 by up to 10% (`.P64`) and 40% (`.P8`) as expected, since it's supposed to be a simpler core which consumes less energy.

Comparing the `VMULL.P8` and `VMULL.P64` implementations on the Apple A7, it can be seen that GCM authentication is 11.76 times faster using the new `VMULL.P64` instruction; AES-128-CTR encryption is 8.1 faster using the new AES instructions. Combined, these results lead to GCM authenticated encryption being 9.3 faster. The A8X processor shows similar results. On the Cortex A53 these speedups are 7.9, 11.9 and 10.6 while on the A57 they are 8.5, 8.5 and 8.4 respectively.

The `VMULL.P64` and `PMULL` implementations offer practically the same performance. This is not surprising since the implementations do not differ greatly. The 64-bit architecture does not make much difference since our code is mostly NEON and practically does not uses regular ARM registers for data processing.

Finally, when comparing the Apple A7 and Cortex A57 processors, we observe that the first is 46% faster for `VMULL.P64` GCM authentication and 34% faster for AES-128-CTR with AES instructions, leading to a 38% faster GCM authenticated encryption.

6 Conclusions and Future Work

The GCM mode is known for the hardness in implementing it in an efficient and secure manner. However, the ARMv8 binary polynomial multiplication and AES instructions can make GCM up to 10 times faster compared to an efficient timing-resistant ARMv7 implementation, making the scheme an ideal choice for protecting communications in smartphones. These instructions enable a natural resistance against timing attacks, since no branches nor table lookups are required.

Finally, our techniques for binary multiplication in $\mathbb{F}_{2^{128}}$ can be extended for larger binary fields which can be used for a fast and secure software implementation of binary elliptic curves.

References

1. Câmara, D., Gouvêa, C.P.L., López, J., Dahab, R.: Fast software polynomial multiplication on ARM processors using the NEON engine. In: Security Engineering and Intelligence Informatics, Lecture Notes in Computer Science, vol. 8128, pp. 137–154. Springer Berlin / Heidelberg (2013)

2. Gueron, S.: Intel's new AES instructions for enhanced performance and security. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 5665, pp. 51–66. Springer Berlin Heidelberg (2009)
3. Gueron, S.: AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR competition. Presented in DIAC 2013: Directions in Authenticated Ciphers (2014), http://2013.diac.cr.yp.to/slides/gueron.pdf
4. Gueron, S., Kounavis, M.E.: Intel carry-less multiplication instruction and its usage for computing the GCM mode. White Paper (2010)
5. 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)
6. 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)
7. Polyakov, A.: The OpenSSL project. OpenSSL Git repository (2014), http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=f8cee9d08181f9e966ef01d3b69ba78b6cb7c8a8
8. Ranger, S.: Internet of things and wearables drive growth for ARM. ZDNet (April 2014), http://www.zdnet.com/internet-of-things-and-wearables-drive-growth-for-arm-7000028684/

A Additional Algorithms

Algorithm 8 256-bit to 128-bit GCM reflected polynomial reduction for ARMv7 from [7]

Input: 128-bit registers `r0q (r0h|r0l)` (lower 128 bits of the operand), `r1q (r1h|r1l)` (higher 128 bits of the operand).

Output: 128-bit register `aq (ah|al)`.

Uses temporary 128-bit registers `t0 (t0h|t0l)`, `t1 (t1h|t1l)`. Clobbers inputs.

```
 1: vshl.i64 t0q, r0q, #57
 2: vshl.i64 t1q, r0q, #62
 3: veor t1q, t1q, t0q
 4: vshl.i64 t0q, r0q, #63
 5: veor t1q, t1q, t0q
 6: veor r0h, r0h, t1l
 7: veor r1l, r1l, t1h
 8: vshr.u64 t1q, r0q, #1
 9: veor r1q, r1q, r0q
10: veor r0q, r0q, t1q
11: vshr.u64 t1q, t1q, #6
12: vshr.u64 r0q, r0q, #1
13: veor r0q, r0q, r1q
14: veor aq, r0q, t1q
```

Algorithm 9 AES-128-CTR encryption of a single block with AES instructions

Input: 128-bit registers `k0-k10` (AES round keys), `ctr` (counter); regular ARM registers `in` (pointer to 128-bit input block), `out` (pointer to 128-bit output block)

Output: Encrypted counter xored with input written to memory pointed by `out`. Uses temporary 128-bit registers `t0`, `t1`.

```
mov.16b t0, ctr
aese.16b t0, k00
aesmc.16b t0, t0
aese.16b t0, k01
aesmc.16b t0, t0
aese.16b t0, k02
aesmc.16b t0, t0
aese.16b t0, k03
aesmc.16b t0, t0
aese.16b t0, k04
aesmc.16b t0, t0
aese.16b t0, k05
aesmc.16b t0, t0
aese.16b t0, k06
aesmc.16b t0, t0
aese.16b t0, k07
aesmc.16b t0, t0
aese.16b t0, k08
aesmc.16b t0, t0
aese.16b t0, k09
eor.16b t0, t0, k10
ld1.16b {t1}, [in], #16
eor.16b t1, t1, t0
st1.16b {t1}, [out], #16
```